Hinc incipit algorismus.
Haec algorismus ars praesens dicitur in qua
talibus indorum fruimur bis quinque figuris
0. 9. 8. 7. 6. 5. 4. 3. 2. 1.

— Friar Alexander de Villa Dei, *Carmen de Algorismo*, (c. 1220)

We should explain, before proceeding, that it is not our object to consider this program with reference to the actual arrangement of the data on the Variables of the engine, but simply as an abstract question of the nature *and* number *of the* operations *required to be perfomed during its complete solution.*

— Ada Augusta Byron King, Countess of Lovelace, translator's notes for Luigi F. Menabrea, "Sketch of the Analytical Engine invented by Charles Babbage, Esq." (1843)

You are right to demand that an artist engage his work consciously, but you confuse two different things: solving the problem and correctly posing the question.

— Anton Chekhov, in a letter to A. S. Suvorin (October 27, 1888)

The more we reduce ourselves to machines in the lower things,
the more force we shall set free to use in the higher.

— Anna C. Brackett, *The Technique of Rest* (1892)

The moment a man begins to talk about technique
that's proof that he is fresh out of ideas.

— Raymond Chandler

0 Introduction

0.1 What is an algorithm?

An algorithm is an explicit, precise, unambiguous, mechanically-executable sequence of elementary instructions. For example, here is an algorithm for singing that annoying song "99 Bottles of Beer on the Wall", for arbitrary values of 99:

```
BottlesOfBeer(n):
  For i ← n down to 1
      Sing "i bottles of beer on the wall, i bottles of beer,"
      Sing "Take one down, pass it around, i − 1 bottles of beer on the wall."
  Sing "No bottles of beer on the wall, no bottles of beer,"
  Sing "Go to the store, buy some more, n bottles of beer on the wall."
```

The word "algorithm" does *not* derive, as algorithmophobic classicists might guess, from the Greek roots *arithmos* (ἀριθμός), meaning "number", and *algos* (ἄλγος), meaning "pain". Rather, it is a corruption of the name of the 9th century Persian mathematician Abū 'Abd Allāh Muḥammad ibn Mūsā al-Khwārizmī.[1] Al-Khwārizmī is perhaps best known as the writer of the treatise *Al-Kitāb al-mukhtaṣar fīhīsāb al-ğabr wa'l-muqābala*[2], from which the modern word *algebra* derives. In another treatise, al-Khwārizmī popularized the modern decimal system for writing and manipulating numbers—in particular, the use of a small circle or *ṣifr* to represent a missing quantity—which had originated in India several

[1] 'Mohammad, father of Adbdulla, son of Moses, the Kwārizmian'. Kwārizm is an ancient city, now called Khiva, in the Khorezm Province of Uzbekistan.

[2] 'The Compendious Book on Calculation by Completion and Balancing'.

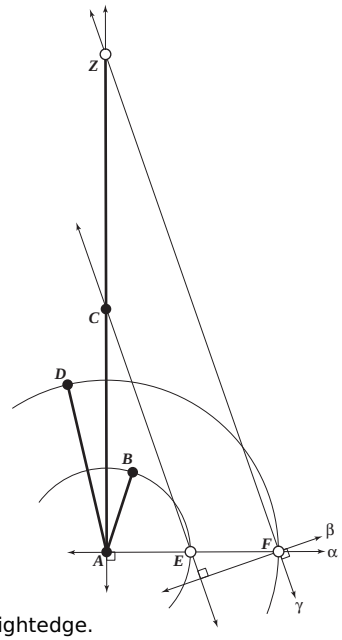
centuries earlier. This system later became known in Europe as *algorism*, and its figures became known in English as *ciphers*.[3] Thanks to the efforts of the medieval Italian mathematician Leonardo of Pisa, better known as Fibonacci, algorism began to replace the abacus as the preferred system of commercial calculation in Europe in the late 12th century. (Indeed, the word *calculate* derives from the Latin word *calculus*, meaning "small rock", referring to the stones on a counting board, or abacus.) Ciphers became truly ubiquitous in Western Europe only after the French revolution 600 years after Fibonacci. The more modern word *algorithm* is a false cognate with the Greek word *arithmos* (ἀριθμός), meaning 'number' (and perhaps the aforementioned ἄλγος).[4] Thus, until very recently, the word *algorithm* referred exclusively to pencil-and-paper methods for numerical calculations. People trained in the reliable execution of these methods were called—you guessed it—*computers*.[5]

0.2 A Few Simple Examples

Multiplication by compass and straightedge

Although they have only been an object of formal study for a few decades, algorithms have been with us since the dawn of civilization, for centuries before Al-Khwārizmī and Fibonacci popularized the cypher. Here is an algorithm, popularized (but almost certainly not discovered) by Euclid about 2500 years ago, for multiplying or dividing numbers using a ruler and compass. The Greek geometers represented numbers using line segments of the appropriate length. In the pseudo-code below, CIRCLE(p, q) represents the circle centered at a point p and passing through another point q. Hopefully the other instructions are obvious.[6]

```
⟨⟨Construct the line perpendicular to ℓ and passing through P.⟩⟩
RIGHTANGLE(ℓ, P):
  Choose a point A ∈ ℓ
  A, B ← INTERSECT(CIRCLE(P, A), ℓ)
  C, D ← INTERSECT(CIRCLE(A, B), CIRCLE(B, A))
  return LINE(C, D)

⟨⟨Construct a point Z such that |AZ| = |AC||AD|/|AB|.⟩⟩
MULTIPLYORDIVIDE(A, B, C, D):
  α ← RIGHTANGLE(LINE(A, C), A)
  E ← INTERSECT(CIRCLE(A, B), α)
  F ← INTERSECT(CIRCLE(A, D), α)
  β ← RIGHTANGLE(LINE(E, C), F)
  γ ← RIGHTANGLE(β, F)
  return INTERSECT(γ, LINE(A, C))
```



Multiplying or dividing using a compass and straightedge.

[3]The Italians transliterated *ṣifr* as *zefiro*, which later evolved into the modern *zero*.

[4]In fact, some medieval English sources claim the Greek prefix "algo-" meant "art" or "introduction". Other sources claimed that algorithms was invented by a Greek philosopher, or a king of India, or perhaps a king of Spain, named "Algus" or "Algor" or "Argus". A few, possibly including Dante Alighieri, even identified the inventor with the mythological Greek shipbuilder and eponymous argonaut. I don't think any serious medieval scholars made the connection to the Greek work for pain, although I'm quite certain their students did.

[5]From the Latin verb *putāre*, which variously means "to trim/prune", "to clean", "to arrange", "to value", "to judge", and "to consider/suppose"; also the source of the English words "dispute", "reputation", and "amputate".

[6]Euclid and his students almost certainly drew their constructions on an ἄβαξ, a table covered in dust or sand (or perhaps very small rocks). Over the next several centuries, the Greek *abax* evolved into the medieval European *abacus*.

This algorithm breaks down the difficult task of multiplication into a series of simple primitive operations: drawing a line between two points, drawing a circle with a given center and boundary point, and so on. These primitive steps are quite non-trivial to execute on a modern digital computer, but this algorithm wasn't designed for a digital computer; it was designed for the Platonic Ideal Classical Greek Mathematician, wielding the Platonic Ideal Compass and the Platonic Ideal Straightedge. In this example, Euclid first defines a new primitive operation, constructing a right angle, by (as modern programmers would put it) writing a subroutine.

Multiplication by duplation and mediation

Here is an even older algorithm for multiplying large numbers, sometimes called *(Russian) peasant multiplication*. A variant of this method was copied into the Rhind papyrus by the Egyptian scribe Ahmes around 1650 BC, from a document he claimed was (then) about 350 years old. This was the most common method of calculation by Europeans before Fibonacci's introduction of Arabic numerals; it was still taught in elementary schools in Eastern Europe in the late 20th century. This algorithm was also commonly used by early digital computers that did not implement integer multiplication directly in hardware.

```
PEASANTMULTIPLY(x, y):
  prod ← 0
  while x > 0
      if x is odd
          prod ← prod + y
      x ← ⌊x/2⌋
      y ← y + y
  return p
```

x	y		$prod$
			0
123	+456	=	456
61	+912	=	1368
30	~~1824~~		
15	+3648	=	5016
7	+7296	=	12312
3	+14592	=	26904
1	+29184	=	**56088**

The peasant multiplication algorithm breaks the difficult task of general multiplication into four simpler operations: (1) determining parity (even or odd), (2) addition, (3) duplation (doubling a number), and (4) mediation (halving a number, rounding down).[7] Of course a full specification of this algorithm requires describing how to perform those four 'primitive' operations. Peasant multiplication requires (a constant factor!) more paperwork to execute by hand, but the necessary operations are easier (for humans) to remember than the 10×10 multiplication table required by the American grade school algorithm.[8]

The correctness of peasant multiplication follows from the following recursive identity, which holds for any non-negative integers x and y:

$$x \cdot y = \begin{cases} 0 & \text{if } x = 0 \\ \lfloor x/2 \rfloor \cdot (y + y) & \text{if } x \text{ is even} \\ \lfloor x/2 \rfloor \cdot (y + y) + y & \text{if } x \text{ is odd} \end{cases}$$

[7] The version of this algorithm actually used in ancient Egypt does not use mediation or parity, but it does use comparisons. To avoid halving, the algorithm pre-computes two tables by repeated doubling: one containing all the powers of 2 not exceeding x, the other containing the same powers of 2 multiplied by y. The powers of 2 that sum to x are then found by greedy subtraction, and the corresponding entries in the other table are added together to form the product.

[8] American school kids learn a variant of the *lattice* multiplication algorithm developed by Indian mathematicians and described by Fibonacci in *Liber Abaci*. The two algorithms are equivalent if the input numbers are represented in binary.

Congressional Apportionment

Here is another good example of an algorithm that comes from outside the world of computing. Article I, Section 2 of the United States Constitution requires that

> Representatives and direct Taxes shall be apportioned among the several States which may be included within this Union, according to their respective Numbers. . . . The Number of Representatives shall not exceed one for every thirty Thousand, but each State shall have at Least one Representative. . . .

Since there are a limited number of seats available in the House of Representatives, exact proportional representation is impossible without either shared or fractional representatives, neither of which are legal. As a result, several different apportionment algorithms have been proposed and used to round the fractional solution fairly. The algorithm actually used today, called *the Huntington-Hill method* or *the method of equal proportions*, was first suggested by Census Bureau statistician Joseph Hill in 1911, refined by Harvard mathematician Edward Huntington in 1920, adopted into Federal law (2 U.S.C. §§2a and 2b) in 1941, and survived a Supreme Court challenge in 1992.[9] The input array $Pop[1..n]$ stores the populations of the n states, and R is the total number of representatives. Currently, $n = 50$ and $R = 435$. The output array $Rep[1..n]$ stores the number of representatives assigned to each state.

```
ApportionCongress(Pop[1..n], R):
  PQ ← NewPriorityQueue
  for i ← 1 to n
      Rep[i] ← 1
      Insert(PQ, i, Pop[i]/√2)
      R ← R − 1

  while R > 0
      s ← ExtractMax(PQ)
      Rep[s] ← Rep[s] + 1
      Insert(PQ, s, Pop[s] / √(Rep[s](Rep[s] + 1)))
      R ← R − 1

  return Rep[1..n]
```

This pseudocode description assumes that you know how to implement a priority queue that supports the operations NewPriorityQueue, Insert, and ExtractMax. (The actual law doesn't assume that, of course.) The output of the algorithm, and therefore its correctness, does not depend *at all* on how the priority queue is implemented. The Census Bureau uses an unsorted array, stored in a column of an Excel spreadsheet; you should have learned a more efficient solution in your undergraduate data structures class.

A bad example

Consider "Martin's algorithm":[10]

[9] Overruling an earlier ruling by a federal district court, the Supreme Court unanimously held that ***any*** apportionment method adopted in good faith by Congress is constitutional (*United States Department of Commerce v. Montana*). The current congressional apportionment algorithm is described in gruesome detail at the U.S. Census Department web site http://www.census.gov/population/www/censusdata/apportionment/computing.html. A good history of the apportionment problem can be found at http://www.thirty-thousand.org/pages/Apportionment.htm. A report by the Congressional Research Service describing various apportionment methods is available at http://www.rules.house.gov/archives/RL31074.pdf.

[10] Steve Martin, "You Can Be A Millionaire", Saturday Night Live, January 21, 1978. Also appears on *Comedy Is Not Pretty*, Warner Bros. Records, 1979.

```
BECOMEAMILLIONAIREANDNEVERPAYTAXES:
  Get a million dollars.
  Don't pay taxes.
  If you get caught,
      Say "I forgot."
```

Pretty simple, except for that first step; it's a doozy. A group of billionaire CEOs might consider this an algorithm, since for them the first step is both unambiguous and trivial, but for the rest of us poor slobs, Martin's procedure is too vague to be considered an actual algorithm. On the other hand, this is a perfect example of a ***reduction***—it *reduces* the problem of being a millionaire and never paying taxes to the 'easier' problem of acquiring a million dollars. We'll see reductions over and over again in this class. As hundreds of businessmen and politicians have demonstrated, if you know how to solve the easier problem, a reduction tells you how to solve the harder one.

Martin's algorithm, like many of our previous examples, is not the kind of algorithm that computer scientists are used to thinking about, because it is phrased in terms of operations that are difficult for computers to perform. In this class, we'll focus (almost!) exclusively on algorithms that can be reasonably implemented on a computer. In other words, each step in the algorithm must be something that either is directly supported by common programming languages (such as arithmetic, assignments, loops, or recursion) or is something that you've already learned how to do in an earlier class (like sorting, binary search, or depth first search).

0.3 Writing down algorithms

Computer programs are concrete representations of algorithms, but algorithms are *not* programs; they should not be described in a particular programming language. The whole *point* of this course is to develop computational techniques that can be used in *any* programming language. The idiosyncratic syntactic details of C, C++, C#, Java, Python, Ruby, Erlang, Haskell, OcaML, Scheme, Scala, Clojure, Visual Basic, Smalltalk, Javascript, Processing, Squeak, Forth, TEX, Fortran, COBOL, INTERCAL, MMIX, LOLCODE, Befunge, Parseltongue, Whitespace, or Brainfuck are of little or no importance in algorithm design, and focusing on them will only distract you from what's *really* going on.[11] What we really want is closer to what you'd write in the *comments* of a real program than the code itself.

On the other hand, a plain English prose description is usually not a good idea either. Algorithms have a lot of structure—especially conditionals, loops, and recursion—that are far too easily hidden by unstructured prose. Natural languages like English are full of ambiguities, subtleties, and shades of meaning, but algorithms must be described as precisely and unambigiously as possible. Finally and more seriously, there is natural tendency to describe repeated operations informally: "Do this first, then do this second, and so on." But as anyone who has taken one of those 'What comes next in this sequence?' tests already knows, specifying what happens in the first few iterations of a loop says very little, of anything, about what happens later iterations. To make the description unambiguous, we must explicitly specify the behavior of *every* iteration. The stupid joke about the programmer dying in the shower has a grain of truth—"Lather, rinse, repeat" *is* ambiguous; what exactly do we repeat, and until when?

[11] This is, of course, a matter of religious conviction. Linguists argue incessantly over the *Sapir-Whorf hypothesis*, which states (more or less) that people think only in the categories imposed by their languages. According to an extreme formulation of this principle, some concepts in one language simply cannot be understood by speakers of other languages, not just because of technological advancement—How would you translate 'jump the shark' or 'blog' into Aramaic?—but because of inherent structural differences between languages and cultures. For a more skeptical view, see Steven Pinker's *The Language Instinct*. There is admittedly some strength to this idea when applied to different programming paradigms. (What's the Y combinator, again? How do templates work? What's an Abstract Factory?) Fortunately, those differences are generally too subtle to have much impact in *this* class. For a compelling counterexample, see Chris Okasaki's thesis/monograph *Functional Data Structures* and its more recent descendants.

In my opinion, the clearest way to present an algorithm is using pseudocode. Pseudocode uses the *structure* of formal programming languages and mathematics to break algorithms into primitive steps; but the primitive steps themselves may be written using mathematics, pure English, or an appropriate mixture of the two. Well-written pseudocode reveals the internal structure of the algorithm but hides irrelevant implementation details, making the algorithm much easier to understand, analyze, debug, and implement.

The precise syntax of pseudocode is a personal choice, but the overriding goal should be clarity and precision. Ideally, pseudocode should allow any competent programmer to implement the underlying algorithm, quickly and correctly, in *their* favorite programming language, *without understanding why the algorithm works*. Here are the guidelines I follow and strongly recommend:

- Be consistent!
- Use standard imperative programming keywords (if/then/else, while, for, repeat/until, case, return) and notation ($variable \leftarrow value$, $Array[index]$, $function(argument)$, $bigger > smaller$, etc.). Keywords should be standard English words: write 'else if' instead of 'elif'.
- Indent everything carefully and consistently; the block structure should be visible from across the room. This rule is especially important for nested loops and conditionals. *Don't* add unnecessary syntactic sugar like braces or begin/end tags; careful indentation is almost always enough.
- Use mnemonic algorithm and variable names. Short variable names are good, but readability is more important than concision; except for idioms like loop indices, short but complete words are better than single letters. Absolutely *never* use pronouns!
- Use standard mathematical notation for standard mathematical things. For example, write $x \cdot y$ instead of $x * y$ for multiplication; write $x \bmod y$ instead of $x \,\%\, y$ for remainder; write $\sqrt{x}$ instead of $sqrt(x)$ for square roots; write a^b instead of $power(a, b)$ for exponentiation; and write ϕ instead of *phi* for the golden ratio.
- Avoid mathematical notation if English is clearer. For example, 'Insert a into X' may be preferable to $\textsc{Insert}(X, a)$ or $X \leftarrow X \cup \{a\}$.
- Each statement should fit on one line, and each line should contain either exactly one statement or exactly one structuring element (for, while, if). (I sometimes make an exception for short and similar statements like $i \leftarrow i + 1;\ j \leftarrow j - 1;\ k \leftarrow 0$.)
- *Don't* use a fixed-width typeface to typeset pseudocode; it's much harder to read than normal typeset text. Similarly, *don't* typeset keywords like 'for' or 'while' in a different **style**; the syntactic sugar is not what you want the reader to look at. On the other hand, I do use *italics* for variables (following the standard mathematical typesetting convention), SMALL CAPS for algorithms and constants, and *a different typeface* for literal strings.

0.4 Analyzing algorithms

It's not enough just to write down an algorithm and say 'Behold!' We must also convince our audience (and ourselves!) that the algorithm actually does what it's supposed to do, and that it does so efficiently.

Correctness

In some application settings, it is acceptable for programs to behave correctly most of the time, on all 'reasonable' inputs. Not in this class; we require algorithms that are correct for *all possible* inputs. Moreover, we must *prove* that our algorithms are correct; trusting our instincts, or trying a few test cases, isn't good enough. Sometimes correctness is fairly obvious, especially for algorithms you've seen in earlier courses. On the other hand, 'obvious' is all too often a synonym for 'wrong'. Many of the algorithms we will discuss in this course will require extra work to prove correct. Correctness proofs almost always involve induction. We *like* induction. Induction is our *friend*.[12]

But before we can formally prove that our algorithm does what it's supposed to do, we have to formally *state* what it's supposed to do! Algorithmic problems are usually presented using standard English, in terms of real-world objects, not in terms of formal mathematical objects. It's up to us, the algorithm designers, to restate these problems in terms of mathematical objects that we can prove things about—numbers, arrays, lists, graphs, trees, and so on. We must also determine if the problem statement carries any hidden assumptions, and state those assumptions explicitly. (For example, in the song "n Bottles of Beer on the Wall", n is always a positive integer.) Restating the problem formally is not only required for proofs; it is also one of the best ways to really understand what a problem is asking for. The hardest part of answering any question is figuring out the right way to ask it!

It is important to remember the distinction between a problem and an algorithm. A problem is a task to perform, like "Compute the square root of x" or "Sort these n numbers" or "Keep n algorithms students awake for t minutes". An algorithm is a set of instructions for accomplishing such a task. The same problem may have hundreds of different algorithms; the same algorithm may solve hundreds of different problems.

Running time

The most common way of ranking different algorithms for the same problem is by how quickly they run. Ideally, we want the fastest possible algorithm for any particular problem. In many application settings, it is acceptable for programs to run efficiently most of the time, on all 'reasonable' inputs. Not in this class; we require algorithms that *always* run efficiently, even in the worst case.

But how do we measure running time? As a specific example, how long does it take to sing the song BOTTLESOFBEER(n)? This is obviously a function of the input value n, but it also depends on how quickly you can sing. Some singers might take ten seconds to sing a verse; others might take twenty. Technology widens the possibilities even further. Dictating the song over a telegraph using Morse code might take a full minute per verse. Downloading an mp3 over the Web might take a tenth of a second per verse. Duplicating the mp3 in a computer's main memory might take only a few microseconds per verse.

What's important here is how the singing time changes as n grows. Singing BOTTLESOFBEER($2n$) takes about twice as long as singing BOTTLESOFBEER(n), no matter what technology is being used. This is reflected in the asymptotic singing time $\Theta(n)$. We can measure time by counting how many times the algorithm executes a certain instruction or reaches a certain milestone in the 'code'. For example, we might notice that the word 'beer' is sung three times in every verse of BOTTLESOFBEER, so the number of times you sing 'beer' is a good indication of the total singing time. For this question, we can give an exact answer: BOTTLESOFBEER(n) uses exactly $3n + 3$ beers.

There are plenty of other songs that have non-trivial singing time. This one is probably familiar to most English-speakers:

[12] If induction is *not* your friend, you will have a hard time in this course.

```
NDaysOfChristmas(gifts[2..n]):
  for i ← 1 to n
      Sing "On the ith day of Christmas, my true love gave to me"
      for j ← i down to 2
          Sing "j gifts[j]"
      if i > 1
          Sing "and"
      Sing "a partridge in a pear tree."
```

The input to NDaysOfChristmas is a list of $n-1$ gifts. It's quite easy to show that the singing time is $\Theta(n^2)$; in particular, the singer mentions the name of a gift $\sum_{i=1}^{n} i = n(n+1)/2$ times (counting the partridge in the pear tree). It's also easy to see that during the first n days of Christmas, my true love gave to me exactly $\sum_{i=1}^{n}\sum_{j=1}^{i} j = n(n+1)(n+2)/6 = \Theta(n^3)$ gifts.

There are many other traditional songs that take quadratic time to sing; examples include "Old MacDonald Had a Farm", "There Was an Old Lady Who Swallowed a Fly", "The House that Jack Built", "Hole in the Bottom of the Sea", "Green Grow the Rushes O", "The Rattlin' Bog", "The Barley-Mow", "Eh, Cumpari!", "Alouette", "Echad Mi Yode'a", "Ist das nicht ein Schnitzelbank?", and "Minkurinn í hænsnakofanum". For further details, consult your nearest preschooler.

```
OldMacDonald(animals[1..n], noise[1..n]):
  for i ← 1 to n
      Sing "Old MacDonald had a farm, E I E I O"
      Sing "And on this farm he had some animals[i], E I E I O"
      Sing "With a noise[i] noise[i] here, and a noise[i] noise[i] there"
      Sing "Here a noise[i], there a noise[i], everywhere a noise[i] noise[i]"
      for j ← i − 1 down to 1
          Sing "noise[j] noise[j] here, noise[j] noise[j] there"
          Sing "Here a noise[j], there a noise[j], everywhere a noise[j] noise[j]"
      Sing "Old MacDonald had a farm, E I E I O."
```

```
Alouette(lapart[1..n]):
  Chantez « Alouette, gentille alouette, alouette, je te plumerais. »
  pour tout i de 1 á n
      Chantez « Je te plumerais lapart[i]. Je te plumerais lapart[i]. »
      pour tout j de i − 1 á bas á 1
          Chantez « Et lapart[j] ! Et lapart[j] ! »
      Chantez « Oooooooo! »
      Chantez « Alouette, gentille alluette, alouette, je te plumerais. »
```

A more modern example of the parametrized cumulative song is "The TELNET Song" by Guy Steele, which takes $O(2^n)$ time to sing; Steele recommended $n = 4$.

For a slightly less facetious example, consider the algorithm ApportionCongress. Here the running time obviously depends on the implementation of the priority queue operations, but we can certainly bound the running time as $O(N+RI+(R-n)E)$, where N denotes the running time of NewPriorityQueue, I denotes the running time of Insert, and E denotes the running time of ExtractMax. Under the reasonable assumption that $R > 2n$ (on average, each state gets at least two representatives), we can simplify the bound to $O(N + R(I+E))$. The Census Bureau implements the priority queue using an unsorted array of size n; this implementation gives us $N = I = \Theta(1)$ and $E = \Theta(n)$, so the overall running time is $O(Rn)$. This is good enough for government work, but we can do better. Implementing the priority queue using a binary heap (or a heap-ordered array) gives us $N = \Theta(1)$ and $I = R = O(\log n)$, which implies an overall running time of $O(R \log n)$.

Sometimes we are also interested in other computational resources: space, randomness, page faults, inter-process messages, and so forth. We can use the same techniques to analyze those resources as we use to analyze running time.

0.5 A Longer Example: Stable Matching

Every year, thousands of new doctors must obtain internships at hospitals around the United States. During the first half of the 20th century, competition among hospitals for the best doctors led to earlier and earlier offers of internships, sometimes as early as the second year of medical school, along with tighter deadlines for acceptance. In the 1940s, medical schools agreed not to release information until a common date during their students' fourth year. In response, hospitals began demanding faster decisions. By 1950, hospitals would regularly call doctors, offer them internships, and demand *immediate* responses. Interns were forced to gamble if their third-choice hospital called first—accept and risk losing a better opportunity later, or reject and risk having no position at all.[13]

Finally, a central clearinghouse for internship assignments, now called the National Resident Matching Program, was established in the early 1950s. Each year, doctors submit a ranked list of all hospitals where they would accept an internship, and each hospital submits a ranked list of doctors they would accept as interns. The NRMP then computes an assignment of interns to hospitals that satisfies the following *stability* requirement. For simplicity, let's assume that there are n doctors and n hospitals; each hospital offers exactly one internship; each doctor ranks all hospitals and vice versa; and finally, there are no ties in the doctors' and hospitals' rankings.[14] We say that a matching of doctors to hospitals is ***unstable*** if there are two doctors α and β and two hospitals A and B, such that

- α is assigned to A, and β is assigned to B;
- α prefers B to A, and B prefers α to β.

In other words, α and B would both be happier with each other than with their current assignment. The goal of the Resident Match is a ***stable matching***, in which no doctor or hospital has an incentive to cheat the system. At first glance, it is not clear that a stable matching exists!

In 1952, the NRMP adopted the "Boston Pool" algorithm to assign interns, so named because it had been previously used by a regional clearinghouse in the Boston area. The algorithm is often misattributed to David Gale and Lloyd Shapley, who formally analyzed the algorithm and first proved that it computes a stable matching in 1962; Gale and Shapley used the metaphor of college admissions.[15] Similar algorithms have since been adopted for other matching markets, including faculty recruiting in France, university admission in Germany, public school admission in New York and Boston, billet assignments for US Navy sailors, and kidney-matching programs. Shapley was awarded the 2012 Nobel Prize in Economics for his research on stable matching, together with Alvin Roth, who significantly extended Shapley's work and used it to develop several real-world exchanges.

[13] The academic job market involves similar gambles, at least in computer science. Some departments start making offers in February with two-week decision deadlines; other departments don't even start interviewing until late March; MIT notoriously waits until May, when all its interviews are over, before making *any* faculty offers.

[14] In reality, most hospitals offer multiple internships, each doctor ranks only a subset of the hospitals and vice versa, and there are typically more internships than interested doctors. And then it starts getting complicated.

[15] The "Gale-Shapley algorithm" is a prime instance of ***Stigler's Law of Eponymy:*** *No scientific discovery is named after its original discoverer.* In his 1980 paper that gives the law its name, the statistician Stephen Stigler claimed that this law was first proposed by sociologist Robert K. Merton. However, similar statements were previously made by Vladimir Arnol'd in the 1970's ("Discoveries are rarely attributed to the correct person."), Carl Boyer in 1968 ("Clio, the muse of history, often is fickle in attaching names to theorems!"), Alfred North Whitehead in 1917 ("Everything of importance has been said before by someone who did not discover it."), and even Stephen's father George Stigler in 1966 ("If we should ever encounter a case where a theory is named for the correct man, it will be noted."). We will see *many* other examples of Stigler's law in this class.

The Boston Pool algorithm proceeds in rounds until every position has been filled. Each round has two stages:

1. An arbitrary unassigned hospital A offers its position to the best doctor α (according to the hospital's preference list) who has not already rejected it.

2. Each doctor ultimately accepts the best offer that she receives, according to her preference list. Thus, if α is currently unassigned, she (tentatively) accepts the offer from A. If α already has an assignment but prefers A, she rejects her existing assignment and (tentatively) accepts the new offer from A. Otherwise, α rejects the new offer.

For example, suppose four doctors (Dr. Quincy, Dr. Rotwang, Dr. Shephard, and Dr. Tam, represented by lower-case letters) and four hospitals (Arkham Asylum, Bethlem Royal Hospital, County General Hospital, and The Dharma Initiative, represented by upper-case letters) rank each other as follows:

q	r	s	t
A	A	B	D
B	D	A	B
C	C	C	C
D	B	D	A

A	B	C	D
t	r	t	s
s	t	r	r
r	q	s	q
q	s	q	t

Given these preferences as input, the Boston Pool algorithm might proceed as follows:

1. Arkham makes an offer to Dr. Tam.
2. Bedlam makes an offer to Dr. Rotwang.
3. County makes an offer to Dr. Tam, who rejects her earlier offer from Arkham.
4. Dharma makes an offer to Dr. Shephard. (From this point on, because there is only one unmatched hospital, the algorithm has no more choices.)
5. Arkham makes an offer to Dr. Shephard, who rejects her earlier offer from Dharma.
6. Dharma makes an offer to Dr. Rotwang, who rejects her earlier offer from Bedlam.
7. Bedlam makes an offer to Dr. Tam, who rejects her earlier offer from County.
8. County makes an offer to Dr. Rotwang, who rejects it.
9. County makes an offer to Dr. Shephard, who rejects it.
10. County makes an offer to Dr. Quincy.

At this point, all pending offers are accepted, and the algorithm terminates with a matching: (A,s), (B,t), (C,q), (D,r). You can (and should) verify by brute force that this matching is stable, even though no doctor was hired by her favorite hospital, and no hospital hired its favorite doctor; in fact, County was forced to hire their *least* favorite doctor. This is not the only stable matching for this list of preferences; the matching $(A,r),(B,s),(C,q),(D,t)$ is also stable.

Running Time

Analyzing the algorithm's running time is relatively straightforward. Each hospital makes an offer to each doctor at most once, so the algorithm requires at most n^2 rounds. In an actual implementation, each doctor and hospital can be identified by a unique integer between 1 and n, and the preference lists can be represented as two arrays $DocPref[1..n][1..n]$ and $HosPref[1..n][1..n]$, where $DocPref[\alpha][r]$

represents the rth hospital in doctor α's preference list, and $HosPref[A][r]$ represents the rth doctor in hospital A's preference list. With the input in this form, the Boston Pool algorithm can be implemented to run in $\boldsymbol{O(n^2)}$ ***time***; we leave the details as an easy exercise.

A somewhat harder exercise is to prove that there are inputs (and choices of who makes offers when) that force $\Omega(n^2)$ rounds before the algorithm terminates. Thus, the $O(n^2)$ upper bound on the worst-case running time cannot be improved; in this case, we say our analysis is ***tight***.

Correctness

But why is the algorithm *correct*? How do we know that the Boston Pool algorithm always computes a *stable matching*? Gale and Shapley proved correctness as follows. The algorithm continues as long as there is at least one unfilled position; conversely, when the algorithm terminates (after at most n^2 rounds), every position is filled. No doctor can accept more than one position, and no hospital can hire more than one doctor. Thus, the algorithm always computes a matching; it remains only to prove that the matching is stable.

Suppose doctor α is assigned to hospital A in the final matching, but prefers B. Because every doctor accepts the best offer she receives, α received no offer she liked more than A. In particular, B never made an offer to α. On the other hand, B made offers to every doctor they like more than β. Thus, B prefers β to α, and so there is no instability.

Surprisingly, the correctness of the algorithm does not depend on which hospital makes its offer in which round. In fact, there is a stronger sense in which the order of offers doesn't matter—no matter which unassigned hospital makes an offer in each round, *the algorithm always computes the same matching*! Let's say that α is a ***feasible*** doctor for A if there is a stable matching that assigns doctor α to hospital A.

Lemma 0.1. *During the Boston Pool algorithm, each hospital A is rejected only by doctors that are infeasible for A.*

Proof: We prove the lemma by induction. Consider an arbitrary round of the Boston Pool algorithm, in which doctor α rejects one hospital A for another hospital B. The rejection implies that α prefers B to A. Every doctor that appears higher than α in B's preference list has already rejected B and therefore, by the inductive hypothesis, is infeasible for B.

Now consider an arbitrary matching that assigns α to A. We already established that α prefers B to A. If B prefers α to its partner, the matching is unstable. On the other hand, if B prefers its partner to α, then (by our earlier argument) its partner is infeasible, and again the matching is unstable. We conclude that there is no stable matching that assigns α to A. □

Now let $\boldsymbol{best(A)}$ denote the highest-ranked *feasible* doctor on A's preference list. Lemma 0.1 implies that every doctor that A prefers to its final assignment is infeasible for A. On the other hand, the final matching is stable, so the doctor assigned to A is feasible for A. The following result is now immediate:

Corollary 0.2. *The Boston Pool algorithm assigns $best(A)$ to A, for every hospital A.*

Thus, from the hospitals' point of view, the Boston Pool algorithm computes the best possible stable matching. It turns out that this matching is also the *worst* possible from the doctors' viewpoint! Let $\boldsymbol{worst(\alpha)}$ denote the lowest-ranked feasible hospital on doctor α's preference list.

Corollary 0.3. *The Boston Pool algorithm assigns α to $worst(\alpha)$, for every doctor α.*

Proof: Suppose the Boston Pool algorithm assigns doctor α to hospital A; we need to show that $A = \mathit{worst}(\alpha)$. Consider an arbitrary stable matching where A is *not* matched with α but with another doctor β. The previous corollary implies that A prefers $\alpha = \mathit{best}(A)$ to β. Because the matching is stable, α must therefore prefer her assigned hopital to A. This argument works for *any* stable assignment, so α prefers *every* other feasible match to A; in other words, $A = \mathit{worst}(\alpha)$. □

A subtle consequence of these two corollaries, discovered by Dubins and Freeman in 1981, is that a doctor can potentially improve her assignment by lying about her preferences, but a hospital cannot. (However, a set of hospitals can collude so that *some* of their assignments improve.) Partly for this reason, the National Residency Matching Program reversed its matching algorithm in 1998, so that potential residents offer to work for hospitals in preference order, and each hospital accepts its best offer. Thus, the new algorithm computes the best possible stable matching for the doctors, and the worst possible stable matching for the hospitals. In practice, however, this modification affected less than 1% of the resident's assignments. As far as I know, the precise effect of this change on the *patients* is an open problem.

0.6 Why are we here, anyway?

This class is ultimately about learning two skills that are crucial for all computer scientists.

1. ***Intuition:*** How to *think* about abstract computation.

2. ***Language:*** How to *talk* about abstract computation.

The first goal of this course is to help you develop algorithmic *intuition*. How do various algorithms really work? When you see a problem for the first time, how should you attack it? How do you tell which techniques will work at all, and which ones will work best? How do you judge whether one algorithm is better than another? How do you tell whether you have the best possible solution? These are *not* easy questions; anyone who says differently is selling something.

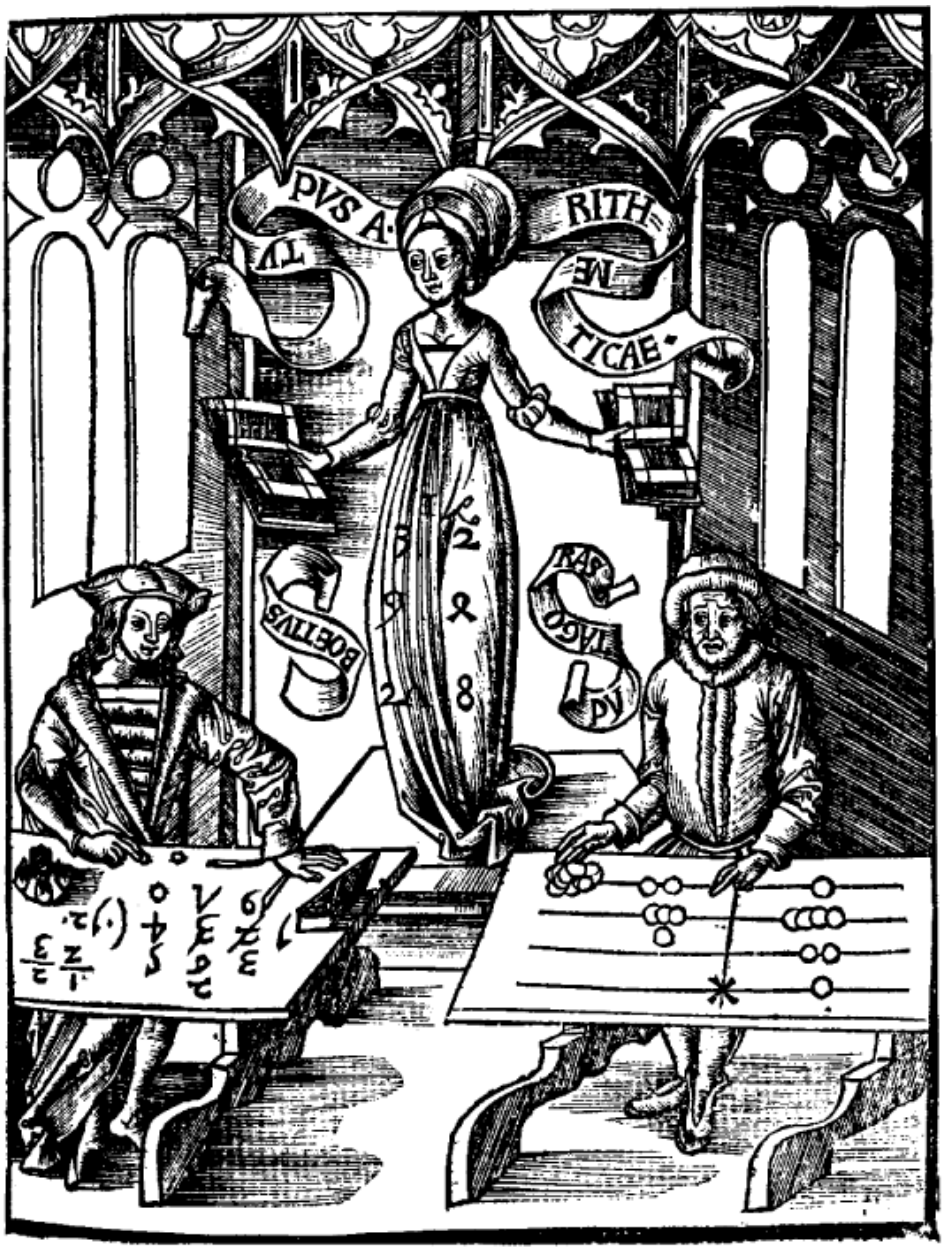
Our second main goal is to help you develop algorithmic *language*. It's not enough just to understand how to solve a problem; you also have to be able to explain your solution to somebody else. I don't mean just how to turn your algorithms into working code—despite what many students (and inexperienced programmers) think, 'somebody else' is *not* just a computer. Nobody programs alone. Code is read far more often than it is written, or even compiled. Perhaps more importantly in the short term, explaining something to somebody else is one of the best ways to clarify your own understanding. As Albert Einstein (or was it Richard Feynman?) apocryphally put it, "You do not really understand something unless you can explain it to your grandmother."

Along the way, you'll pick up a bunch of algorithmic facts—mergesort runs in $\Theta(n \log n)$ time; the amortized time to search in a splay tree is $O(\log n)$; greedy algorithms usually don't produce optimal solutions; the traveling salesman problem is NP-hard—but these aren't the point of the course. You can always look up mere facts in a textbook or on the web, provided you have enough intuition and experience to know what to look for. That's why we let you bring cheat sheets to the exams; we don't want you wasting your study time trying to memorize all the facts you've seen.

You'll also practice a lot of algorithm design and analysis skills—finding useful (counter)examples, developing induction proofs, solving recurrences, using big-Oh notation, using probability, giving problems crisp mathematical descriptions, and so on. These skills are *incredibly* useful, and it's impossible to develop good intuition and good communication skills without them, but they aren't the main point of the course either. At this point in your educational career, you should be able to pick up most of those skills on your own, once you know what you're trying to do.

Unfortunately, there is no systematic procedure—no algorithm—to determine which algorithmic techniques are most effective at solving a given problem, or finding good ways to explain, analyze, optimize, or implement a given algorithm. Like many other activities (music, writing, juggling, acting, martial arts, sports, cooking, programming, teaching, etc.), the *only* way to master these skills is to make them your own, through practice, practice, and more practice. You can only develop good problem-solving skills by solving problems. You can only develop good communication skills by communicating. Good intuition is the product of experience, not its replacement. We *can't* teach you how to do well in this class. All we can do (and what we will do) is lay out some fundamental tools, show you how to use them, create opportunities for you to practice with them, and give you honest feedback, based on our own hard-won experience and intuition. The rest is up to you.

Good algorithms are extremely useful, elegant, surprising, deep, even beautiful, but most importantly, algorithms are *fun*! I hope you will enjoy playing with them as much as I do.



Boethius the algorist versus Pythagoras the abacist.
from *Margarita Philosophica* by Gregor Reisch (1503)

Exercises

0. Describe and analyze an efficient algorithm that determines, given a legal arrangement of standard pieces on a standard chess board, which player will win at chess from the given starting position if both players play perfectly. *[Hint: There is a trivial one-line solution!]*

1. "The Barley Mow" is a cumulative drinking song which has been sung throughout the British Isles for centuries. (An early version entitled "Giue vs once a drinke" appears in Thomas Ravenscroft's song collection *Deuteromelia*, which was published in 1609, but the song is almost certainly much older.) The song has many variants, but one version traditionally sung in Devon and Cornwall has the following pseudolyrics, where *vessel*$[i]$ is the name of a vessel that holds 2^i ounces of beer. The traditional song uses the following vessels: nipperkin, gill pot, half-pint, pint, quart, pottle, gallon, half-anker, anker, firkin, half-barrel, barrel, hogshead, pipe, well, river, and ocean. (Every vessel in this list is twice as big as its predecessor, except that a firkin is actually 2.25 ankers, and the last three units are just silly.)

```
BARLEYMOW(n):
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  "We'll drink it out of the jolly brown bowl,"
  "Here's a health to the barley-mow!"
  "Here's a health to the barley-mow, my brave boys,"
  "Here's a health to the barley-mow!"

  for i ← 1 to n
      "We'll drink it out of the vessel[i], boys,"
      "Here's a health to the barley-mow!"
      for j ← i downto 1
          "The vessel[j],"
      "And the jolly brown bowl!"
      "Here's a health to the barley-mow!"
      "Here's a health to the barley-mow, my brave boys,"
      "Here's a health to the barley-mow!"
```

   (a) Suppose each name *vessel*$[i]$ is a single word, and you can sing four words a second. How long would it take you to sing BARLEYMOW(n)? (Give a tight asymptotic bound.)

   (b) If you want to sing this song for arbitrarily large values of n, you'll have to make up your own vessel names. To avoid repetition, these names must become progressively longer as n increases. ("We'll drink it out of the hemisemidemiyottapint, boys!") Suppose *vessel*$[n]$ has $\Theta(\log n)$ syllables, and you can sing six syllables per second. Now how long would it take you to sing BARLEYMOW(n)? (Give a tight asymptotic bound.)

   (c) Suppose each time you mention the name of a vessel, you actually drink the corresponding amount of beer: one ounce for the jolly brown bowl, and 2^i ounces for each *vessel*$[i]$. Assuming for purposes of this problem that you are at least 21 years old, *exactly* how many ounces of beer would you drink if you sang BARLEYMOW(n)? (Give an *exact* answer, not just an asymptotic bound.)

2. Describe and analyze the Boston Pool stable matching algorithm in more detail, so that the worst-case running time is $O(n^2)$, as claimed earlier in the notes.

3. Prove that it is possible for the Boston Pool algorithm to execute $\Omega(n^2)$ rounds. (You need to describe both a suitable input and a sequence of $\Omega(n^2)$ valid proposals.)

4. Describe and analyze an efficient algorithm to determine whether a given set of hospital and doctor preferences has to a *unique* stable matching.

5. Consider a generalization of the stable matching problem, where some doctors do not rank all hospitals and some hospitals do not rank all doctors, and a doctor can be assigned to a hospital only if each appears in the other's preference list. In this case, there are three additional unstable situations:

   - A hospital prefers an unmatched doctor to its assigned match.
   - A doctor prefers an unmatched hospital to her assigned match.
   - An unmatched doctor and an unmatched hospital appear in each other's preference lists.

   Describe and analyze an efficient algorithm that computes a stable matching in this setting.

   Note that a stable matching may leave some doctors and hospitals unmatched, even though their preference lists are non-empty. For example, if every doctor lists Harvard as their only acceptable hospital, and every hospital lists Dr. House as their only acceptable intern, then only House and Harvard will be matched.

6. Recall that the input to the Huntington-Hill apportionment algorithm ApportionCongress is an array $P[1..n]$, where $P[i]$ is the population of the ith state, and an integer R, the total number of representatives to be allotted. The output is an array $r[1..n]$, where $r[i]$ is the number of representatives allotted to the ith state by the algorithm.

   Let $P = \sum_{i=1}^{n} P[i]$ denote the total population of the country, and let $r_i^* = R \cdot P[i]/P$ denote the ideal number of representatives for the ith state.

   (a) Prove that $r[i] \geq \lfloor r_i^* \rfloor$ for all i.

   (b) Describe and analyze an algorithm that computes exactly the same congressional apportionment as ApportionCongress in $O(n \log n)$ time. (Recall that the running time of ApportionCongress depends on R, which could be arbitrarily larger than n.)

   ⋆(c) If a state's population is small relative to the other states, its ideal number r_i^* of representatives could be close to zero; thus, tiny states are over-represented by the Huntington-Hill apportionment process. Surprisingly, this can also be true of very large states. Let $\alpha = (1+\sqrt{2})/2 \approx 1.20710678119$. Prove that for any $\varepsilon > 0$, there is an input to ApportionCongress with $\max_i P[i] = P[1]$, such that $r[1] > (\alpha - \varepsilon) r_1^*$.

   ★(d) Can you improve the constant α in the previous question?

> *Whence it is manifest that if we could find characters or signs appropriate for expressing all our thoughts. . . , we could in all subjects*—in so far as they are amenable to reasoning—*accomplish what is done in Arithmetic and Geometry. For all inquiries which depend on reasoning would be performed by the transposition of characters and by a kind of calculus, which would immediately facilitate the discovery of beautiful results. . . . And if someone would doubt my results, I should say to him: "Let us calculate, Sir," and thus by taking to pen and ink, we should soon settle the question.*
>
> — Gottfried Wilhelm Leibniz, *Preface to the General Science* (1677), translated by Philip Wiener (1951)

> *I hope the reader sees that the alphabet can be understood by any intelligent being who has any one of the five senses left him,—by all rational men, that is, excepting the few eyeless deaf persons who have lost both taste and smell in some complete paralysis. . . . Whales in the sea can telegraph as well as senators on land, if they will only note the difference between long spoutings and short ones. . . . A tired listener at church, by properly varying his long yawns and his short ones, may express his opinion of the sermon to the opposite gallery before the sermon is done.*
>
> — Edward Everett Hale, "The Dot and Line Alphabet", Altlantic Monthy (October 1858)

> *If indeed, as Hilbert asserted, mathematics is a meaningless game played with meaningless marks on paper, the only mathematical experience to which we can refer is the making of marks on paper.*
>
> — Eric Temple Bell, *The Queen of the Sciences* (1931)

1 Strings

Throughout this course, we will discuss dozens of algorithms and computational models that manipulate sequences: one-dimensional arrays, linked lists, blocks of text, walks in graphs, sequences of executed instructions, and so on. Ultimately the input and output of any algorithm must be representable as a finite string of symbols—the raw contents of some contiguous portion of the computer's memory. Reasoning about computation requires reasoning about strings.

This note lists several formal definitions and formal induction proofs related to strings. These definitions and proofs are *intentionally* much more detailed than normally used in practice—most people's intuition about strings is fairly accurate—but the extra precision is necessary for any sort of formal proof. It may be helpful to think of this material as part of the "assembly language" of theoretical computer science. We normally think about computation at a *much* higher level of abstraction, but ultimately every argument must "compile" down to these (and similar) definitions.

1.1 Definitions

Fix an arbitrary finite set Σ called the ***alphabet***; the elements of Σ are called ***symbols*** or ***characters***. As a notational convention, I will always use lower-case letters near the start of the English alphabet ($a, b, c, \ldots$) as symbol variables, and *never* as explicit symbols. For explicit symbols, I will always use fixed-width upper-case letters (`A`, `B`, `C`, ...), digits (`0`, `1`, `2`, ...), or other symbols (`◇`, `$`, `#`, `•`, ...) that are clearly distinguishable from variables.

A ***string*** (or ***word***) over Σ is a finite sequence of zero or more symbols from Σ. Formally, a string w over Σ is defined recursively as either

- the empty string, denoted by the Greek letter ε (epsilon),or
- an ordered pair (a, x), where a is a symbol in Σ and x is a string over Σ.

We normally write either $\boldsymbol{a \cdot x}$ or simply $\boldsymbol{ax}$ to denote the ordered pair (a, x). Similarly, we normally write explicit strings as sequences of symbols instead of nested ordered pairs; for example, `STRING` is convenient shorthand for the formal expression $(\texttt{S}, (\texttt{T}, (\texttt{R}, (\texttt{I}, (\texttt{N}, (\texttt{G}, \varepsilon))))))$. As a notational convention, I will always use lower-case letters near the end of the alphabet $(\ldots, w, x, y, z)$ to represent unknown strings, and `SHOUTY⋄MONOSPACED⋄TEXT` to represent explicit symbols and (non-empty) strings.

The set of all strings over Σ is denoted $\boldsymbol{\Sigma^*}$ (pronounced "sigma star"). It is very important to remember that every element of Σ^* is a *finite* string, although Σ^* itself is an infinite set containing strings of every possible *finite* length.

The ***length*** $\boldsymbol{|w|}$ of a string w is the number of symbols in w, defined formally as follows:

$$|w| := \begin{cases} 0 & \text{if } w = \varepsilon, \\ 1 + |x| & \text{if } w = ax. \end{cases}$$

For example, the string `SEVEN` has length 5. Although they are formally different objects, we do not normally distinguish between symbols and strings of length 1.

The ***concatenation*** of two strings x and y, denoted either $\boldsymbol{x \bullet y}$ or simply $\boldsymbol{xy}$, is the unique string containing the characters of x in order, followed by the characters in y in order. For example, the string `NOWHERE` is the concatenation of the strings `NOW` and `HERE`; that is, $\texttt{NOW} \bullet \texttt{HERE} = \texttt{NOWHERE}$. (On the other hand, $\texttt{HERE} \bullet \texttt{NOW} = \texttt{HERENOW}$.) Formally, concatenation is defined recusively as follows:

$$w \bullet z := \begin{cases} z & \text{if } w = \varepsilon \\ a \cdot (x \bullet z) & \text{if } w = ax \end{cases}$$

(Here I'm using a larger dot $\bullet$ to formally distinguish the operator that concatenates two arbitrary strings from from the operator $\cdot$ that builds a string from a single character and a string.)

When we describe the concatenation of more than two strings, we normally omit all dots and parentheses, writing $wxyz$ instead of $(w \bullet (x \bullet y)) \bullet z$, for example. This simplification is justified by the fact (which we will prove shortly) that $\bullet$ is associative.

1.2 Induction on Strings

Induction is *the* standard technique for proving statements about recursively defined objects. Hopefully you are already comfortable proving statements about *natural numbers* via induction, but induction actually a far more general technique. Several different variants of induction can be used to prove statements about more general structures; here I describe the variant that I recommend (and actually use in practice). This variant follows two primary design considerations:

- ***The case structure of the proof should mirror the case structure of the recursive definition.*** For example, if you are proving something about all *strings*, your proof should have two cases: Either $w = \varepsilon$, or $w = ax$ for some symbol a and string x.
- ***The inductive hypothesis should be as strong as possible.*** The (strong) inductive hypothesis for statements about natural numbers is *always* "Assume there is no counterexample k such that $k < n$." I recommend adopting a similar inductive hypothesis for strings: "Assume there is no counterexample x such that $|x| < |w|$." Then for the case $w = ax$, we have $|x| = |w| - 1 < |w|$ by definition of $|w|$, so the inductive hypothesis applies to x.

Thus, string-induction proofs have the following boilerplate structure. Suppose we want to prove that every string is perfectly cromulent, whatever that means. The white boxes hide additional proof details that, among other things, depend on the precise definition of "perfectly cromulent".

Proof: Let w be an arbitrary string.
Assume, for every string x such that $|x| < |w|$, that x is perfectly cromulent.
There are two cases to consider.

- Suppose $w = \varepsilon$.

  Therefore, w is perfectly cromulent.

- Suppose $w = ax$ for some symbol a and string x.
  The induction hypothesis implies that x is perfectly cromulent.

  Therefore, w is perfectly cromulent.

In both cases, we conclude that w is perfectly cromulent. □

Here are three canonical examples of this proof structure. When developing proofs in this style, I strongly recommend first ***mindlessly*** writing the green text (the boilerplate) with lots of space for each case, then filling in the red text (the actual theorem and the induction hypothesis), and only then starting to actually think.

Lemma 1.1. *For every string w, we have $w \bullet \varepsilon = w$.*

Proof: Let w be an arbitrary string. Assume that $x \bullet \varepsilon = x$ for every string x such that $|x| < |w|$.
There are two cases to consider:

- Suppose $w = \varepsilon$.

$$\begin{aligned} w \bullet \varepsilon &= \varepsilon \bullet \varepsilon && \text{because } w = \varepsilon, \\ &= \varepsilon && \text{by definition of concatenation,} \\ &= w && \text{because } w = \varepsilon. \end{aligned}$$

- Suppose $w = ax$ for some symbol a and string x.

$$\begin{aligned} w \bullet \varepsilon &= (a \cdot x) \bullet \varepsilon && \text{because } w = ax, \\ &= a \cdot (x \bullet \varepsilon) && \text{by definition of concatenation,} \\ &= a \cdot x && \text{by the inductive hypothesis,} \\ &= w && \text{because } w = ax. \end{aligned}$$

In both cases, we conclude that $w \bullet \varepsilon = w$. □

Lemma 1.2. *Concatenation adds length: $|w \bullet x| = |w| + |x|$ for all strings w and x.*

Proof: Let w and x be arbitrary strings. Assume that $|y \bullet x| = |y| + |x|$ for every string y such that $|y| < |w|$. (Notice that we are using induction only on w, not on x.) There are two cases to consider:

- Suppose $w = \varepsilon$.

$$\begin{aligned} |w \bullet x| &= |\varepsilon \bullet x| && \text{because } w = \varepsilon \\ &= |x| && \text{by definition of } |\,| \\ &= |\varepsilon| + |x| && |e| = 0 \text{ by definition of } |\,| \\ &= |w| + |x| && \text{because } w = \varepsilon \end{aligned}$$

- Suppose $w = ay$ for some symbol a and string y.

$$\begin{aligned}
|w \bullet x| &= |ay \bullet x| && \text{because } w = ay \\
&= |a \cdot (y \bullet x)| && \text{by definition of } \bullet \\
&= 1 + |y \bullet x| && \text{by definition of } |\,| \\
&= 1 + |y| + |x| && \text{by the inductive hypothesis} \\
&= |ay| + |x| && \text{by definition of } |\,| \\
&= |w| + |x| && \text{because } w = ay
\end{aligned}$$

In both cases, we conclude that $|w \bullet x| = |w| + |x|$. □

Lemma 1.3. *Concatenation is associative:* $(w \bullet x) \bullet y = w \bullet (x \bullet y)$ *for all strings* w*,* x*, and* y*.*

Proof: Let w, x, and y be arbitrary strings. Assume that $(z \bullet x) \bullet y = w \bullet (x \bullet y)$ for every string z such that $|z| < |w|$. (Again, we are using induction only on w.) There are two cases to consider.

- Suppose $w = \varepsilon$.

$$\begin{aligned}
(w \bullet x) \bullet y &= (\varepsilon \bullet x) \bullet y && \text{because } w = \varepsilon \\
&= x \bullet y && \text{by definition of } \bullet \\
&= \varepsilon \bullet (x \bullet y) && \text{by definition of } \bullet \\
&= w \bullet (x \bullet y) && \text{because } w = \varepsilon
\end{aligned}$$

- Suppose $w = az$ for some symbol a and some string z.

$$\begin{aligned}
(w \bullet x) \bullet y &= (az \bullet x) \bullet y && \text{because } w = az \\
&= (a \cdot (z \bullet x)) \bullet y && \text{by definition of } \bullet \\
&= a \cdot ((z \bullet x) \bullet y) && \text{by definition of } \bullet \\
&= a \cdot (z \bullet (x \bullet y)) && \text{by the inductive hypothesis} \\
&= az \bullet (x \bullet y) && \text{by definition of } \bullet \\
&= w \bullet (x \bullet y) && \text{because } w = az
\end{aligned}$$

In both cases, we conclude that $(w \bullet x) \bullet y = w \bullet (x \bullet y)$. □

This is not the only boilerplate that one can use for induction proofs on strings. For example, we can modify the inductive case analysis using the following observation: A *non-empty* string w is either a single symbol or the concatenation of two non-empty strings, which (by Lemma 1.2) must be shorter than w. Here is an alternate proof of Lemma 1.3 that uses this alternative recursive structure:

Proof: Let w, x, and y be arbitrary strings. Assume that $(z \bullet x) \bullet y = w \bullet (x \bullet y)$ for every string z such that $|z| < |w|$. There are ***three*** cases to consider.

- Suppose $w = \varepsilon$.

$$\begin{aligned}
(w \bullet x) \bullet y &= (\varepsilon \bullet x) \bullet y && \text{because } w = \varepsilon \\
&= x \bullet y && \text{by definition of } \bullet \\
&= \varepsilon \bullet (x \bullet y) && \text{by definition of } \bullet \\
&= w \bullet (x \bullet y) && \text{because } w = \varepsilon
\end{aligned}$$

- Suppose w is equal to some symbol a.

$$\begin{aligned}
(w \bullet x) \bullet y &= (a \bullet x) \bullet y && \text{because } w = a \\
&= (a \cdot x) \bullet y && \text{because } a \bullet z = a \cdot z \text{ by definition of } \bullet \\
&= a \cdot (x \bullet y) && \text{by definition of } \bullet \\
&= a \bullet (x \bullet y) && \text{because } a \bullet z = a \cdot z \text{ by definition of } \bullet \\
&= w \bullet (x \bullet y) && \text{because } w = a
\end{aligned}$$

- Suppose $w = uv$ for some nonempty strings u and v.

$$\begin{aligned}
(w \bullet x) \bullet y &= ((u \bullet v) \bullet x) \bullet y && \text{because } w = uv \\
&= (u \bullet (v \bullet x)) \bullet y && \text{by the inductive hypothesis, because } |u| < |w| \\
&= u \bullet ((v \bullet x) \bullet y) && \text{by the inductive hypothesis, because } |u| < |w| \\
&= u \bullet (v \bullet (x \bullet y)) && \text{by the inductive hypothesis, because } |v| < |w| \\
&= (u \bullet v) \bullet (x \bullet y) && \text{by the inductive hypothesis, because } |u| < |w| \\
&= w \bullet (x \bullet y) && \text{because } w = uv
\end{aligned}$$

In both cases, we conclude that $(w \bullet x) \bullet y = w \bullet (x \bullet y)$. □

1.3 Indices, Substrings, and Subsequences

For any string w and any integer $1 \le i \le |w|$, the expression $\boldsymbol{w_i}$ denotes the ith symbol in w, counting from left to right. More formally, w_i is recursively defined as follows:

$$w_i := \begin{cases} a & \text{if } w = ax \text{ and } i = 1 \\ x_{i-1} & \text{if } w = ax \text{ and } i > 1 \end{cases}$$

As one might reasonably expect, w_i is formally undefined if $i < 1$ or $w = \varepsilon$, and therefore (by induction) if $i > |w|$. The integer i is called the ***index*** of w_i.

We sometimes write strings as a concatenation of their constituent symbols using this subscript notation: $w = w_1 w_2 \cdots w_{|w|}$. While standard, this notation is slightly misleading, since it *incorrectly* suggests that the string w contains at least three symbols, when in fact w could be a single symbol or even the empty string.

In actual code, subscripts are usually expressed using the bracket notation $\boldsymbol{w[i]}$. Brackets were introduced as a typographical convention over a hundred years ago because subscripts and superscripts[1] were difficult or impossible to type.[2] We sometimes write strings as explicit arrays $w[1..n]$, with the

[1] The same bracket notation is also used for bibliographic references, instead of the traditional footnote/endnote superscripts, for exactly the same reasons.

[2] A ***typewriter*** is an obsolete mechanical device loosely resembling a computer keyboard. Pressing a key on a typewriter moves a lever (called a "typebar") that strikes a cloth ribbon full of ink against a piece of paper, leaving the image of a single character. Many historians believe that the ordering of letters on modern keyboards (`QWERTYUIOP`) evolved in the late 1800s, reaching its modern form on the 1874 Sholes & Glidden Type-Writer™, in part to separate many common letter pairs, to prevent typebars from jamming against each other; this is also why the keys on most modern keyboards are arranged in a slanted grid. (The common folk theory that the ordering was deliberately intended to slow down typists doesn't withstand careful scrutiny.) A more recent theory suggests that the ordering was influenced by telegraph[3] operators, who found older alphabetic arrangements confusing, in part because of ambiguities in American Morse Code.

understanding that $n = |w|$. Again, this notation is potentially misleading; always remember that n might be zero; the string/array could be empty.

A ***substring*** of a string w is another string obtained from w by deleting zero or more symbols from the beginning and from the end. Formally, a string y is a substring of w if and only if there are strings x and z such that $w = xyz$. Extending the array notation for strings, we write $\boldsymbol{w[i..j]}$ to denote the substring of w starting at w_i and ending at w_j. More formally, we define

$$w[i..j] := \begin{cases} \varepsilon & \text{if } j < i, \\ w_i \cdot w[i+1..j] & \text{otherwise.} \end{cases}$$

A ***proper substring*** of w is any substring other than w itself. For example, LAUGH is a proper substring of SLAUGHTER. Whenever y is a (proper) substring of w, we also call w a (proper) ***superstring*** of y.

A ***prefix*** of $w[1..n]$ is any substring of the form $w[1..j]$. Equivalently, a string p is a ***prefix*** of another string w if and only if there is a string x such that $px = w$. A ***proper prefix*** of w is any prefix except w itself. For example, DIE is a proper prefix of DIET.

Similarly, a suffix of $w[1..n]$ is any substring of the form $w[i..n]$. Equivalently, a string s is a ***suffix*** of a string w if and only if there is a string x such that $xs = w$. A ***proper suffix*** of w is any suffix except w itself. For example, YES is a proper suffix of EYES, and HE is both a proper prefix and a proper suffix of HEADACHE.

A ***subsequence*** of a string w is a strong obtained by deleting zero or more symbols from *anywhere* in w. More formally, z is a subsequence of w if and only if

- $z = \varepsilon$, or
- $w = ax$ for some symbol a and some string x such that z is a subsequence of x.
- $w = ax$ and $z = ay$ for some symbol a and some strings x and y, and y is a subsequence of x.

A ***proper subsequence*** of w is any subsequence of w other than w itself. Whenever z is a (proper) subsequence of w, we also call w a (proper) ***supersequence*** of z.

Substrings and subsequences are not the same objects; don't confuse them! Every substring of w is also a subsequence of w, but not every subsequence is a substring. For example, METAL is a subsequence, but not a substring, of MEATBALL. To emphasize the distinction, we sometimes redundantly refer to substrings of w as ***contiguous*** substrings, meaning all their symbols appear together in w.

[3] A ***telegraph*** is an obsolete electromechanical communication device consisting of an electrical circuit with a switch at one end and an electromagnet at the other. The sending operator would press and release a key, closing and opening the circuit, originally causing the electromagnet to push a stylus onto a moving paper tape, leaving marks that could be decoded by the receiving operator. (Operators quickly discovered that they could directly decode the clicking sounds made by the electromagnet, and so the paper tape became obsolete almost immediately.) The most common scheme within the US to encode symbols, developed by Alfred Vail and Samuel Morse in 1837, used (mostly) short (·) and long (−) marks—now called "dots" and "dashes", or "dits" and "dahs"—separated by gaps of various lengths. American Morse code (as it became known) was ambiguous; for example, the letter Z and the string SE were both encoded by the sequence $\cdots \cdot$ ("di-di-dit, dit"). This ambiguity has been blamed for the S key's position on the typewriter keyboard near E and Z.

Vail and Morse were of course not the first people to propose encoding symbols as strings of bits. That honor apparently falls to Francis Bacon, who devised a five-bit binary encoding of the alphabet (except for the letters J and U) in 1605 as the basis for a steganographic code—a method or hiding secret message in otherwise normal text.

Exercises

Most of the following exercises ask for proofs of various claims about strings. For each claim, give a complete, self-contained, formal proof by inductive definition-chasing, using the boilerplate structure recommended in Section 1.2. You can use Lemmas 1.1, 1.2, and 1.3, but don't assume any other facts about strings that you have not proved. Do not use the words "obvious" or "clearly" or "just". Most of these claims **are** in fact obvious; the real exercise is understanding **why** they're obvious.

1. For any symbol a and any string w, let $\#(a, w)$ denote the number of occurrences of a in w. For example, $\#(\texttt{A}, \texttt{BANANA}) = 3$ and $\#(\texttt{X}, \texttt{FLIBBERTIGIBBET}) = 0$.

   (a) Give a formal recursive definition of the function $\# : \Sigma \times \Sigma^* \to \mathbb{N}$.

   (b) Prove that $\#(a, xy) = \#(a, x) + \#(a, y)$ for every symbol a and all strings x and y. Your proof must rely on both your answer to part (a) and the formal recursive definition of string concatenation.

2. Recursively define a set L of strings over the alphabet $\{\texttt{0}, \texttt{1}\}$ as follows:

   - The empty string ε is in L.
   - For any two strings x and y in L, the string $\texttt{0}x\texttt{1}y\texttt{0}$ is also in L.
   - These are the only strings in L.

   (a) Prove that the string $\texttt{000010101010010100}$ is in L.

   (b) Prove by induction that every string in L has exactly twice as many $\texttt{0}$s as $\texttt{1}$s. (You may assume the identity $\#(a, xy) = \#(a, x) + \#(a, y)$ for any symbol a and any strings x and y; see Exercise 1(b).)

   (c) Give an example of a string with exactly twice as many $\texttt{0}$s as $\texttt{1}$s that is *not* in L.

3. For any string w and any non-negative integer n, let w^n denote the string obtained by concatenating n copies of w; more formally, we define

$$w^n := \begin{cases} \varepsilon & \text{if } n = 0 \\ w \bullet w^{n-1} & \text{otherwise} \end{cases}$$

   For example, $(\texttt{BLAH})^5 = \texttt{BLAHBLAHBLAHBLAHBLAH}$ and $\varepsilon^{374} = \varepsilon$.

   Prove that $w^m \bullet w^n = w^{m+n}$ for every string w and all integers non-negative integer n and m.

4. Let w be an arbitrary string, and let $n = |w|$. Prove each of the following statements.

   (a) w has exactly $n + 1$ prefixes.

   (b) w has exactly n proper suffixes.

   (c) w has at most $n(n + 1)/2$ distinct substrings.

   (d) w has at most $2^n - 1$ proper subsequences.

5. The ***reversal*** w^R of a string w is defined recursively as follows:

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = a \cdot x \end{cases}$$

   (a) Prove that $|w^R| = |w|$ for every string w.
   (b) Prove that $(wx)^R = x^R w^R$ for all strings w and x.
   (c) Prove that $(w^R)^n = (w^n)^R$ for every string w and every integer $n \geq 0$. (See Exercise 1.)
   (d) Prove that $(w^R)^R = w$ for every string w.

6. Let w be an arbitrary string, and let $n = |w|$. Prove the following statements for all indices $1 \leq i \leq j \leq k \leq n$.
   (a) $\big|w[i..j]\big| = j - i + 1$
   (b) $w[i..j] \bullet w[j+1..k] = w[i..k]$
   (c) $w^R[i..j] = (w[i'..j'])^R$ where $i' = |w| + 1 - j$ and $j' = |w| + 1 - i$.

7. A ***palindrome*** is a string that is equal to its reversal.
   (a) Give a recursive definition of a palindrome over the alphabet Σ.
   (b) Prove that any string p meets your recursive definition of a palindrome if and only if $p = p^R$.

8. A string $w \in \Sigma^*$ is called a ***shuffle*** of two strings $x, y \in \Sigma^*$ if at least one of the following recursive conditions is satisfied:
   - $w = x = y = \varepsilon$.
   - $w = aw'$ and $x = ax'$ and w' is a shuffle of x' and y, for some $a \in \Sigma$ and some $w', x' \in \Sigma^*$.
   - $w = aw'$ and $y = ay'$ and w' is a shuffle of x and y', for some $a \in \Sigma$ and some $w', y' \in \Sigma^*$.

   For example, the string BANANANANANAS is a shuffle of the strings BANANA and ANANAS.
   (a) Prove that if w is a shuffle of x and y, then $|w| = |x| + |y|$.
   (b) Prove that if w is a shuffle of x and y, then w^R is a shuffle of x^R and y^R.

9. Consider the following pair of mutually recursive functions on strings:

$$evens(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ odds(x) & \text{if } w = ax \end{cases} \qquad odds(w) := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ a \cdot evens(x) & \text{if } w = ax \end{cases}$$

   (a) Prove the following identity for all strings w and x:

$$evens(w \bullet x) = \begin{cases} evens(w) \bullet evens(x) & \text{if } |w| \text{ is even,} \\ evens(w) \bullet odds(x) & \text{if } |w| \text{ is odd.} \end{cases}$$

(b) State and prove a similar identity for $odds(w \bullet x)$.

10. For any positive integer n, the ***Fibonacci string F_n*** is defined recursively as follows:

$$F_n = \begin{cases} \texttt{0} & \text{if } n = 1, \\ \texttt{1} & \text{if } n = 2, \\ F_{n-2} \bullet F_{n-1} & \text{otherwise.} \end{cases}$$

For example, $F_6 = \texttt{10101101}$ and $F_7 = \texttt{0110110101101}$.

(a) Prove that for every integer $n \geq 2$, the string F_n can also be obtained from F_{n-1} by replacing every occurrence of `0` with `1` and replacing every occurrence of `1` with `01`. More formally, prove that $F_n = Finc(F_{n-1})$, where

$$Finc(w) = \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ \texttt{1} \cdot Finc(x) & \text{if } w = \texttt{0}x \\ \texttt{01} \bullet Finc(x) & \text{if } w = \texttt{1}x \end{cases}$$

[Hint: First prove that $Finc(x \bullet y) = Finc(x) \bullet Finc(y)$.]

(b) Prove that `00` and `111` are not substrings of any Fibonacci string F_n.

11. Prove that the following three properties of strings are in fact identical.

    - A string $w \in \{0,1\}^*$ is ***balanced*** if it satisfies one of the following conditions:
      - $w = \varepsilon$,
      - $w = 0x1$ for some balanced string x, or
      - $w = xy$ for some balanced strings x and y.
    - A string $w \in \{0,1\}^*$ is ***erasable*** if it satisfies one of the following conditions:
      - $w = \varepsilon$, or
      - $w = x01y$ for some strings x and y such that xy is erasable. (The strings x and y are not necessarily erasable.)
    - A string $w \in \{0,1\}^*$ is ***conservative*** if it satisfies ***both*** of the following conditions:
      - w has an equal number of 0s and 1s, and
      - no prefix of w has more 0s than 1s.

    (a) Prove that every balanced string is erasable.

    (b) Prove that every erasable string is conservative.

    (c) Prove that every conservative string is balanced.

    *[Hint: To develop intuition, it may be helpful to think of 0s as left brackets and 1s as right brackets, but **don't** invoke this intuition in your proofs.]*

12. A string $w \in \{0,1\}^*$ ***equitable*** if it has an equal number of 0s and 1s.

    (a) Prove that a string w is equitable if and only if it satisfies one of the following conditions:

    - $w = \varepsilon$,
    - $w = 0x1$ for some equitable string x,
    - $w = 1x0$ for some equitable string x, or
    - $w = xy$ for some equitable strings x and y.

    (b) Prove that a string w is equitable if and only if it satisfies one of the following conditions:

    - $w = \varepsilon$,
    - $w = x01y$ for some strings x and y such that xy is equitable, or
    - $w = x10y$ for some strings x and y such that xy is equitable.

    In the last two cases, the individual strings x and y are not necessarily equitable.

    (c) Prove that a string w is equitable if and only if it satisfies one of the following conditions:

    - $w = \varepsilon$,
    - $w = xy$ for some balanced string x and some equitable string y, or
    - $w = x^R y$ for some for some balanced string x and some equitable string y.

    (See the previous exercise for the definition of "balanced".)

Caveat lector: This is the first edition of this lecture note. Please send bug reports and suggestions to jeffe@illinois.edu.

> *But the Lord came down to see the city and the tower the people were building. The Lord said, "If as one people speaking the same language they have begun to do this, then nothing they plan to do will be impossible for them. Come, let us go down and confuse their language so they will not understand each other."*
>
> — Genesis 11:6–7 (New International Version)

> *Imagine a piano keyboard, eh, 88 keys, only 88 and yet, and yet, hundreds of new melodies, new tunes, new harmonies are being composed upon hundreds of different keyboards every day in Dorset alone. Our language, tiger, our language: hundreds of thousands of available words, frillions of legitimate new ideas, so that I can say the following sentence and be utterly sure that nobody has ever said it before in the history of human communication: "**Hold the newsreader's nose squarely, waiter, or friendly milk will countermand my trousers.**" Perfectly ordinary words, but never before put in that precise order. A unique child delivered of a unique mother.*
>
> — Stephen Fry, *A Bit of Fry and Laurie*, Series 1, Episode 3 (1989)

2 Regular Languages

2.1 Definitions

A ***formal language*** (or just a ***language***) is a set of strings over some finite alphabet Σ, or equivalently, an arbitrary subset of Σ^*. For example, each of the following sets is a language:

- The empty set $\emptyset$.[1]
- The set $\{\varepsilon\}$.
- The set $\{0,1\}^*$.
- The set $\{\texttt{THE}, \texttt{OXFORD}, \texttt{ENGLISH}, \texttt{DICTIONARY}\}$.
- The set of all subsequences of $\texttt{THE}\diamond\texttt{OXFORD}\diamond\texttt{ENGLISH}\diamond\texttt{DICTIONARY}$.
- The set of all words in *The Oxford English Dictionary*.
- The set of all strings in $\{0,1\}^*$ with an odd number of 1s.
- The set of all strings in $\{0,1\}^*$ that represent a prime number in base 13.
- The set of all sequences of turns that solve the Rubik's cube (starting in some fixed configuration)
- The set of all python programs that print "Hello World!"

As a notational convention, I will always use italic upper-case letters (usually L, but also A, B, C, and so on) to represent languages.

Formal languages are not "languages" in the same sense that English, Klingon, and Python are "languages". Strings in a formal language do not necessarily carry any "meaning", nor are they necessarily assembled into larger units ("sentences" or "paragraphs" or "packages") according to some "grammar".

It is ***very*** important to distinguish between three "empty" objects. Many beginning students have trouble keeping these straight.

[1] The empty set symbol $\emptyset$ derives from the Norwegian letter Ø, pronounced like a sound of disgust or a German ö, and *not* from the Greek letter ϕ. Calling the empty set "fie" or "fee" makes the baby Jesus cry.

- $\emptyset$ is the empty *language*, which is a set containing zero strings. $\emptyset$ is not a string.
- $\{\varepsilon\}$ is a language containing exactly one string, which has length zero. $\{\varepsilon\}$ is not empty, and it is not a string.
- ε is the empty *string*, which is a sequence of length zero. ε is not a language.

2.2 Building Languages

Languages can be combined and manipulated just like any other sets. Thus, if A and B are languages over Σ, then their union $A \cup B$, intersection $A \cap B$, difference $A \setminus B$, and symmetric difference $A \oplus B$ are also languages over Σ, as is the complement $\overline{A} := \Sigma^* \setminus A$. However, there are two more useful operators that are specific to sets of *strings*.

The ***concatenation*** of two languages A and B, again denoted $A \bullet B$ or just AB, is the set of all strings obtained by concatenating an arbitrary string in A with an arbitrary string in B:

$$A \bullet B := \{xy \mid x \in A \text{ and } y \in B\}.$$

For example, if $A = \{\texttt{HOCUS}, \texttt{ABRACA}\}$ and $B = \{\texttt{POCUS}, \texttt{DABRA}\}$, then $A \bullet B = \{\texttt{HOCUSPOCUS}, \texttt{ABRACAPOCUS}, \texttt{HOCUSDABRA}, \texttt{ABRACADABRA}\}$. In particular, for every language A, we have

$$\emptyset \bullet A = A \bullet \emptyset = \emptyset \qquad \text{and} \qquad \{\varepsilon\} \bullet A = A \bullet \{\varepsilon\} = A.$$

The ***Kleene closure*** or ***Kleene star***[2] of a language L, denoted L^*, is the set of all strings obtained by concatenating a sequence of zero or more strings from L. For example, $\{\texttt{0}, \texttt{11}\}^* = \{\varepsilon, \texttt{0}, \texttt{00}, \texttt{11}, \texttt{000}, \texttt{011}, \texttt{110}, \texttt{0000}, \texttt{0011}, \texttt{0110}, \texttt{1100}, \texttt{1111}, \texttt{00000}, \texttt{00011}, \texttt{00110}, \ldots, \texttt{000111100110111111}, \ldots\}$. More formally, L^* is defined recursively as the set of all strings w such that either

- $w = \varepsilon$, or
- $w = xy$, for some strings $x \in L$ and $y \in L^*$.

This definition immediately implies that

$$\emptyset^* = \{\varepsilon\}^* = \{\varepsilon\}.$$

For any other language L, the Kleene closure L^* is infinite and contains arbitrarily long (but *finite!*) strings. Equivalently, L^* can also be defined as the smallest superset of L that contains the empty string ε and is closed under concatenation (hence "closure"). The set of all strings Σ^* is, just as the notation suggests, the Kleene closure of the alphabet Σ (where each symbol is viewed as a string of length 1).

A useful variant of the Kleene closure operator is the ***Kleene plus***, defined as $L^+ := L \bullet L^*$. Thus, L^+ is the set of all strings obtained by concatenating a sequence of ***one*** or more strings from L.

2.3 Regular Languages and Regular Expressions

A language L is ***regular*** if and only if it satisfies one of the following (recursive) conditions:

- L is empty;
- L contains a single string (which could be the empty string ε);
- L is the union of two regular languages;

[2] after Stephen Kleene, who pronounced his last name "***clay***-knee", not "clean" or "cleanie" or "claynuh" or "dimaggio".

- L is the concatenation of two regular languages; or
- L is the Kleene closure of a regular language.

Regular languages are normally described using a slightly more compact representation called ***regular expressions***, which omit braces around one-string sets, use $+$ to represent union instead of $\cup$, and juxtapose subexpressions to represent concatenation instead of using an explicit operator $\bullet$. By convention, in the absence of parentheses, the * operator has highest precedence, followed by the (implicit) concatenation operator, followed by $+$. Thus, for example, the regular expression $\mathtt{10}^*$ is shorthand for $\{\mathtt{1}\} \bullet \{\mathtt{0}\}^*$. As a larger example, the regular expression

$$\mathtt{0} + \mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1} + \mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$$

represents the language

$$\{\mathtt{0}\} \cup \big(\{\mathtt{0}\}^* \bullet \{\mathtt{1}\} \bullet \big((\{\mathtt{1}\} \bullet \{\mathtt{0}\}^* \bullet \{\mathtt{1}\}) \cup (\{\mathtt{0}\} \bullet \{\mathtt{1}\}^* \bullet \{\mathtt{0}\})\big)^* \bullet \{\mathtt{1}\} \bullet \{\mathtt{0}\}^*\big).$$

Here are a few more examples of regular expressions and the languages they represent.

- $\mathtt{0}^*$ — the set of all strings of $\mathtt{0}$s, including the empty string.
- $\mathtt{00000}^*$ — the set of all strings consisting of at least four $\mathtt{0}$s.
- $(\mathtt{00000})^*$ — the set of all strings of $\mathtt{0}$s whose length is a multiple of 5.
- $(\varepsilon + \mathtt{1})(\mathtt{01})^*(\varepsilon + \mathtt{0})$ — the set of all strings of alternating $\mathtt{0}$s and $\mathtt{1}$s, or equivalently, the set of all binary strings that do not contain the substrings $\mathtt{00}$ or $\mathtt{11}$.
- $((\varepsilon + \mathtt{0} + \mathtt{00} + \mathtt{000})\mathtt{1})^*(\varepsilon + \mathtt{0} + \mathtt{00} + \mathtt{000})$ — the set of all binary strings that do not contain the substring $\mathtt{0000}$.
- $((\mathtt{0} + \mathtt{1})(\mathtt{0} + \mathtt{1}))^*$ — the set of all binary strings whose length is even.
- $\mathtt{1}^*(\mathtt{01}^*\mathtt{01}^*)^*$ — the set of all binary strings with an even number of $\mathtt{0}$s.
- $\mathtt{0} + \mathtt{1}(\mathtt{0} + \mathtt{1})^*\mathtt{00}$ — the set of all non-negative binary numerals divisible by 4 and with no redundant leading $\mathtt{0}$s.
- $\mathtt{0} + \mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1} + \mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$ — the set of all non-negative binary numerals divisible by 3, possibly with redundant leading $\mathtt{0}$s.

The last example should ***not*** be obvious. It is straightforward, but rather tedious, to prove by induction that every string in $\mathtt{0} + \mathtt{0}^*\mathtt{1}(\mathtt{10}^*\mathtt{1} + \mathtt{01}^*\mathtt{0})^*\mathtt{10}^*$ is the binary representation of a non-negative multiple of 3. It is similarly straightforward, and similarly tedious, to prove that the binary representation of *every* non-negative multiple of 3 matches this regular expression. In a later note, we will see a systematic method for deriving regular expressions for some languages that avoids (or more accurately, automates) this tedium.

Most of the time we do not distinguish between regular expressions and the languages they represent, for the same reason that we do not normally distinguish between the arithmetic expression "2+2" and the integer 4, or the symbol π and the area of the unit circle. However, we sometimes need to refer to regular expressions themselves *as strings*. In those circumstances, we write $\boldsymbol{L(R)}$ to denote the language represented by the regular expression R. String w ***matches*** regular expression R if and only if $w \in L(R)$.

Two regular expressions R and R' are ***equivalent*** if they describe the same language; for example, the regular expressions $(0+1)^*$ and $(1+0)^*$ are equivalent, because the union operator is commutative.

Almost every regular language can be represented by infinitely many distinct but equivalent regular expressions, even if we ignore ultimately trivial equivalences like $L = (L\emptyset)^*L\varepsilon + \emptyset$. The following identities, which we state here without (easy) proofs, are useful for designing, simplifying, or understanding regular expressions.

Lemma 2.1. *The following identities hold for all languages A, B, and C:*
(a) $\emptyset A = A\emptyset = \emptyset$.
(b) $\varepsilon A = A\varepsilon = A$.
(c) $A + B = B + A$.
(d) $(A + B) + C = A + (B + C)$.
(e) $(AB)C = A(BC)$.
(f) $A(B + C) = AB + AC$.

Lemma 2.2. *The following identities hold for every language L:*
(a) $L^* = \varepsilon + L^+ = L^*L^* = (L + \varepsilon)^* = (L \setminus \varepsilon)^* = \varepsilon + L + L^+L^+$.
(b) $L^+ = L^* \setminus \varepsilon = LL^* = L^*L = L^+L^* = L^*L^+ = L + L^+L^+$.
(c) $L^+ = L^*$ *if and only if* $\varepsilon \in L$.

Lemma 2.3 (Arden's Rule). *For any languages A, B, and L such that $L = AL + B$, we have $A^*B \subseteq L$. Moreover, if A does not contain the empty string, then $L = AL + B$ if and only if $L = A^*B$.*

2.4 Things What Ain't Regular Expressions

Many computing environments and programming languages support patterns called ***regexen*** (singular ***regex***, pluralized like *ox*) that are considerably more general and powerful than regular expressions. Regexen include special symbols representing negation, character classes (for example, upper-case letters, or digits), contiguous ranges of characters, line and word boundaries, limited repetition (as opposed to the unlimited repetition allowed by *), back-references to earlier subexpressions, and even local variables. Despite its obvious etymology, a regex is ***not*** necessarily a regular expression, and it does ***not*** necessarily describe a regular language![3]

Another type of pattern that is often confused with regular expression are ***globs***, which are patterns used in most Unix shells and some scripting languages to represent sets file names. Globs include symbols for arbitrary single characters (`?`), single characters from a specified range (`[a-z]`), arbitrary substrings (`*`), and substrings from a specified finite set (`{foo,ba{r,z}}`). Globs are significantly *less* powerful than regular expressions.

2.5 Not Every Language is Regular

You may be tempted to conjecture that *all* languages are regular, but in fact, the following cardinality argument *almost all* languages are *not* regular. To make the argument concrete, let's consider languages over the single-symbol alphabet $\{\diamond\}$.

- Every regular expression over the one-symbol alphabet $\{\diamond\}$ is itself a string over the 7-symbol alphabet $\{\diamond, +, (,), *, \varepsilon, \emptyset\}$. By interpreting these symbols as the digits 1 through 7, we can interpret any string over this larger alphabet as the base-8 representation of some unique integer. Thus, the set of all regular expressions over $\{\diamond\}$ is *at most* as large as the set of integers, and is therefore countably infinite. It follows that the set of all regular *languages* over $\{\diamond\}$ is also countably infinite.

[3]However, regexen are not all-powerful, either; see http://stackoverflow.com/a/1732454/775369.

- On the other hand, for any real number $0 \le \alpha < 1$, we can define a corresponding language

$$L_\alpha = \left\{ \diamond^n \;\middle|\; \alpha 2^n \bmod 1 \ge 1/2 \right\}.$$

  In other words, L_α contains the string $\diamond^n$ if and only if the $(n+1)$th bit in the binary representation of α is equal to 1. For any distinct real numbers $\alpha \neq \beta$, the binary representations of α and β must differ in some bit, so $L_\alpha \neq L_\beta$. We conclude that the set of ***all*** languages over $\{\diamond\}$ is *at least* as large as the set of real numbers between 0 and 1, and is therefore uncountably infinite.

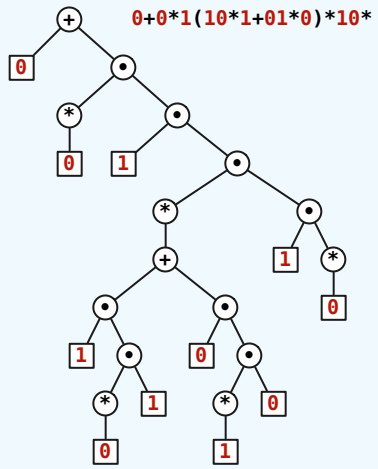
We will see several explicit examples of non-regular languages in future lectures. For example, the set of all regular expressions over $\{0, 1\}$ is not itself a regular language!

2.6 Parsing Regular Expressions

Most algorithms for regular expressions require them in the form of *regular expression trees*, rather than as raw strings. A regular expression tree is one of the following:

- A leaf node labeled $\emptyset$.
- A leaf node labeled with a string in Σ^*.
- A node labeled $+$ with two children, each the root of an expression tree.
- A node labeled $*$ with one child, which is the root of an expression tree.
- A node labeled $\bullet$ with two children, each the root of an expression tree.

In other words, a regular expression tree directly encodes a sequence of alternation, concatenation and Kleene closure operations that defines a regular language. Similarly, when we want to prove things about regular expressions or regular languages, it is more natural to think of subexpressions *as subtrees* rather than *as substrings*.



★★★

Given any regular expression of length n, we can ***parse*** it into an equivalent regular expression tree in $O(n)$ time. Thus, when we see an algorithmic problem that starts "Given a regular expression. . . ", we can assume without loss of generality that we're actually given a regular expression *tree*.

We'll see more on this topic later.

Exercises

1. (a) Prove that $\{\varepsilon\} \bullet L = L \bullet \{\varepsilon\} = L$, for any language L.
   (b) Prove that $\emptyset \bullet L = L \bullet \emptyset = \emptyset$, for any language L.
   (c) Prove that $(A \bullet B) \bullet C = A \bullet (B \bullet C)$, for all languages A, B, and C.

(d) Prove that $|A \bullet B| = |A| \cdot |B|$, for all languages A and B. (The second $\cdot$ is multiplication!)

(e) Prove that L^* is finite if and only if $L = \emptyset$ or $L = \{\varepsilon\}$.

(f) Prove that $AB = BC$ implies $A^*B = BC^* = A^*BC^*$, for all languages A, B, and C.

2. Recall that the reversal w^R of a string w is defined recursively as follows:

$$w^R := \begin{cases} \varepsilon & \text{if } w = \varepsilon \\ x^R \bullet a & \text{if } w = a \cdot x \end{cases}$$

The reversal L^R of any language L is the set of reversals of all strings in L:

$$L^R := \{w^R \mid w \in L\}.$$

(a) Prove that $(AB)^R = B^R A^R$ for all languages A and B.

(b) Prove that $(L^R)^R = L$ for every language L.

(c) Prove that $(L^*)^R = (L^R)^*$ for every language L.

3. Prove that each of the following regular expressions is equivalent to $(\mathtt{0} + \mathtt{1})^*$.

(a) $\varepsilon + \mathtt{0}(\mathtt{0} + \mathtt{1})^* + \mathtt{1}(\mathtt{1} + \mathtt{0})^*$

(b) $\mathtt{0}^* + \mathtt{0}^*\mathtt{1}(\mathtt{0} + \mathtt{1})^*$

(c) $((\varepsilon + \mathtt{0})(\varepsilon + \mathtt{1}))^*$

(d) $\mathtt{0}^*(\mathtt{10}^*)^*$

(e) $(\mathtt{1}^*\mathtt{0})^*(\mathtt{0}^*\mathtt{1})^*$

4. For each of the following languages in $\{\mathtt{0}, \mathtt{1}\}^*$, describe an equivalent regular expression. There are infinitely many correct answers for each language. (This problem will become significantly simpler after we've seen finite-state machines, in the next lecture note.)

(a) Strings that end with the suffix $\mathtt{0}^9 = \mathtt{000000000}$.

(b) All strings except $\mathtt{010}$.

(c) Strings that contain the substring $\mathtt{010}$.

(d) Strings that contain the subsequence $\mathtt{010}$.

(e) Strings that do not contain the substring $\mathtt{010}$.

(f) Strings that do not contain the subsequence $\mathtt{010}$.

(g) Strings that contain an even number of occurrences of the substring $\mathtt{010}$.

*(h) Strings that contain an even number of occurrences of the substring $\mathtt{000}$.

(i) Strings in which every occurrence of the substring $\mathtt{00}$ appears before every occurrence of the substring $\mathtt{11}$.

(j) Strings w such that *in every prefix of* w, the number of $\mathtt{0}$s and the number of $\mathtt{1}$s differ by at most 1.

*(k) Strings w such that *in every prefix of w*, the number of `0`s and the number of `1`s differ by at most 2.

*(l) Strings in which the number of `0`s and the number of `1`s differ by a multiple of 3.

*(m) Strings that contain an even number of `1`s and an odd number of `0`s.

★(n) Strings that represent a number divisible by 5 in binary.

5. Prove that for any regular expression R such that $L(R)$ is nonempty, there is a regular expression equivalent to R that does not use the empty-set symbol $\emptyset$.

6. Prove that if L is a regular language, then L^R is also a regular language. *[Hint: How do you reverse a regular expression?]*

7. (a) Describe and analyze an efficient algorithm to determine, given a regular expression R, whether $L(R)$ is empty.

   (b) Describe and analyze an efficient algorithm to determine, given a regular expression R, whether $L(R)$ is infinite.

   In each problem, assume you are given R as a regular expression tree, not just a raw string.

Caveat lector: This is the first edition of this lecture note. Please send bug reports and suggestions to jeffe@illinois.edu.

> *Imagine a piano keyboard, eh, 88 keys, only 88 and yet, and yet, hundreds of new melodies, new tunes, new harmonies are being composed upon hundreds of different keyboards every day in Dorset alone. Our language, tiger, our language: hundreds of thousands of available words, frillions of legitimate new ideas, so that I can say the following sentence and be utterly sure that nobody has ever said it before in the history of human communication: "**Hold the newsreader's nose squarely, waiter, or friendly milk will countermand my trousers.**" Perfectly ordinary words, but never before put in that precise order. A unique child delivered of a unique mother.*
>
> — Stephen Fry, *A Bit of Fry and Laurie*, Series 1, Episode 3 (1989)

2½ Context-Free Languages and Grammars

2½.1 Definitions

Intuitively, a language is regular if it can be built from individual strings by concatenation, union, and repetition. In this note, we consider a wider class of ***context-free*** languages, which are languages that can be built from individual strings by concatenation, union, and *recursion*.

Formally, a language is context-free if and only if it has a certain type of recursive description known as a ***context-free grammar***, which is a structure with the following components:

- A finite set Σ, whose elements are called ***symbols*** or ***terminals***.
- A finite set Γ disjoint from Σ, whose elements are called ***non-terminals***.
- A finite set R of ***production rules*** of the form $A \to w$, where $A \in \Gamma$ is a non-terminal and $w \in (\Sigma \cup \Gamma)^*$ is a string of symbols and variables.
- A ***starting*** non-terminal, typically denoted S.

For example, the following eight production rules describe a context free grammar with terminals $\Sigma = \{0, 1\}$ and non-terminals $\Gamma = \{S, A, B\}$:

$$\begin{array}{llll} S \to A & A \to 0A & B \to B1 & C \to \varepsilon \\ S \to B & A \to 0C & B \to C1 & C \to 0C1 \end{array}$$

Normally we write grammars more compactly by combining the right sides of all rules for each non-terminal into one list, with alternatives separated by vertical bars.[1] For example, the previous grammar can be written more compactly as follows:

$$\begin{aligned} S &\to A \mid B \\ A &\to 0A \mid 0C \\ B &\to B1 \mid C1 \\ C &\to \varepsilon \mid 0C1 \end{aligned}$$

For the rest of this lecture, I will *almost* always use the following notational conventions.

[1] Yes, this means we now have *three* symbols $\cup$, $+$, and $\mid$ with exactly the same meaning. Sigh.

- Monospaced digits (`0`, `1`, `2`, ...), and symbols (`◇`, `$`, `#`, `•`, ...) are explicit terminals.
- Early lower-case Latin letters ($a, b, c, \ldots$) represent unknown or arbitrary terminals in Σ.
- Upper-case Latin letters ($A, B, C, \ldots$) and the letter S represent non-terminals in Γ.
- Late lower-case Latin letters ($\ldots, w, x, y, z$) represent strings in $(\Sigma \cup \Gamma)^*$, whose characters could be either terminals or non-terminals.

We can ***apply*** a production rule to a string in $(\Sigma \cup \Gamma)^*$ by replacing any instance of the non-terminal on the left of the rule with the string on the right. More formally, for any strings $x, y, z \in (\Sigma \cup \Gamma)^*$ and any non-terminal $A \in \Gamma$, applying the production rule $A \to y$ to the string xAz yields the string xyz. We use the notation $\boldsymbol{xAz \rightsquigarrow xyz}$ to describe this application. For example, we can apply the rule $C \to \texttt{0}C\texttt{1}$ to the string $\texttt{00}C\texttt{1}BAC\texttt{0}$ in two different ways:

$$\texttt{00}\underline{C}\texttt{1}BAC\texttt{0} \rightsquigarrow \texttt{00}\underline{\texttt{0}C\texttt{1}}\texttt{1}BAC\texttt{0} \qquad \texttt{00}C\texttt{1}BA\underline{C}\texttt{0} \rightsquigarrow \texttt{00}C\texttt{1}BA\underline{\texttt{0}C\texttt{1}}\texttt{0}$$

More generally, for any strings $x, z \in (\Sigma \cup \Gamma)^*$, we say that $\boldsymbol{z}$ ***derives from*** $\boldsymbol{x}$, written $\boldsymbol{x \rightsquigarrow^* z}$, if we can transform x into z by applying a finite sequence of production rules, or more formally, if either

- $x = z$, or
- $x \rightsquigarrow y$ and $y \rightsquigarrow^* z$ for some string $y \in (\Sigma \cup \Gamma)^*$.

Straightforward definition-chasing implies that, for any strings $w, x, y, z \in (\sigma \cup \gamma)^*$, if $x \rightsquigarrow^* y$, then $wxz \rightsquigarrow^* wyz$.

The ***language*** $\boldsymbol{L(w)}$ of any string $w \in (\Sigma \cup \Gamma)^*$ is the set of all strings in Σ^* that derive from w:

$$L(w) := \{x \in \Sigma^* \mid w \rightsquigarrow^* x\}.$$

The language ***generated by*** a context-free grammar G, denoted $\boldsymbol{L(G)}$, is the language of its starting non-terminal. Finally, a language is ***context-free*** if it is generated by some context-free grammar.

Context-free grammars are sometimes used to model natural languages. In this context, the symbols are *words*, and the strings in the languages are *sentences*. For example, the following grammar describes a simple subset of English sentences. (Here I diverge from the usual notation conventions. Strings in ⟨angle brackets⟩ are non-terminals, and regular strings are terminals.)

⟨sentence⟩ → ⟨noun phrase⟩⟨verb phrase⟩⟨noun phrase⟩
⟨noun phrase⟩ → ⟨adjective phrase⟩⟨noun⟩
⟨adj. phrase⟩ → ⟨article⟩ | ⟨possessive⟩ | ⟨adjective phrase⟩⟨adjective⟩
⟨verb phrase⟩ → ⟨verb⟩ | ⟨adverb⟩⟨verb phrase⟩

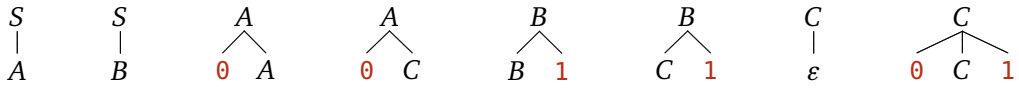
⟨noun⟩ → dog | trousers | daughter | nose | homework | time lord | pony | ⋯
⟨article⟩ → the | a | some | every | that | ⋯
⟨possessive⟩ → ⟨noun phrase⟩'s | my | your | his | her | ⋯
⟨adjective⟩ → friendly | furious | moist | green | severed | timey-wimey | little | ⋯
⟨verb⟩ → ate | found | wrote | killed | mangled | saved | invented | broke | ⋯
⟨adverb⟩ → squarely | incompetently | barely | sort of | awkwardly | totally | ⋯

2½.2 Parse Trees

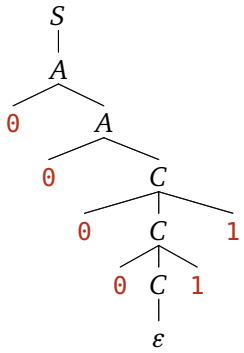
It is often useful to visualize derivations of strings in $L(G)$ using a ***parse tree***. The parse tree for a string $w \in L(G)$ is a rooted ordered tree where

- Each leaf is labeled with a terminal or the empty string ε. Concatenating these in order from left to right yields the string w.
- Each internal node is labeled with a non-terminal. In particular, the root is labeled with the start non-terminal S.
- For each internal node v, there is a production rule $A \to \omega$ where A is the label of v and the symbols in ω are the labels of the children of v in order from left to right.

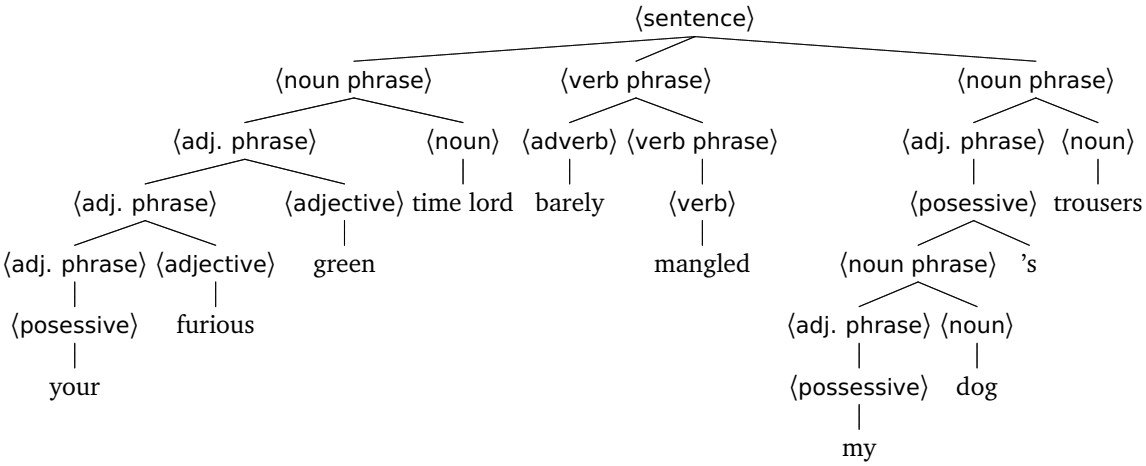
In other words, the production rules of the grammar describe *template trees* that can be assembled into larger parse trees. For example, the simple grammar on the previous page has the following templates, one for each production rule:



The same grammar gives us the following parse tree for the string `000011`:



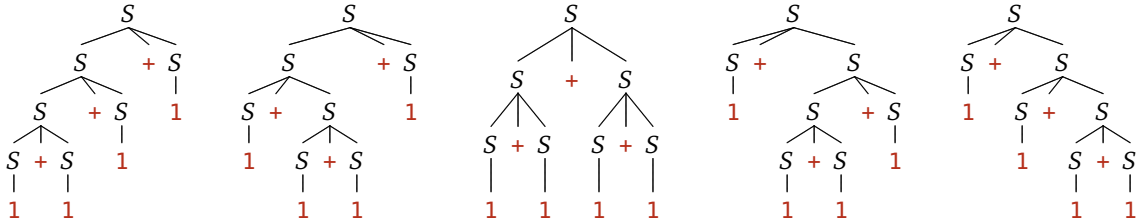
Our more complicated "English" grammar gives us parse trees like the following:



Any parse tree that contains at least one node with more than one non-terminal child corresponds to several different derivations. For example, when deriving an "English" sentence, we have a choice of

whether to expand the first ⟨noun phrase⟩ ("your furious green time lord") before or after the second ("my dog's trousers").

A string w is ***ambiguous*** with respect to a grammar if there is more than one parse tree for w, and a grammar G is ***ambiguous*** is some string is ambiguous with respect to G. Neither of the previous example grammars is ambiguous. However, the grammar $S \to 1 \mid S{+}S$ is ambiguous, because the string $1{+}1{+}1{+}1$ has five different parse trees:



A context-free language L is ***inherently ambiguous*** if every context-free grammar that generates L is ambiguous. The language generated by the previous grammar (the regular language $(1{+})^*1$) is *not* inherently ambiguous, because the unambiguous grammar $S \to 1 \mid 1{+}S$ generates the same language.

2½.3 From Grammar to Language

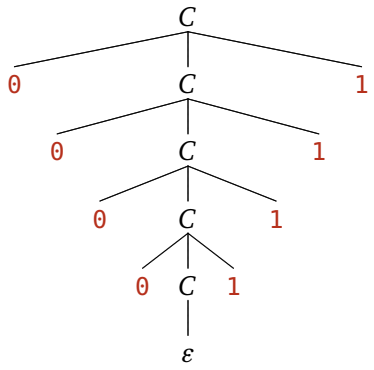
Let's figure out the language generated by our first example grammar

$$S \to A \mid B \qquad A \to 0A \mid 0C \qquad B \to B1 \mid C1 \qquad C \to \varepsilon \mid 0C1.$$

Since the production rules for non-terminal C do not refer to any other non-terminal, let's begin by figuring out $L(C)$. After playing around with the smaller grammar $C \to \varepsilon \mid 0C1$ for a few seconds, you can probably guess that its language is $\{\varepsilon, 01, 0011, 000111, \ldots\}$, that is, the set all of strings of the form 0^n1^n for some integer n. For example, we can derive the string 00001111 from the start non-terminal S using the following derivation:

$$C \rightsquigarrow 0C1 \rightsquigarrow 00C11 \rightsquigarrow 000C111 \rightsquigarrow 0000C1111 \rightsquigarrow 0000\varepsilon1111 = 00001111$$

The same derivation can be viewed as the following parse tree:



In fact, it is not hard to *prove* by induction that $L(C) = \{0^n1^n \mid n \geq 0\}$ as follows. As usual when we prove that two sets X and Y are equal, the proof has two stages: one stage to prove $X \subseteq Y$, the other to prove $Y \subseteq X$.

- First we prove that $C \rightsquigarrow^* 0^n1^n$ for every non-negative integer n.

  Fix an arbitrary non-negative integer n. Assume that $C \rightsquigarrow^* 0^k1^k$ for every non-negative integer $k < n$. There are two cases to consider.

  - If $n = 0$, then $0^n1^n = \varepsilon$. The rule $C \to \varepsilon$ implies that $C \rightsquigarrow \varepsilon$ and therefore $C \rightsquigarrow^* \varepsilon$.
  - Suppose $n > 0$. The inductive hypothesis implies that $C \rightsquigarrow^* 0^{n-1}1^{n-1}$. Thus, the rule $C \to 0C1$ implies that $C \rightsquigarrow 0C1 \rightsquigarrow^* 0(0^{n-1}1^{n-1})1 = 0^n1^n$.

  In both cases, we conclude that that $C \rightsquigarrow^* 0^n1^n$, as claimed.

- Next we prove that for every string $w \in \Sigma^*$ such that $C \rightsquigarrow^* w$, we have $w = 0^n1^n$ for some non-negative integer n.

  Fix an arbitrary string w such that $C \rightsquigarrow^* w$. Assume that for any string x such that $|x| < |w|$ and $C \rightsquigarrow^* x$, we have $x = 0^k1^k$ for some non-negative integer k. There are two cases to consider, one for each production rule.

  - If $w = \varepsilon$, then $w = 0^01^0$.
  - Suppose $w = 0x1$ for some string x such that $C \rightsquigarrow^* x$. Because $|x| = |w| - 2 < |w|$, the inductive hypothesis implies that $x = 0^k1^k$ for some integer k. Then we have $w = 0^{k+1}1^{k+1}$.

  In both cases, we conclude that that $w = 0^n1^n$ for some non-negative integer n, as claimed.

The first proof uses induction on strings, following the boilerplate proposed in the previous lecture; in particular, the case analysis mirrors the recursive definition of "string". The second proof uses *structural induction* on the grammar; the case analysis mirrors the recursive definition of the language of S, as described by the production rules. In both proofs, the inductive hypothesis is "Assume there is no smaller counterexample."

Similar analysis implies that $L(A) = \{0^m1^n \mid m > n\}$ and $L(B) = \{0^m1^n \mid m < n\}$, and therefore $L(S) = \{0^m1^n \mid m \neq n\}$.

2½.4 More Examples

★★★

Give three or four examples of simple but interesting context-free grammars. Some possibilities:

- Same number of 0s and 1s
- Different number of 0s and 1s
- Palindromes
- Balanced parentheses
- Arithmetic/algebraic expressions
- Regular expressions

2½.5 Regular Languages are Context-Free

The following inductive argument proves that every regular language is also a context-free language. Let L be an arbitrary regular language, encoded by some regular expression R. Assume that any regular expression shorter than R represents a context-free language. ("Assume no smaller counterexample.") We construct a context-free grammar for L as follows. There are several cases to consider.

- Suppose L is empty. Then L is generated by the trivial grammar $S \to S$.
- Suppose $L = \{w\}$ for some string $w \in \Sigma^*$. Then L is generated by the grammar $S \to w$.
- Suppose L is the union of some regular languages L_1 and L_2. The inductive hypothesis implies that L_1 and L_2 are context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1, and let G_2 be a context-free language for L_2 with starting non-terminal S_2, where the non-terminal sets in G_1 and G_2 are disjoint. Then $L = L_1 \cup L_2$ is generated by the production rule $S \to S_1 \mid S_2$.

- Suppose L is the concatenation of some regular languages L_1 and L_2. The inductive hypothesis implies that L_1 and L_2 are context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1, and let G_2 be a context-free language for L_2 with starting non-terminal S_2, where the non-terminal sets in G_1 and G_2 are disjoint. Then $L = L_1L_2$ is generated by the production rule $S \rightarrow S_1S_2$.

- Suppose L is the Kleene closure of some regular language L_1. The inductive hypothesis implies that L_1 is context-free. Let G_1 be a context-free language for L_1 with starting non-terminal S_1. Then $L = L_1^*$ is generated by the production rule $S \rightarrow \varepsilon \mid S_1S$.

In every case, we have found a context-free grammar that generates L, which means L is context-free.

In the next lecture note, we will prove that the context-free language $\{0^n1^n \mid n \geq 0\}$ is not regular. (In fact, this is the *canonical example* of a non-regular language.) Thus, context-free grammars are strictly more expressive than regular expressions.

2½.6 Not Every Language is Context-Free

Again, you may be tempted to conjecture that *every* language is context-free, but a variant of our earlier cardinality argument implies that this is not the case.

Any context-free grammar over the alphabet Σ can be encoded as a string over the alphabet $\Sigma \cup \Gamma \cup \{\varepsilon, \rightarrow, \mid, \$\}$, where \$ indicates the end of the production rules for each non-terminal. For example, our example grammar

$$S \rightarrow A \mid B \qquad A \rightarrow 0A \mid 0C \qquad B \rightarrow B1 \mid C1 \qquad C \rightarrow \varepsilon \mid 0C1$$

can be encoded as the string

S→A|B\$A→0A|0C\$B→B1|C1\$C→Ɛ|0C1\$

We can further encode any such string as a *binary* string by associating each symbol in the set $\Sigma \cup \Gamma \cup \{\varepsilon, \rightarrow, \mid, \$\}$ with a different binary substring. Specifically, if we encode each of the grammar symbols $\varepsilon, \rightarrow, \mid, \$$ as a string of the form 11^*0, each terminal in Σ as a string of the form 011^*0, and each non-terminal as a string of the form 0011^*0, we can unambiguously recover the grammar from the encoding. For example, applying the code

Ɛ ↦ 10	0 ↦ 010	S ↦ 0010
→ ↦ 110	1 ↦ 0110	A ↦ 00110
| ↦ 1110		B ↦ 001110
\$ ↦ 11110		C ↦ 0011110

transforms our example grammar into the 135-bit string

```
00101100011011100011101111000110
11001000110111001000111101111000
11101100011100110111000111100110
11110001111011010111001000111100
1011110.
```

Adding a 1 to the start of this bit string gives us the binary encoding of the integer

$$51\,115\,617\,766\,581\,763\,757\,672\,062\,401\,233\,529\,937\,502.$$

Our construction guarantees that two different context-free grammars over the same language (ignoring changing the names of the non-terminals) yield different positive integers. Thus, the set of context-free grammars over any alphabet is *at most* as large as the set of integers, and is therefore countably infinite. (Most integers are not encodings of context-free grammars, but that only helps us.) It follows that the set of all context-free *languages* over any fixed alphabet is also countably infinite. But we already showed that the set of *all* languages over any alphabet is uncountably infinite. So almost all languages are non-context-free!

Although we will probably not see them in this course, there are techniques for proving that certain languages are not context-free, just as there are for proving certain languages are not regular. In particular, the $\{0^n1^n0^n \mid n \geq 0\}$ is not context-free. (In fact, this is the *canonical example* of a non-context-free language.)

*2½.7 Chomsky Normal Form

For many algorithmic problems involving context-free grammars, it is helpful to consider grammars with a particular special structure called ***Chomsky normal form***, abbreviated ***CNF***:

- The starting non-terminal S does not appear on the right side of any production rule.
- The starting non-terminal S *may* have the production rule $S \to \varepsilon$.
- The right side of every other production rule is either a single terminal symbol or a string of exactly two non-terminals—that is, every other production rule has the form $A \to BC$ or $A \to a$.

A particularly attractive feature of CNF grammars is that they yield *full binary* parse trees; in particular, every parse tree for a string of length $n > 0$ has exactly $2n - 1$ non-terminal nodes. Consequently, any string of length n in the language of a CNF grammar can be derived in exactly $2n - 1$ production steps. It follows that we can actually determine whether a string belongs to the language of a CNF grammar by brute-force consideration of all possible derivations of the appropriate length.

For arbitrary context-free grammars, there is no similar upper bound on the length of a derivation, and therefore no similar brute-force membership algorithm, because the grammar may contain additional ***ε-productions*** of the form $A \to \varepsilon$ and/or ***unit productions*** of the form $A \to B$, where both A and B are non-terminals. Unit productions introduce nodes of degree 1 into any parse tree, and ε-productions introduce leaves that do not contribute to the word being parsed.

Fortunately, it *is* possible to determine membership in the language of an arbitrary context-free grammar, thanks to the following theorem. Two context-free grammars are ***equivalent*** if they define the same language.

Every context-free grammar is equivalent to a grammar in Chomsky normal form.

To be more specific, define the ***total length*** of a context-free grammar to be the number of symbols needed to write down the grammar; up to constant factors, the total length is the sum of the lengths of the production rules.

Theorem 2½.1. *Every context-free grammar with total length L can be mechanically converted into an equivalent grammar in Chomsky normal form with total length $O(L^2)$ in $O(L^2)$ time.*

Converting an arbitrary grammar into Chomsky normal form is a complex task. Fortunately, for most applications of context-free grammars, it's enough to know that the algorithm exists. For the sake of completeness, however, I will describe one such conversion algorithm here. This algorithm consists of several relatively straightforward stages. Efficient implementation of some of these stages requires standard graph-traversal algorithms, which we will describe much later in the course.

0. Add a new starting non-terminal. Add a new non-terminal S' and a production rule $S' \to S$, where S is the starting non-terminal for the given grammar. S' will be the starting non-terminal for the resulting CNF grammar. (In fact, this step is necessary only when $S \rightsquigarrow^* \varepsilon$, but at this point in the conversion process, we don't yet know whether that's true.)

1. Decompose long production rules. For each production rule $A \to \omega$ whose right side w has length greater than two, add new production rules of length two that still permit the derivation $A \rightsquigarrow^* \omega$. Specifically, suppose $\omega = \alpha\chi$ for some symbol $\alpha \in \Sigma \cup \Gamma$ and string $\chi \in (\Sigma \cup \Gamma)^*$. The algorithm replaces $A \to \omega$ with two new production rules $A \to \alpha B$ and $B \to \chi$, where B is a new non-terminal, and then (if necessary) recursively decomposes the production rule $B \to \chi$. For example, we would replace the long production rule $A \to \mathtt{0}BC\mathtt{1}CB$ with the following sequence of short production rules, where each A_i is a new non-terminal:

$$A \to \mathtt{0}A_1 \qquad A_1 \to BA_2 \qquad A_2 \to CA_3 \qquad A_3 \to \mathtt{1}A_4 \qquad A_4 \to CB$$

This stage can significantly increase the number of non-terminals and production rules, but it increases the *total length* of all production rules by at most a small constant factor.[2] The running time of this stage is $O(L)$.

2. Identify nullable non-terminals. A non-terminal A is ***nullable*** if and only if $A \rightsquigarrow^* \varepsilon$. The recursive definition of $\rightsquigarrow^*$ implies that A is nullable if and only if the grammar contains a production rule $A \to \omega$ where ω consists entirely of nullable non-terminals (in particular, if $\omega = \varepsilon$). You may be tempted to transform this recursive characterization directly into a recursive algorithm, but this is a bad idea; the resulting algorithm would fall into an infinite loop if (for example) the same non-terminal appeared on both sides of the same production rule. Instead, we apply the following ***fixed-point*** algorithm, which repeatedly scans through the entire grammar until a complete scan discovers no new nullable non-terminals.

```
NULLABLES(Σ, Γ, R, S):
  Γ_ε ← ∅        ⟨⟨known nullable non-terminals⟩⟩
  done ← FALSE
  while ¬done
      done ← TRUE
      for each non-terminal A ∈ Γ \ Γ_ε
          for each production rule A → ω
              if ω ∈ Γ_ε*
                  add A to Γ_ε
                  done ← FALSE
  return Γ_ε
```

At this point in the conversion algorithm, if S' is ***not*** identified as nullable, then we can safely remove it from the grammar and use the original starting nonterminal S instead.

As written, NULLABLES runs in $O(nL) = O(L^2)$ time, where n is the number of non-terminals in Γ. Each iteration of the main loop except the last adds at least one non-terminal to Γ_ε, so the algorithm

[2] In most textbook descriptions of this conversion algorithm, this stage is performed *last*, after removing ε-productions and unit productions. But with the stages in that traditional order, removing ε-productions could *exponentially* increase the length of the grammar in the worst case! Consider the production rule $A \to (BC)^k$, where B is nullable but C is not. Decomposing this rule first and then removing ε-productions introduces about $3k$ new rules; whereas, removing ε-productions first introduces 2^k new rules, most of which then must then be further decomposed.

halts after at most $n+1 \le L$ iterations, and in each iteration, we examine at most L production rules. There is a faster implementation of NULLABLES that runs in $O(n+L) = O(L)$ time,[3] but since other parts of the conversion algorithm already require $O(L^2)$ time, we needn't bother.

3. Eliminate ε-productions. First, remove every production rule of the form $A \to \varepsilon$. Then for each production rule $A \to w$, add all possible new production rules of the form $A \to w'$, where w' is a ***non-empty*** string obtained from w by removing one nullable non-terminal. For example, if if the grammar contained the production rule $A \to BC$, where B and C are both nullable, we would add two new production rules $A \to B \mid C$. (Adding these productions may increase the size of the grammar exponentially!) Finally, if S' was identified as nullable in the previous stage, add the production rule $S' \to \varepsilon$; this will be the *only* ε-production in the final grammar. This phase of the conversion runs in $O(L)$ time and at most triples the number of production rules.

4. Merge equivalent non-terminals. We say that two non-terminals A and B are ***equivalent*** if they can be derived from each other: $A \rightsquigarrow^* B$ and $B \rightsquigarrow^* A$. Because we have already removed ε-productions, any such derivation must consist entirely of unit productions. For example, in the grammar

$$S \to B \mid C, \qquad A \to B \mid D \mid CC \mid \mathtt{0}, \qquad B \to C \mid AD \mid \mathtt{1}, \qquad C \to A \mid DA, \qquad D \to BA \mid CS,$$

non-terminals A, B, C are all equivalent, but S is not in that equivalence class (because we cannot derive S from A) and neither is D (because we cannot derive A from D).

Construct a directed graph G whose vertices are the non-terminals and whose edges correspond to unit productions, in $O(L)$ time. Then two non-terminals are equivalent if and only if they are in the same strong component of G. Compute the strong components of G in $O(L)$ time using, for example, the algorithm of Kosaraju and Sharir. Then merge all the non-terminals in each equivalence class into a single non-terminal. Finally, remove any unit productions of the form $A \to A$. The total running time for this phase is $O(L)$. Starting with our example grammar above, merging B and C with A and removing the production $A \to A$ gives us the simpler grammar

$$S \to A, \qquad A \to AA \mid D \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad D \to AA \mid AS.$$

We could further simplify the grammar by merging all non-terminals reachable from S using only unit productions (in this case, merging non-terminals S and S), but this further simplification is unnecessary.

5. Remove unit productions. Once again, we construct a directed graph G whose vertices are the non-terminals and whose edges correspond to unit productions, in $O(L)$ time. Because no two non-terminals are equivalent, G is acyclic. Thus, using topological sort, we can index the non-terminals $A_1, A_2, \ldots, A_n$ such that for every unit production $A_i \to A_j$ we have $i < j$, again in $O(L)$ time; moreover, we can assume that the starting non-terminal is A_1. (In fact, both the dag G and the linear ordering of non-terminals was already computed in the previous phase!!)

Then for each index j in decreasing order, for each unit production $A_i \to A_j$ and each production $A_j \to \omega$, we add a new production rule $A_i \to \omega$. At this point, all unit productions are redundant and can be removed. Applying this algorithm to our example grammar above gives us the grammar

$$S \to AA \mid AS \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad A \to AA \mid AS \mid DA \mid \mathtt{0} \mid \mathtt{1}, \qquad D \to AA \mid AS.$$

[3] Consider the bipartite graph whose vertices correspond to non-terminals and the right sides of production rules, with one edge per rule. The faster algorithm is a modified breadth-first search of this graph, starting at the vertex representing ε.

In the worst case, each production rule for A_n is copied to each of the other $n-1$ non-terminals. Thus, this phase runs in $\Theta(nL) = O(L^2)$ time and increases the length of the grammar to $\Theta(nL) = O(L^2)$ in the worst case.

This phase dominates the running time of the CNF conversion algorithm. Unlike previous phases, no faster algorithm for removing unit transformations is known! There are grammars of length L with unit productions such that any equivalent grammar without unit productions has length $\Omega(L^{1.499999})$ (for any desired number of 9s), but this lower bound does not rule out the possibility of an algorithm that runs in only $O(L^{3/2})$ time. Closing the gap between $\Omega(L^{3/2-\varepsilon})$ and $O(L^2)$ has been an open problem since the early 1980s.

6. Protect terminals. Finally, for each terminal $a \in \Sigma$, we introduce a new non-terminal A_a and a new production rule $A_a \to a$, and then replace a with A_a in every production rule of length 2. This completes the conversion to Chomsky normal form. As claimed, the total running time of the algorithm is $O(L^2)$, and the total length of the output grammar is also $O(L^2)$.

CNF Conversion Example

As a running example, let's apply these stages one at a time to our first example grammar.

$$S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad C \to \varepsilon \mid \mathtt{0}C\mathtt{1}$$

0. Add a new starting non-terminal S'.

$$\underline{S' \to S} \qquad S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad C \to \varepsilon \mid \mathtt{0}C\mathtt{1}$$

1. Decompose the long production rule $C \to \mathtt{0}C\mathtt{1}$.

$$S' \to S \qquad S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \qquad B \to B\mathtt{1} \mid C\mathtt{1} \qquad \underline{C \to \varepsilon \mid \mathtt{0}D} \qquad \underline{D \to C\mathtt{1}}$$

2. Identify C as the only nullable non-terminal. Because S' is not nullable, remove the production rule $S' \to S$.

3. Eliminate the ε-production $C \to \varepsilon$.

$$S \to A \mid B \qquad A \to \mathtt{0}A \mid \mathtt{0}C \mid \underline{\mathtt{0}} \qquad B \to B\mathtt{1} \mid C\mathtt{1} \mid \underline{\mathtt{1}} \qquad C \to \mathtt{0}D \qquad D \to C\mathtt{1} \mid \underline{\mathtt{1}}$$

4. No two non-terminals are equivalent, so there's nothing to merge.

5. Remove the unit productions $S' \to S$, $S \to A$, and $S \to B$.

$$\underline{S \to \mathtt{0}A \mid \mathtt{0}C \mid B\mathtt{1} \mid C\mathtt{1} \mid \mathtt{0} \mid \mathtt{1}} \qquad A \to \mathtt{0}A \mid \mathtt{0}C \mid \mathtt{0} \qquad B \to B\mathtt{1} \mid C\mathtt{1} \mid \mathtt{1} \qquad C \to \mathtt{0}D \qquad D \to C\mathtt{1} \mid \mathtt{1}.$$

6. Finally, protect the terminals $\mathtt{0}$ and $\mathtt{1}$ to obtain the final CNF grammar.

$$\begin{aligned}
S &\to \underline{EA \mid EC \mid BF \mid CF \mid \mathtt{0} \mid \mathtt{1}} \\
A &\to \underline{EA \mid EC} \mid \mathtt{0} \\
B &\to \underline{BF \mid CF} \mid \mathtt{1} \\
C &\to \underline{ED} \\
D &\to \underline{CF} \mid \mathtt{1} \\
\underline{E} &\underline{\to \mathtt{0}} \\
\underline{F} &\underline{\to \mathtt{1}}
\end{aligned}$$

Exercises

1. Describe context-free grammars that generate each of the following languages. The function $\#(x, w)$ returns the number of occurrences of the ***substring*** x in the string w. For example, $\#(\texttt{0}, \texttt{101001}) = 3$ and $\#(\texttt{010}, \texttt{1010100011}) = 2$.

   (a) All strings in $\{\texttt{0}, \texttt{1}\}^*$ whose length is divisible by 5.

   (b) All strings in $\{\texttt{0}, \texttt{1}\}^*$ representing a non-negative multiple of 5 in binary.

   (c) $\{w \in \{\texttt{0}, \texttt{1}\}^* \mid \#(\texttt{0}, w) = \#(\texttt{1}, w)\}$

   (d) $\{w \in \{\texttt{0}, \texttt{1}\}^* \mid \#(\texttt{0}, w) \neq \#(\texttt{1}, w)\}$

   (e) $\{w \in \{\texttt{0}, \texttt{1}\}^* \mid \#(\texttt{00}, w) = \#(\texttt{11}, w)\}$

   (f) $\{w \in \{\texttt{0}, \texttt{1}\}^* \mid \#(\texttt{01}, w) = \#(\texttt{10}, w)\}$

   (g) $\{w \in \{\texttt{0}, \texttt{1}\}^* \mid \#(\texttt{0}, w) = \#(\texttt{1}, w)$ and $|w|$ is a multiple of $3\}$

   (h) $\{\texttt{0}, \texttt{1}\}^* \setminus \{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$

   (i) $\{\texttt{0}^n\texttt{1}^{2n} \mid n \geq 0\}$

   (j) $\{\texttt{0}, \texttt{1}\}^* \setminus \{\texttt{0}^n\texttt{1}^{2n} \mid n \geq 0\}$

   (k) $\{\texttt{0}^n\texttt{1}^m \mid 0 \leq 2m \leq n < 3m\}$

   (l) $\{\texttt{0}^i\texttt{1}^j\texttt{2}^{i+j} \mid i, j \geq 0\}$

   (m) $\{\texttt{0}^i\texttt{1}^j\texttt{2}^k \mid i = j$ or $j = k\}$

   (n) $\{\texttt{0}^i\texttt{1}^j\texttt{2}^k \mid i \neq j$ or $j \neq k\}$

   (o) $\{\texttt{0}^i\texttt{1}^j\texttt{0}^j\texttt{1}^i \mid i, j \geq 0\}$

   (p) $\left\{w\texttt{\$0}^{\#(\texttt{0},w)} \;\middle|\; w \in \{\texttt{0}, \texttt{1}\}^*\right\}$

   (q) $\{xy \mid x, y \in \{\texttt{0}, \texttt{1}\}^*$ and $x \neq y$ and $|x| = |y|\}$

   (r) $\left\{x\texttt{\$}y^R \;\middle|\; x, y \in \{\texttt{0}, \texttt{1}\}^*$ and $x \neq y\right\}$

   (s) $\{x\texttt{\$}y \mid x, y \in \{\texttt{0}, \texttt{1}\}^*$ and $\#(\texttt{0}, x) = \#(\texttt{1}, y)\}$

   (t) $\{\texttt{0}, \texttt{1}\}^* \setminus \{ww \mid w \in \{\texttt{0}, \texttt{1}\}^*\}$

   (u) All strings in $\{\texttt{0}, \texttt{1}\}^*$ that are *not* palindromes.

   (v) All strings in $\{\texttt{(}, \texttt{)}, \texttt{◇}\}^*$ in which the parentheses are balanced and the symbol ◇ appears at most four times. For example, `()(())` and `(◇◇(()()◇)()())◇` and `◇◇◇` are strings in this language, but `)(()` and `(◇◇◇)◇◇` are not.

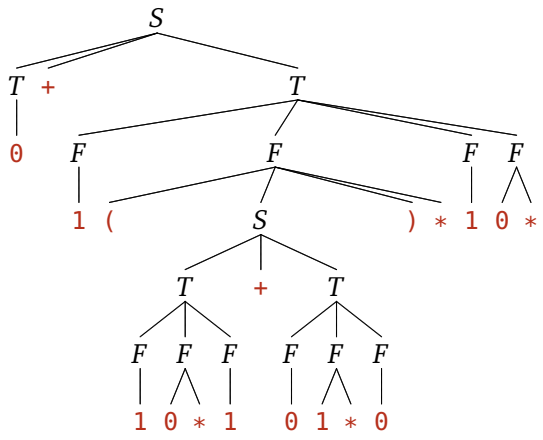
2. Prove that if L is a context-free language, then L^R is also a context-free language. *[Hint: How do you reverse a context-free grammar?]*

3. Consider a generalization of context-free grammars that allows any *regular expression* over $\Sigma \cup \Gamma$ to appear on the right side of a production rule. Without loss of generality, for each non-terminal $A \in \Gamma$, the generalized grammar contains a single regular expression $R(A)$. To apply a production rule to a string, we replace any non-terminal A with an arbitrary word in the language described by $R(A)$. As usual, the language of the generalized grammar is the set of all strings that can be derived from its start non-terminal.

For example:, the following generalized context-free grammar describes the language of all regular expressions over the alphabet {0, 1}:

$$S \to (T\texttt{+})^* T + \emptyset \qquad \text{(Regular expressions)}$$
$$T \to \varepsilon + F^* F \qquad \text{(Terms = summable expressions)}$$
$$F \to (\texttt{0} + \texttt{1} + \texttt{(}S\texttt{)})(\texttt{*} + \varepsilon) \qquad \text{(Factors = concatenable expressions)}$$

Here is a parse tree for the regular expression `0+1(10*1+01*0)*10*` (which represents the set of all binary numbers divisible by 3):



Prove that every *generalized* context-free grammar describes a context-free language. In other words, show that allowing regular expressions to appear in production rules does not increase the expressive power of context-free grammars.

Caveat lector! This is the first edition of this lecture note. A few topics are missing, and there are almost certainly a few serious errors. Please send bug reports and suggestions to jeffe@illinois.edu.

O Marvelous! what new configuration will come next?
I am bewildered with multiplicity.

— William Carlos Williams

Life only avails, not the having lived. Power ceases in the instant of repose; it resides in the moment of transition from a past to a new state, in the shooting of the gulf, in the darting to an aim.

— Ralph Waldo Emerson, "Self Reliance", *Essays, First Series* (1841)

3 Finite-State Machines

3.1 Intuition

Suppose we want to determine whether a given string $w[1..n]$ of bits represents a multiple of 5 in binary. After a bit of thought, you might realize that you can read the bits in w one at a time, from left to right, keeping track of the value modulo 5 of the prefix you have read so far.

```
MULTIPLEOF5(w[1..n]):
  rem ← 0
  for i ← 1 to n
      rem ← (2 · rem + w[i]) mod 5
  if rem = 0
      return TRUE
  else
      return FALSE
```

Aside from the loop index i, which we need just to read the entire input string, this algorithm has a single local variable *rem*, which has only four different values (0, 1, 2, 3, or 4).

This algorithm already runs in $O(n)$ time, which is the best we can hope for—after all, we have to read every bit in the input—but we can speed up the algorithm *in practice*. Let's define a ***change*** or ***transition*** function $\delta : \{0,1,2,3,4\} \times \{0,1\} \to \{0,1,2,3,4\}$ as follows:

$$\delta(q,a) = (2q + a) \bmod 5.$$

(Here I'm implicitly converting the symbols `0` and `1` to the corresponding integers 0 and 1.) Since we already know all values of the transition function, we can store them in a precomputed table, and then replace the computation in the main loop of MULTIPLEOF5 with a simple array lookup.

We can also modify the return condition to check for different values modulo 5. To be completely general, we replace the final if-then-else lines with another array lookup, using an array $A[0..4]$ of booleans describing which final mod-5 values are "acceptable".

After both of these modifications, our algorithm can be rewritten as follows, either iteratively or recursively (with $q = 0$ in the initial call):

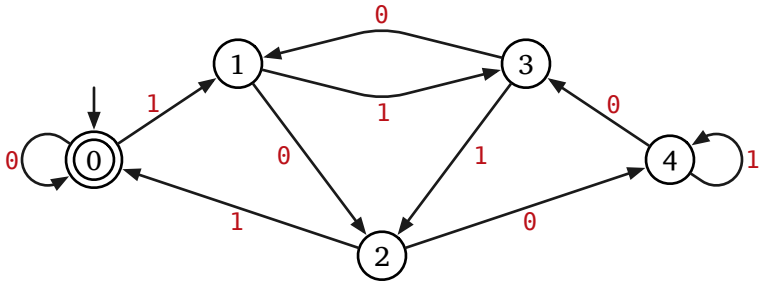
```
DoSomethingCool(w[1..n]):
  q ← 0
  for i ← 1 to n
      q ← δ[q, w[i]]
  return A[q]
```

```
DoSomethingCool(q, w):
  if w = ε
      return A[q]
  else
      decompose w = a · x
      return DoSomethingCool(δ(q, a), x)
```

If we want to use our new DoSomethingCool algorithm to implement MultipleOf5, we simply give the arrays δ and A the following hard-coded values:

q	$\delta[q,0]$	$\delta[q,1]$	$A[q]$
0	0	1	True
1	2	3	False
2	4	0	False
3	1	2	False
4	3	4	False

We can also visualize the behavior of DoSomethingCool by drawing a directed graph, whose vertices represent possible values of the variable q—the possible ***states*** of the algorithm—and whose edges are labeled with input symbols to represent transitions between states. Specifically, the graph includes the labeled directed edge $p \xrightarrow{a} q$ if and only if $\delta(p,a) = q$. To indicate the proper return value, we draw the "acceptable" final states using doubled circles. Here is the resulting graph for MultipleOf5:



State-transition graph for MultipleOf5

If we run the MultipleOf5 algorithm on the string `00101110110` (representing the number 374 in binary), the algorithm performs the following sequence of transitions:

$$0 \xrightarrow{0} 0 \xrightarrow{0} 0 \xrightarrow{1} 1 \xrightarrow{0} 2 \xrightarrow{1} 0 \xrightarrow{1} 1 \xrightarrow{1} 3 \xrightarrow{0} 1 \xrightarrow{1} 3 \xrightarrow{1} 2 \xrightarrow{0} 4$$

Because the final state is not the "acceptable" state 0, the algorithm correctly returns False. We can also think of this sequence of transitions as a walk in the graph, which is completely determined by the start state 0 and the sequence of edge labels; the algorithm returns True if and only if this walk ends at an "acceptable" state.

3.2 Formal Definitions

The object we have just described is an example of a ***finite-state machine***. A finite-state machine is a formal model of any system/machine/algorithm that can exist in a finite number of ***states*** and that transitions among those states based on sequence of ***input*** symbols.

Finite-state machines are also commonly called ***deterministic finite-state automata***, abbreviated ***DFAs***. The word "deterministic" means that the behavior of the machine is completely *determined* by

the input string; we'll discuss nondeterministic automata in the next lecture. The word "automaton" (plural "automata") comes from ancient Greek αὐτόματος meaning "self-acting", from the roots αὐτό- ("self") and -ματος ("thinking, willing", the root of Latin *mentus*).

Formally, every finite-state machine consists of five components:

- An arbitrary finite set Σ, called the ***input alphabet***.
- Another arbitrary finite set Q, whose elements are called ***states***.
- An arbitrary ***transition*** function $\delta : Q \times \Sigma \to Q$.
- A ***start state*** $s \in Q$.
- A subset $A \subseteq Q$ of ***accepting states***.

The behavior of a finite-state machine is governed by an ***input string*** w, which is a finite sequence of symbols from the input alphabet Σ. The machine ***reads*** the symbols in w one at a time in order (from left to right). At all times, the machine has a *current state* q; initially q is the machine's start state s. Each time the machine reads a symbol a from the input string, its current state ***transitions*** from q to $\delta(q, a)$. After all the characters have been read, the machine ***accepts*** w if the current state is in A and ***rejects*** w otherwise. In other words, every finite state machine runs the algorithm DoSomethingCool! The ***language*** of a finite state machine M, denoted $L(M)$ is the set of all strings that M accepts.

More formally, we extend the transition function $\delta : Q \times \Sigma^* \to Q$ of any finite-state machine to a function $\delta^* : Q \times \Sigma^* \to Q$ that transitions on *strings* as follows:

$$\delta^*(q, w) := \begin{cases} q & \text{if } w = \varepsilon, \\ \delta^*(\delta(q, a), x) & \text{if } w = ax. \end{cases}$$

Finally, a finite-state machine ***accepts*** a string w if and only if $\delta^*(s, w) \in A$, and ***rejects*** w otherwise. (Compare this definition with the recursive formulation of DoSomethingCool!)

For example, our final MultipleOf5 algorithm is a DFA with the following components:

- input alphabet: $\Sigma = \{0, 1\}$
- state set: $Q = \{0, 1, 2, 3, 4\}$
- transition function: $\delta(q, a) = (2q + a) \bmod 5$
- start state: $s = 0$
- accepting states: $A = \{0\}$

This machine rejects the string 00101110110, because

$$\begin{aligned} \delta^*(0, 00101110110) &= \delta^*(\delta(0, 0), 0101110110) \\ &= \delta^*(0, 0101110110) = \delta^*(\delta(0, 0), 101110110) \\ &= \delta^*(0, 101110110) = \delta^*(\delta(0, 1), 01110110) \\ &= \delta^*(1, 01110110) = \delta^*(\delta(1, 0), 1110110) = \cdots \\ &\vdots \\ \cdots &= \delta^*(1, 110) = \delta^*(\delta(1, 1), 10) \\ &= \delta^*(3, 10) = \delta^*(\delta(3, 1), 0) \\ &= \delta^*(2, 0) = \delta^*(\delta(3, 0), \varepsilon) \\ &= \delta^*(4, \varepsilon) = 4 \notin A. \end{aligned}$$

We have already seen a more graphical representation of this entire sequence of transitions:

$$0 \xrightarrow{0} 0 \xrightarrow{0} 0 \xrightarrow{1} 1 \xrightarrow{0} 2 \xrightarrow{1} 0 \xrightarrow{1} 1 \xrightarrow{1} 3 \xrightarrow{0} 1 \xrightarrow{1} 3 \xrightarrow{1} 2 \xrightarrow{0} 4$$

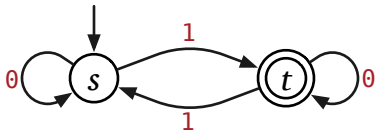
The arrow notation is easier to read and write for specific examples, but surprisingly, most people actually find the more formal functional notation easier to use in formal proofs. Try them both!

We can equivalently define a DFA as a directed graph whose vertices are the states Q, whose edges are labeled with symbols from Σ, such that every vertex has exactly one outgoing edge with each label. In our drawings of finite state machines, the start state s is always indicated by an incoming arrow, and the accepting states A are always indicted by doubled circles. By induction, for any string $w \in \Sigma^*$, this graph contains a unique walk that starts at s and whose edges are labeled with the symbols in w in order. The machine accepts w if this walk ends at an accepting state. This graphical formulation of DFAs is incredibly useful for developing intuition and even designing DFAs. For proofs, it's largely a matter of taste whether to write in terms of extended transition functions or labeled graphs, but (as much as I wish otherwise) I actually find it easier to write ***correct*** proofs using the functional formulation.

3.3 Another Example

The following drawing shows a finite-state machine with input alphabet $\Sigma = \{0, 1\}$, state set $Q = \{s, t\}$, start state s, a single accepting state t, and the transition function

$$\delta(s, 0) = s, \quad \delta(s, 1) = t, \quad \delta(t, 0) = t, \quad \delta(t, 1) = s.$$



A simple finite-state machine.

For example, the two-state machine M at the top of this page accepts the string 00101110100 after the following sequence of transitions:

$$s \xrightarrow{0} s \xrightarrow{0} s \xrightarrow{1} t \xrightarrow{0} t \xrightarrow{1} s \xrightarrow{1} t \xrightarrow{1} s \xrightarrow{0} s \xrightarrow{1} t \xrightarrow{0} t \xrightarrow{0} t.$$

The same machine M rejects the string 11100101 after the following sequence of transitions:

$$s \xrightarrow{1} t \xrightarrow{1} s \xrightarrow{1} t \xrightarrow{0} t \xrightarrow{0} s \xrightarrow{1} t \xrightarrow{0} t \xrightarrow{1} s.$$

Finally, M rejects the empty string, because the start state s is not an accepting state.

From these examples and others, it is easy to conjecture that the language of M is the set of all strings of 0s and 1s with an odd number of 1s. So let's prove it!

Proof (tedious case analysis): Let $\#(a, w)$ denote the number of times symbol a appears in string w. We will prove the following stronger claims, for any string w.

$$\delta^*(s, w) = \begin{cases} s & \text{if } \#(1, w) \text{ is even} \\ t & \text{if } \#(1, w) \text{ is odd} \end{cases} \qquad \text{and} \qquad \delta^*(t, w) = \begin{cases} t & \text{if } \#(1, w) \text{ is even} \\ s & \text{if } \#(1, w) \text{ is odd} \end{cases}$$

Let w be an arbitrary string. Assume that for any string x that is shorter than w, we have $\delta^*(s, x) = s$ and $\delta^*(t, x) = t$ if x has an even number of 1s, and $\delta^*(s, x) = t$ and $\delta^*(t, x) = s$ if x has an odd number of 1s. There are five cases to consider.

- If $w = \varepsilon$, then w contains an even number of 1s and $\delta^*(s, w) = s$ and $\delta^*(t, w) = t$ by definition.
- Suppose $w = 1x$ and $\#(1, w)$ is even. Then $\#(1, x)$ is odd, which implies

$$\begin{aligned}
\delta^*(s, w) &= \delta^*(\delta(s, 1), x) && \text{by definition of } \delta^* \\
&= \delta^*(t, x) && \text{by definition of } \delta \\
&= s && \text{by the inductive hypothesis} \\
\delta^*(t, w) &= \delta^*(\delta(t, 1), x) && \text{by definition of } \delta^* \\
&= \delta^*(s, x) && \text{by definition of } \delta \\
&= T && \text{by the inductive hypothesis}
\end{aligned}$$

  Since the remaining cases are similar, I'll omit the line-by-line justification.
- If $w = 1x$ and $\#(1, w)$ is odd, then $\#(1, x)$ is even, so the inductive hypothesis implies

$$\begin{aligned}
\delta^*(s, w) &= \delta^*(\delta(s, 1), x) = \delta^*(t, x) = t \\
\delta^*(t, w) &= \delta^*(\delta(t, 1), x) = \delta^*(s, x) = s
\end{aligned}$$

- If $w = 0x$ and $\#(1, w)$ is even, then $\#(1, x)$ is even, so the inductive hypothesis implies

$$\begin{aligned}
\delta^*(s, w) &= \delta^*(\delta(s, 0), x) = \delta^*(s, x) = s \\
\delta^*(t, w) &= \delta^*(\delta(t, 0), x) = \delta^*(t, x) = t
\end{aligned}$$

- Finally, if $w = 0x$ and $\#(1, w)$ is odd, then $\#(1, x)$ is odd, so the inductive hypothesis implies

$$\begin{aligned}
\delta^*(s, w) &= \delta^*(\delta(s, 0), x) = \delta^*(s, x) = t \\
\delta^*(t, w) &= \delta^*(\delta(t, 0), x) = \delta^*(t, x) = s
\end{aligned}$$

□

Notice that this proof contains $|Q|^2 \cdot |\Sigma| + |Q|$ separate inductive arguments. For every pair of states p and q, we must argue about the language so strings w such that $\delta^*(p, w) = q$, and we must consider each first symbol in w. We must also argue about $\delta(p, \varepsilon)$ for every state p. Each of those arguments is typically straightforward, but it's easy to get lost in the deluge of cases.

For this particular proof, however, we can reduce the number of cases by switching from tail recursion to *head* recursion. The following identity holds for all strings $x \in \Sigma^*$ and symbols $a \in \Sigma$:

$$\boxed{\delta^*(q, xa) = \delta(\delta^*(q, x), a)}$$

We leave the inductive proof of this identity as a straightforward exercise (hint, hint).

Proof (clever renaming, head induction): Let's rename the states 0 and 1 instead of s and t. Then the transition function can be described concisely as $\boldsymbol{\delta(q, a) = (q + a) \bmod 2}$.

Now we claim that for every string w, we have $\delta^*(0, w) = \#(1, w) \bmod 2$. So let w be an arbitrary string, and assume that for any string x that is shorter than w that $\delta^*(0, x) = \#(1, x) \bmod 2$. There are only two cases to consider: either w is empty or it isn't.

- If $w = \varepsilon$, then $\delta^*(0, w) = 0 = \#(1, w) \bmod 2$ by definition.

- Otherwise, $w = xa$ for some string x and some symbol a, and we have

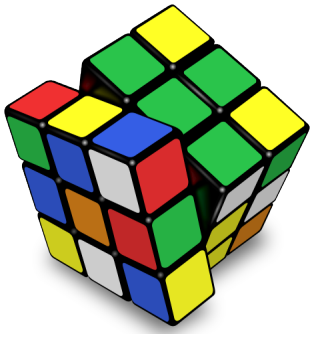
$$\begin{aligned}
\delta^*(0, w) &= \delta(\delta^*(0, x), a) \\
&= \delta(\#(1, x) \bmod 2, a) && \text{by the inductive hypothesis} \\
&= (\#(1, x) \bmod 2 + a) \bmod 2 && \text{by definition of } \delta \\
&= (\#(1, x) + a) \bmod 2 && \text{by definition of mod 2} \\
&= (\#(1, x) + \#(1, a)) \bmod 2 && \text{because } \#(1, 0) = 0 \text{ and } \#(1, 1) = 1 \\
&= (\#(1, xa)) \bmod 2 && \text{by definition of } \# \\
&= (\#(1, w)) \bmod 2 && \text{because } w = xa
\end{aligned}$$

□

Hmmm. This "clever" proof is certainly shorter than the earlier brute-force proof, but is it really "better"? "Simpler"? More intuitive? Easier to understand? I'm skeptical. Sometimes brute force really is more effective.

3.4 Yet Another Example

As a more complex example, consider the ***Rubik's cube***, a well-known mechanical puzzle invented independently by Ernő Rubik in Hungary and Terutoshi Ishigi in Japan in the mid-1970s. This puzzle has precisely 519,024,039,293,878,272,000 distinct configurations. In the unique *solved* configuration, each of the six faces of the cube shows exactly one color. We can change the configuration of the cube by rotating one of the six faces of the cube by 90 degrees, either clockwise or counterclockwise. The cube has six faces (front, back, left, right, up, and down), so there are exactly twelve possible turns, typically represented by the symbols $\mathtt{R}, \mathtt{L}, \mathtt{F}, \mathtt{B}, \mathtt{U}, \mathtt{D}, \bar{\mathtt{R}}, \bar{\mathtt{L}}, \bar{\mathtt{F}}, \bar{\mathtt{B}}, \bar{\mathtt{U}}, \bar{\mathtt{D}}$, where the letter indicates which face to turn and the presence or absence of a bar over the letter indicates turning counterclockwise or clockwise, respectively. Thus, we can represent a Rubik's cube as a finite-state machine with 519,024,039,293,878,272,000 states and an input alphabet with 12 symbols; or equivalently, as a directed graph with 519,024,039,293,878,272,000 vertices, each with 12 outgoing edges. In practice, the number of states is *far* too large for us to actually draw the machine or explicitly specify its transition function; nevertheless, the number of states is still finite. If we let the start state s and the sole accepting state be the solved state, then the language of this finite state machine is the set of all move sequences that leave the cube unchanged.



A complicated finite-state machine.

3.5 Building DFAs

This section describes a few examples of building DFAs that accept particular languages, thereby proving that those languages are automatic. As usual in algorithm design, there is no purely mechanical recipe—no *automatic* method—no *algorithm*—for building DFAs in general. However, the following examples show several useful design strategies.

3.5.1 Superstrings

Perhaps the simplest rule of thumb is to try to construct an algorithm that looks like MULTIPLEOF5: A simple for-loop through the symbols, using a *constant* number of variables, where each variable (except the loop index) has only a *constant* number of possible values. Here, "constant" means an actual number that is not a function of the input size n. You should be able to compute the number of possible values for each variable *at compile time*.

For example, the following algorithm determines whether a given string in $\Sigma = \{0, 1\}$ contains the substring 11.

```
CONTAINS11(w[1..n]):
  found ← FALSE
  for i ← 1 to n
      if i = 1
          last2 ← w[1]
      else
          last2 ← w[1] · w[2]
      if last = 11
          found ← TRUE
  return found
```

Aside from the loop index, this algorithm has exactly two variables.

- A boolean flag *found* indicating whether we have seen the substring 11. This variable has exactly two possible values: TRUE and FALSE.
- A string *last2* containing the last (up to) three symbols we have read so far. This variable has exactly 7 possible values: ε, 0, 1, 00, 01, 10, and 11.

Thus, altogether, the algorithm can be in at most $2 \times 7 = 14$ possible states, one for each possible pair (*found*, *last2*). Thus, we can encode the behavior of CONTAINS11 as a DFA with fourteen states, where the start state is (FALSE, ε) and the accepting states are all seven states of the form (TRUE, $*$). The transition function is described in the following table (split into two parts to save space):

q	$\delta[q, 0]$	$\delta[q, 1]$
(FALSE, ε)	(FALSE, 0)	(FALSE, 1)
(FALSE, 0)	(FALSE, 00)	(FALSE, 01)
(FALSE, 1)	(FALSE, 10)	(**TRUE**, 11)
(FALSE, 00)	(FALSE, 00)	(FALSE, 01)
(FALSE, 01)	(FALSE, 10)	(**TRUE**, 11)
(FALSE, 10)	(FALSE, 00)	(FALSE, 01)
(FALSE, 11)	(FALSE, 10)	(**TRUE**, 11)

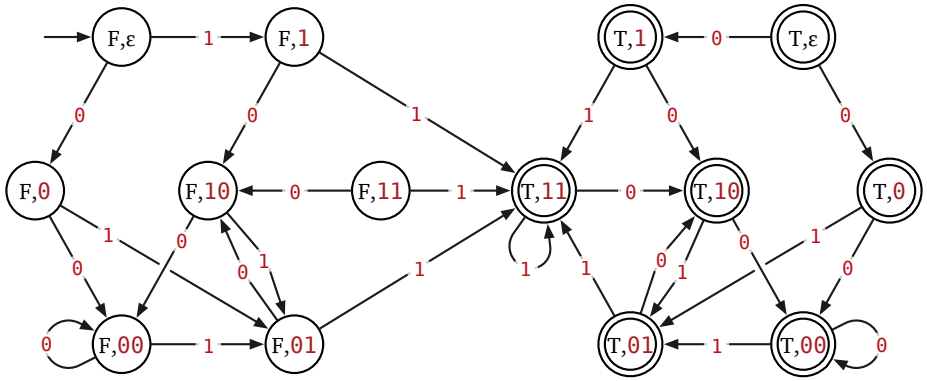
q	$\delta[q, 0]$	$\delta[q, 1]$
(TRUE, ε)	(TRUE, 0)	(TRUE, 1)
(TRUE, 0)	(TRUE, 00)	(TRUE, 01)
(TRUE, 1)	(TRUE, 10)	(TRUE, 11)
(TRUE, 00)	(TRUE, 00)	(TRUE, 01)
(TRUE, 01)	(TRUE, 10)	(TRUE, 11)
(TRUE, 10)	(TRUE, 00)	(TRUE, 01)
(TRUE, 11)	(TRUE, 10)	(TRUE, 11)

For example, given the input string 10010$\underline{11}$100, this DFA performs the following sequence of transitions and then accepts.

$$(\text{FALSE}, \varepsilon) \xrightarrow{1} (\text{FALSE}, 1) \xrightarrow{0} (\text{FALSE}, 10) \xrightarrow{0} (\text{FALSE}, 00) \xrightarrow{1} (\text{FALSE}, 01) \xrightarrow{0} (\text{FALSE}, 10) \xrightarrow{1} (\text{FALSE}, 01) \xrightarrow{1} (\text{TRUE}, 11) \xrightarrow{1} (\text{TRUE}, 11) \xrightarrow{0} (\text{TRUE}, 10) \xrightarrow{0} (\text{TRUE}, 00)$$

3.5.2 Reducing states

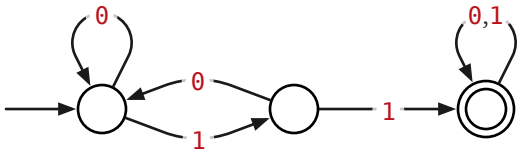
You can probably guess that the brute-force DFA we just constructed has considerably more states than necessary, especially after seeing its transition graph:



Our brute-force DFA for strings containing the substring 11

For example, we don't need actually to remember both of the last two symbols, but only the penultimate symbol, because the last symbol is the one we're currently reading. This observation allows us to reduce the number of states from fourteen to only six. Once the flag part of the state is set to TRUE, we know the machine will eventually accept, so we might as well merge the two accepting states together. Finally, and more subtly, because all transitions out of (FALSE, ε) and (FALSE, 0) are identical, we can merge those two states together as well. In the end, we obtain the following DFA with just three states:

- The start state, which indicates that the machine has not read the substring 11 an did not just read the symbol 1.
- An intermediate state, which indicates that the machine has not read the substring 11 but just read the symbol 1.
- A unique accept state, which indicates that the machine has read the substring 11.



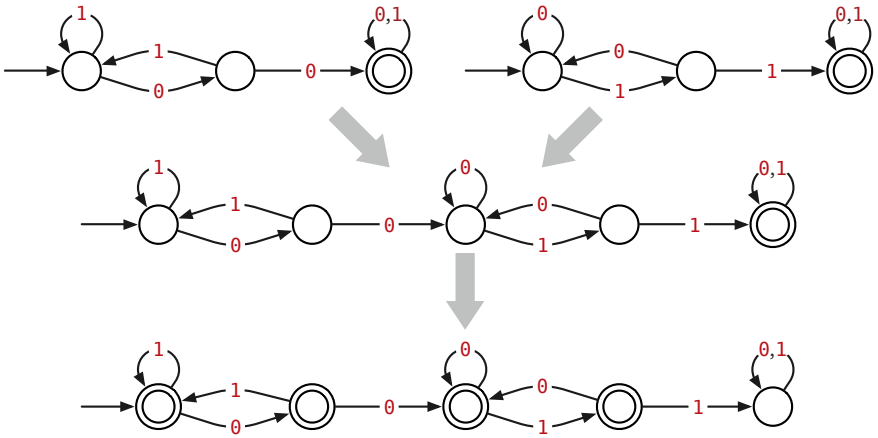
A minimal DFA for superstrings of 11

At the end of this note, I'll describe an efficient algorithm to transform any given DFA into an equivalent DFA with the fewest possible states. Given that this minimization algorithm exists, there is very little incentive to optimize DFAs *by hand*. Clarity is infinitely more important than brevity, especially in this class.

3.5.3 Every this after that

Suppose we want to accept the set of strings in which every occurrence of the substring 00 occurs after every occurrence of the substring 11. Equivalently, we want to *reject* every string in which some 00 occurs before 11. Often the easiest way to design a DFA to check whether a string is *not* in some set is first to build a DFA that *is* in that set and then invert which states in that machine are accepting.

From the previous example, we know that there is a three-state DFA M_{11} that accepts the set of strings with the substring `11` and a nearly identical DFA M_{00} that accepts the set of strings containing the substring `00`. By identifying the accept state of M_{00} with the start state of M_{11}, we obtain a five-state DFA that accepts the set of strings with `00` before `11`. Finally, by inverting which states are accepting, we obtain the DFA we want.



Building a DFA for the language of strings in which every `00` is after every `11`.

3.5.4 Both This and That: The Product Construction

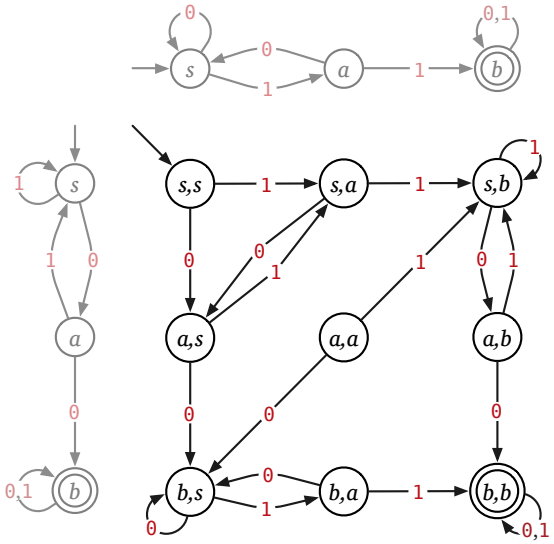
Now suppose we want to accept all strings that contain both `00` and `11` as substrings, in either order. Intuitively, we'd like to run two of our earlier DFAs in parallel—the DFA M_{00} to detect superstrings of `00` and the DFA M_{11} to detect superstrings of `11`—and then accept the input string if and only if both of these DFAs accept. In fact, we can encode precisely this "parallel computation" into a single DFA, whose states are all ordered pairs (p,q), where p is a state in M_{00} and q is a state in M_{11}. The new "parallel" DFA includes the transition $(p,q) \xrightarrow{a} (p',q')$ if and only if M_{00} contains the transition $p \xrightarrow{a} p'$ and M_{11} contains the transition $q \xrightarrow{a} q'$. Finally, the state (p,q) is accepting if and only if p and q are accepting states in their respective machines. The resulting nine-state DFA is shown on the next page.

More generally, let $M_1 = (\Sigma, Q_1, \delta_1, s_1, A_1)$ be an arbitrary DFA that accepts some language L_1, and let $M_2 = (\Sigma, Q_2, \delta_2, s_2, A_2)$ be an arbitrary DFA that accepts some language L_2 (over the same alphabet Σ). We can construct a third DFA $M = (\Sigma, Q, \delta, s, A)$ that accepts the intersection language $L_1 \cap L_2$ as follows.

$$\begin{aligned} Q &:= Q_1 \times Q_2 = \{(p,q) \mid p \in Q_1 \text{ and } q \in Q_2\} \\ s &:= (s_1, s_2) \\ A &:= A_1 \times A_2 = \{(p,q) \mid p \in A_1 \text{ and } q \in A_2\} \\ \delta((p,q),a) &:= \big(\delta_1(p,a),\ \delta_2(q,a)\big) \end{aligned}$$

To convince yourself that this product construction is actually correct, consider the extended transition function $\delta^* : (Q \times Q') \times \Sigma^* \to (Q \times Q')$, which acts on strings instead of individual symbols. Recall that this function is defined recursively as follows:

$$\delta^*\big((p,q),w\big) := \begin{cases} q & \text{if } w = \varepsilon, \\ \delta^*\big(\delta((p,q),a),\ x\big) & \text{if } w = ax. \end{cases}$$



Building a DFA for the language of strings in which every 00 is after every 11.

Inductive definition-chasing gives us the identity $\delta^*((p,q),w) = \big(\delta_1^*(p,w),\ \delta_2^*(q,w)\big)$ for any string w:

$$\begin{aligned}
\delta^*\big((p,q),\varepsilon\big) &= (p,q) && \text{by the definition of } \delta^* \\
&= \big(\delta_1^*(p,\varepsilon),\ \delta_2^*(q,\varepsilon)\big) && \text{by the definitions of } \delta_1^* \text{ and } \delta_2^*; \\
\delta^*\big((p,q),ax\big) &= \delta^*\big(\delta((p,q),a),\ x\big) && \text{by the definition of } \delta^* \\
&= \delta^*\big((\delta_1(p,a),\ \delta_2(q,a)),\ x\big) && \text{by the definition of } \delta \\
&= \big(\delta_1^*((\delta_1(p,a),\ x),\ \delta_2^*(\delta_2(q,a),x)\big) && \text{by the induction hypothesis} \\
&= \big(\delta_1^*(p,ax),\ \delta_2^*(q,ax)\big) && \text{by the definitions of } \delta_1^* \text{ and } \delta_2^*.
\end{aligned}$$

It now follows from this seemingly impenetrable wall of notation that for any string w, we have $\delta^*(s,w) \in A$ if and only if both $\delta_1^*(s_1,w) \in A_1$ and $\delta_2^*(s_2,w) \in A_2$. In other words, M accepts w if and only if both M_1 and M_2 accept w, as required.

As usual, this construction technique does not necessarily yield *minimal* DFAs. For example, in our first example of a product DFA, illustrated above, the central state (a,a) cannot be reached by any other state and is therefore redundant. Whatever.

Similar product constructions can be used to build DFAs that accept any other boolean combination of languages; in fact, the only part of the construction that needs to be changed is the choice of accepting states. For example:

- To accept the union $L_1 \cup L_2$, define $A = \big\{(p,q) \;\big|\; p \in A_1 \textbf{ or } q \in A_2\big\}$.
- To accept the difference $L_1 \setminus L_2$, define $A = \big\{(p,q) \;\big|\; p \in A_1 \textbf{ but not } q \notin A_2\big\}$.
- To accept the symmetric difference $L_1 \oplus L_2$, define $A = \big\{(p,q) \;\big|\; p \in A_1 \textbf{ xor } q \in A_2\big\}$.

Moreover, by cascading this product construction, we can construct DFAs that accept arbitrary boolean combinations of arbitrary finite collections of regular languages.

3.6 Decision Algorithms

It's unclear how much we can say here, since we haven't yet talked about graph algorithms, or even really about graphs. Perhaps this discussion should simply be moved to the graph-traversal notes.

- ***Is $w \in L(M)$?*** Follow the unique path from q_0 with label w. By definition, $w \in L(M)$ if and only if this path leads to an accepting state.
- ***Is $L(M)$ empty?*** The language $L(M)$ is empty if and only if no accepting state is reachable from q_0. This condition can be checked in $O(n)$ time via whatever-first search, where n is the number of states. Alternatively, but less usefully, $L(M) = \emptyset$ if and only if $L(M)$ contains no string w such that $|w| < n$.

★★★

- ***Is $L(M)$ finite?*** Remove all states unreachable from q_0 (via whatever first search). Then $L(M)$ is finite if and only if the reduced DFA is a dag; this condition can be checked by depth-first search. Alternatively, but less usefully, $L(M)$ is finite if and only if $L(M)$ contains no string w such that $n \le |w| < 2n$.
- ***Is $L(M) = \Sigma^*$?*** Remove all states unreachable from q_0 (via whatever first search). Then $L(M) = \Sigma^*$ if and only if every state in M is an accepting state.
- ***Is $L(M) = L(M')$?*** Build a DFA N such that $L(N) = L(M) \setminus L(M')$ using a standard product construction, and then check whether $L(N) = \emptyset$.

3.7 Closure Properties

We haven't yet proved that automatic languages are regular yet, so formally, for now, some of these are closure properties of ***automatic*** languages.

- Complement (easy for DFAs, hard for regular expressions.)
- Concatenation (trivial for regular expressions, hard for DFAs)
- Union (trivial for regular expressions, easy for DFAs via product)
- Intersection (hard for regular expressions, easy for DFAs via product)
- Difference (hard for regular expressions, easy for DFAs via product)
- Kleene star: wait for NFAs (trivial for regular expression, hard for DFAs)
- Homomorphism: only mention in passing
- Inverse homomorphism: only mention in passing

★★★

3.8 Fooling Sets

Fix an arbitrary language L over an arbitrary alphabet Σ. For any strings $x, y, z \in \Sigma^*$, we say that ***z distinguishes x from y*** if exactly one of the strings xz and yz is in L. If no string distinguishes x and y, we say that x and y are ***L-equivalent*** and write $x \equiv_L y$. Thus,

$$x \equiv_L y \iff \text{For every string } z \in \Sigma^*, \text{ we have } xz \in L \text{ if and only if } yz \in L.$$

For example, let L_{eo} denote the language of strings over $\{0, 1\}$ with an even number of 0s and an odd number of 1s. Then the strings $x = 01$ and $y = 0011$ are distinguished by the string $z = 100$, because

$$xz = 01 \bullet 100 = 01100 \in L_{eo}$$
$$yz = 0011 \bullet 100 = 0011100 \notin L_{eo}.$$

On the other hand, it is quite easy to prove (hint, hint) that the strings 0001 and 1011 are L_{eo}-equivalent.

Let M be an arbitrary DFA for an arbitrary language L, and let x be y be arbitrary strings. If x and y lead to the same state in M—that is, if $\delta^*(s,x) = \delta^*(s,y)$—then we have

$$\delta^*(s,xz) = \delta^*(\delta^*(s,x),z) = \delta^*(\delta^*(s,y),z) = \delta^*(s,yz)$$

for any string z. In particular, either M accepts both x and y, or M rejects both x and y, and therefore $x \equiv_L y$. It follows that if x and y are not L-equivalent, then ***any*** DFA that accepts L has at least two distinct states $\delta^*(s,x) \neq \delta^*(s,y)$.

Finally, a ***fooling set*** for L is a set F of strings such that *every* pair of strings in F has a distinguishing suffix. For example, $F = \{\texttt{01}, \texttt{101}, \texttt{010}, \texttt{1010}\}$ is a fooling set for the language L_{eo} of strings with an even number of `0`s and an odd number of `1`s, because each pair of strings in F has a distinguishing suffix:

- `0` distinguishes `01` and `101`;
- `0` distinguishes `01` and `010`;
- `0` distinguishes `01` and `1010`;
- `10` distinguishes `101` and `010`;
- `1` distinguishes `101` and `1010`;
- `1` distinguishes `010` and `1010`.

The pigeonhole principle now implies that for any integer k, if language L is accepted by a DFA with k states, then *every* fooling set for L contains at most k strings. This simple observation has two immediate corollaries.

First, for any integer k, if L has a fooling set of size k, then *every* DFA that accepts L has at least k states. For example, the fooling set $\{\texttt{01}, \texttt{101}, \texttt{010}, \texttt{1010}\}$ proves that any DFA for L_{eo} has at least four states. Thus, we can use fooling sets to prove that certain DFAs are as small as possible.

Second, and more interestingly, if a language L is accepted by *any* DFA, then *every* fooling set for L must be finite. Equivalently: ***If L has an infinite fooling set, then L is not accepted by any DFA.*** This is arguably both the simplest and most powerful method for proving that a language is non-regular. Here are a few canonical examples of the fooling-set technique in action.

Lemma 3.1. *The language $L = \{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$ is not regular.*

Proof: Consider the set $F = \{\texttt{0}^n \mid n \geq 0\}$, or more simply $F = \texttt{0}^*$. Let x and y be arbitrary distinct strings in F. Then we must have $x = \texttt{0}^i$ and $y = \texttt{0}^j$ for some integers $i \neq j$. The suffix $z = \texttt{1}^i$ distinguishes x and y, because $xz = \texttt{0}^i\texttt{1}^i \in L$, but $yz = \texttt{0}^i\texttt{1}^j \notin L$. We conclude that F is a fooling set for L. Because F is infinite, L cannot be regular. □

Lemma 3.2. *The language $L = \{ww^R \mid w \in \Sigma^*\}$ of even-length palindromes is not regular.*

Proof: Let x and y be arbitrary distinct strings in $\texttt{0}^*\texttt{1}$. Then we must have $x = \texttt{0}^i\texttt{1}$ and $y = \texttt{0}^j\texttt{1}$ for some integers $i \neq j$. The suffix $z = \texttt{10}^i$ distinguishes x and y, because $xz = \texttt{0}^i\texttt{110}^i \in L$, but $yz = \texttt{0}^i\texttt{110}^j \notin L$. We conclude that $\texttt{0}^*\texttt{1}$ is a fooling set for L. Because $\texttt{0}^*\texttt{1}$ is infinite, L cannot be regular. □

Lemma 3.3. *The language $L = \{\texttt{0}^{2^n} \mid n \geq 0\}$ is not regular.*

Proof: Let x and y be arbitrary distinct strings in L. Then we must have $x = \texttt{0}^{2^i}$ and $y = \texttt{0}^{2^j}$ for some integers $i \neq j$. The suffix $z = \texttt{0}^{2^i}$ distinguishes x and y, because $xz = \texttt{0}^{2^i+2^i} = \texttt{0}^{2^{i+1}} \in L$, but $yz = \texttt{0}^{2^i+2^j} \notin L$. We conclude that L itself is a fooling set for L. Because L is infinite, L cannot be regular. □

Lemma 3.4. *The language $L = \{0^p \mid p \text{ is prime}\}$ is not regular.*

Proof: Again, we use 0^* as our fooling set, but but the actual argument is somewhat more complicated than in our earlier examples.

Let x and y be arbitrary distinct strings in 0^*. Then we must have $x = 0^i$ and $y = 0^j$ for some integers $i \neq j$. Without loss of generality, assume that $i < j$. Let p be any prime number larger than i. Because $p + 0(j-i)$ is prime and $p + p(j-i) > p$ is not, there must be a positive integer $k \leq p$ such that $p + (k-1)(j-i)$ is prime but $p + k(j-i)$ is not. Then the suffix $0^{p+(k-1)j-ki}$ distinguishes x and y:

$$xz = 0^i\, 0^{p+(k-1)j-ki} = 0^{p+(k-1)(j-i)} \in L \qquad \text{because } p + (k-1)(j-i) \text{ is prime;}$$

$$yz = 0^j\, 0^{p+(k-1)j-ki} = 0^{p+k(j-i)} \notin L \qquad \text{because } p + k(j-i) \text{ is not prime.}$$

(Because $i < j$ and $i < p$, the suffix $0^{p+(k-1)j-ki} = 0^{(p-i)+(k-1)(j-i)}$ has positive length and therefore *actually exists!*) We conclude that 0^* is indeed a fooling set for L, which implies that L is not regular. □

One natural question that many students ask is "How did you come up with that fooling set?" Perhaps the simplest rule of thumb is that for most languages L—in particular, for almost all languages that students are asked to prove non-regular on homeworks or exams—either some simple regular language like 0^* or 10^*1 is a fooling set, or the language L itself is a fooling set. (Of course, there are well-engineered counterexamples.)

*3.9 The Myhill-Nerode Theorem

The fooling set technique implies a *necessary* condition for a language to be accepted by a DFA—the language must have no infinite fooling sets. In fact, this condition is also *sufficient*. The following powerful theorem was first proved by Anil Nerode in 1958, strengthening a 1957 result of John Myhill.[1]

The Myhill-Nerode Theorem. *For any language L, the following are equal:*
(a) the minimum number of states in a DFA that accepts L,
(b) the maximum size of a fooling set for L, and
(c) the number of equivalence classes of $\equiv_L$.
In particular, L is accepted by a DFA if and only if every fooling set for L is finite.

Proof: Let L be an arbitrary language.

We have already proved that the size of any fooling set for L is at most the number of states in any DFA that accepts L, so (a)≤(b). It also follows directly from the definitions that $F \subseteq \Sigma^*$ is a fooling set for L if and only if F contains at most one string in each equivalence class of $\equiv_L$; thus, (b)=(c). We complete the proof by showing that (a)≥(c).

We have already proved that if $\equiv_L$ has an infinite number of equivalence classes, there is no DFA that accepts L, so assume that the number of equivalence classes is finite. For any string w, let $[w]$ denote its equivalence class. We define a DFA $M_{\equiv} = (\Sigma, Q, s, A, \delta)$ as follows:

$$\begin{aligned} Q &:= \{[w] \mid w \in \Sigma^*\} \\ s &:= [\varepsilon] \\ A &:= \{[w] \mid w \in L\} \\ \delta([w], a) &:= [w \bullet a] \end{aligned}$$

[1] Myhill considered the finer equivalence relation $x \sim_L y$, meaning $wxz \in L$ if and only if $wyz \in L$ for all strings w and z, and proved that L is regular if and only if $\sim_L$ defines a finite number of equivalence classes. Like most of Myhill's early automata research, this result appears in an unpublished Air Force technical report. The modern Myhill-Nerode theorem appears (in an even more general form) as a minor lemma in Nerode's 1958 paper, which (not surprisingly) does not cite Myhill.

We claim that this DFA accepts the language L; this claim completes the proof of the theorem.

But before we can prove anything about this DFA, we first need to verify that it is actually well-defined. Let x and y be two strings such that $[x] = [y]$. By definition of L-equivalence, for any string z, we have $xz \in L$ if and only if $yz \in L$. It immediately follows that for any symbol $a \in \Sigma$ and any string z', we have $xaz' \in L$ if and only if $yaz' \in L$. Thus, by definition of L-equivalence, we have $[xa] = [ya]$ for every symbol $a \in \Sigma$. We conclude that the function δ is indeed well-defined.

An easy inductive proof implies that $\delta^*([\varepsilon], x) = [x]$ for every string x. Thus, M accepts string x if and only if $[x] = [w]$ for some string $w \in L$. But if $[x] = [w]$, then by definition (setting $z = \varepsilon$), we have $x \in L$ if and only if $w \in L$. So M accepts x if and only if $x \in L$. In other words, M accepts L, as claimed, so the proof is complete. □

*3.10 Minimal Automata

Given a DFA $M = (\Sigma, Q, s, A, \delta)$, suppose we want to find another DFA $M' = (\Sigma, Q', s', A', \delta')$ with the fewest possible states that accepts the same language. In this final section, we describe an efficient algorithm to minimize DFAs, first described (in slightly different form) by Edward Moore in 1956. We analyze the running time of Moore's in terms of two parameters: $n = |Q|$ and $\sigma = |\Sigma|$.

In the preprocessing phase, we find and remove any states that cannot be reached from the start state s; this filtering can be performed in $O(n\sigma)$ time using any graph traversal algorithm. So from now on we assume that all states are reachable from s.

Now define two states p and q in the trimmed DFA to be ***distingusiable***, written $\boldsymbol{p \not\sim q}$, if at least one of the following conditions holds:

- $p \in A$ and $q \notin A$,
- $p \notin A$ and $q \in A$, or
- $\delta(p, a) \not\sim \delta(q, a)$ for some $a \in \Sigma$.

Equivalently, $p \not\sim q$ if and only if there is a string z such that exactly one of the states $\delta^*(p, z)$ and $\delta^*(q, z)$ is accepting. (Sound familiar?) Intuitively, the main algorithm assumes that all states are equivalent until proven otherwise, and then repeatedly looks for state pairs that can be proved distinguishable.

The main algorithm maintains a two-dimensional table, indexed by the states, where $Dist[p, q] = \text{TRUE}$ indicates that we have proved states p and q are distinguished. Initially, for all states p and q, we set $Dist[p, q] \leftarrow \text{TRUE}$ if $p \in A$ and $q \notin A$ or vice versa, and $Dist[p, q] = \text{FALSE}$ otherwise. Then we repeatedly consider each pair of states and each symbol to find more distinguished pairs, until we make a complete pass through the table without modifying it. The table-filling algorithm can be summarized as follows:[2]

★★★ Don't just whine; actually explain Moore's algorithm as a dynamic programming. Need to prove that if two states can be distinguished at all, they can be distinguished by a string of length at most n.

[2] More experienced readers should become violently ill at the mere suggestion that any algorithm is merely *filling in a table* instead of *evaluating a recurrence*; this algorithm is no exception. Consider the boolean function $Dist(p, q, k)$, which equals TRUE if and only if p and q can be distinguished by some string of length at most k. This function obeys the following recurrence:

$$Dist(p, q, k) = \begin{cases} (p \in A) \oplus (q \in A) & \text{if } k = 0, \\ Dist(p, q, k-1) \vee \bigvee_{a \in \Sigma} Dist(\delta(p, a), \delta(q, a), k-1) & \text{otherwise.} \end{cases}$$

The "table-filling" algorithm presented here is just a space-efficient dynamic programming algorithm to evaluate this recurrence.

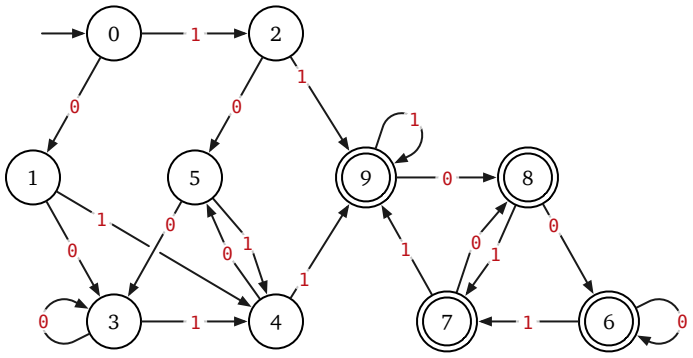
```
MINDFATABLE(Σ, Q, s, A, δ):
  for all p ∈ Q
      for all q ∈ Q
          if (p ∈ A and q ∉ A) or (p ∉ A and q ∈ A)
              Dist[p, q] ← TRUE
          else
              Dist[p, q] ← FALSE
  notdone ← TRUE
  while notdone
      notdone ← FALSE
      for all p ∈ Q
          for all q ∈ Q
              if Dist[p, q] = FALSE
              for all a ∈ Σ
                  if Dist[δ(p, a), δ(q, a)]
                      Dist[p, q] ← TRUE
                      notdone ← TRUE
  return Dist
```

The algorithm must eventually halt, because there are only a finite number of entries in the table that can be marked. In fact, the main loop is guaranteed to terminate after at most n iterations, which implies that the entire algorithm runs in $\boldsymbol{O(\sigma n^3)}$ time. Once the table is filled, any two states p and q such that $Dist(p, q) = \text{FALSE}$ are equivalent and can be merged into a single state. The remaining details of constructing the minimized DFA are straightforward.

With more care, Moore's minimization algorithm can be modified to run in $O(\sigma n^2)$ time. A faster DFA minimization algorithm, due to John Hopcroft, runs in $O(\sigma n \log n)$ time.

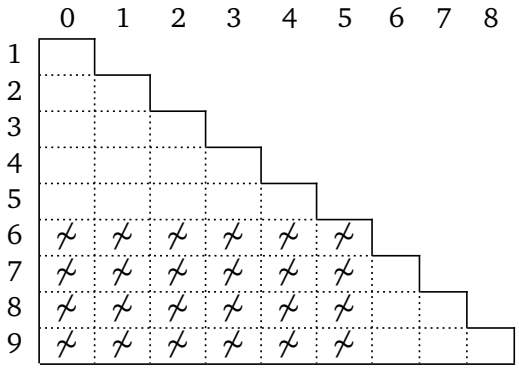
Example

To get a better idea how this algorithm works, let's visualize the algorithm running on our earlier brute-force DFA for strings containing the substring `11`. This DFA has four unreachable states: (FALSE, `11`), (TRUE, ε), (TRUE, `0`), and (TRUE, `1`). We remove these states, and relabel the remaining states for easier reference. (In an actual implementation, the states would almost certainly be represented by indices into an array anyway, not by mnemonic labels.)

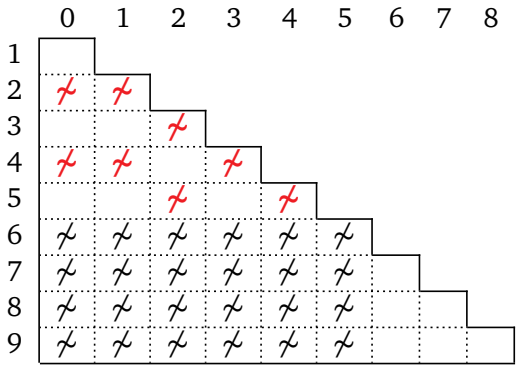


Our brute-force DFA for strings containing the substring `11`, after removing all four unreachable states

The main algorithm initializes (the bottom half of) a 10×10 table as follows. (In the implementation, cells marked $\not\sim$ have value TRUE and blank cells have value FALSE.)

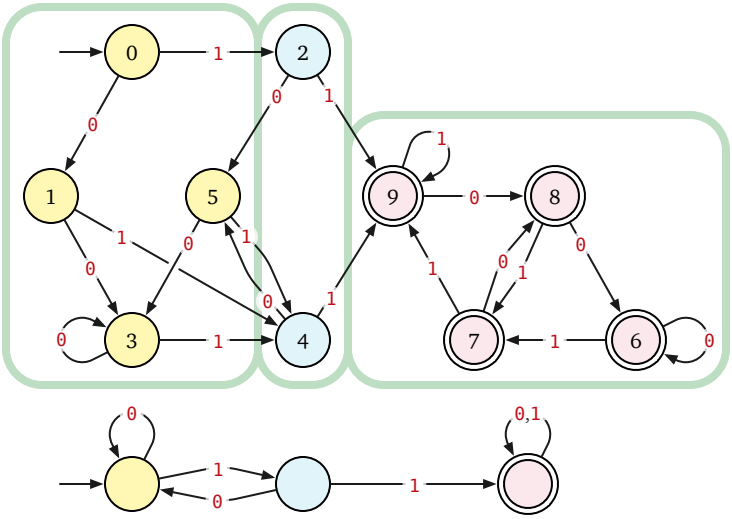


In the first iteration of the main loop, the algorithm discovers several distinguishable pairs of states. For example, the algorithm sets $Dist[0,2] \leftarrow \text{TRUE}$ because $Dist[\delta(0,1), \delta(2,1)] = Dist[2,9] = \text{TRUE}$. After the iteration ends, the table looks like this:



The second iteration of the while loop makes no further changes to the table—We got lucky!—so the algorithm terminates.

The final table implies that the states of our trimmed DFA fall into exactly three equivalence classes: $\{0,1,3,5\}$, $\{2,4\}$, and $\{6,7,8,9\}$. Replacing each equivalence class with a single state gives us the three-state DFA that we already discovered.



Equivalence classes of states in the trimmed DFA, and the resulting minimal equivalent DFA.

Exercises

1. For each of the following languages in $\{0,1\}^*$, describe a deterministic finite-state machine that accepts that language. There are infinitely many correct answers for each language. "Describe" does not necessarily mean "draw".

   (a) Only the string 0110.

   (b) Every string except 0110.

   (c) Strings that contain the substring 0110.

   (d) Strings that do not contain the substring 0110.

   *(e) Strings that contain an even number of occurrences of the substring 0110. (For example, this language contains the strings 0110110 and 01011.)

   (f) Strings that contain the *subsequence* 0110.

   (g) Strings that do not contain the *subsequence* 0110.

   (h) Strings that contain an even number of 1s and an odd number of 0s.

   (i) Strings that represent a number divisible by 7 in binary.

   (j) Strings whose reversals represent a number divisible by 7 in binary.

   (k) Strings in which the substrings 01 and 10 appear the same number of times.

   (l) Strings such that in every prefix, the number of 0s and the number of 1s differ by at most 1.

   (m) Strings such that in every prefix, the number of 0s and the number of 1s differ by at most 4.

   (n) Strings that end with $0^{10} = 0000000000$.

   (o) Strings in which the number of 1s is even, the number of 0s is divisible by 3, the overall length is divisible by 5, the binary value is divisible by 7, and the binary value of the reversal is divisible by 11. *[Hint: This is more tedious than difficult.]*

2. (a) Let $L \subseteq 0^*$ be an arbitrary *unary* language. Prove that L^* is regular.

   (b) Prove that there is a binary language $L \subseteq (0+1)^*$ such that L^* is not regular.

3. Describe and analyze algorithms for the following problems. In each case, the input is a DFA M over the alphabet $\Sigma = \{0,1\}$.

★★★ Move these to the graph traversal notes?

   (a) Does M accept any string whose length is a multiple of 5?

   (b) Does M accept every string that represents a number divisible by 7 in binary?

   (c) Does M accept an infinite number of strings containing an odd number of 0s?

   (d) Does M accept a finite number of strings that contain the substring 0110110 and whose length is divisible by five?

   (e) Does M accept *only* strings whose lengths are perfect squares?

   (f) Does M accept any string whose length is *composite*?

   *(g) Does M accept any string whose length is *prime*?

4. Prove that each of the following languages cannot be accepted by a DFA.

   (a) $\{\mathtt{0}^{n^2} \mid n \geq 0\}$

   (b) $\{\mathtt{0}^{n^3} \mid n \geq 0\}$

   (c) $\{\mathtt{0}^{f(n)} \mid n \geq 0\}$, where $f(n)$ is *any* fixed polynomial in n with degree at least 2.

   (d) $\{\mathtt{0}^n \mid n \text{ is composite}\}$

   (e) $\{\mathtt{0}^n\mathtt{1}\mathtt{0}^n \mid n \geq 0\}$

   (f) $\{\mathtt{0}^i\mathtt{1}^j \mid i \neq j\}$

   (g) $\{\mathtt{0}^i\mathtt{1}^j \mid i < 3j\}$

   (h) $\{\mathtt{0}^i\mathtt{1}^j \mid i \text{ and } j \text{ are relatively prime}\}$

   (i) $\{\mathtt{0}^i\mathtt{1}^j \mid j - i \text{ is a perfect square}\}$

   (j) $\{w\#w \mid w \in (\mathtt{0}+\mathtt{1})^*\}$

   (k) $\{ww \mid w \in (\mathtt{0}+\mathtt{1})^*\}$

   (l) $\{w\#\mathtt{0}^{|w|} \mid w \in (\mathtt{0}+\mathtt{1})^*\}$

   (m) $\{w\mathtt{0}^{|w|} \mid w \in (\mathtt{0}+\mathtt{1})^*\}$

   (n) $\{xy \mid w, x \in (\mathtt{0}+\mathtt{1})^* \text{ and } |x| = |y| \text{ but } x \neq y\}$

   (o) $\{\mathtt{0}^m\mathtt{1}^n\mathtt{0}^{m+n} \mid m, n \geq 0\}$

   (p) $\{\mathtt{0}^m\mathtt{1}^n\mathtt{0}^{mn} \mid m, n \geq 0\}$

   (q) Strings in which the substrings `00` and `11` appear the same number of times.

   (r) The set of all palindromes in $(\mathtt{0}+\mathtt{1})^*$ whose length is divisible by 7.

   (s) $\{w \in (\mathtt{0}+\mathtt{1})^* \mid w \text{ is the binary representation of a perfect square}\}$

   ★(t) $\{w \in (\mathtt{0}+\mathtt{1})^* \mid w \text{ is the binary representation of a prime number}\}$

5. For each of the following languages over the alphabet $\Sigma = \{\mathtt{0}, \mathtt{1}\}$, either describe a DFA that accepts the language or prove that no such DFA exists. Recall that Σ^+ denotes the set of all *nonempty* strings over Σ. *[Hint: Believe it or not, most of these languages* ***can*** *be accepted by DFAs.]*

   (a) $\{wxw \mid w, x \in \Sigma^*\}$

   (b) $\{wxw \mid w, x \in \Sigma^+\}$

   (c) $\{wxw^R \mid w, x \in \Sigma^+\}$

   (d) $\{wwx \mid w, x \in \Sigma^+\}$

   (e) $\{ww^Rx \mid w, x \in \Sigma^+\}$

   (f) $\{wxwy \mid w, x, y \in \Sigma^+\}$

   (g) $\{wxw^Ry \mid w, x, y \in \Sigma^+\}$

   (h) $\{xwwy \mid w, x, y \in \Sigma^+\}$

   (i) $\{xww^Ry \mid w, x, y \in \Sigma^+\}$

   (j) $\{wxxw \mid w, x \in \Sigma^+\}$

   *(k) $\{wxw^Rx \mid w, x \in \Sigma^+\}$

Caveat lector! This is the first edition of this lecture note. Some topics are incomplete, and there are almost certainly a few serious errors. Please send bug reports and suggestions to jeffe@illinois.edu.

The art of art, the glory of expression and the sunshine of the light of letters is simplicity. Nothing is better than simplicity . . . . nothing can make up for excess or for the lack of definiteness.

— Walt Whitman, Preface to *Leaves of Grass* (1855)

Freedom of choice
Is what you got.
Freedom from choice
Is what you want.

— Devo, "Freedom of Choice", *Freedom of Choice* (1980)

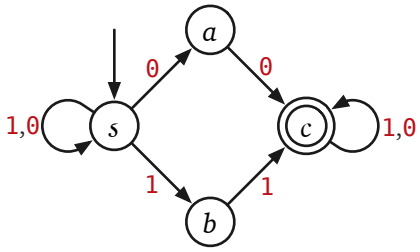
Nondeterminism means never having to say you are wrong.

— BSD 4.3 fortune(6) file (c.1985)

4 Nondeterminism

4.1 Nondeterministic State Machines

The following diagram shows something that looks like a finite-state machine over the alphabet {0, 1}, but on closer inspection, it is not consistent with our earlier definitions. On one hand, there are two transitions out of s for each input symbol. On the other hand, states a and b are each missing an outgoing transition.



A nondeterministic finite-state automaton

Nevertheless, there is a sense in which this machine "accepts" the set of all strings that contain either 00 or 11 as a substring. Imagine that when the machine reads a symbol in state s, it makes a choice about which transition to follow. If the input string contains the substring 00, then it is *possible* for the machine to end in the accepting state c, by *choosing* to move into state a when it reads a 0 immediately before another 0. Similarly, if the input string contains the substring 11, it is *possible* for the machine to end in the accepting state c. On the other hand, if the input string does not contain either 00 or 11—or in other words, if the input alternates between 0 and 1—there are no choices that lead the machine to the accepting state. If the machine incorrectly chooses to transition to state a and then reads a 1, or transitions to b and then reads 0, it explodes; the only way to avoid an explosion is to stay in state s.

This object is an example of a ***nondeterministic finite-state automaton***, or ***NFA***, so named because its behavior is not uniquely *determined* by the input string. Formally, every NFA has five components:

- An arbitrary finite set Σ, called the ***input alphabet***.
- Another arbitrary finite set Q, whose elements are called ***states***.

- An arbitrary ***transition*** function $\delta : Q \times \Sigma \to 2^Q$.
- A ***start state*** $s \in Q$.
- A subset $A \subseteq Q$ of ***accepting states***.

The only difference from the formal definition of *deterministic* finite-state automata is the domain of the transition function. In a DFA, the transition function always returns a single state; in an NFA, the transition function returns a *set* of states, which could be empty, or all of Q, or anything in between.

Just like DFAs, the behavior of an NFA is governed by an ***input string*** $w \in \Sigma^*$, which the machine reads one symbol at a time, from left to right. Unlike DFAs, however, an NFA does not maintain a single current state, but rather a *set* of current states. Whenever the NFA reads a symbol a, its set of current states changes from C to $\bigcup_{q \in C} \delta(q, a)$. After all symbols have been read, the NFA ***accepts*** w if its current state set contains *at least one* accepting state and ***rejects*** w otherwise. In particular, if the set of current states ever becomes empty, it will stay empty forever, and the NFA will reject.

More formally, we define the function $\delta^* : Q \times \Sigma^* \to 2^Q$ that transitions on *strings* as follows:

$$\delta^*(q, w) := \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \bigcup_{r \in \delta(q,a)} \delta^*(r, x) & \text{if } w = ax. \end{cases}$$

The NFA $(Q, \Sigma, \delta, s, A)$ ***accepts*** $w \in \Sigma^*$ if and only if $\delta^*(s, w) \cap A \neq \emptyset$.

We can equivalently define an NFA as a directed graph whose vertices are the states Q, whose edges are labeled with symbols from Σ. We no longer require that every vertex has exactly one outgoing edge with each label; it may have several such edges or none. An NFA accepts a string w if the graph contains *at least one* walk from the start state to an accepting state whose label is w.

4.2 Intuition

There are at least three useful ways to think about non-determinism.

Clairvoyance: Whenever an NFA reads symbol a in state q, it *chooses* the next state from the set $\delta(q, a)$, always *magically* choosing a state that leads to the NFA accepting the input string, unless no such choice is possible. As the BSD fortune file put it, "Nondeterminism means never having to say you're wrong."[1] Of course real machines can't actually look into the future; that's why I used the word "magic".

Parallel threads: An arguably more "realistic" view is that when an NFA reads symbol a in state q, it spawns an independent execution thread for each state in $\delta(q, a)$. In particular, if $\delta(q, a)$ is empty, the current thread simply dies. The NFA accepts if *at least one* thread is in an accepting state after it reads the last input symbol.

Equivalently, we can imagine that when an NFA reads symbol a in state q, it branches into several parallel universes, one for each state in $\delta(q, a)$. If $\delta(q, a)$ is empty, the NFA destroys the universe (including itself). Similarly, if the NFA finds itself in a non-accepting state when the input ends, the NFA destroys the universe. Thus, when the input is gone, only universes in which the NFA somehow chose a path to an accept state still exist. One slight disadvantage of this metaphor is that if an NFA reads a string that is not in its language, it destroys *all* universes.

Proofs/oracles: Finally, we can treat NFAs not as a mechanism for *computing* something, but only as a mechanism for *verifying proofs*. If we want to *prove* that a string w contains one of the suffixes `00` or `11`, it suffices to demonstrate a single walk in our example NFA that starts at s and ends at c, and whose

[1] This sentence is a riff on a horrible aphorism that was (sadly) popular in the US in the 70s and 80s. Fortunately, everyone seems to have forgotten the original saying, except for that one time it was parodied on the Simpsons.

edges are labeled with the symbols in w. Equivalently, whenever the NFA faces a nontrivial choice, the prover can simply tell the NFA which state to move to next.

This intuition can be formalized as follows. Consider a *deterministic* finite state machine whose input alphabet is the product $\Sigma \times \Omega$ of an ***input*** alphabet Σ and an ***oracle*** alphabet Ω. Equivalently, we can imagine that this DFA reads simultaneously from two strings of the same length: the *input* string w and the *oracle* string ω. In either formulation, the transition function has the form $\delta : Q \times \Sigma \times \Omega \to Q$. As usual, this DFA accepts the pair $(w, \omega) \in (\Sigma \times \Gamma)^*$ if and only if $\delta^*(s, w, \omega) \in A$. Finally, M ***nondeterministically accepts*** the string $w \in \Sigma^*$ if there is an oracle string $\omega \in \Omega^*$ with $|\omega| = |w|$ such that $(w, \omega) \in L(M)$.

4.3 ε-Transitions

It is fairly common for NFAs to include so-called ***ε-transitions***, which allow the machine to change state without reading an input symbol. An NFA with ε-transitions accepts a string w if and only if there is a sequence of transitions $s \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_\ell} q_\ell$ where the final state q_ℓ is accepting, each a_i is either ε or a symbol in Σ, and $a_1 a_2 \cdots a_\ell = w$.

More formally, the transition function in an NFA with ε-transitions has a slightly larger domain $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$. The ***$\varepsilon$-reach*** of a state $q \in Q$ consists of all states r that satisfy one of the following conditions:

- $r = q$
- $r \in \delta(q', \varepsilon)$ for some state q' in the ε-reach of q.

In other words, r is in the ε-reach of q if there is a (possibly empty) sequence of ε-transitions leading from q to r. Now we redefine the extended transition function $\delta^* : Q \times \Sigma^* \to 2^Q$, which transitions on arbitrary strings, as follows:

$$\delta^*(q, w) := \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \displaystyle\bigcup_{r \in \varepsilon\text{-reach}(q)} \bigcup_{r' \in \delta(r,a)} \delta^*(r', x) & \text{if } w = ax. \end{cases}$$

As usual, the modified NFA accepts a string w if and only if $\delta^*(s, w) \cap A \neq \emptyset$.

Given an NFA $M = (\Sigma, Q, s, A, \delta)$ with ε-transitions, we can easily construct an equivalent NFA $M' = (\Sigma, Q', s', A', \delta')$ without ε-transitions as follows:

$$\begin{aligned} Q' &:= Q \\ s' &= s \\ A' &= \left\{ q \in Q \;\middle|\; \varepsilon\text{-reach}(q) \cap A \neq \emptyset \right\} \\ \delta'(q, a) &= \bigcup_{r \in \varepsilon\text{-reach}(q)} \delta(r, a) \end{aligned}$$

Straightforward definition-chasing implies that M and M' accept exactly the same language. Thus, whenever we reason about or design NFAs, we are free to either allow or forbid ε-transitions, whichever is more convenient for the task at hand.

4.4 Kleene's Theorem

We are now finally in a position to prove the following fundamental fact, first observed by Steven Kleene:

Theorem 4.1. *A language L can be described by a regular expression if and only if L is the language accepted by a DFA.*

We will prove Kleene's fundamental theorem in four stages:

- Every DFA can be transformed into an equivalent NFA.
- Every NFA can be transformed into an equivalent DFA.
- Every regular expression can be transformed into an NFA.
- Every NFA can be transformed into an equivalent regular expression.

The first of these four transformations is completely trivial; a DFA is just a special type of NFA where the transition function always returns a single state. Unfortunately, the other three transformations require a bit more work.

4.5 DFA from NFAs: The Subset Construction

In the parallel-thread model of NFA execution, an NFA does not have a single current state, but rather a *set* of current states. The evolution of this set of states is *determined* by a modified transition function $\delta' : 2^Q \times \Sigma \to 2^Q$, defined by setting $\delta'(P, a) := \bigcup_{p \in P} \delta(p, a)$ for any set of states $P \subseteq Q$ and any symbol $a \in \Sigma$. When the NFA finishes reading its input string, it accepts if and only if the current set of states intersects the set A of accepting states.

This formulation makes the NFA completely deterministic! We have just shown that any NFA $M = (\Sigma, Q, s, A, \delta)$ is equivalent to a DFA $M' = (\Sigma, Q', s', A', \delta')$ defined as follows:

$$\begin{aligned} Q' &:= 2^Q \\ s' &:= \{s\} \\ A' &:= \left\{ S \subseteq Q \;\middle|\; S \cap A \neq \emptyset \right\} \\ \delta'(q', a) &:= \bigcup_{p \in q'} \delta(p, a) \qquad \text{for all } q' \subseteq Q \text{ and } a \in \Sigma. \end{aligned}$$

Similarly, any NFA with ε-transitions is equivalent to a DFA with the transition function

$$\delta'(q', a) = \bigcup_{p \in q'} \bigcup_{r \in \varepsilon\text{-reach}(p)} \delta(r, a)$$

for all $q' \subseteq Q$ and $a \in \Sigma$. This conversion from NFA to DFA is often called the ***subset construction***, but that name is somewhat misleading; it's not a "construction" so much as a change in perspective.

One disadvantage of this "construction" is that it usually leads to DFAs with far more states than necessary, in part because most of those states are unreachable. These unreachable states can be avoided by constructing the DFA incrementally, essentially by performing a breadth-first search of the DFA graph, starting at its start state.

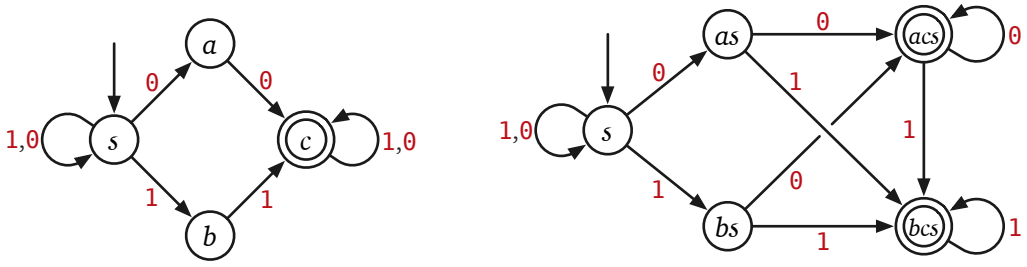
To execute this algorithm by hand, we prepare a table with $|\Sigma| + 3$ columns, with one row for each DFA state we discover. In order, these columns record the following information:

- The DFA state (as a subset of NFA states)
- The ε-reach of the corresponding subset of NFA states
- Whether the DFA state is accepting (that is, whether the ε-reach intersects A)
- The output of the transition function for each symbol in Σ.

We start with DFA-state s in the first row and first column. Whenever we discover an unexplored state in one of the last $|\Sigma|$ columns, we copy it to the left column in a new row.

For example, given the NFA on the first page of this note, this incremental algorithm produces the following table, yielding a five-state DFA. For this example, the second column is redundant, because the NFA has no ε-transitions, but we will see another example with ε-transitions in the next subsection. To simplify notation, we write each set of states as a simple string, omitting braces and commas.

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	s		as	bs
as	as		acs	bs
bs	bs		as	bcs
acs	acs	✓	acs	bcs
bcs	bcs	✓	acs	bcs



Our example NFA, and the output of the incremental subset construction for that NFA.

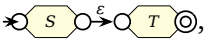
4.6 NFAs from Regular Expressions: Thompson's Algorithm

Lemma 4.2. *Every regular language is accepted by a non-deterministic finite automaton.*

Proof: In fact, we will prove the following stronger claim: Every regular language is accepted by an NFA *with exactly one accepting state, which is different from its start state*. The following construction was first described by Ken Thompson in 1968. Thompson's algorithm actually proves a stronger statement: For any regular language L, there is an NFA that accepts L that has exactly one accepting state t, which is distinct from the starting state s.

Let R be an arbitrary regular expression over an arbitrary finite alphabet Σ. Assume that for any sub-expression S of R, the language described by S is accepted by an NFA with one accepting state distinct from its start state, which we denote pictorially by . There are six cases to consider—three base cases and three recursive cases—mirroring the recursive definition of a regular expression.

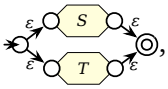
- If $R = \emptyset$, then $L(R) = \emptyset$ is accepted by the empty NFA: .
- If $R = \varepsilon$, then $L(R) = \{\varepsilon\}$ is accepted by the NFA .
- If $R = a$ for some symbol $a \in \Sigma$, then $L(R) = \{a\}$ is accepted by the NFA . (The case where R is a single *string* with length greater than 1 reduces to the single-symbol case by concatenation, as described in the next case.)
- Suppose $R = ST$ for some regular expressions S and T. The inductive hypothesis implies that the languages $L(S)$ and $L(T)$ are accepted by NFAs and , respectively. Then

$L(R) = L(ST) = L(S) \bullet L(T)$ is accepted by the NFA 

, built by connecting the two component NFAs in series.

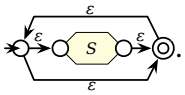
- Suppose $R = S + T$ for some regular expressions S and T. The inductive hypothesis implies that the language $L(S)$ and $L(T)$ are accepted by NFAs 

and 

, respectively. Then $L(R) = L(S + T) = L(S) \cup L(T)$ is accepted by the NFA 

, built by connecting the two component NFAs in parallel with new start and accept states.

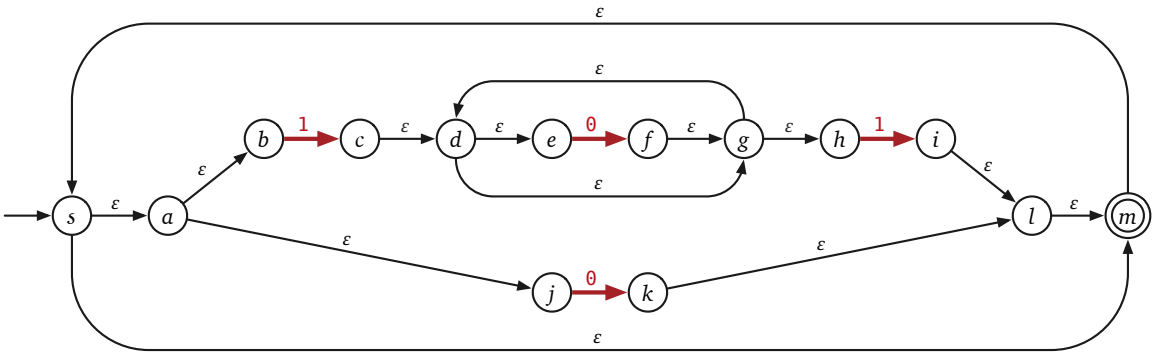
- Finally, suppose $R = S^*$ for some regular expression S. The inductive hypothesis implies that the language $L(S)$ is accepted by an NFA 

. Then the language $L(R) = L(S^*) = L(S)^*$ is accepted by the NFA 

.

In all cases, the language $L(R)$ is accepted by an NFA with one accepting state, which is different from its start state, as claimed. □

As an example, given the regular expression $(0 + 10^*1)^*$ of strings containing an even number of 1s, Thompson's algorithm produces a 14-state NFA shown on the next page. As this example shows, Thompson's algorithm tends to produce NFAs with many redundant states. Fortunately, just as there are for DFAs, there are algorithms that can reduce any NFA to an equivalent NFA with the smallest possible number of states.

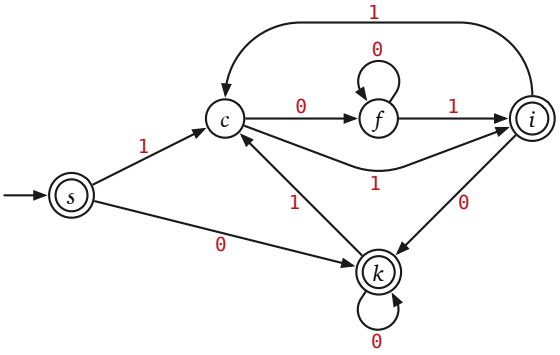


The NFA constructed by Thompson's algorithm for the regular expression $(0 + 10^*1)^*$.
The four non-ε-transitions are drawn with with bold red arrows for emphasis.

Interestingly, applying the incremental subset algorithm to Thompson's NFA tends to yield a DFA with relatively *few* states, in part because the states in Thompson's NFA tend to have large ε-reach, and in part because relatively few of those states are the targets of non-ε-transitions. Starting with the NFA shown above, for example, the incremental subset construction yields a DFA for the language $(0 + 10^*1)^*$ with just five states:

This DFA can be further simplified to just two states, by observing that all three accepting states are all equivalent, and that both non-accepting states are equivalent. But still, five states is pretty good, especially compared with the $2^{13} = 8096$ states that the naïve subset construction would yield!

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	$sabjm$	✓	k	c
k	$sabjklm$	✓	k	c
c	$cdegh$		f	i
f	$degh$		f	i
i	$sabjilm$	✓	k	c



The DFA computed by the incremental subset algorithm from Thompson's NFA for $(0 + 10^*1)^*$.

*4.7 NFAs from Regular Expressions: Glushkov's Algorithm

Thompson's algorithm is actually a modification of an earlier algorithm, which was independently discovered by V. I. Glushkov in 1961 and Robert McNaughton and Hisao Yamada in 1960. Given a regular expression containing n symbols (not counting the parentheses and pluses and stars), Glushkov's algorithm produces an NFA with exactly $n + 1$ states.

Glushkov's algorithm combines six functions on regular expressions:

- *index*(R) is the regular expression obtained by replacing the symbols in R with the integers 1 through n, in order from left to right. For example, $\mathit{index}((0 + 10^*1)^*) = (1 + 23^*4)^*$.
- *symbols*(R) denotes the string obtained by removing all non-symbols from R. For example, $\mathit{symbols}((0 + 10^*1)^*) = 0101$.
- *has-ε*(R) is TRUE if $\varepsilon \in L(R)$ and FALSE otherwise.
- *first*(R) is the set of all initial symbols of strings in $L(R)$.
- *last*(R) is the set of all final symbols of strings in $L(R)$.
- *middle*(R) is the set of all pairs (a, b) such that ab is a substring of some string in $L(R)$.

The last four functions obey the following recurrences:

$$\mathit{has\text{-}\varepsilon}(\emptyset) = \emptyset$$

$$\mathit{has\text{-}\varepsilon}(w) = \begin{cases} \text{TRUE} & \text{if } w = \varepsilon \\ \text{FALSE} & \text{otherwise} \end{cases}$$

$$\mathit{has\text{-}\varepsilon}(S + T) = \mathit{has\text{-}\varepsilon}(S) \vee \mathit{has\text{-}\varepsilon}(T)$$

$$\mathit{has\text{-}\varepsilon}(ST) = \mathit{has\text{-}\varepsilon}(S) \wedge \mathit{has\text{-}\varepsilon}(T)$$

$$\mathit{has\text{-}\varepsilon}(S^*) = \text{TRUE}$$

$$\mathit{first}(\emptyset) = \emptyset$$

$$\mathit{first}(w) = \begin{cases} \emptyset & \text{if } w = \varepsilon \\ \{a\} & \text{if } w = ax \end{cases}$$

$$\mathit{first}(S + T) = \mathit{first}(S) \cup \mathit{first}(T)$$

$$\mathit{first}(ST) = \begin{cases} \mathit{first}(S) \cup \mathit{first}(T) & \text{if } \mathit{has\text{-}\varepsilon}(S) \\ \mathit{first}(T) & \text{otherwise} \end{cases}$$

$$\mathit{first}(S^*) = \mathit{first}(S)$$

$$\mathit{last}(\emptyset) = \emptyset$$

$$\mathit{last}(w) = \begin{cases} \emptyset & \text{if } w = \varepsilon \\ \{a\} & \text{if } w = xa \end{cases}$$

$$\mathit{last}(S + T) = \mathit{last}(S) \cup \mathit{last}(T)$$

$$\mathit{last}(ST) = \begin{cases} \mathit{last}(S) \cup \mathit{last}(T) & \text{if } \mathit{has\text{-}\varepsilon}(T) \\ \mathit{last}(T) & \text{otherwise} \end{cases}$$

$$\mathit{last}(S^*) = \mathit{last}(S)$$

$$
\begin{aligned}
middle(\emptyset) &= \emptyset \\
middle(w) &= \begin{cases} \emptyset & \text{if } |w| \le 1 \\ \{(a,b)\} \cup middle(bx) & \text{if } w = abx \end{cases} \\
middle(S+T) &= middle(S) \cup middle(T) \\
middle(ST) &= middle(S) \cup (last(S) \times first(T)) \cup middle(T) \\
middle(S^*) &= middle(S) \cup (last(S) \times first(S))
\end{aligned}
$$

For example, the set $middle((1+23^*4)^*)$ can be computed recursively as follows. If we're doing this by hand, we can skip many of the steps in this derivation, because we know what the functions *first*, *middle*, *last*, and *has-ε* actually mean, but a mechanical recursive evaluation would necessarily evaluate every step.

$$
\begin{aligned}
&middle((1+23^*4)^*) \\
&\quad = middle(1+23^*4) \cup \big(last(1+23^*4) \times first(1+23^*4)\big) \\
&\quad = middle(1) \cup middle(23^*4) \cup \big(last(1+23^*4) \times first(1+23^*4)\big) \\
&\quad = \emptyset \cup middle(23^*4) \cup \big(last(1+23^*4) \times first(1+23^*4)\big) \\
&\quad = middle(2) \cup \big(last(2) \times first(3^*4)\big) \cup middle(3^*4) \cup \big(last(1+23^*4) \times first(1+23^*4)\big) \\
&\quad = \emptyset \cup \big(\{2\} \times first(3^*4)\big) \cup middle(3^*4) \cup \big(last(1+23^*4) \times first(1+23^*4)\big) \\
&\quad = \big(\{2\} \times \big(first(3^*) \cup first(4)\big)\big) \cup middle(3^*4) \cup \big(last(1+23^*4) \times first(1+23^*4)\big) \\
&\quad = \big(\{2\} \times \big(first(3) \cup first(4)\big)\big) \cup middle(3^*4) \cup \big(last(1+23^*4) \times first(1+23^*4)\big) \\
&\quad = \big(\{2\} \times \{3,4\}\big) \cup middle(3^*4) \cup \big(last(1+23^*4) \times first(1+23^*4)\big) \\
&\quad = \big\{(2,3),(3,4)\big\} \cup middle(3^*4) \cup \big(last(1+23^*4) \times first(1+23^*4)\big) \\
&\quad \vdots \\
&\quad = \big\{(1,1),(1,2),(2,3),(2,4),(3,3),(3,4),(4,1),(4,2)\big\}
\end{aligned}
$$

Finally, given any regular expression R, Glushkov's algorithm constructs the NFA $M_R = (\Sigma, Q, s, A, \delta)$ that accepts exactly the language $L(R)$ as follows:

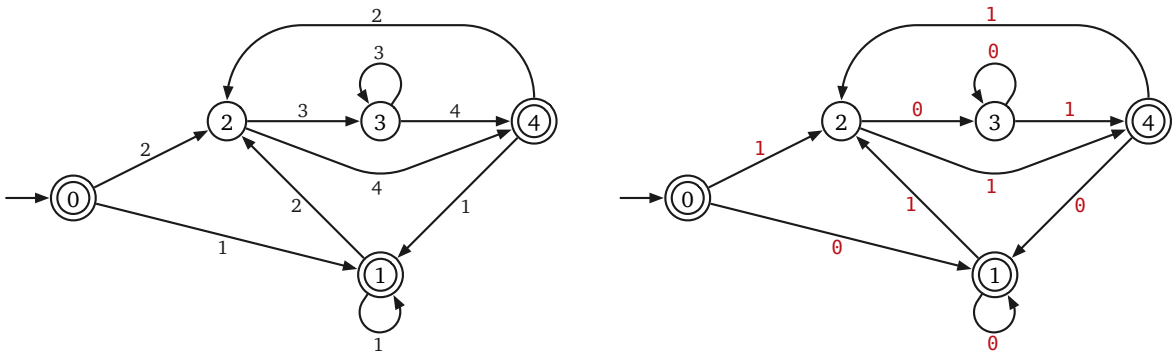
$$
\begin{aligned}
Q &= \big\{0, 1, \ldots, |symbols(R)|\big\} \\
s &= 0 \\
A &= \begin{cases} \{0\} \cup last(index(R)) & \text{if } has\text{-}\varepsilon(R) \\ last(index(R)) & \text{otherwise} \end{cases} \\
\delta(0,a) &= \big\{ j \in first(index(R)) \;\big|\; a = symbols(R)[j] \big\} \\
\delta(i,a) &= \big\{ j \;\big|\; (i,j) \in middle(index(R)) \text{ and } a = symbols(R)[j] \big\}
\end{aligned}
$$

There are a few natural ways to think about Glushkov's algorithm that are somewhat less impenetrable than the wall of definitions. One viewpoint is that Glushkov's algorithm first computes a *DFA* for the indexed regular expression $index(R)$—in fact, a DFA with the fewest possible states, except for an extra start state—and then replaces each index with the corresponding symbol in $symbols(R)$ to get an NFA for the original expression R. Another useful observation is that Glushkov's NFA is identical to the result of removing all ε-transitions from Thompson's NFA for the same regular expression.

For example, given the regular expression $R = (\texttt{0} + \texttt{10}^*\texttt{1})^*$, Glushkov's algorithm computes

$$\begin{aligned}
\mathit{index}(R) &= (1 + 23^*4)^* \\
\mathit{symbols}(R) &= \texttt{0101} \\
\mathit{has\text{-}\varepsilon}(R) &= \textsc{True} \\
\mathit{first}(\mathit{index}(R)) &= \{1, 2\} \\
\mathit{last}(\mathit{index}(R)) &= \{1, 4\} \\
\mathit{middle}(\mathit{index}(R)) &= \big\{(1,1), (1,2), (2,3), (2,4), (3,3), (3,4), (4,1), (4,2)\big\}
\end{aligned}$$

and then constructs the following five-state NFA.

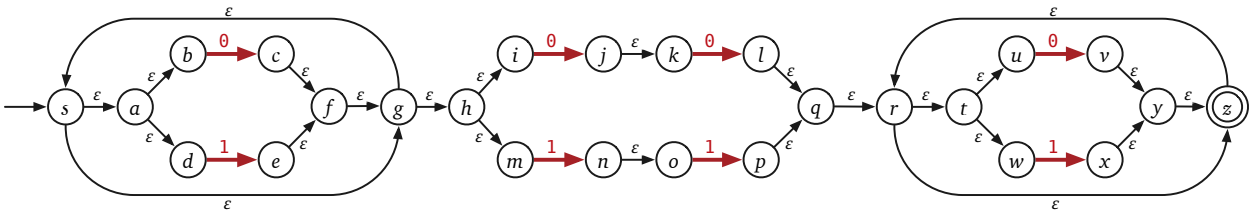


Glushkov's DFA for the index expression $(1 + 23^*4)^*$ and Glushkov's NFA for the regular expression $(\texttt{0} + \texttt{10}^*\texttt{1})^*$.

Hey, look, Glushkov's algorithm actually gave us a DFA! In fact, it gave us *precisely* the same DFA that we constructed earlier by sending Thompson's NFA through the incremental subset algorithm! Unfortunately, that's just a coincidence; in general the output of Glushkov's algorithm is *not* deterministic. We'll see a more typical example in the next section.

4.8 Another Example

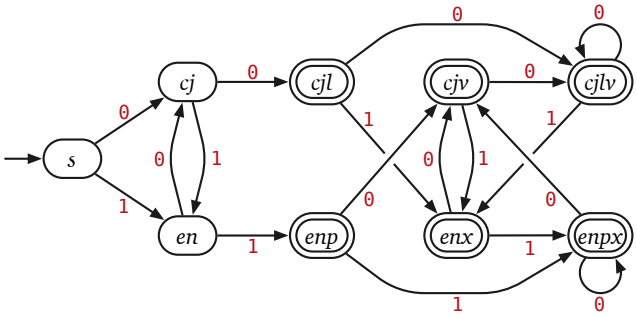
Here is another example of all the algorithms we've seen so far, starting with the regular expression $(\texttt{0}+\texttt{1})^*(\texttt{00}+\texttt{11})(\texttt{0}+\texttt{1})^*$, which describes the language accepted by our very first example NFA. Thompson's algorithm constructs the following 26-state monster:



Thompson's NFA for the regular expression $(\texttt{0} + \texttt{1})^*(\texttt{00} + \texttt{11})(\texttt{0} + \texttt{1})^*$

Given this NFA as input, the incremental subset construction computes the following table, leading to a DFA with just nine states. Yeah, the ε-reaches get a bit ridiculous; unfortunately, this *is* typical for Thompson's NFA.

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	$sabdghim$		cj	en
cj	$sabdfghijkm$		cjl	en
en	$sabdfghmno$		cj	enp
cjl	$sabdfghijklmqrtuwz$	✓	$cjlv$	enx
enp	$sabdfghmnopqrtuwz$	✓	cjv	$enpx$
$cjlv$	$sabdfghijklmqrtuvwyz$	✓	$cjlv$	enx
enx	$sabdfghmnopqrtuwxyz$	✓	cjv	$enpx$
cjv	$sabdfghijkmrtuvwyz$	✓	$cjlv$	enx
$enpx$	$sabdfghmnopqrtuwxyz$	✓	cjv	$enpx$



The DFA computed by the incremental subset algorithm from Thompson's NFA for $(0+1)^*(00+11)(0+1)^*$.

This DFA has far more states that necessary, intuitively because it keeps looking for 00 and 11 substrings even after it's already found one. After all, when Thompson's NFA finds a00 and 11 substring, it doesn't kill all the other parallel threads, because it *can't*. NFAs often have significantly fewer states than equivalent DFAs, but that efficiency also makes them kind of stupid.

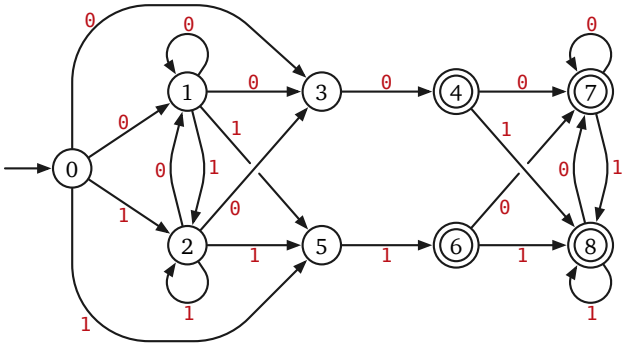
Glushkov's algorithm recursively computes the following values for the same regular expression $R = (0+1)^*(00+11)(0+1)^*$:

$$\begin{aligned}
\mathit{index}(R) &= (1+2)^*(34+56)(7+8)^* \\
\mathit{symbols}(R) &= 01001101 \\
\mathit{has\text{-}\varepsilon}(R) &= \textsc{False} \\
\mathit{first}(\mathit{index}(R)) &= \{1,2,3,5\} \\
\mathit{last}(\mathit{index}(R)) &= \{4,6,7,8\} \\
\mathit{middle}(\mathit{index}(R)) &= \big\{(1,1),(1,2),(2,1),(2,2),(1,3),(1,5),(2,3),(2,5),(3,4), \\
&\qquad (5,6),(4,7),(4,8),(6,7),(6,8),(7,7),(7,8),(8,7),(8,8)\big\}
\end{aligned}$$

These values imply the nine-state NFA shown on the next page. Careful readers should confirm that running the incremental subset construction on this NFA yields exactly the same DFA (with different state names) as it did for Thompson's NFA.

*4.9 Regular Expressions from NFAs: Han and Wood's Algorithm

The only component of Kleene's theorem we still have to prove is that every language accepted by a DFA or NFA is regular. As usual, it is actually easier to prove a stronger result. We consider a more general class of finite-state machines called ***expression automata***, introduced by Yo-Sub Han and Derick Wood



Glushkov's NFA for $(0+1)^*(00+11)(0+1)^*$

in 2005.[2] Formally, an expression automaton consists of the following components:

- A finite set Σ called the ***input alphabet***
- Another finite set Q whose elements are called ***states***
- A ***start state*** $s \in Q$
- A single ***terminal state*** $t \in Q \setminus \{s\}$
- A ***transition function*** $R\colon (Q \setminus \{t\}) \times (Q \setminus \{s\}) \to \mathit{Reg}(\Sigma)$, where $\mathit{Reg}(\Sigma)$ is the set of regular expressions over Σ

Less formally, an expression automaton is a directed graph that includes a directed edge $p \to q$ labeled with a regular expression $R(p \to q)$, from *every* vertex p to *every* vertex q (including $q = p$), where by convention, we require that $R(q \to s) = R(t \to q) = \emptyset$ for every vertex q.

We say that string w ***matches*** a transition $p \to q$ if w matches the regular expression $R(p \to q)$. In particular, if $R(p \to q) = \emptyset$, then no string matches $p \to q$. More generally, w matches a sequence of states $q_0 \to q_1 \to \cdots \to q_k$ if w matches the regular expression $R(q_0 \to q_1) \bullet R(q_1 \to q_2) \bullet \cdots \bullet R(q_{k-1} \to q_k)$. Equivalently, w matches the sequence $q_0 \to q_1 \to \cdots \to q_k$ if either

- $w = \varepsilon$ and the sequence has only one state ($k = 0$), or
- $w = xy$ for some string x that matches the regular expression $R(q_0 \to q_1)$ and some string y that matches the remaining sequence $q_1 \to \cdots \to q_k$.

An expression automaton ***accepts*** any string that matches at least one sequence of states that starts at s and ends at t. The ***language*** of an expression automaton E is the set of all strings that E accepts.

Expression automata are nondeterministic. A single string could match several (even infinitely many) state sequences that start with s, and it could match each of those state sequences in several different ways. A string is accepted if *at least one* of the state sequences it matches ends at t. Conversely, a string might match *no* state sequences; all such strings are rejected.

Two special cases of expression automata are already familiar. First, every regular language is clearly the language of an expression automaton with exactly two states. Second, with only minor modifications, any DFA or NFA can be converted into an expression automaton with trivial transition expressions. Thompson's algorithm can be used to transform any expression automaton into an NFA, by recursively expanding any nontrivial transition. To complete the proof of Kleene's theorem, we show how to convert any expression automaton into a regular expression by repeatedly deleting vertices.

[2] Yo-Sub Han* and Derick Wood. The generalization of generalized automata: Expression automata. International Journal of Foundations of Computer Science 16(3):499–510, 2005.

Lemma 4.3. *Every expression automaton accepts a regular language.*

Proof: Let $E = (Q, \Sigma, R, s, t)$ be an arbitrary expression automaton. Assume that any expression automaton with fewer states than E accepts a regular language. There are two cases to consider, depending on the number of states in Q:

- If $Q = \{s, t\}$, then trivially, E accepts the regular language $R(s \to t)$.
- On the other hand, suppose there is a state $q \in Q \setminus \{s, a\}$. We can modify the expression automaton so that state q is redundant and can be removed. Define a new transition function $R' : Q \times Q \to Reg(\Sigma)$ by setting

$$R'(p \to r) := R(p \to r) + R(p \to q)\,R(q \to q)^*\,R(q \to r).$$

  With this modified transition function, any string w that matches the sequence $p \to q \to q \to \cdots \to q \to r$ (for any number of q's) also matches the single transition $p \to r$. Thus, by induction, if w matches a sequence of states, it also matches the subsequence obtained by removing all q's. Let E' be the expression automaton with states $Q' = Q \setminus \{q\}$ that uses this modified transition function R'. This new automaton accepts exactly the same strings as the original automaton E. Because E' has fewer states than E, the inductive hypothesis implies E' accepts a regular language.

In both cases, we conclude that E accepts a regular language. □

This proof can be mechanically translated into an algorithm to convert any NFA—in particular, any DFA—into an equivalent regular expression. Given an NFA with n states (including s and a), the algorithm iteratively removes $n-2$ states, updating $O(n^2)$ transition expressions in each iteration. If the concatenation and Kleene star operations could be performed in constant time, the resulting algorithm would run in $O(n^3)$ time. However, in each iteration, the transition expressions grows in length by roughly a factor of 4 in the worst case, so the final expression has length $\Theta(4^n)$. If we insist on representing the expressions as explicit strings, the worst-case running time is actually $\Theta(4^n)$.

A figure on the next page shows this conversion algorithm in action for a simple DFA. First we convert the DFA into an expression automaton by adding new start and accept states and merging two transitions, and then we remove each of the three original states, updating the transition expressions between any remaining states at each iteration. For the sake of clarity, edges $p \to q$ with $R(p \to q) = \emptyset$ are omitted from the figures.

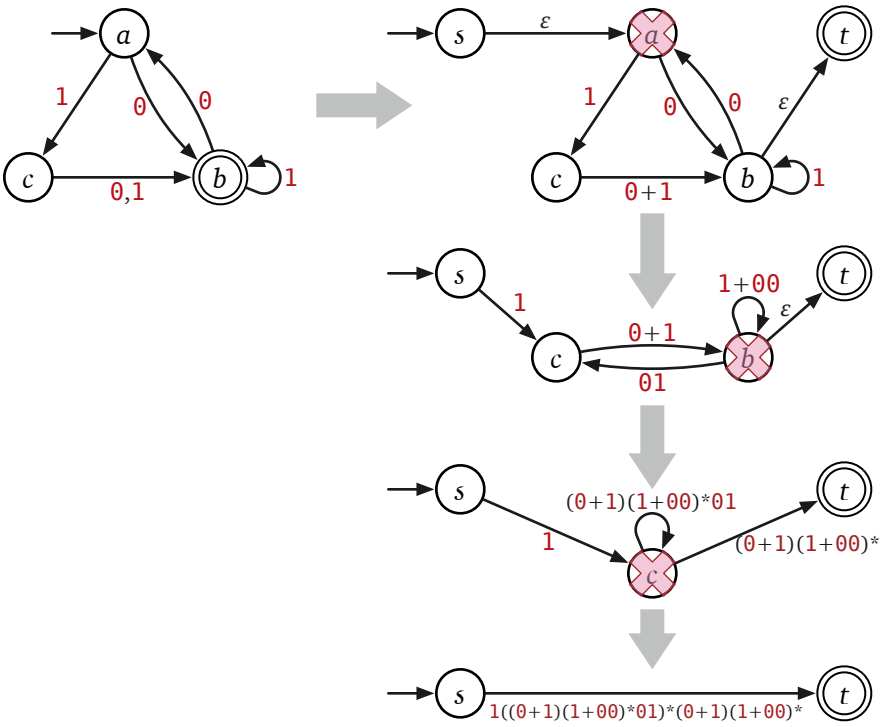
*4.10 Recursive Automata

★★★ Move to separate starred note?

All the flavors of finite-state automata we have seen so far describe/encode/accept/compute *regular* language; these are precisely the languages that can be constructed from individual strings by union, concatenation, and unbounded repetition. The more general class of *context-free* languages can be constructed from individual strings by union, concatenation, and *recursion*. Just as context-free grammars are recursive generalizations of regular expressions, we can define a class of machines called *recursive automata*, which generalize (nondeterministic) finite-state automata.

Formally, a ***recursive automaton*** consists of the following components:

- A non-empty finite set Σ, called the ***input alphabet***
- Another non-empty finite set N, disjoint from Σ, whose elements are called ***module names***
- A ***start name*** $S \in N$



Converting a DFA into an equivalent regular expression.

- A set $M = \{M_A \mid A \in N\}$ of NFAs over the alphabet $\Sigma \cup N$ called ***modules***, each with a single accepting state. Each module M_A has the following components:
  - A finite set Q_A of ***states***, such that $Q_A \cap Q_B = \emptyset$ for all $A \neq B$
  - A ***start*** state $s_A \in Q_A$
  - A ***terminal*** or ***accepting*** state $t_A \in Q_A$
  - A ***transition function*** $\delta_A : Q_A \times (\Sigma \cup \{\varepsilon\} \cup N) \to 2^{Q_A}$.

Equivalently, we have a single global transition function $\delta : Q \times (\Sigma \cup \{\varepsilon\} \cup N) \to 2^Q$, where $Q = \bigcup_{A \in N} Q_A$, such that for any name A and any state $q \in Q_A$ we have $\delta(q) \subseteq Q_A$. Machine M_S is called the ***main module***.

A ***configuration*** of a recursive automaton is a triple (w, q, s), where w is a string in Σ^* called the ***input***, q is a state in Q called the ***local state***, and σ is a string in Q^* called the ***stack***. The module containing the local state q is called the ***active module***. A configuration can be changed by three types of transitions.

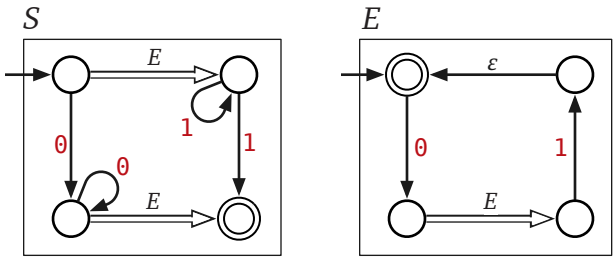
- A ***read*** transition consumes the first symbol in the input and changes the local state within the current module, just like a standard NFA.
- An ***epsilon*** transition changes the local state within the current module, without consuming any input characters, just like a standard NFA.
- A ***call*** transition chooses an arbitrary name A, changes the current state q_0 to some state in $\delta(q, A)$, and pushes the corresponding start state s_A onto the stack (thereby changing the active module to M_A), without consuming any input characters.
- Finally, if the current state is the terminal state of some module *and* the stack is non-empty, a ***return*** transition pops the top state off the stack and makes it the new local state (thereby possibly changing the active module), without consuming any input characters.

Symbolically, we can describe these transitions as follows:

read:	$(ax, q, \sigma) \longmapsto (x, q', \sigma)$	for some $q' \in \delta(q, a)$
epsilon:	$(w, q, \sigma) \longmapsto (w, q', \sigma)$	for some $q' \in \delta(q, \varepsilon)$
call:	$(w, q, \sigma) \longmapsto (w, s_A, q' \cdot \sigma)$	for some $A \in N$ and $q' \in \delta(q, A)$
return:	$(w, t_A, q \cdot \sigma) \longmapsto (w, q, \sigma)$	

A recursive automaton ***accepts*** a string w if there is a *finite* sequence of transitions starting at the start configuration (w, s_S, ε) and ending at the terminal configuration $(\varepsilon, t_S, \varepsilon)$.

For example, the following recursive automaton accepts the language $\{0^m 1^n \mid m \neq n\}$. The recursive automaton has two component machines; the start machine named S and a "subroutine" named E (for "equal") that accepts the language $\{0^n 1^n \mid n \geq 0\}$. White arrows indicate recursive transitions.



A recursive automaton for the language $\{0^m 1^n \mid m \neq n\}$

Lemma 4.4. *Every context-free language is accepted by a recursive automaton.*

Proof:

★★★ Direct construction from the CFG, with one module per nonterminal.

□

For example, the context-free grammar

$$S \to 0A \mid B1$$
$$A \to 0A \mid E$$
$$B \to B1 \mid E$$
$$E \to \varepsilon \mid 0E0$$

leads to the following recursive automaton with four modules:

★★★ Figure!

Lemma 4.5. *Every recursive automaton accepts a context-free language.*

Proof (sketch): Let $R = (\Sigma, N, S, \delta, M)$ be an arbitrary recursive automaton. We define a context-free grammar G that describes the language accepted by R as follows.

The set of nonterminals in G is isomorphic the state set Q; that is, for each state $q \in Q$, the grammar contains a corresponding nonterminal $[q]$. The language of $[q]$ will be the set of strings w such that there is a finite sequence of transitions starting at the start configuration (w, q, ε) and ending at the terminal configuration $(\varepsilon, t, \varepsilon)$, where t is the terminal state of the module containing q.

The grammar has four types of production rules, corresponding to the four types of transitions:

- ***read:*** For each symbol a and each pair of states p and q such that $p \in \delta(q, a)$, the grammar contains the production rule $[q] \to a[p]$.
- ***epsilon:*** For any two states p and q such that $p \in \delta(q, \varepsilon)$, the grammar contains the production rule $[q] \to [p]$.
- ***call:*** Each name A and each pair of states states p and q such that $p \in \delta(q, A)$, the grammar contains the production rule $[q] \to [s_A][p]$.
- ***return:*** Each name A, the grammar contains the production rule $[t_A] \to \varepsilon$.

Finally, the starting nonterminal of G is $[s_S]$, which corresponds to the start state of the main module.

We can now argue inductively that the grammar G and the recursive automaton R describe the same language. Specifically, any sequence of transitions in R from (w, s_S, ε) to $(\varepsilon, t_S, \varepsilon)$ can be transformed mechanically into a derivation of w from the nonterminal $[s_S]$ in G. Symmetrically, the ***leftmost*** derivation of any string w in G can be mechanically transformed into an accepting sequence of transitions in R. We omit the straightforward but tedious details. □

For example, the recursive automaton on the previous page gives us the following context-free grammar. To make the grammar more readable, I've renamed the nonterminals corresponding to start and terminal states: $S = [s_S]$, $T = [t_S]$, and $E = [s_E] = [t_E]$:

$$\begin{aligned} S &\to EA \mid \texttt{0}B & E &\to \varepsilon \mid \texttt{0}X \\ A &\to \texttt{1}A \mid \texttt{1}T & X &\to EY \\ B &\to \texttt{0}B \mid ET & Y &\to \texttt{1}Z \\ T &\to \varepsilon & Z &\to E \end{aligned}$$

Our earlier proofs imply that we can forbid ε-transitions or even allow regular-expression transitions in our recursive automata without changing the set of languages they accept.

Exercises

1. For each of the following NFAs, describe an equivalent DFA. ("Describe" does not necessarily mean "draw"!)

★★★ Half a dozen examples.

2. For each of the following regular expressions, draw an equivalent NFA.

★★★ Half a dozen examples.

3. For each of the following regular expressions, describe an equivalent DFA. ("Describe" does not necessarily mean "draw"!)

★★★ Half a dozen examples.

4. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular.

   (a) $\mathit{ones}(L) := \left\{ w \in 1^* \;\middle|\; |w| = |x| \text{ for some } x \in L \right\}$

   (b) $\mathit{reverse}(L) := \left\{ w \in \Sigma^* \;\middle|\; w^R \in L \right\}$. (Recall that w^R denotes the reversal of string w.)

   (c) $\mathit{prefix}(L) := \{x \in \Sigma^* \mid xy \in L \text{ for some } y \in \Sigma^*\}$

   (d) $\mathit{suffix}(L) := \{y \in \Sigma^* \mid xy \in L \text{ for some } x \in \Sigma^*\}$

   (e) $\mathit{substring}(L) := \{y \in \Sigma^* \mid xyz \in L \text{ for some } x, z \in \Sigma^*\}$

   (f) $\mathit{cycle}(L) := \{xy \mid x, y \in \Sigma^* \text{ and } yx \in L\}$

   (g) $\mathit{prefmax}(L) := \{x \in L \mid xy \in L \iff y = \varepsilon\}$.

   (h) $\mathit{sufmin}(L) := \{xy \in L \mid y \in L \iff x = \varepsilon\}$.

   (i) $\mathit{everyother}(L) := \{\mathit{everyother}(w) \mid w \in L\}$, where $\mathit{everyother}(w)$ is the subsequence of w containing every other symbol. For example, $\mathit{everyother}(\texttt{EVERYOTHER}) = \texttt{VROHR}$.

   (j) $\mathit{rehtoyreve}(L) := \{w \in \Sigma^* \mid \mathit{everyother}(w) \in L\}$.

   (k) $\mathit{subseq}(L) := \{x \in \Sigma^* \mid x \text{ is a subsequence of some } y \in L\}$

   (l) $\mathit{superseq}(L) := \{x \in \Sigma^* \mid \text{some } y \in L \text{ is a subsequence of } x\}$

   (m) $\mathit{left}(L) := \{x \in \Sigma^* \mid xy \in L \text{ for some } y \in \Sigma^* \text{ where } |x| = |y|\}$

   (n) $\mathit{right}(L) := \{y \in \Sigma^* \mid xy \in L \text{ for some } x \in \Sigma^* \text{ where } |x| = |y|\}$

   (o) $\mathit{middle}(L) := \{y \in \Sigma^* \mid xyz \in L \text{ for some } x, z \in \Sigma^* \text{ where } |x| = |y| = |z|\}$

   (p) $\mathit{half}(L) := \{w \in \Sigma^* \mid ww \in L\}$

   (q) $\mathit{third}(L) := \{w \in \Sigma^* \mid www \in L\}$

   (r) $\mathit{reflect}(L) := \left\{ w \in \Sigma^* \;\middle|\; ww^R \in L \right\}$

   *(s) $\mathit{sqrt}(L) := \left\{ x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = |x|^2 \right\}$

   *(t) $\mathit{log}(L) := \left\{ x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = 2^{|x|} \right\}$

   *(u) $\mathit{flog}(L) := \left\{ x \in \Sigma^* \;\middle|\; xy \in L \text{ for some } y \in \Sigma^* \text{ such that } |y| = F_{|x|} \right\}$, where F_n is the nth Fibonacci number.

*5. Let $L \subseteq \Sigma^*$ be an arbitrary regular language. Prove that the following languages are regular. *[Hint: For each language, there is an accepting NFA with at most q^q states, where q is the number of states in some DFA that accepts L.]*

   (a) $\mathit{repeat}(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for some } n \geq 0\}$

   (b) $\mathit{allreps}(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for every } n \geq 0\}$

   (c) $\mathit{manyreps}(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for infinitely many } n \geq 0\}$

   (d) $\mathit{fewreps}(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for finitely many } n \geq 0\}$

   (e) $\mathit{powers}(L) := \left\{ w \in \Sigma^* \;\middle|\; w^{2^n} \in L \text{ for some } n \geq 0 \right\}$

★(f) $whatthe_N(L) := \{w \in \Sigma^* \mid w^n \in L \text{ for some } n \in N\}$, where N is an ***arbitrary*** fixed set of non-negative integers. *[Hint: You only have to prove that an accepting NFA exists; you don't have to describe how to construct it.]*

6. For each of the following expression automata, describe an equivalent DFA *and* an equivalent regular expression.

★★★ Half a dozen examples.

7. Describe recursive automata that accept the following languages. ("Describe" does not necessarily mean "draw"!)

★★★ Half a dozen examples.

The control of a large force is the same principle as the control of a few men: it is merely a question of dividing up their numbers.

— Sun Zi, *The Art of War* (c. 400 C.E.), translated by Lionel Giles (1910)

Our life is frittered away by detail. . . . Simplify, simplify.

— Henry David Thoreau, *Walden* (1854)

Nothing is particularly hard if you divide it into small jobs.

— Henry Ford

Do the hard jobs first. The easy jobs will take care of themselves.

— Dale Carnegie

5 Recursion

5.1 Reductions

Reduction is the single most common technique used in designing algorithms. Reducing one problem X to another problem Y means to write an algorithm for X that uses an algorithm for Y as a black box or subroutine. Crucially, the correctness of the resulting algorithm cannot depend in any way on *how* the algorithm for Y works. The only thing we can assume is that the black box solves Y correctly. The inner workings of the black box are simply *none of our business*; they're somebody else's problem. It's often best to literally think of the black box as functioning by magic.

For example, the Huntington-Hill algorithm described in Lecture 0 reduces the problem of apportioning Congress to the problem of maintaining a priority queue that supports the operations INSERT and EXTRACTMAX. The abstract data type "priority queue" is a black box; the correctness of the apportionment algorithm does not depend on any specific priority queue data structure. Of course, the *running time* of the apportionment algorithm depends on the *running time* of the INSERT and EXTRACTMAX algorithms, but that's a separate issue from the *correctness* of the algorithm. The beauty of the reduction is that we can create a more efficient apportionment algorithm by simply swapping in a new priority queue data structure. Moreover, the designer of that data structure does not need to know or care that it will be used to apportion Congress.

Similarly, if we want to design an algorithm to compute the smallest deterministic finite-state machine equivalent to a given regular expression, we don't have to start from scratch. Instead we can reduce the problem to three subproblems for which algorithms can be found in earlier lecture notes: (1) build an NFA from the regular expression, using either Thompson's algorithm or Glushkov's algorithm; (2) transform the NFA into an equivalent DFA, using the (incremental) subset construction; and (3) transform the DFA into the smallest equivalent DFA, using Moore's algorithm, for example. Even if your class skipped over the automata notes, merely knowing that those component algorithms exist (Trust me!) allows you to combine them into more complex algorithms; you don't *need* to know the details. (But you should, because they're totally cool. Trust me!) Again swapping in a more efficient algorithm for any of those three subproblems automatically yields a more efficient algorithm for the problem as a whole.

When we design algorithms, we may not know exactly how the basic building blocks we use are implemented, or how our algorithms might be used as building blocks to solve even bigger problems. Even when you do know precisely how your components work, it is often *extremely* useful to pretend that you don't. (Trust *yourself*!)

5.2 Simplify and Delegate

Recursion is a particularly powerful kind of reduction, which can be described loosely as follows:

- If the given instance of the problem is small or simple enough, just solve it.
- Otherwise, reduce the problem to one or more ***simpler instances of the same problem***.

If the self-reference is confusing, it's helpful to imagine that someone else is going to solve the simpler problems, just as you would assume for other types of reductions. I like to call that someone else the Recursion Fairy. Your *only* task is to *simplify* the original problem, or to solve it directly when simplification is either unnecessary or impossible; the Recursion Fairy will magically take care of all the simpler subproblems for you, using Methods That Are None Of Your Business So *Butt Out*.[1] Mathematically sophisticated readers might recognize the Recursion Fairy by its more formal name, the Induction Hypothesis.

There is one mild technical condition that must be satisfied in order for any recursive method to work correctly: There must be no infinite sequence of reductions to 'simpler' and 'simpler' subproblems. Eventually, the recursive reductions must stop with an elementary *base case* that can be solved by some other method; otherwise, the recursive algorithm will loop forever. This finiteness condition is almost always satisfied trivially, but we should always be wary of "obvious" recursive algorithms that actually recurse forever. (All too often, "obvious" is a synonym for "false".)

5.3 Tower of Hanoi

The Tower of Hanoi puzzle was first published by the mathematician François Éduoard Anatole Lucas in 1883, under the pseudonym "N. Claus (de Siam)" (an anagram of "Lucas d'Amiens"). The following year, Henri de Parville described the puzzle with the following remarkable story:[2]

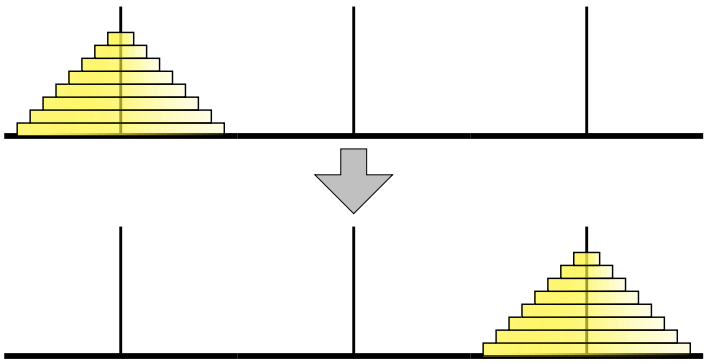
> *In the great temple at Benares beneath the dome which marks the centre of the world, rests a brass plate in which are fixed three diamond needles, each a cubit high and as thick as the body of a bee. On one of these needles, at the creation, God placed sixty-four discs of pure gold, the largest disc resting on the brass plate, and the others getting smaller and smaller up to the top one. This is the Tower of Bramah. Day and night unceasingly the priests transfer the discs from one diamond needle to another according to the fixed and immutable laws of Bramah, which require that the priest on duty must not move more than one disc at a time and that he must place this disc on a needle so that there is no smaller disc below it. When the sixty-four discs shall have been thus transferred from the needle on which at the creation God placed them to one of the other needles, tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.*

Of course, as good computer scientists, our first instinct on reading this story is to substitute the variable n for the hardwired constant 64. And following standard practice (since most physical instances of the puzzle are made of wood instead of diamonds and gold), we will refer to the three possible locations for the disks as "pegs" instead of "needles". How can we move a tower of n disks from one peg to another, using a third peg as an occasional placeholder, without ever placing a disk on top of a smaller disk?

The trick to solving this puzzle is to think recursively. Instead of trying to solve the entire puzzle all at once, let's concentrate on moving just the largest disk. We can't move it at the beginning, because all the other disks are covering it; we have to move those $n-1$ disks to the third peg before we can move the nth disk. And then after we move the nth disk, we have to move those $n-1$ disks back on top of it. So now all we have to figure out is how to...

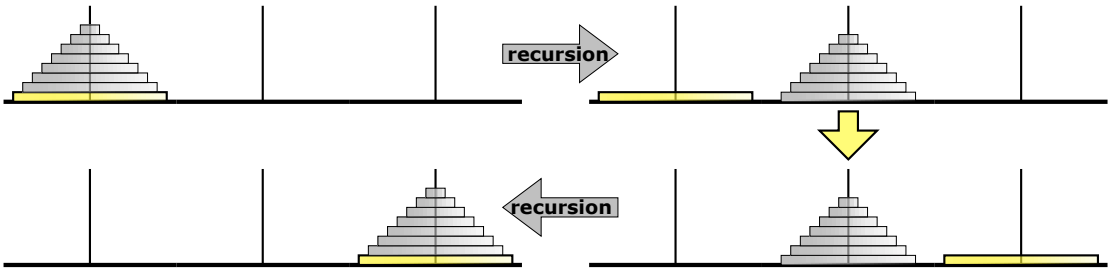
[1] When I was a student, I used to attribute recursion to "elves" instead of the Recursion Fairy, referring to the Brothers Grimm story about an old shoemaker who leaves his work unfinished when he goes to bed, only to discover upon waking that elves ("Wichtelmänner") have finished everything overnight. Someone more entheogenically experienced than I might recognize them as Terence McKenna's "self-transforming machine elves".

[2] This English translation is from W. W. Rouse Ball and H. S. M. Coxeter's book *Mathematical Recreations and Essays*.



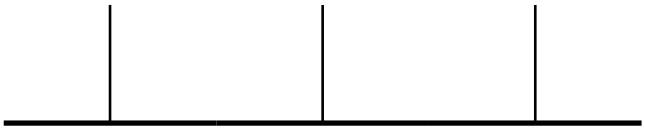
The Tower of Hanoi puzzle

STOP!! That's it! We're done! We've successfully reduced the n-disk Tower of Hanoi problem to two instances of the $(n-1)$-disk Tower of Hanoi problem, which we can gleefully hand off to the Recursion Fairy (or, to carry the original story further, to the junior monks at the temple).



The Tower of Hanoi algorithm; ignore everything but the bottom disk

Our recursive reduction does make one subtle but important assumption: *There is a largest disk*. In other words, our recursive algorithm works for any $n \geq 1$, but it breaks down when $n = 0$. We must handle that base case directly. Fortunately, the monks at Benares, being good Buddhists, are quite adept at moving zero disks from one peg to another in no time at all.



The base case for the Tower of Hanoi algorithm. There is no spoon.

While it's tempting to think about how all those smaller disks get moved—or more generally, what happens when the recursion is unrolled—it's not necessary. For even slightly more complicated algorithms, unrolling the recursion is far more confusing than illuminating. Our ***only*** task is to reduce the problem to one or more simpler instances, or to solve the problem directly if such a reduction is impossible. Our algorithm is trivially correct when $n = 0$. For any $n \geq 1$, the Recursion Fairy correctly moves (or more formally, the inductive hypothesis implies that our recursive algorithm correctly moves) the top $n-1$ disks, so (by induction) our algorithm must be correct.

Here's the recursive Hanoi algorithm in more typical pseudocode. This algorithm moves a stack of n disks from a source peg (*src*) to a destination peg (*dst*) using a third temporary peg (*tmp*) as a placeholder.

```
HANOI(n, src, dst, tmp):
  if n > 0
      HANOI(n − 1, src, tmp, dst)
      move disk n from src to dst
      HANOI(n − 1, tmp, dst, src)
```

Let $T(n)$ denote the number of moves required to transfer n disks—the running time of our algorithm. Our vacuous base case implies that $T(0) = 0$, and the more general recursive algorithm implies that $T(n) = 2T(n-1) + 1$ for any $n \geq 1$. The annihilator method (or guessing and checking by induction) quickly gives us the closed form solution $\boldsymbol{T(n) = 2^n - 1}$. In particular, moving a tower of 64 disks requires $2^{64} - 1 =$ 18,446,744,073,709,551,615 individual moves. Thus, even at the impressive rate of one move per second, the monks at Benares will be at work for approximately 585 billion years before tower, temple, and Brahmins alike will crumble into dust, and with a thunderclap the world will vanish.

5.4 Mergesort

Mergesort is one of the earliest algorithms proposed for sorting. According to Donald Knuth, it was proposed by John von Neumann as early as 1945.

1. Divide the input array into two subarrays of roughly equal size.
2. Recursively mergesort each of the subarrays.
3. Merge the newly-sorted subarrays into a single sorted array.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Divide:	S	O	R	T	I	N	G	E	X	A	M	P	L
Recurse:	I	N	O	S	R	T	A	E	G	L	M	P	X
Merge:	A	E	G	I	L	M	N	O	P	R	S	T	X

A mergesort example.

The first step is completely trivial—we only need to compute the median array index—and we can delegate the second step to the Recursion Fairy. All the real work is done in the final step; the two sorted subarrays can be merged using a simple linear-time algorithm. Here's a complete description of the algorithm; to keep the recursive structure clear, we separate out the merge step as an independent subroutine.

```
MERGESORT(A[1..n]):
  if n > 1
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])
      MERGESORT(A[m+1..n])
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1;  j ← m + 1
  for k ← 1 to n
      if j > n
          B[k] ← A[i];  i ← i + 1
      else if i > m
          B[k] ← A[j];  j ← j + 1
      else if A[i] < A[j]
          B[k] ← A[i];  i ← i + 1
      else
          B[k] ← A[j];  j ← j + 1
  for k ← 1 to n
      A[k] ← B[k]
```

To prove that this algorithm is correct, we apply our old friend induction twice, first to the MERGE subroutine then to the top-level MERGESORT algorithm.

- We prove MERGE is correct by induction on $n - k + 1$, which is the total size of the two sorted subarrays $A[i..m]$ and $A[j..n]$ that remain to be merged into $B[k..n]$ when the kth iteration of the main loop begins. There are five cases to consider. Yes, five.
  - If $k > n$, the algorithm correctly merges the two empty subarrays by doing absolutely nothing. (This is the base case of the inductive proof.)
  - If $i \le m$ and $j > n$, the subarray $A[j..n]$ is empty. Because both subarrays are sorted, the smallest element in the union of the two subarrays is $A[i]$. So the assignment $B[k] \leftarrow A[i]$ is correct. The inductive hypothesis implies that the remaining subarrays $A[i+1..m]$ and $A[j..n]$ are correctly merged into $B[k+1..n]$.
  - Similarly, if $i > m$ and $j \le n$, the assignment $B[k] \leftarrow A[j]$ is correct, and The Recursion Fairy correctly merges—sorry, I mean the inductive hypothesis implies that the MERGE algorithm correctly merges—the remaining subarrays $A[i..m]$ and $A[j+1..n]$ into $B[k+1..n]$.
  - If $i \le m$ and $j \le n$ and $A[i] < A[j]$, then the smallest remaining element is $A[i]$. So $B[k]$ is assigned correctly, and the Recursion Fairy correctly merges the rest of the subarrays.
  - Finally, if $i \le m$ and $j \le n$ and $A[i] \ge A[j]$, then the smallest remaining element is $A[j]$. So $B[k]$ is assigned correctly, and the Recursion Fairy correctly does the rest.
- Now we prove MERGESORT correct by induction; there are two cases to consider. Yes, two.
  - If $n \le 1$, the algorithm correctly does nothing.
  - Otherwise, the Recursion Fairy correctly sorts—sorry, I mean the induction hypothesis implies that our algorithm correctly sorts—the two smaller subarrays $A[1..m]$ and $A[m+1..n]$, after which they are correctly MERGEd into a single sorted array (by the previous argument).

What's the running time? Because the MERGESORT algorithm is recursive, its running time will be expressed by a recurrence. MERGE clearly takes linear time, because it's a simple for-loop with constant work per iteration. We immediately obtain the following recurrence for MERGESORT:

$$T(n) = T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + O(n).$$

As in most divide-and-conquer recurrences, we can safely strip out the floors and ceilings using a domain transformation,[3] giving us the simpler recurrence

$$T(n) = 2T(n/2) + O(n).$$

The "all levels equal" case of the recursion tree method now immediately implies the closed-form solution $\boldsymbol{T(n) = O(n \log n)}$. (Recursion trees and domain transformations are described in detail in a separate note on solving recurrences.)

5.5 Quicksort

Quicksort is another recursive sorting algorithm, discovered by Tony Hoare in 1962. In this algorithm, the hard work is splitting the array into subsets so that merging the final result is trivial.

1. Choose a *pivot* element from the array.
2. Partition the array into three subarrays containing the elements smaller than the pivot, the pivot element itself, and the elements larger than the pivot.

[3] See the course notes on solving recurrences for more details.

3. Recursively quicksort the first and last subarray.

Input:	S	O	R	T	I	N	G	E	X	A	M	P	L
Choose a pivot:	S	O	R	T	I	N	G	E	X	A	M	P	L
Partition:	A	G	E	I	L	N	R	O	X	S	M	P	T
Recurse:	A	E	G	I	L	M	N	O	P	R	S	T	X

A quicksort example.

Here's a more detailed description of the algorithm. In the separate PARTITION subroutine, the input parameter p is index of the pivot element in the unsorted array; the subroutine partitions the array and returns the new index of the pivot.

```
QUICKSORT(A[1..n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← PARTITION(A, p)
      QUICKSORT(A[1..r−1])
      QUICKSORT(A[r+1..n])
```

```
PARTITION(A[1..n], p):
  swap A[p] ↔ A[n]
  i ← 0
  j ← n
  while (i < j)
      repeat i ← i+1 until (i ≥ j or A[i] ≥ A[n])
      repeat j ← j−1 until (i ≥ j or A[j] ≤ A[n])
      if (i < j)
          swap A[i] ↔ A[j]
  swap A[i] ↔ A[n]
  return i
```

Just like mergesort, proving QUICKSORT is correct requires two separate induction proofs: one to prove that PARTITION correctly partitions the array, and the other to prove that QUICKSORT correctly sorts *assuming* PARTITION is correct. I'll leave the gory details as an exercise for the reader.

The analysis is also similar to mergesort. PARTITION runs in $O(n)$ time: $j-i=n$ at the beginning, $j-i=0$ at the end, and we do a constant amount of work each time we increment i or decrement j. For QUICKSORT, we get a recurrence that depends on r, the *rank* of the chosen pivot element:

$$T(n) = T(r-1) + T(n-r) + O(n)$$

If we could somehow choose the pivot to be the *median* element of the array A, we would have $r = \lceil n/2 \rceil$, the two subproblems would be as close to the same size as possible, the recurrence would become

$$T(n) = 2T\big(\lceil n/2 \rceil - 1\big) + T\big(\lfloor n/2 \rfloor\big) + O(n) \le 2T(n/2) + O(n),$$

and we'd have $T(n) = O(n \log n)$ by the recursion tree method.

In fact, as we will see shortly, we *can* locate the median element in an unsorted array in linear time. However, the algorithm is fairly complicated, and the hidden constant in the $O(\cdot)$ notation is large enough to make the resulting sorting algorithm impractical. In practice, most programmers settle for something simple, like choosing the first or last element of the array. In this case, r take any value between 1 and n, so we have

$$T(n) = \max_{1 \le r \le n} \big(T(r-1) + T(n-r) + O(n)\big).$$

In the worst case, the two subproblems are completely unbalanced—either $r = 1$ or $r = n$—and the recurrence becomes $T(n) \le T(n-1) + O(n)$. The solution is $\boldsymbol{T(n) = O(n^2)}$.

Another common heuristic is called "median of three"—choose three elements (usually at the beginning, middle, and end of the array), and take the median of those three elements the pivot.

Although this heuristic is somewhat more efficient in practice than just choosing one element, especially when the array is already (nearly) sorted, we can still have $r = 2$ or $r = n-1$ in the worst case. With the median-of-three heuristic, the recurrence becomes $T(n) \leq T(1) + T(n-2) + O(n)$, whose solution is still $T(n) = O(n^2)$.

Intuitively, the pivot element will 'usually' fall somewhere in the middle of the array, say between $n/10$ and $9n/10$. This observation suggests that the *average-case* running time is $O(n \log n)$. Although this intuition is actually correct (at least under the right formal assumptions), we are still far from a *proof* that quicksort is usually efficient. We will formalize this intuition about average-case behavior in a later lecture.

5.6 The Pattern

Both mergesort and and quicksort follow a general three-step pattern shared by all divide and conquer algorithms:

1. **Divide** the given instance of the problem into several *independent smaller* instances.
2. **Delegate** each smaller instance to the Recursion Fairy.
3. **Combine** the solutions for the smaller instances into the final solution for the given instance.

If the size of any subproblem falls below some constant threshold, the recursion bottoms out. Hopefully, at that point, the problem is trivial, but if not, we switch to a different algorithm instead.

Proving a divide-and-conquer algorithm correct almost always requires induction. Analyzing the running time requires setting up and solving a recurrence, which usually (but unfortunately not always!) can be solved using recursion trees, perhaps after a simple domain transformation.

5.7 Median Selection

So how *do* we find the median element of an array in linear time? The following algorithm was discovered by Manuel Blum, Bob Floyd, Vaughan Pratt, Ron Rivest, and Bob Tarjan in the early 1970s. Their algorithm actually solves the more general problem of selecting the kth largest element in an n-element array, given the array and the integer g as input, using a variant of an algorithm called either "quickselect" or "one-armed quicksort". The basic quickselect algorithm chooses a pivot element, partitions the array using the PARTITION subroutine from QUICKSORT, and then recursively searches only *one* of the two subarrays.

```
QUICKSELECT(A[1..n], k):
  if n = 1
      return A[1]
  else
      Choose a pivot element A[p]
      r ← PARTITION(A[1..n], p)

      if k < r
          return QUICKSELECT(A[1..r-1], k)
      else if k > r
          return QUICKSELECT(A[r+1..n], k-r)
      else
          return A[r]
```

The worst-case running time of QUICKSELECT obeys a recurrence similar to the quicksort recurrence. We don't know the value of r or which subarray we'll recursively search, so we'll just assume the worst.

$$T(n) \leq \max_{1 \leq r \leq n} (\max\{T(r-1), T(n-r)\} + O(n))$$

We can simplify the recurrence by using ℓ to denote the length of the recursive subproblem:

$$T(n) \leq \max_{0 \leq \ell \leq n-1} T(\ell) + O(n) \leq T(n-1) + O(n)$$

As with quicksort, we get the solution $T(n) = O(n^2)$ when $\ell = n-1$, which happens when the chosen pivot element is either the smallest element or largest element of the array.

On the other hand, we could avoid this quadratic behavior if we could somehow magically choose a *good* pivot, where $\ell \leq \alpha n$ for some constant $\alpha < 1$. In this case, the recurrence would simplify to

$$T(n) \leq T(\alpha n) + O(n).$$

This recurrence expands into a descending geometric series, which is dominated by its largest term, so $T(n) = O(n)$.

The Blum-Floyd-Pratt-Rivest-Tarjan algorithm chooses a good pivot for one-armed quicksort by *recursively computing the median* of a carefully-selected subset of the input array.

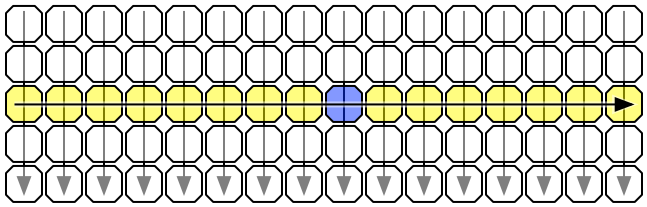
```
MOM5SELECT(A[1..n], k):
  if n ≤ 25
      use brute force
  else
      m ← ⌈n/5⌉
      for i ← 1 to m
          M[i] ← MEDIANOFFIVE(A[5i−4..5i])   ⟨⟨Brute force!⟩⟩
      mom ← MOMSELECT(M[1..m], ⌊m/2⌋)        ⟨⟨Recursion!⟩⟩

      r ← PARTITION(A[1..n], mom)

      if k < r
          return MOMSELECT(A[1..r−1], k)     ⟨⟨Recursion!⟩⟩
      else if k > r
          return MOMSELECT(A[r+1..n], k−r)   ⟨⟨Recursion!⟩⟩
      else
          return mom
```

The recursive structure of the algorithm requires a slightly larger base case. There's absolutely nothing special about the constant 25 in the pseudocode; for theoretical purposes, any other constant like 42 or 666 or 8765309 would work just as well.

If the input array is too large to handle by brute force, we divide it into $\lceil n/5 \rceil$ *blocks*, each containing exactly 5 elements, except possibly the last. (If the last block isn't full, just throw in a few ∞s.) We find the median of each block by brute force and collect those medians into a new array $M[1..\lceil n/5 \rceil]$. Then we recursively compute the median of this new array. Finally we use the median of medians — hence 'mom' — as the pivot in one-armed quicksort.

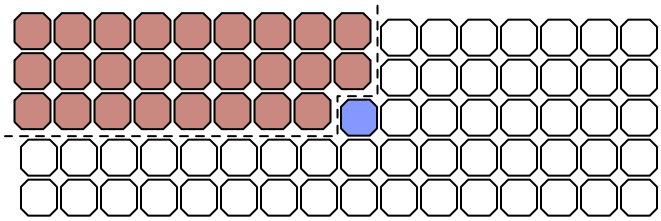
The key insight is that neither of these two subarrays can be too large. The median of medians is larger than $\lceil \lceil n/5 \rceil / 2 \rceil - 1 \approx n/10$ block medians, and each of those medians is larger than two other elements in its block. Thus, mom is larger than at least $3n/10$ elements in the input array, and symmetrically, mom is smaller than at least $3n/10$ input elements. Thus, in the worst case, the final recursive call searches an array of size $7n/10$.

We can visualize the algorithm's behavior by drawing the input array as a $5 \times \lceil n/5 \rceil$ grid, which each column represents five consecutive elements. For purposes of illustration, imagine that we sort every column from top down, and then we sort the columns by their middle element. (Let me emphasize that *the algorithm does not actually do this!*) In this arrangement, the median-of-medians is the element closest to the center of the grid.



Visualizing the median of medians

The left half of the first three rows of the grid contains $3n/10$ elements, each of which is smaller than the median-of-medians. If the element we're looking for is larger than the median-of-medians, our algorithm will throw away *everything* smaller than the median-of-median, including those $3n/10$ elements, before recursing. Thus, the input to the recursive subproblem contains at most $7n/10$ elements. A symmetric argument applies when our target element is smaller than the median-of-medians.



Discarding approximately 3/10 of the array

We conclude that the worst-case running time of the algorithm obeys the following recurrence:

$$T(n) \leq O(n) + T(n/5) + T(7n/10).$$

The recursion tree method implies the solution $T(n) = O(n)$.

Finer analysis reveals that the constant hidden by the $O()$ is quite large, even if we count only comparisons; this is not a practical algorithm for small inputs. (In particular, mergesort uses fewer comparisons in the worst case when $n < 4{,}000{,}000$.) Selecting the median of 5 elements requires at most 6 comparisons, so we need at most $6n/5$ comparisons to set up the recursive subproblem. We need another $n-1$ comparisons to partition the array after the recursive call returns. So a more accurate recurrence for the total number of comparisons is

$$T(n) \leq 11n/5 + T(n/5) + T(7n/10).$$

The recursion tree method implies the upper bound

$$T(n) \leq \frac{11n}{5} \sum_{i \geq 0} \left(\frac{9}{10}\right)^i = \frac{11n}{5} \cdot 10 = 22n.$$

5.8 Multiplication

Adding two n-digit numbers takes $O(n)$ time by the standard iterative 'ripple-carry' algorithm, using a lookup table for each one-digit addition. Similarly, multiplying an n-digit number by a one-digit number takes $O(n)$ time, using essentially the same algorithm.

What about multiplying two n-digit numbers? In most of the world, grade school students (supposedly) learn to multiply by breaking the problem into n one-digit multiplications and n additions:

```
   31415962
 × 27182818
 ----------
  251327696
   31415962
  251327696
  62831924
 251327696
  31415962
 219911734
 62831924
 ----------
 853974377340916
```

We could easily formalize this algorithm as a pair of nested for-loops. The algorithm runs in $\Theta(n^2)$ time—altogether, there are $\Theta(n^2)$ digits in the partial products, and for each digit, we spend constant time. The Egyptian/Russian peasant multiplication algorithm described in the first lecture also runs in $\Theta(n^2)$ time.

Perhaps we can get a more efficient algorithm by exploiting the following identity:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

Here is a divide-and-conquer algorithm that computes the product of two n-digit numbers x and y, based on this formula. Each of the four sub-products e, f, g, h is computed recursively. The last line does not involve any multiplications, however; to multiply by a power of ten, we just shift the digits and fill in the right number of zeros.

```
MULTIPLY(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m
      d ← ⌊y/10^m⌋;  c ← y mod 10^m
      e ← MULTIPLY(a, c, m)
      f ← MULTIPLY(b, d, m)
      g ← MULTIPLY(b, c, m)
      h ← MULTIPLY(a, d, m)
      return 10^(2m) e + 10^m (g + h) + f
```

You can easily prove by induction that this algorithm is correct. The running time for this algorithm is given by the recurrence

$$T(n) = 4T(\lceil n/2 \rceil) + \Theta(n), \qquad T(1) = 1,$$

which solves to $T(n) = \Theta(n^2)$ by the recursion tree method (after a simple domain transformation). Hmm... I guess this didn't help after all.

In the mid-1950s, the famous Russian mathematician Andrey Kolmogorov conjectured that there is *no* algorithm to multiply two n-digit numbers in $o(n^2)$ time. However, in 1960, after Kolmogorov posed his conjecture at a seminar at Moscow University, Anatoliĭ Karatsuba, one of the students in the seminar, discovered a remarkable counterexample. According to Karastuba himself,

> After the seminar I told Kolmogorov about the new algorithm and about the disproof of the n^2 conjecture. Kolmogorov was very agitated because this contradicted his very plausible conjecture. At the next meeting of the seminar, Kolmogorov himself told the participants about my method, and at that point the seminar was terminated.

Karastuba observed that the middle coefficient $bc + ad$ can be computed from the other two coefficients ac and bd using only *one* more recursive multiplication, via the following algebraic identity:

$$ac + bd - (a-b)(c-d) = bc + ad$$

This trick lets us replace the last three lines in the previous algorithm as follows:

```
FastMultiply(x, y, n):
  if n = 1
      return x · y
  else
      m ← ⌈n/2⌉
      a ← ⌊x/10^m⌋;  b ← x mod 10^m
      d ← ⌊y/10^m⌋;  c ← y mod 10^m
      e ← FastMultiply(a, c, m)
      f ← FastMultiply(b, d, m)
      g ← FastMultiply(a − b, c − d, m)
      return 10^(2m) e + 10^m (e + f − g) + f
```

The running time of Karatsuba's FastMultiply algorithm is given by the recurrence

$$T(n) \le 3T(\lceil n/2 \rceil) + O(n), \qquad T(1) = 1.$$

After a domain transformation, we can plug this into a recursion tree to get the solution $T(n) = O(n^{\lg 3}) = O(n^{1.585})$, a significant improvement over our earlier quadratic-time algorithm.[4] Karastuba's algorithm arguably launched the design and analysis of algorithms as a formal field of study.

Of course, in practice, all this is done in binary instead of decimal.

We can take this idea even further, splitting the numbers into more pieces and combining them in more complicated ways, to obtain even faster multiplication algorithms. Andrei Toom and Stephen Cook discovered an infinite family of algorithms that split any integer into k parts, each with n/k digits, and then compute the product using only $2k-1$ recursive multiplications. For any fixed k, the resulting algorithm runs in $O(n^{1+1/(\lg k)})$ time, where the hidden constant in the $O(\cdot)$ notation depends on k.

Ultimately, this divide-and-conquer strategy led Gauss (yes, really) to the discovery of the *Fast Fourier transform*, which we discuss in detail in the next lecture note. The fastest multiplication algorithm known, published by Martin Fürer in 2007 and based on FFTs, runs in $n \log n 2^{O(\log^* n)}$ time. Here, $\log^* n$ is the slowly growing *iterated logarithm* of n, which is the number of times one must take the logarithm of n before the value is less than 1:

$$\lg^* n = \begin{cases} 1 & \text{if } n \le 2, \\ 1 + \lg^*(\lg n) & \text{otherwise.} \end{cases}$$

[4] Karatsuba actually proposed an algorithm based on the formula $(a+c)(b+d) - ac - bd = bc + ad$. This algorithm also runs in $O(n^{\lg 3})$ time, but the actual recurrence is a bit messier: $a-b$ and $c-d$ are still m-digit numbers, but $a+b$ and $c+d$ might have $m+1$ digits. The simplification presented here is due to Donald Knuth. The same technique was used by Gauss in the 1800s to multiply two complex numbers using only three real mutliplications.

(For all practical purposes, $\log^* n \le 6$.) It is widely conjectured that the best possible algorithm for multiply two n-digit numbers runs in $\Theta(n \log n)$ time.

5.9 Exponentiation

Given a number a and a positive integer n, suppose we want to compute a^n. The standard naïve method is a simple for-loop that does $n-1$ multiplications by a:

```
SLOWPOWER(a, n):
  x ← a
  for i ← 2 to n
      x ← x · a
  return x
```

This iterative algorithm requires n multiplications.

Notice that the input a could be an integer, or a rational, or a floating point number. In fact, it doesn't need to be a number at all, as long as it's something that we know how to multiply. For example, the same algorithm can be used to compute powers modulo some finite number (an operation commonly used in cryptography algorithms) or to compute powers of matrices (an operation used to evaluate recurrences and to compute shortest paths in graphs). All we really require is that a belong to a multiplicative group.[5] Since we don't know what kind of things we're multiplying, we can't know how long a multiplication takes, so we're forced analyze the running time in terms of the number of multiplications.

There is a much faster divide-and-conquer method, using the following simple recursive formula:

$$a^n = a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil}.$$

What makes this approach more efficient is that once we compute the first factor $a^{\lfloor n/2 \rfloor}$, we can compute the second factor $a^{\lceil n/2 \rceil}$ using at most one more multiplication.

```
FASTPOWER(a, n):
  if n = 1
      return a
  else
      x ← FASTPOWER(a, ⌊n/2⌋)
      if n is even
          return x · x
      else
          return x · x · a
```

The total number of multiplications satisfies the recurrence $T(n) \le T(\lfloor n/2 \rfloor) + 2$, with the base case $T(1) = 0$. After a domain transformation, recursion trees give us the solution $T(n) = O(\log n)$.

Incidentally, this algorithm is asymptotically optimal—any algorithm for computing a^n must perform at least $\Omega(\log n)$ multiplications. In fact, when n is a power of two, this algorithm is *exactly* optimal. However, there are slightly faster methods for other values of n. For example, our divide-and-conquer algorithm computes a^{15} in six multiplications ($a^{15} = a^7 \cdot a^7 \cdot a$; $a^7 = a^3 \cdot a^3 \cdot a$; $a^3 = a \cdot a \cdot a$), but only five multiplications are necessary ($a \to a^2 \to a^3 \to a^5 \to a^{10} \to a^{15}$). It is an open question whether the absolute minimum number of multiplications for a given exponent n can be computed efficiently.

[5] A *multiplicative group* $(G, \otimes)$ is a set G and a function $\otimes : G \times G \to G$, satisfying three axioms:

1. There is a *unit* element $1 \in G$ such that $1 \otimes g = g \otimes 1$ for any element $g \in G$.
2. Any element $g \in G$ has a *inverse* element $g^{-1} \in G$ such that $g \otimes g^{-1} = g^{-1} \otimes g = 1$
3. The function is *associative*: for any elements $f, g, h \in G$, we have $f \otimes (g \otimes h) = (f \otimes g) \otimes h$.

Exercises

1. Prove that the Russian peasant multiplication algorithm runs in $\Theta(n^2)$ time, where n is the total number of input digits.

2. (a) Professor George O'Jungle has a 27-node binary tree, in which every node is labeled with a unique letter of the Roman alphabet or the character **&**. Preorder and postorder traversals of the tree visit the nodes in the following order:
   - Preorder: **I Q J H L E M V O T S B R G Y Z K C A & F P N U D W X**
   - Postorder: **H E M L J V Q S G Y R Z B T C P U D N F W & X A K O I**

   Draw George's binary tree.

   (b) Prove that there is no algorithm to reconstruct an *arbtirary* binary tree from its preorder and postorder node sequences.

   (c) Recall that a binary tree is *full* if every non-leaf node has exactly two children. Describe and analyze a recursive algorithm to reconstruct an arbitrary *full* binary tree, given its preorder and postorder node sequences as input.

   (d) Describe and analyze a recursive algorithm to reconstruct an arbtirary binary tree, given its preorder and *inorder* node sequences as input.

   (e) Describe and analyze a recursive algorithm to reconstruct an arbitrary binary *search* tree, given only its preorder node sequence. Assume all input keys are distinct. For extra credit, describe an algorithm that runs in $O(n)$ time.

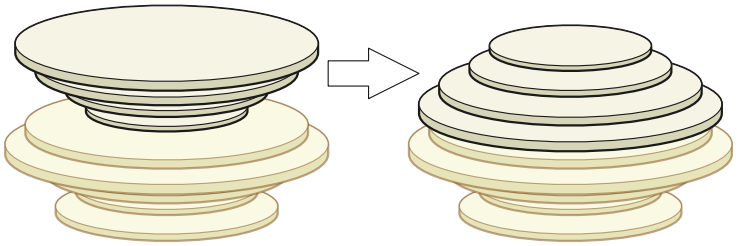
   In parts (b), (c), and (d), assume that all keys are distinct and that the input is consistent with at least one binary tree.

3. Consider a $2^n \times 2^n$ chessboard with one (arbitrarily chosen) square removed.

   (a) Prove that any such chessboard can be tiled without gaps or overlaps by L-shaped pieces, each composed of 3 squares.

   (b) Describe and analyze an algorithm to compute such a tiling, given the integer n and two n-bit integers representing the row and column of the missing square. The output is a list of the positions and orientations of $(4^n - 1)/3$ tiles. Your algorithm should run in $O(4^n)$ time.

4. Suppose you are given a stack of n pancakes of different sizes. You want to sort the pancakes so that smaller pancakes are on top of larger pancakes. The only operation you can perform is a *flip*—insert a spatula under the top k pancakes, for some integer k between 1 and n, and flip them all over.



Flipping the top four pancakes.

(a) Describe an algorithm to sort an arbitrary stack of n pancakes using as few flips as possible. *Exactly* how many flips does your algorithm perform in the worst case?

(b) Now suppose one side of each pancake is burned. Describe an algorithm to sort an arbitrary stack of n pancakes, so that the burned side of every pancake is facing down, using as few flips as possible. *Exactly* how many flips does your algorithm perform in the worst case?

5. Prove that the original recursive Tower of Hanoi algorithm is *exactly equivalent* to each of the following non-recursive algorithms. In other words, prove that all three algorithms move exactly the same sequence of disks, to and from the same pegs, in the same order. The pegs are labeled 0, 1, and 2, and our problem is to move a stack of n disks from peg 0 to peg 2 (as shown on page 3).

   (a) Repeatedly make the only legal move that satisfies the following constraints:
   - Never move the same disk twice in a row.
   - If n is even, always move the smallest disk forward ($\cdots \to 0 \to 1 \to 2 \to 0 \to \cdots$).
   - If n is odd, always move the smallest disk backward ($\cdots \to 0 \to 2 \to 1 \to 0 \to \cdots$).

   If there is no move that satisfies these three constraints, the puzzle is solved.

   (b) Start by putting your finger on the top of peg 0. Then repeat the following steps:
   i. If n is odd, move your finger to the next peg ($\cdots \to 0 \to 1 \to 2 \to 0 \to \cdots$).
   ii. If n is even, move your finger to the previous peg ($\cdots \to 0 \to 2 \to 1 \to 0 \to \cdots$).
   iii. Make the only legal move that does not require you to lift your finger. If there is no such move, the puzzle is solved.

   (c) Let $\rho(n)$ denote the smallest integer k such that $n/2^k$ is not an integer. For example, $\rho(42) = 2$, because $42/2^1$ is an integer but $42/2^2$ is not. (Equivalently, $\rho(n)$ is one more than the position of the least significant 1 in the binary representation of n.) Because its behavior resembles the marks on a ruler, $\rho(n)$ is sometimes called the *ruler function*:

   $$1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 5, 1, 2, 1, 3, 1, 2, 1, 4, 1, 2, 1, 3, 1, 2, 1, 6, 1, 2, 1, 3, 1, \ldots$$

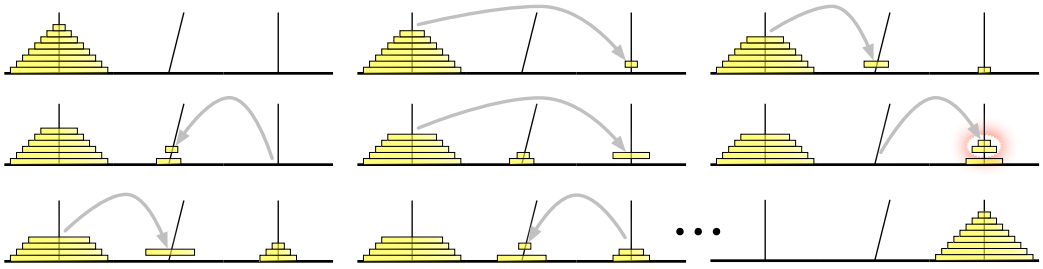
   Here's the non-recursive algorithm in one line:

   In step i, move disk $\rho(i)$ forward if $n - i$ is even, backward if $n - i$ is odd.

   When this rule requires us to move disk $n + 1$, the puzzle is solved.

6. A less familiar chapter in the Tower of Hanoi's history is its brief relocation of the temple from Benares to Pisa in the early 13th century. The relocation was organized by the wealthy merchant-mathematician Leonardo Fibonacci, at the request of the Holy Roman Emperor Frederick II, who had heard reports of the temple from soldiers returning from the Crusades. The Towers of Pisa and their attendant monks became famous, helping to establish Pisa as a dominant trading center on the Italian peninsula.

   Unfortunately, almost as soon as the temple was moved, one of the diamond needles began to lean to one side. To avoid the possibility of the leaning tower falling over from too much use, Fibonacci convinced the priests to adopt a more relaxed rule: ***Any number of disks on the leaning needle can be moved together to another needle in a single move.*** It was still forbidden to place a larger disk on top of a smaller disk, and disks had to be moved one at a time *onto* the leaning needle or between the two vertical needles.



The Towers of Pisa. In the fifth move, two disks are taken off the leaning needle.

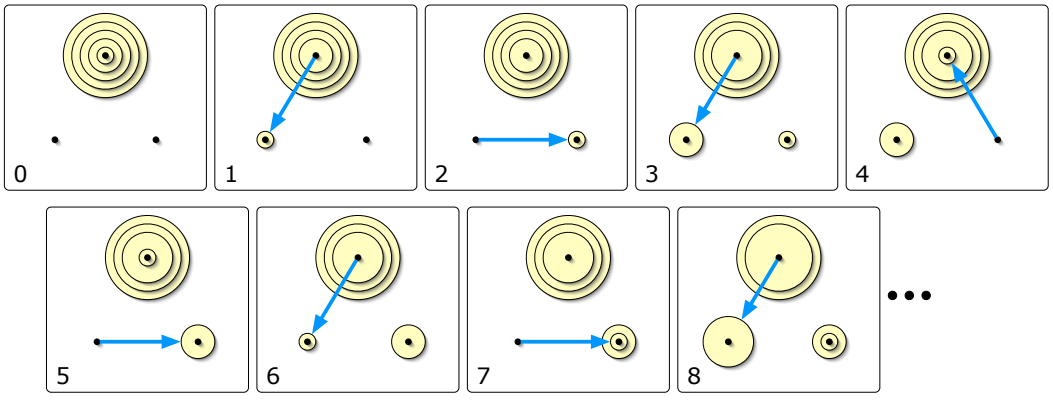
Thanks to Fibonacci's new rule, the priests could bring about the end of the universe somewhat faster from Pisa then they could than could from Benares. Fortunately, the temple was moved from Pisa back to Benares after the newly crowned Pope Gregory IX excommunicated Frederick II, making the local priests less sympathetic to hosting foreign heretics with strange mathematical habits. Soon afterward, a bell tower was erected on the spot where the temple once stood; it too began to lean almost immediately.

Describe an algorithm to transfer a stack of n disks from one *vertical* needle to the other *vertical* needle, using the smallest possible number of moves. *Exactly* how many moves does your algorithm perform?

7. Consider the following restricted variants of the Tower of Hanoi puzzle. In each problem, the pegs are numbered 0, 1, and 2, as in problem 5, and your task is to move a stack of n disks from peg 1 to peg 2.

   (a) Suppose you are forbidden to move any disk directly between peg 1 and peg 2; *every* move must involve peg 0. Describe an algorithm to solve this version of the puzzle in as few moves as possible. *Exactly* how many moves does your algorithm make?

   (b) Suppose you are only allowed to move disks from peg 0 to peg 2, from peg 2 to peg 1, or from peg 1 to peg 0. Equivalently, suppose the pegs are arranged in a circle and numbered in clockwise order, and you are only allowed to move disks counterclockwise. Describe an algorithm to solve this version of the puzzle in as few moves as possible. How many moves does your algorithm make?



A top view of the first eight moves in a counterclockwise Towers of Hanoi solution

★(c) Finally, suppose your only restriction is that you may never move a disk directly from peg 1 to peg 2. Describe an algorithm to solve this version of the puzzle in as few moves as possible.

How many moves does your algorithm make? *[Hint: This variant is considerably harder to analyze than the other two.]*

8. A German mathematician developed a new variant of the Towers of Hanoi game, known in the US literature as the "Liberty Towers" game.[6] In this variant, there is a row of k pegs, numbered from 1 to k. In a single turn, you are allowed to move the smallest disk on peg i to either peg $i-1$ or peg $i+1$, for any index i; as usual, you are not allowed to place a bigger disk on a smaller disk. Your mission is to move a stack of n disks from peg 1 to peg k.

   (a) Describe a recursive algorithm for the case $k = 3$. *Exactly* how many moves does your algorithm make? (This is the same as part (a) of the previous question.)

   (b) Describe a recursive algorithm for the case $k = n + 1$ that requires at most $O(n^3)$ moves. *[Hint: Use part (a).]*

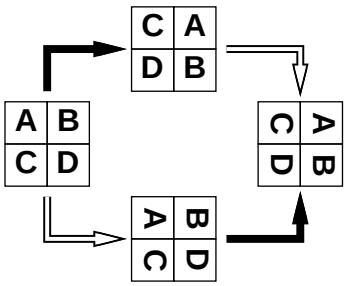
   (c) Describe a recursive algorithm for the case $k = n + 1$ that requires at most $O(n^2)$ moves. *[Hint: Don't use part (a).]*

   (d) Describe a recursive algorithm for the case $k = \sqrt{n}$ that requires at most a polynomial number of moves. (What polynomial??)

   *(e) Describe and analyze a recursive algorithm for arbitrary n and k. How small must k be (as a function of n) so that the number of moves is bounded by a polynomial in n?

9. Most graphics hardware includes support for a low-level operation called *blit*, or **bl**ock **t**ransfer, which quickly copies a rectangular chunk of a pixel map (a two-dimensional array of pixel values) from one location to another. This is a two-dimensional version of the standard C library function `memcpy()`.

   Suppose we want to rotate an $n \times n$ pixel map 90° clockwise. One way to do this, at least when n is a power of two, is to split the pixel map into four $n/2 \times n/2$ blocks, move each block to its proper position using a sequence of five blits, and then recursively rotate each block. (Why five? For the same reason the Tower of Hanoi puzzle needs a third peg.) Alternately, we could *first* recursively rotate the blocks and *then* blit them into place.

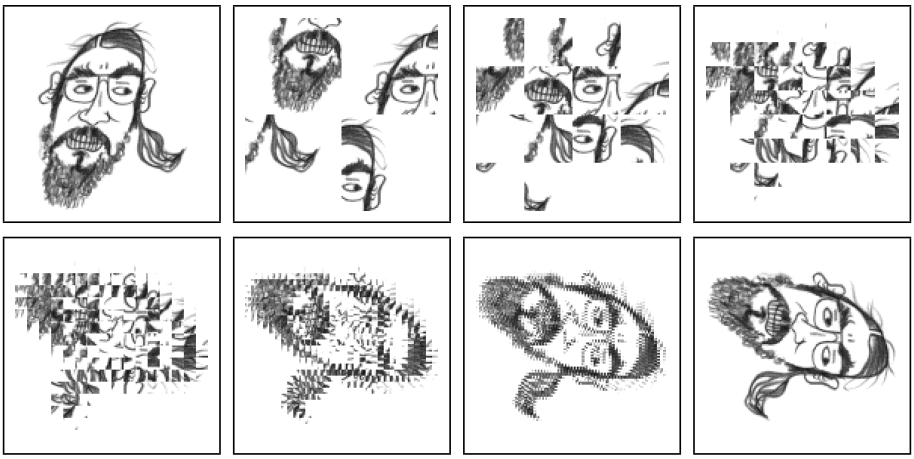


Two algorithms for rotating a pixel map.
Black arrows indicate blitting the blocks into place; white arrows indicate recursively rotating the blocks.

   (a) Prove that both versions of the algorithm are correct when n is a power of 2.

   (b) *Exactly* how many blits does the algorithm perform when n is a power of 2?

[6] No it isn't.



The first rotation algorithm (blit then recurse) in action.

(c) Describe how to modify the algorithm so that it works for arbitrary n, not just powers of 2. How many blits does your modified algorithm perform?

(d) What is your algorithm's running time if a $k \times k$ blit takes $O(k^2)$ time?

(e) What if a $k \times k$ blit takes only $O(k)$ time?

10. Prove that quicksort with the median-of-three heuristic requires $\Omega(n^2)$ time to sort an array of size n in the worst case. Specifically, for any integer n, describe a permutation of the integers 1 through n, such that in every recursive call to median-of-three-quicksort, the pivot is always the second smallest element of the array. Designing this permutation requires intimate knowledge of the PARTITION subroutine.

(a) As a warm-up exercise, assume that the PARTITION subroutine is *stable*, meaning it preserves the existing order of all elements smaller than the pivot, and it preserves the existing order of all elements smaller than the pivot.

(b) Assume that the PARTITION subroutine uses the specific algorithm listed on page 6 of this lecture note, which is *not* stable.

11. (a) Prove that the following algorithm actually sorts its input.

```
STOOGESORT(A[0..n-1]):
  if n = 2 and A[0] > A[1]
      swap A[0] ↔ A[1]
  else if n > 2
      m = ⌈2n/3⌉
      STOOGESORT(A[0..m-1])
      STOOGESORT(A[n-m..n-1])
      STOOGESORT(A[0..m-1])
```

(b) Would STOOGESORT still sort correctly if we replaced $m = \lceil 2n/3 \rceil$ with $m = \lfloor 2n/3 \rfloor$? Justify your answer.

(c) State a recurrence (including the base case(s)) for the number of comparisons executed by STOOGESORT.

(d) Solve the recurrence, and prove that your solution is correct. *[Hint: Ignore the ceiling.]*

(e) Prove that the number of *swaps* executed by StoogeSort is at most $\binom{n}{2}$.

12. Consider the following cruel and unusual sorting algorithm.

```
Cruel(A[1..n]):
  if n > 1
      Cruel(A[1..n/2])
      Cruel(A[n/2+1..n])
      Unusual(A[1..n])
```

```
Unusual(A[1..n]):
  if n = 2
      if A[1] > A[2]                          ⟨⟨the only comparison!⟩⟩
          swap A[1] ↔ A[2]
  else
      for i ← 1 to n/4                        ⟨⟨swap 2nd and 3rd quarters⟩⟩
          swap A[i + n/4] ↔ A[i + n/2]
      Unusual(A[1..n/2])                      ⟨⟨recurse on left half⟩⟩
      Unusual(A[n/2+1..n])                    ⟨⟨recurse on right half⟩⟩
      Unusual(A[n/4+1..3n/4])                 ⟨⟨recurse on middle half⟩⟩
```

Notice that the comparisons performed by the algorithm do not depend at all on the values in the input array; such a sorting algorithm is called ***oblivious***. Assume for this problem that the input size n is always a power of 2.

(a) Prove by induction that Cruel correctly sorts any input array. *[Hint: Consider an array that contains $n/4$ 1s, $n/4$ 2s, $n/4$ 3s, and $n/4$ 4s. Why is this special case enough?]*

(b) Prove that Cruel would *not* correctly sort if we removed the for-loop from Unusual.

(c) Prove that Cruel would *not* correctly sort if we swapped the last two lines of Unusual.

(d) What is the running time of Unusual? Justify your answer.

(e) What is the running time of Cruel? Justify your answer.

13. You are a visitor at a political convention (or perhaps a faculty meeting) with n delegates; each delegate is a member of exactly one political party. It is impossible to tell which political party any delegate belongs to; in particular, you will be summarily ejected from the convention if you ask. However, you can determine whether any pair of delegates belong to the *same* party or not simply by introducing them to each other—members of the same party always greet each other with smiles and friendly handshakes; members of different parties always greet each other with angry stares and insults.

(a) Suppose more than half of the delegates belong to the same political party. Describe an efficient algorithm that identifies all members of this majority party.

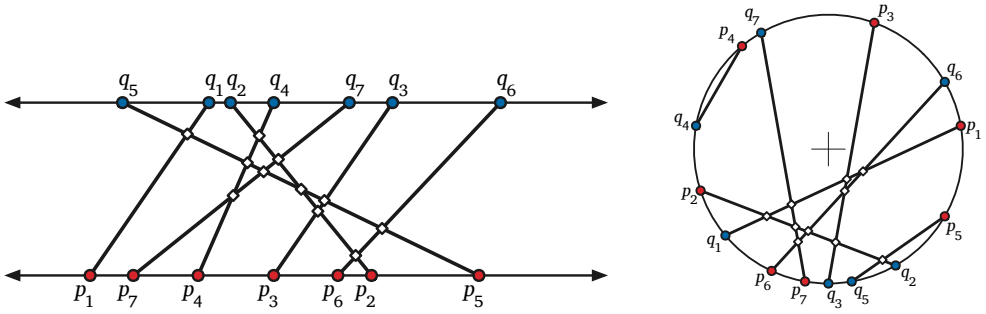
(b) Now suppose exactly k political parties are represented at the convention and one party has a *plurality*: more delegates belong to that party than to any other. Present a practical procedure to pick out the people from the plurality political party as parsimoniously as possible. (Please.)

14. An *inversion* in an array $A[1..n]$ is a pair of indices (i, j) such that $i < j$ and $A[i] > A[j]$. The number of inversions in an n-element array is between 0 (if the array is sorted) and $\binom{n}{2}$ (if the array is sorted backward). Describe and analyze an algorithm to count the number of inversions in an n-element array in $O(n \log n)$ time. *[Hint: Modify mergesort.]*

15. (a) Suppose you are given two sets of n points, one set $\{p_1, p_2, \ldots, p_n\}$ on the line $y = 0$ and the other set $\{q_1, q_2, \ldots, q_n\}$ on the line $y = 1$. Create a set of n line segments by connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect, in $O(n \log n)$ time.

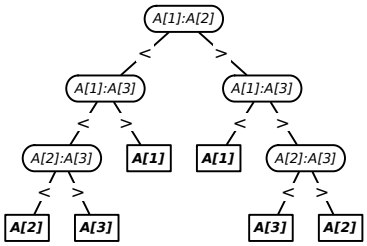
    (b) Now suppose you are given two sets $\{p_1, p_2, \ldots, p_n\}$ and $\{q_1, q_2, \ldots, q_n\}$ of n points *on the unit circle*. Connect each point p_i to the corresponding point q_i. Describe and analyze a divide-and-conquer algorithm to determine how many pairs of these line segments intersect in $O(n \log^2 n)$ time. *[Hint: Use your solution to part (a).]*

    (c) Solve the previous problem in $O(n \log n)$ time.



Eleven intersecting pairs of segments with endpoints on parallel lines, and ten intersecting pairs of segments with endpoints on a circle.

16. Describe an algorithm to compute the median of an array $A[1..5]$ of distinct numbers using at most 6 comparisons. Instead of writing pseudocode, describe your algorithm using a ***decision tree***: A binary tree where each internal node contains a comparison of the form "$A[i] \gtrless A[j]$?" and each leaf contains an index into the array.



Finding the median of a 3-element array using at most 3 comparisons

17. Suppose we are given a set S of n items, each with a *value* and a *weight*. For any element $x \in S$, we define two subsets

    - $S_{<x}$ is the set of all elements of S whose value is smaller than the value of x.
    - $S_{>x}$ is the set of all elements of S whose value is larger than the value of x.

For any subset $R \subseteq S$, let $w(R)$ denote the sum of the weights of elements in R. The ***weighted median*** of R is any element x such that $w(S_{<x}) \le w(S)/2$ and $w(S_{>x}) \le w(S)/2$.

Describe and analyze an algorithm to compute the weighted median of a given weighted set in $O(n)$ time. Your input consists of two unsorted arrays $S[1..n]$ and $W[1..n]$, where for each index i, the ith element has value $S[i]$ and weight $W[i]$. You may assume that all values are distinct and all weights are positive.

18. Consider the following generalization of the Blum-Floyd-Pratt-Rivest-Tarjan SELECT algorithm, which partitions the input array into $\lceil n/b \rceil$ blocks of size b, instead of $\lceil n/5 \rceil$ blocks of size 5, but is otherwise identical. In the pseudocode below, the necessary modifications are indicated in red.

```
MOM_bSELECT(A[1..n], k):
  if n ≤ b^2
      use brute force
  else
      m ← ⌈n/b⌉
      for i ← 1 to m
          M[i] ← MEDIANOFB(A[b(i−1)+1 .. bi])
      mom_b ← MOM_bSELECT(M[1..m], ⌊m/2⌋)
      r ← PARTITION(A[1..n], mom_b)
      if k < r
          return MOM_bSELECT(A[1..r−1], k)
      else if k > r
          return MOM_bSELECT(A[r+1..n], k−r)
      else
          return mom_b
```

(a) State a recurrence for the running time of $\text{MOM}_b\text{SELECT}$, assuming that b is a constant (so the subroutine MEDIANOFB runs in $O(1)$ time). In particular, how do the sizes of the recursive subproblems depend on the constant b? Consider even b and odd b separately.

(b) What is the running time of $\text{MOM}_1\text{SELECT}$? *[Hint: This is a trick question.]*

*(c) What is the running time of $\text{MOM}_2\text{SELECT}$? *[Hint: This is an unfair question.]*

(d) What is the running time of $\text{MOM}_3\text{SELECT}$?

(e) What is the running time of $\text{MOM}_4\text{SELECT}$?

(f) For any constants $b \ge 5$, the algorithm $\text{MOM}_b\text{SELECT}$ runs in $O(n)$ time, but different values of b lead to different constant factors. Let $M(b)$ denote the minimum number of comparisons required to find the median of b numbers. The exact value of $M(b)$ is known only for $b \le 13$:

b	1	2	3	4	5	6	7	8	9	10	11	12	13
$M(b)$	0	1	3	4	6	8	10	12	14	16	18	20	23

For each b between 5 and 13, find an upper bound on the running time of $\text{MOM}_b\text{SELECT}$ of the form $T(n) \le \alpha_b n$ for some explicit constant α_b. (For example, on page 8 we showed that $\alpha_5 \le 22$.)

(g) Which value of b yields the smallest constant α_b? *[Hint: This is a trick question.]*

19. An array $A[0..n-1]$ of n distinct numbers is ***bitonic*** if there are unique indices i and j such that $A[(i-1) \bmod n] < A[i] > A[(i+1) \bmod n]$ and $A[(j-1) \bmod n] > A[j] < A[(j+1) \bmod n]$. In other words, a bitonic sequence either consists of an increasing sequence followed by a decreasing sequence, or can be circularly shifted to become so. For example,

4	6	9	8	7	5	1	2	3

is bitonic, but

3	6	9	8	7	5	1	2	4

is *not* bitonic.

Describe and analyze an algorithm to find the *smallest* element in an n-element bitonic array in $O(\log n)$ time. You may assume that the numbers in the input array are distinct.

20. Suppose we are given an array $A[1..n]$ with the special property that $A[1] \geq A[2]$ and $A[n-1] \leq A[n]$. We say that an element $A[x]$ is a *local minimum* if it is less than or equal to both its neighbors, or more formally, if $A[x-1] \geq A[x]$ and $A[x] \leq A[x+1]$. For example, there are six local minima in the following array:

9	7	7	2	1	3	7	5	4	7	3	3	4	8	6	9

We can obviously find a local minimum in $O(n)$ time by scanning through the array. Describe and analyze an algorithm that finds a local minimum in $O(\log n)$ time. *[Hint: With the given boundary conditions, the array **must** have at least one local minimum. Why?]*

21. Suppose you are given a sorted array of n distinct numbers that has been *rotated* k steps, for some ***unknown*** integer k between 1 and $n-1$. That is, you are given an array $A[1..n]$ such that the prefix $A[1..k]$ is sorted in increasing order, the suffix $A[k+1..n]$ is sorted in increasing order, and $A[n] < A[1]$.

For example, you might be given the following 16-element array (where $k = 10$):

9	13	16	18	19	23	28	31	37	42	−4	0	2	5	7	8

(a) Describe and analyze an algorithm to compute the unknown integer k.

(b) Describe and analyze an algorithm to determine if the given array contains a given number x.

22. You are a contestant on the hit game show "Beat Your Neighbors!" You are presented with an $m \times n$ grid of boxes, each containing a unique number. It costs \$100 to open a box. Your goal is to find a box whose number is larger than its neighbors in the grid (above, below, left, and right). If you spend less money than any of your opponents, you win a week-long trip for two to Las Vegas and a year's supply of Rice-A-Roni™, to which you are hopelessly addicted.

(a) Suppose $m = 1$. Describe an algorithm that finds a number that is bigger than either of its neighbors. How many boxes does your algorithm open in the worst case?

*(b) Suppose $m = n$. Describe an algorithm that finds a number that is bigger than any of its neighbors. How many boxes does your algorithm open in the worst case?

*(c) Prove that your solution to part (b) is optimal up to a constant factor.

23. (a) Suppose we are given two sorted arrays $A[1..n]$ and $B[1..n]$ and an integer k. Describe an algorithm to find the kth smallest element in the union of A and B in $\Theta(\log n)$ time. For example, if $k = 1$, your algorithm should return the smallest element of $A \cup B$; if $k = n$, your algorithm should return the median of $A \cup B$.) You can assume that the arrays contain no duplicate elements. *[Hint: First solve the special case $k = n$.]*

    (b) Now suppose we are given *three* sorted arrays $A[1..n]$, $B[1..n]$, and $C[1..n]$, and an integer k. Describe an algorithm to find the kth smallest element in $A \cup B \cup C$ in $O(\log n)$ time.

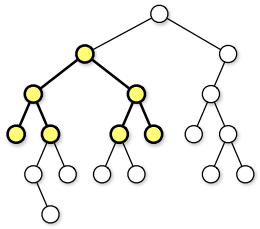
    (c) Finally, suppose we are given a two dimensional array $A[1..m][1..n]$ in which every row $A[i][\,]$ is sorted, and an integer k. Describe an algorithm to find the kth smallest element in A as quickly as possible. How does the running time of your algorithm depend on m? *[Hint: Use the linear-time* SELECT *algorithm as a subroutine.]*

24. (a) Describe an algorithm that sorts an input array $A[1..n]$ by calling a subroutine SQRTSORT(k), which sorts the subarray $A\left[k+1..k+\sqrt{n}\right]$ in place, given an arbitrary integer k between 0 and $n-\sqrt{n}$ as input. (To simplify the problem, assume that $\sqrt{n}$ is an integer.) Your algorithm is ***only*** allowed to inspect or modify the input array by calling SQRTSORT; in particular, your algorithm must not directly compare, move, or copy array elements. How many times does your algorithm call SQRTSORT in the worst case?

    (b) Prove that your algorithm from part (a) is optimal up to constant factors. In other words, if $f(n)$ is the number of times your algorithm calls SQRTSORT, prove that no algorithm can sort using $o(f(n))$ calls to SQRTSORT. *[Hint: See Lecture 19.]*

    (c) Now suppose SQRTSORT is implemented recursively, by calling your sorting algorithm from part (a). For example, at the second level of recursion, the algorithm is sorting arrays roughly of size $n^{1/4}$. What is the worst-case running time of the resulting sorting algorithm? (To simplify the analysis, assume that the array size n has the form 2^{2^k}, so that repeated square roots are always integers.)

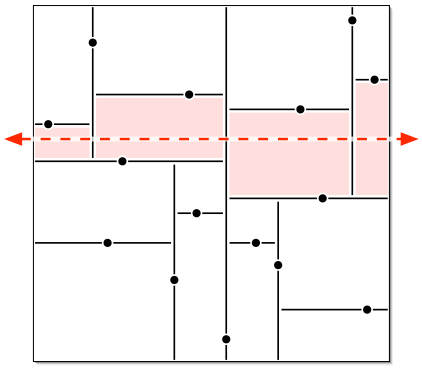
25. For this problem, a *subtree* of a binary tree means any connected subgraph. A binary tree is *complete* if every internal node has two children, and every leaf has exactly the same depth. Describe and analyze a recursive algorithm to compute the *largest complete subtree* of a given binary tree. Your algorithm should return the root and the depth of this subtree.



The largest complete subtree of this binary tree has depth 2.

26. Suppose we have n points scattered inside a two-dimensional box. A *kd-tree* recursively subdivides the points as follows. First we split the box into two smaller boxes with a *vertical* line, then we split each of those boxes with *horizontal* lines, and so on, always alternating between horizontal and vertical splits. Each time we split a box, the splitting line partitions the rest of the interior points

as evenly as possible by passing through a median point inside the box (*not* on its boundary). If a box doesn't contain any points, we don't split it any more; these final empty boxes are called *cells*.



A kd-tree for 15 points. The dashed line crosses the four shaded cells.

(a) How many cells are there, as a function of n? Prove your answer is correct.

(b) In the worst case, *exactly* how many cells can a horizontal line cross, as a function of n? Prove your answer is correct. Assume that $n = 2^k - 1$ for some integer k. *[Hint: There is more than one function f such that $f(16) = 4$.]*

(c) Suppose we are given n points stored in a kd-tree. Describe and analyze an algorithm that counts the number of points above a horizontal line (such as the dashed line in the figure) as quickly as possible. *[Hint: Use part (b).]*

(d) Describe an analyze an efficient algorithm that counts, given a kd-tree storing n points, the number of points that lie inside a rectangle R with horizontal and vertical sides. *[Hint: Use part (c).]*

*27. Bob Ratenbur, a new student in CS 225, is trying to write code to perform preorder, inorder, and postorder traversal of binary trees. Bob understands the basic idea behind the traversal algorithms, but whenever he tries to implement them, he keeps mixing up the recursive calls. Five minutes before the deadline, Bob frantically submitted code with the following structure:

```
PREORDER(v):
  if v = NULL
    return
  else
    print label(v)
    [redacted]ORDER(left(v))
    [redacted]ORDER(right(v))
```

```
INORDER(v):
  if v = NULL
    return
  else
    [redacted]ORDER(left(v))
    print label(v)
    [redacted]ORDER(right(v))
```

```
POSTORDER(v):
  if v = NULL
    return
  else
    [redacted]ORDER(left(v))
    [redacted]ORDER(right(v))
    print label(v)
```

Each [redacted] hides one of the prefixes PRE, IN, or POST. Moreover, each of the following function calls appears exactly once in Bob's submitted code:

PREORDER(*left*(v))	INORDER(*left*(v))	POSTORDER(*left*(v))
PREORDER(*right*(v))	INORDER(*right*(v))	POSTORDER(*right*(v))

Thus, there are precisely 36 possibilities for Bob's code. Unfortunately, Bob accidentally deleted his source code after submitting the executable, so neither you nor he knows which functions were called where.

Now suppose you are given the output of Bob's traversal algorithms, executed on some ***unknown*** binary tree T. Bob's output has been helpfully parsed into three arrays $Pre[1..n]$, $In[1..n]$, and $Post[1..n]$. You may assume that these traversal sequences are consistent with exactly one binary tree T; in particular, the vertex labels of the unknown tree T are distinct, and every internal node in T has exactly two children.

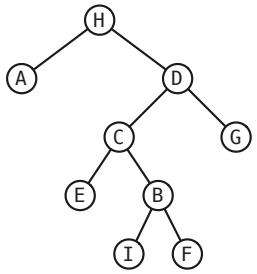
(a) Describe an algorithm to reconstruct the unknown tree T from the given traversal sequences.

(b) Describe an algorithm that either reconstruct Bob's code from the given traversal sequences, or correctly reports that the traversal sequences are consistent with more than one set of algorithms.

For example, given the input

$$Pre[1..n] = [\texttt{H A E C B I F G D}]$$
$$In[1..n] = [\texttt{A H D C E I F B G}]$$
$$Post[1..n] = [\texttt{A E I B F C D G H}]$$

your first algorithm should return the following tree:



and your second algorithm should reconstruct the following code:

```
PREORDER(v):
  if v = NULL
    return
  else
    print label(v)
    PREORDER(left(v))
    POSTORDER(right(v))
```

```
INORDER(v):
  if v = NULL
    return
  else
    POSTORDER(left(v))
    print label(v)
    PREORDER(right(v))
```

```
POSTORDER(v):
  if v = NULL
    return
  else
    INORDER(left(v))
    INORDER(right(v))
    print label(v)
```

28. Consider the following classical recursive algorithm for computing the factorial $n!$ of a non-negative integer n:

```
FACTORIAL(n):
  if n = 0
    return 1
  else
    return n · FACTORIAL(n − 1)
```

(a) How many multiplications does this algorithm perform?

(b) How many bits are required to write $n!$ in binary? Express your answer in the form $\Theta(f(n))$, for some familiar function $f(n)$. *[Hint:* $(n/2)^{n/2} < n! < n^n$*.]*

(c) Your answer to (b) should convince you that the number of multiplications is *not* a good estimate of the actual running time of FACTORIAL. We can multiply any k-digit number and any l-digit number in $O(k \cdot l)$ time using the grade-school algorithm (or the Russian peasant algorithm). What is the running time of FACTORIAL if we use this multiplication algorithm as a subroutine?

*(d) The following algorithm also computes the factorial function, but using a different grouping of the multiplications:

```
FACTORIAL2(n, m):        ⟨⟨Compute n!/(n−m)!⟩⟩
  if m = 0
      return 1
  else if m = 1
      return n
  else
      return FACTORIAL2(n, ⌊m/2⌋) · FACTORIAL2(n − ⌊m/2⌋, ⌈m/2⌉)
```

What is the running time of FACTORIAL2(n, n) if we use grade-school multiplication? *[Hint: Ignore the floors and ceilings.]*

(e) Describe and analyze a variant of Karastuba's algorithm that can multiply any k-digit number and any l-digit number, where $k \geq l$, in $O(k \cdot l^{\lg 3-1}) = O(k \cdot l^{0.585})$ time.

*(f) What are the running times of FACTORIAL(n) and FACTORIAL2(n, n) if we use the modified Karatsuba multiplication from part (e)?

To resolve the question by a careful enumeration of solutions via trial and error, continued Gauss, would take only an hour or two. Apparently such inelegant work held little attraction for Gauss, for he does not seem to have carried it out, despite outlining in detail how to go about it.

— Paul Campbell, "Gauss and the Eight Queens Problem:
A Study in Miniature of the Propagation of Historical Error" (1977)

I dropped my dinner, and ran back to the laboratory. There, in my excitement, I tasted the contents of every beaker and evaporating dish on the table. Luckily for me, none contained any corrosive or poisonous liquid.

— Constantine Fahlberg on his discovery of saccharin,
Scientific American (1886)

7 Backtracking

In this lecture, I want to describe another recursive algorithm strategy called ***backtracking***. A backtracking algorithm tries to build a solution to a computational problem incrementally. Whenever the algorithm needs to decide between multiple alternatives to the next component of the solution, it simply tries all possible options recursively.

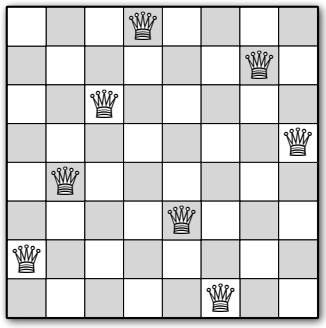
7.1 *n* Queens

The prototypical backtracking problem is the classical ***n Queens Problem***, first proposed by German chess enthusiast Max Bezzel in 1848 (under his pseudonym "Schachfreund") for the standard 8×8 board and by François-Joseph Eustache Lionnet in 1869 for the more general $n \times n$ board. The problem is to place n queens on an $n \times n$ chessboard, so that no two queens can attack each other. For readers not familiar with the rules of chess, this means that no two queens are in the same row, column, or diagonal.

Obviously, in any solution to the n-Queens problem, there is exactly one queen in each row. So we will represent our possible solutions using an array $Q[1..n]$, where $Q[i]$ indicates which square in row i contains a queen, or 0 if no queen has yet been placed in row i. To find a solution, we put queens on the board row by row, starting at the top. A *partial* solution is an array $Q[1..n]$ whose first $r-1$ entries are positive and whose last $n-r+1$ entries are all zeros, for some integer r.

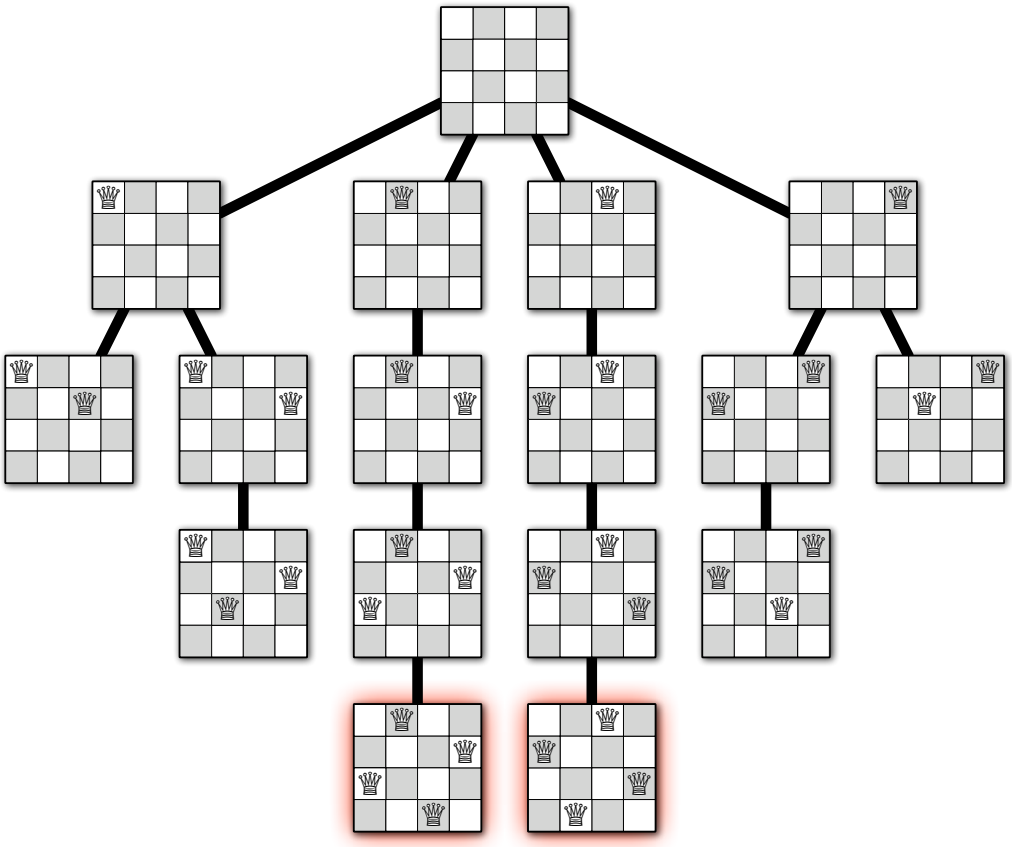
The following recursive algorithm, essentially due to Gauss (who called it "methodical groping"), recursively enumerates all complete n-queens solutions that are consistent with a given partial solution. The input parameter r is the first empty row. Thus, to compute all n-queens solutions with no restrictions, we would call RECURSIVENQUEENS$(Q[1..n], 1)$.

```
RecursiveNQueens(Q[1..n], r):
  if r = n + 1
      print Q
  else
      for j ← 1 to n
          legal ← TRUE
          for i ← 1 to r − 1
              if (Q[i] = j) or (Q[i] = j + r − i) or (Q[i] = j − r + i)
                  legal ← FALSE
          if legal
              Q[r] ← j
              RecursiveNQueens(Q[1..n], r + 1)
```



One solution to the 8 queens problem, represented by the array [4,7,3,8,2,5,1,6]

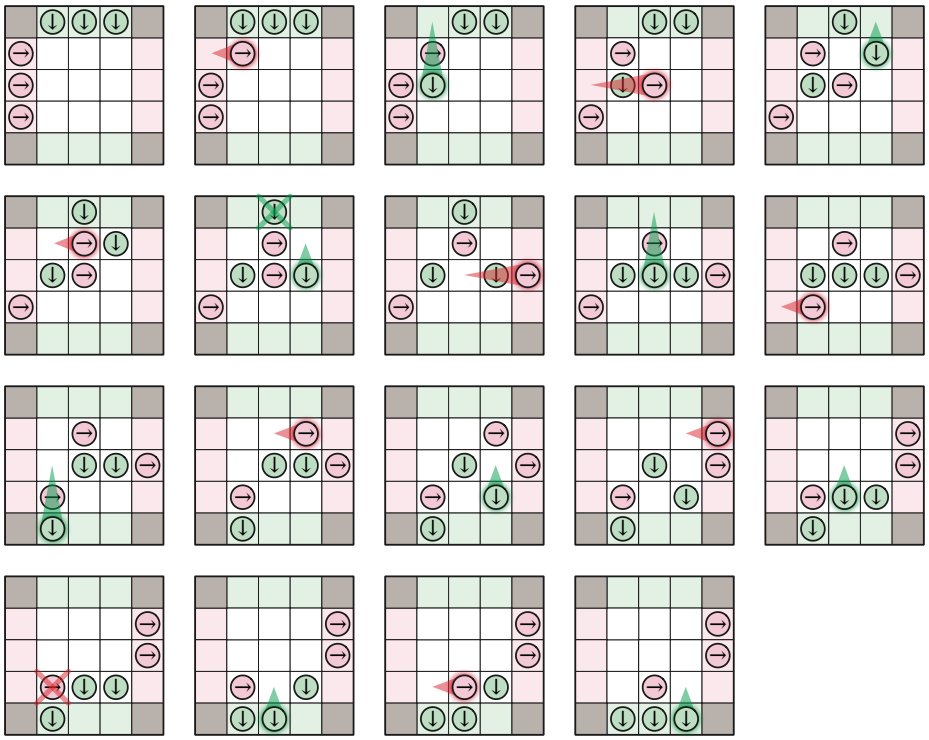
Like most recursive algorithms, the execution of a backtracking algorithm can be illustrated using a ***recursion tree***. The root of the recursion tree corresponds to the original invocation of the algorithm; edges in the tree correspond to recursive calls. A path from the root down to any node shows the history of a partial solution to the n-Queens problem, as queens are added to successive rows. The leaves correspond to partial solutions that cannot be extended, either because there is already a queen on every row, or because every position in the next empty row is in the same row, column, or diagonal as an existing queen. The backtracking algorithm simply performs a depth-first traversal of this tree.



The complete recursion tree for our algorithm for the 4 queens problem.

7.2 Game Trees

Consider the following simple two-player game played on an $n \times n$ square grid with a border of squares; let's call the players Horatio Fahlberg-Remsen and Vera Rebaudi.[1] Each player has n tokens that they move across the board from one side to the other. Horatio's tokens start in the left border, one in each row, and move to the right; symmetrically, Vera's tokens start in the top border, one in each column, and move down. The players alternate turns. In each of his turns, Horatio either *moves* one of his tokens one step to the right into an empty square, or *jumps* one of his tokens over exactly one of Vera's tokens into an empty square two steps to the right. However, if no legal moves or jumps are available, Horatio simply passes. Similarly, Vera either moves or jumps one of her tokens downward in each of her turns, unless no moves or jumps are possible. The first player to move all their tokens off the edge of the board wins.



Vera wins the 3×3 game.

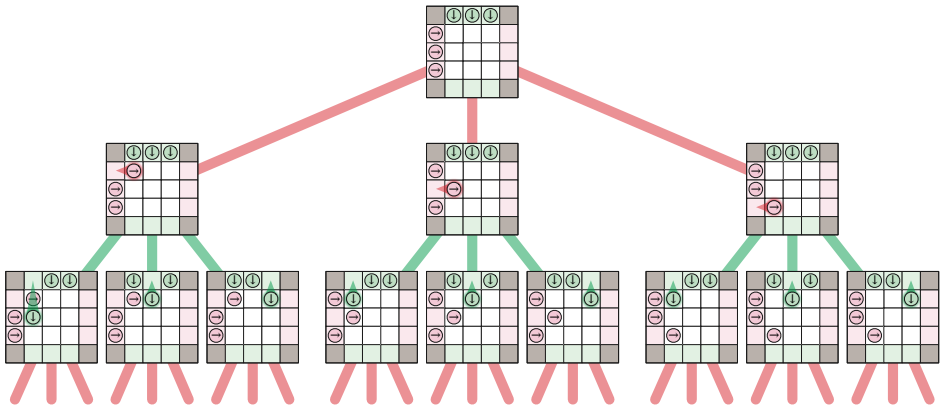
We can use a simple backtracking algorithm to determine the best move for each player at each turn. The *state* of the game consists of the locations of all the pieces and the player whose turn it is. We recursively define a game state to be *good* or *bad* as follows:

- A game state is *bad* if all the opposing player's tokens have reached their goals.
- A game state is *good* if the current player can move to a state that is bad for the opposing player.
- A configuration is *bad* if every move leads to a state that is good for the opposing player.

[1] I don't know what this game is called, or even if I'm remembering the rules correctly. I learned it (or something like it) from Lenny Pitt, who recommended playing it with sweetener packets at restaurants.

Constantin Fahlberg and Ira Remsen synthesized saccharin for the first time in 1878, while Fahlberg was a postdoc in Remsen's lab investigating coal tar derivatives. In 1900, Ovidio Rebaudi published the first chemical analysis of *ka'a he'ê*, a medicinal plant cultivated by the Guaraní for more than 1500 years, now more commonly known as *Stevia rebaudiana*.

This recursive definition immediately suggests a recursive backtracking algorithm to determine whether a given state of the game is good or bad. Moreover, for any good state, the backtracking algorithm finds a move leading to a bad state for the opposing player. Thus, by induction, any player that finds the game in a good state on their turn can win the game, even if their opponent plays perfectly; on the other hand, starting from a bad state, a player can win only if their opponent makes a mistake.



The first two levels of the game tree.

All computer game players are ultimately based on this simple backtracking strategy. However, since most games have an enormous number of states, it is not possible to traverse the entire game tree in practice. Instead, game programs employ other heuristics[2] to *prune* the game tree, by ignoring states that are obviously good or bad (or at least obviously better or worse that other states), and/or by cutting off the tree at a certain depth (or *ply*) and using a more efficient heuristic to evaluate the leaves.

7.3 Subset Sum

Let's consider a more complicated problem, called SUBSETSUM: Given a set X of positive integers and *target* integer T, is there a subset of elements in X that add up to T? Notice that there can be more than one such subset. For example, if $X = \{8, 6, 7, 5, 3, 10, 9\}$ and $T = 15$, the answer is TRUE, thanks to the subsets $\{8, 7\}$ or $\{7, 5, 3\}$ or $\{6, 9\}$ or $\{5, 10\}$. On the other hand, if $X = \{11, 6, 5, 1, 7, 13, 12\}$ and $T = 15$, the answer is FALSE.

There are two trivial cases. If the target value T is zero, then we can immediately return TRUE, because empty set is a subset of *every* set X, and the elements of the empty set add up to zero.[3] On the other hand, if $T < 0$, or if $T \neq 0$ but the set X is empty, then we can immediately return FALSE.

For the general case, consider an arbitrary element $x \in X$. (We've already handled the case where X is empty.) There is a subset of X that sums to T if and only if one of the following statements is true:

- There is a subset of X that *includes* x and whose sum is T.
- There is a subset of X that *excludes* x and whose sum is T.

In the first case, there must be a subset of $X \setminus \{x\}$ that sums to $T - x$; in the second case, there must be a subset of $X \setminus \{x\}$ that sums to T. So we can solve SUBSETSUM(X, T) by reducing it to two simpler instances: SUBSETSUM$(X \setminus \{x\}, T - x)$ and SUBSETSUM$(X \setminus \{x\}, T)$. Here's how the resulting recusive algorithm might look if X is stored in an array.

[2] A heuristic is an algorithm that doesn't work.

[3] There's no base case like the vacuous base case!

```
SUBSETSUM(X[1..n], T):
  if T = 0
      return TRUE
  else if T < 0 or n = 0
      return FALSE
  else
      return (SUBSETSUM(X[1..n-1], T) ∨ SUBSETSUM(X[1..n-1], T - X[n]))
```

Proving this algorithm correct is a straightforward exercise in induction. If $T = 0$, then the elements of the empty subset sum to T, so TRUE is the correct output. Otherwise, if T is negative or the set X is empty, then no subset of X sums to T, so FALSE is the correct output. Otherwise, if there is a subset that sums to T, then either it contains $X[n]$ or it doesn't, and the Recursion Fairy correctly checks for each of those possibilities. Done.

The running time $T(n)$ clearly satisfies the recurrence $T(n) \leq 2T(n-1) + O(1)$, which we can solve using either recursion trees or annihilators (or just guessing) to obtain the upper bound $T(n) = O(2^n)$. In the worst case, the recursion tree for this algorithm is a complete binary tree with depth n.

Here is a similar recursive algorithm that actually *constructs* a subset of X that sums to T, if one exists. This algorithm also runs in $O(2^n)$ time.

```
CONSTRUCTSUBSET(X[1..n], T):
  if T = 0
      return ∅
  if T < 0 or n = 0
      return NONE

  Y ← CONSTRUCTSUBSET(X[1..n-1], T)
  if Y ≠ NONE
      return Y

  Y ← CONSTRUCTSUBSET(X[1..n-1], T - X[n])
  if Y ≠ NONE
      return Y ∪ {X[n]}

  return NONE
```

7.4 The General Pattern

★★★

Find a small choice whose correct answer would reduce the problem size. For each possible answer, temporarily adopt that choice and recurse. (Don't try to be clever about which choices to try; just try them all.) The recursive subproblem is often more general than the original target problem; in each recursive subproblem, we must consider *only* solutions that are consistent with the choices we have already made.

7.5 NFA acceptance

Recall that a nondeterministic finite-state automaton, or NFA, can be described as a directed graph, whose edges are called *states* and whose edges have *labels* drawn from a finite set Σ called the *alphabet*. Every NFA has a designated *start* state and a subset of *accepting* states. Any walk in this graph has a label, which is a string formed by concatenating the labels of the edges in the walk. A string w is *accepted* by an NFA if and only if there is a walk from the start state to one of the accepting states whose label is w.

More formally (or at least, more symbolically), an NFA consists of a finite set Q of states, a start state $s \in Q$, a set of accepting states $A \subseteq Q$, and a transition function $\delta : Q \times \Sigma \to 2^Q$. We recursively extend

the transition function to strings by defining

$$\delta^*(q,w) = \begin{cases} \{q\} & \text{if } w = \varepsilon, \\ \bigcup_{r \in \delta(q,a)} \delta^*(r,x) & \text{if } w = ax. \end{cases}$$

The NFA accepts string w if and only if the set $\delta^*(s,w)$ contains at least one accepting state.

We can express this acceptance criterion more directly as follows. We define a boolean function *Accepts?*(q,w), which is TRUE if the NFA would accept string w if we started in state q, and FALSE otherwise. This function has the following recursive definition:

$$\textit{Accepts?}(q,w) := \begin{cases} \textsc{True} & \text{if } w = \varepsilon \text{ and } q \in A \\ \textsc{False} & \text{if } w = \varepsilon \text{ and } q \in A \\ \bigvee_{r \in \delta(q,a)} \textit{Accepts?}(r,x) & \text{if } w = ax \end{cases}$$

The NFA accepts w if and only if *Accepts?*$(s,w) = \textsc{True}$.

In the magical world of non-determinism, we can imagine that the NFA always magically makes the right decision when faces with multiple transitions, or perhaps spawns off an independent parallel thread for each possible choice. Alas, real computers are neither clairvoyant nor (despite the increasing use of multiple cores) infinitely parallel. To simulate the NFA's behavior directly, we must recursively explore the consequences of each choice explicitly.

The recursive definition of *Accepts?* translates directly into the following recursive backtracking algorithm. Here, the transition function δ and the accepting states A are represented as global boolean arrays, where $\delta[q,a,r] = \textsc{True}$ if and only if $r \in \delta(q,a)$, and $A[q] = \textsc{True}$ if and only if $q \in A$.

```
ACCEPTS?(q, w[1..n]):
  if n = 0
      return A[q]
  for all states r
      if δ[q, w[1], r] and ACCEPTS?(r, w[2..n])
          return TRUE
  return FALSE
```

To determine whether the NFA accepts a string w, we call ACCEPTS?(δ, A, s, w).

The running time of this algorithm satisfies the recursive inequailty $T(n) \leq O(|Q|) \cdot T(n-1)$, which immediately implies that $T(n) = O(|Q|^n)$.

7.6 Longest Increasing Subsequence

Now suppose we are given a sequence of integers, and we want to find the longest subsequence whose elements are in increasing order. More concretely, the input is an array $A[1..n]$ of integers, and we want to find the longest sequence of indices $1 \leq i_1 < i_2 < \cdots i_k \leq n$ such that $A[i_j] < A[i_{j+1}]$ for all j.

To derive a recursive algorithm for this problem, we start with a recursive definition of the kinds of objects we're playing with: sequences and subsequences.

> A *sequence of integers* is either empty
> or an integer followed by a sequence of integers.

This definition suggests the following strategy for devising a recursive algorithm. If the input sequence is empty, there's nothing to do. Otherwise, we only need to figure out what to do with the first element of the input sequence; the Recursion Fairy will take care of everything else. We can formalize this strategy somewhat by giving a recursive definition of subsequence (using array notation to represent sequences):

> The only *subsequence* of the empty sequence is the empty sequence.
> A *subsequence* of $A[1..n]$ is either a subsequence of $A[2..n]$
> or $A[1]$ followed by a subsequence of $A[2..n]$.

We're not just looking for just *any* subsequence, but a *longest* subsequence with the property that elements are in *increasing* order. So let's try to add those two conditions to our definition. (I'll omit the familiar vacuous base case.)

> The *LIS* of $A[1..n]$ is
> either the LIS of $A[2..n]$
> or $A[1]$ followed by the LIS of $A[2..n]$ *with elements larger than* $A[1]$,
> *whichever is longer*.

This definition is correct, but it's not quite recursive—we're defining the object 'longest increasing subsequence' in terms of the slightly *different* object 'longest increasing subsequence with elements larger than x', which we haven't properly defined yet. Fortunately, this second object has a very similar recursive definition. (Again, I'm omitting the vacuous base case.)

> If $A[1] \le x$, the LIS of $A[1..n]$ with elements larger than x is
> the LIS of $A[2..n]$ with elements larger than x.
>
> Otherwise, the LIS of $A[1..n]$ with elements larger than x is
> either the LIS of $A[2..n]$ with elements larger than x
> or $A[1]$ followed by the LIS of $A[2..n]$ with elements larger than $A[1]$,
> whichever is longer.

The longest increasing subsequence without restrictions can now be redefined as the longest increasing subsequence with elements larger than $-\infty$. Rewriting this recursive definition into pseudocode gives us the following recursive algorithm.

```
LIS(A[1..n]):
  return LISBIGGER(−∞, A[1..n])
```

```
LISBIGGER(prev, A[1..n]):
  if n = 0
    return 0
  else
    max ← LISBIGGER(prev, A[2..n])
    if A[1] > prev
      L ← 1 + LISBIGGER(A[1], A[2..n])
      if L > max
        max ← L
    return max
```

The running time of this algorithm satisfies the recurrence $T(n) \le 2T(n-1) + O(1)$, which as usual implies that $T(n) = O(2^n)$. We really shouldn't be surprised by this running time; in the worst case, the algorithm examines each of the 2^n subsequences of the input array.

The following alternative strategy avoids defining a new object with the "larger than x" constraint. We still only have to decide whether to include or exclude the first element $A[1]$. We consider the case where $A[1]$ is excluded exactly the same way, but to consider the case where $A[1]$ is included, we remove any elements of $A[2..n]$ that are larger than $A[1]$ *before* we recurse. This new strategy gives us the following algorithm:

```
FILTER(A[1..n], x):
  j ← 1
  for i ← 1 to n
      if A[i] > x
          B[j] ← A[i];  j ← j + 1
  return B[1..j]
```

```
LIS(A[1..n]):
  if n = 0
      return 0
  else
      max ← LIS(prev, A[2..n])
      L ← 1 + LIS(A[1], FILTER(A[2..n], A[1]))
      if L > max
          max ← L
      return max
```

The FILTER subroutine clearly runs in $O(n)$ time, so the running time of LIS satisfies the recurrence $T(n) \le 2T(n-1) + O(n)$, which solves to $T(n) \le O(2^n)$ by the annihilator method. This upper bound pessimistically assumes that FILTER never actually removes any elements; indeed, if the input sequence is sorted in increasing order, this assumption is correct.

7.7 Optimal Binary Search Trees

★★★

Retire this example? It's not a *bad* example, exactly—certainly it's infinitely better than the execrable matrix-chain multiplication problem from Aho, Hopcroft, and Ullman—but it's not the best *first* example of tree-like backtracking. Minimum-ink triangulation of convex polygons is both more intuitive (geometry FTW!) and structurally equivalent. CFG parsing and regular expression matching (really just a special case of parsing) have similar recursive structure, but are a bit more complicated.

Our next example combines recursive backtracking with the divide-and-conquer strategy. Recall that the running time for a successful search in a binary search tree is proportional to the number of ancestors of the target node.[4] As a result, the worst-case search time is proportional to the depth of the tree. Thus, to minimize the worst-case search time, the height of the tree should be as small as possible; by this metric, the ideal tree is perfectly balanced.

In many applications of binary search trees, however, it is more important to minimize the total cost of several searches rather than the worst-case cost of a single search. If x is a more 'popular' search target than y, we can save time by building a tree where the depth of x is smaller than the depth of y, even if that means increasing the overall depth of the tree. A perfectly balanced tree is *not* the best choice if some items are significantly more popular than others. In fact, a totally unbalanced tree of depth $\Omega(n)$ might actually be the best choice!

This situation suggests the following problem. Suppose we are given a sorted array of ***keys*** $A[1..n]$ and an array of corresponding ***access frequencies*** $f[1..n]$. Our task is to build the binary search tree that minimizes the *total* search time, assuming that there will be exactly $f[i]$ searches for each key $A[i]$.

Before we think about how to solve this problem, we should first come up with a good recursive definition of the function we are trying to optimize! Suppose we are also given a binary search tree T with n nodes. Let v_i denote the node that stores $A[i]$, and let r be the index of the root node. Ignoring

[4] An *ancestor* of a node v is either the node itself or an ancestor of the parent of v. A *proper* ancestor of v is either the parent of v or a proper ancestor of the parent of v.

constant factors, the cost of searching for $A[i]$ is the number of nodes on the path from the root v_r to v_i. Thus, the total cost of performing all the binary searches is given by the following expression:

$$Cost(T, f[1..n]) = \sum_{i=1}^{n} f[i] \cdot \#\text{nodes between } v_r \text{ and } v_i$$

Every search path includes the root node v_r. If $i < r$, then all other nodes on the search path to v_i are in the left subtree; similarly, if $i > r$, all other nodes on the search path to v_i are in the right subtree. Thus, we can partition the cost function into three parts as follows:

$$\begin{aligned} Cost(T, f[1..n]) = &\sum_{i=1}^{r-1} f[i] \cdot \#\text{nodes between } \mathit{left}(v_r) \text{ and } v_i \\ &+ \sum_{i=1}^{n} f[i] \\ &+ \sum_{i=r+1}^{n} f[i] \cdot \#\text{nodes between } \mathit{right}(v_r) \text{ and } v_i \end{aligned}$$

Now the first and third summations look exactly like our original expression (*) for $Cost(T, f[1..n])$. Simple substitution gives us our recursive definition for *Cost*:

$$\boxed{Cost(T, f[1..n]) = Cost(\mathit{left}(T), f[1..r-1]) \;+\; \sum_{i=1}^{n} f[i] \;+\; Cost(\mathit{right}(T), f[r+1..n])}$$

The base case for this recurrence is, as usual, $n = 0$; the cost of performing no searches in the empty tree is zero.

Now our task is to compute the tree T_{opt} that minimizes this cost function. Suppose we somehow magically knew that the root of T_{opt} is v_r. Then the recursive definition of $Cost(T, f)$ immediately implies that the left subtree $\text{left}(T_{\text{opt}})$ must be the optimal search tree for the keys $A[1..r-1]$ and access frequencies $f[1..r-1]$. Similarly, the right subtree $\text{right}(T_{\text{opt}})$ must be the optimal search tree for the keys $A[r+1..n]$ and access frequencies $f[r+1..n]$. **Once we choose the correct key to store at the root, the Recursion Fairy automatically constructs the rest of the optimal tree.** More formally, let $OptCost(f[1..n])$ denote the total cost of the optimal search tree for the given frequency counts. We immediately have the following recursive definition.

$$\boxed{OptCost(f[1..n]) = \min_{1 \le r \le n} \left\{ OptCost(f[1..r-1]) \;+\; \sum_{i=1}^{n} f[i] \;+\; OptCost(f[r+1..n]) \right\}}$$

Again, the base case is $OptCost(f[1..0]) = 0$; the best way to organize no keys, which we will plan to search zero times, is by storing them in the empty tree!

This recursive definition can be translated mechanically into a recursive algorithm, whose running time $T(n)$ satisfies the recurrence

$$T(n) = \Theta(n) + \sum_{k=1}^{n} \big(T(k-1) + T(n-k) \big).$$

The $\Theta(n)$ term comes from computing the total number of searches $\sum_{i=1}^{n} f[i]$.

Yeah, that's one ugly recurrence, but it's actually easier to solve than it looks. To transform it into a more familiar form, we regroup and collect identical terms, subtract the recurrence for $T(n-1)$ to get rid of the summation, and then regroup again.

$$\begin{aligned} T(n) &= \Theta(n) + 2\sum_{k=0}^{n-1} T(k) \\ T(n-1) &= \Theta(n-1) + 2\sum_{k=0}^{n-2} T(k) \\ T(n) - T(n-1) &= \Theta(1) + 2T(n-1) \\ T(n) &= 3T(n-1) + \Theta(1) \end{aligned}$$

The solution $\boldsymbol{T(n) = \Theta(3^n)}$ now follows from the annihilator method.

Let me emphasize that this recursive algorithm does *not* examine all possible binary search trees. The number of binary search trees with n nodes satisfies the recurrence

$$N(n) = \sum_{r=1}^{n-1} \big(N(r-1) \cdot N(n-r)\big),$$

which has the closed-from solution $N(n) = \Theta(4^n/\sqrt{n})$. Our algorithm saves considerable time by searching *independently* for the optimal left and right subtrees. A full enumeration of binary search trees would consider all possible *pairings* of left and right subtrees; hence the product in the recurrence for $N(n)$.

7.8 CFG Parsing

Our final example is the ***parsing*** problem for context-free languages. Given a string w and a context-free grammar G, does w belong to the language generated by G? Recall that a context-free grammar over the alphabet Σ consists of a finite set Γ of *non-terminals* (disjoint from Σ) and a finite set of *production rules* of the form $A \to w$, where A is a nonterminal and w is a string over $\Sigma \cup \Gamma$.

Real-world applications of parsing normally require more information than just a single bit. For example, compilers require parsers that output a parse tree of the input code; some natural language applications require the *number* of distinct parse trees for a given string; others assign probabilities to the production rules and then ask for the *most likely* parse tree for a given string. However, these more general problems can be solved using relatively straightforward generalizations of the following decision algorithm.

★★★ Backtracking recurrence behind CYK

Exercises

1. (a) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common subsequence* of A and B is both a subsequence of A and a subsequence of B. Give a simple recursive definition for the function $lcs(A, B)$, which gives the length of the *longest* common subsequence of A and B.

   (b) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common supersequence* of A and B is another sequence that contains both A and B as subsequences. Give a simple recursive definition for the function $scs(A, B)$, which gives the length of the *shortest* common supersequence of A and B.

(c) Call a sequence $X[1..n]$ *oscillating* if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Give a simple recursive definition for the function $los(A)$, which gives the length of the longest oscillating subsequence of an arbitrary array A of integers.

(d) Give a simple recursive definition for the function $sos(A)$, which gives the length of the shortest oscillating supersequence of an arbitrary array A of integers.

(e) Call a sequence $X[1..n]$ *accelerating* if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i. Give a simple recursive definition for the function $lxs(A)$, which gives the length of the longest accelerating subsequence of an arbitrary array A of integers.

For more backtracking exercises, see the next two lecture notes!

Those who cannot remember the past are doomed to repeat it.

— George Santayana, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

The 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word 'research'*. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term* 'research' *in his presence. You can imagine how he felt, then, about the term* 'mathematical'*. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?*

— Richard Bellman, on the origin of his term 'dynamic programming' (1984)

If we all listened to the professor, we may be all looking for professor jobs.

— Pittsburgh Steelers' head coach Bill Cowher, responding to David Romer's dynamic-programming analysis of football strategy (2003)

8 Dynamic Programming

8.1 Fibonacci Numbers

8.1.1 Recursive Definitions Are Recursive Algorithms

The Fibonacci numbers F_n, named after Leonardo Fibonacci Pisano[1], the mathematician who popularized 'algorism' in Europe in the 13th century, are defined as follows: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. The recursive definition of Fibonacci numbers immediately gives us a recursive algorithm for computing them:

```
RecFibo(n):
  if (n < 2)
      return n
  else
      return RecFibo(n-1) + RecFibo(n-2)
```

How long does this algorithm take? Except for the recursive calls, the entire algorithm requires only a constant number of steps: one comparison and possibly one addition. If $T(n)$ represents the number of recursive calls to RecFibo, we have the recurrence

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + 1.$$

This looks an awful lot like the recurrence for Fibonacci numbers! The annihilator method gives us an asymptotic bound of $\Theta(\phi^n)$, where $\phi = (\sqrt{5}+1)/2 \approx 1.61803398875$, the so-called *golden ratio*, is the largest root of the polynomial $r^2 - r - 1$. But it's fairly easy to prove (hint, hint) the exact solution $\boldsymbol{T(n) = 2F_{n+1} - 1}$. In other words, computing F_n using this algorithm takes more than twice as many steps as just counting to F_n!

Another way to see this is that the RecFibo is building a big binary tree of additions, with nothing but zeros and ones at the leaves. Since the eventual output is F_n, our algorithm must call RecRibo(1) (which returns 1) exactly F_n times. A quick inductive argument implies that RecFibo(0) is called exactly F_{n-1} times. Thus, the recursion tree has $F_n + F_{n-1} = F_{n+1}$ leaves, and therefore, because it's a full binary tree, it must have $2F_{n+1} - 1$ nodes.

[1] literally, "Leonardo, son of Bonacci, of Pisa"

8.1.2 Memo(r)ization: Remember Everything

The obvious reason for the recursive algorithm's lack of speed is that it computes the same Fibonacci numbers over and over and over. A single call to RECFIBO(n) results in one recursive call to RECFIBO($n-1$), two recursive calls to RECFIBO($n-2$), three recursive calls to RECFIBO($n-3$), five recursive calls to RECFIBO($n-4$), and in general F_{k-1} recursive calls to RECFIBO($n-k$) for any integer $0 \le k < n$. Each call is recomputing some Fibonacci number from scratch.

We can speed up our recursive algorithm considerably just by writing down the results of our recursive calls and looking them up again if we need them later. This process was dubbed *memoization* by Richard Michie in the late 1960s.[2]

```
MEMFIBO(n):
  if (n < 2)
      return n
  else
      if F[n] is undefined
          F[n] ← MEMFIBO(n − 1) + MEMFIBO(n − 2)
      return F[n]
```

Memoization clearly decreases the running time of the algorithm, but by how much? If we actually trace through the recursive calls made by MEMFIBO, we find that the array $F[\,]$ is filled from the bottom up: first $F[2]$, then $F[3]$, and so on, up to $F[n]$. This pattern can be verified by induction: Each entry $F[i]$ is filled only after its predecessor $F[i-1]$. If we ignore the time spent in recursive calls, it requires only constant time to evaluate the recurrence for each Fibonacci number F_i. But by design, the recurrence for F_i is evaluated only once for each index i! We conclude that MEMFIBO performs only $O(n)$ additions, an *exponential* improvement over the naïve recursive algorithm!

8.1.3 Dynamic Programming: Fill Deliberately

But once we see how the array $F[\,]$ is filled, we can replace the recursion with a simple loop that intentionally fills the array in order, instead of relying on the complicated recursion to do it for us 'accidentally'.

```
ITERFIBO(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

Now the time analysis is immediate: ITERFIBO clearly uses ***O(n) additions*** and stores $O(n)$ *integers*.

This gives us our first explicit *dynamic programming* algorithm. The dynamic programming paradigm was developed by Richard Bellman in the mid-1950s, while working at the RAND Corporation. Bellman deliberately chose the name 'dynamic programming' to hide the mathematical character of his work from his military bosses, who were actively hostile toward anything resembling mathematical research. Here, the word 'programming' does not refer to writing code, but rather to the older sense of *planning* or *scheduling*, typically by filling in a table. For example, sports programs and theater programs are schedules of important events (with ads); television programming involves filling each available time slot with a show (and ads); degree programs are schedules of classes to be taken (with ads). The Air Force funded Bellman and others to develop methods for constructing training and logistics schedules,

[2] "My name is Elmer J. Fudd, millionaire. I own a mansion and a yacht."

or as they called them, 'programs'. The word 'dynamic' is meant to suggest that the table is filled in over time, rather than all at once (as in 'linear programming', which we will see later in the semester).[3]

8.1.4 Don't Remember Everything After All

In many dynamic programming algorithms, it is not necessary to retain *all* intermediate results through the entire computation. For example, we can significantly reduce the space requirements of our algorithm ITERFIBO by maintaining only the two newest elements of the array:

```
ITERFIBO2(n):
  prev ← 1
  curr ← 0
  for i ← 1 to n
      next ← curr + prev
      prev ← curr
      curr ← next
  return curr
```

(This algorithm uses the non-standard but perfectly consistent base case $F_{-1} = 1$ so that ITERFIBO2(0) returns the correct value 0.)

8.1.5 Faster! Faster!

Even this algorithm can be improved further, using the following wonderful fact:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x+y \end{bmatrix}$$

In other words, multiplying a two-dimensional vector by the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$ does exactly the same thing as one iteration of the inner loop of ITERFIBO2. This might lead us to believe that multiplying by the matrix n times is the same as iterating the loop n times:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}.$$

A quick inductive argument proves this fact. So if we want the nth Fibonacci number, we just have to compute the nth power of the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$. If we use repeated squaring, computing the nth power of something requires only $O(\log n)$ multiplications. In this case, that means $O(\log n)$ 2×2 matrix multiplications, each of which reduces to a constant number of integer multiplications and additions. Thus, we can compute F_n in only ***O(log n) integer arithmetic operations***.

This is an exponential speedup over the standard iterative algorithm, which was already an exponential speedup over our original recursive algorithm. Right?

8.1.6 Whoa! Not so fast!

Well, not exactly. Fibonacci numbers grow exponentially fast. The nth Fibonacci number is approximately $n \log_{10} \phi \approx n/5$ decimal digits long, or $n \log_2 \phi \approx 2n/3$ bits. So we can't possibly compute F_n in logarithmic time — we need $\Omega(n)$ time just to write down the answer!

[3] "I thought dynamic programming was a good name. It was something not even a Congressman could object to. So I used it as an umbrella for my activities."

The way out of this apparent paradox is to observe that ***we can't perform arbitrary-precision arithmetic in constant time***. Let $M(n)$ denote the time required to multiply two n-digit numbers. The matrix-based algorithm's actual running time obeys the recurrence $T(n) = T(\lfloor n/2 \rfloor) + M(n)$, which solves to $T(n) = \boldsymbol{O(M(n))}$ using recursion trees. The fastest known multiplication algorithm runs in time $O(n \log n 2^{O(\log^* n)})$, so that is also the running time of the fastest algorithm known to compute Fibonacci numbers.

Is this algorithm slower than our initial "linear-time" iterative algorithm? No! Addition isn't free, either. Adding two n-digit numbers takes $O(n)$ time, so the running time of the iterative algorithm is $\boldsymbol{O(n^2)}$. (Do you see why?) The matrix-squaring algorithm really is faster than the iterative addition algorithm, but not exponentially faster.

In the original recursive algorithm, the extra cost of arbitrary-precision arithmetic is overwhelmed by the huge number of recursive calls. The correct recurrence is $T(n) = T(n-1) + T(n-2) + O(n)$, for which the annihilator method still implies the solution $T(n) = O(\phi^n)$.

8.2 Longest Increasing Subsequence

In a previous lecture, we developed a recursive algorithm to find the length of the longest increasing subsequence of a given sequence of numbers. Given an array $A[1..n]$, the length of the longest increasing subsequence is computed by the function call LISBIGGER$(-\infty, A[1..n])$, where LISBIGGER is the following recursive algorithm:

```
LISBIGGER(prev, A[1..n]):
  if n = 0
    return 0
  else
    max ← LISBIGGER(prev, A[2..n])
    if A[1] > prev
      L ← 1 + LISBIGGER(A[1], A[2..n])
      if L > max
        max ← L
    return max
```

We can simplify our notation slightly with two simple observations. First, the input variable *prev* is always either $-\infty$ or an element of the input array. Second, the second argument of LISBIGGER is always a *suffix* of the original input array. If we add a new sentinel value $A[0] = -\infty$ to the input array, we can identify any recursive subproblem with two array indices.

Thus, we can rewrite the recursive algorithm as follows. Add the sentinel value $A[0] = -\infty$. Let $LIS(i, j)$ denote the length of the longest increasing subsequence of $A[j..n]$ with all elements larger than $A[i]$. Our goal is to compute $LIS(0, 1)$. For all $i < j$, we have

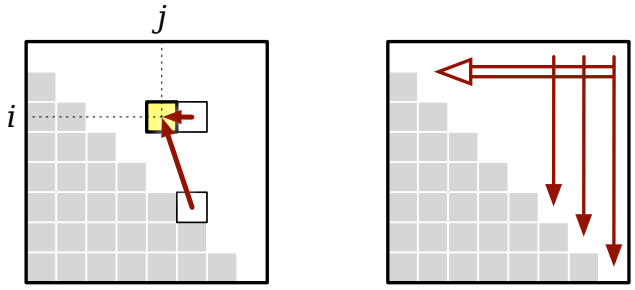
$$LIS(i,j) = \begin{cases} 0 & \text{if } j > n \\ LIS(i, j+1) & \text{if } A[i] \geq A[j] \\ \max\{LIS(i, j+1),\ 1 + LIS(j, j+1)\} & \text{otherwise} \end{cases}$$

Because each recursive subproblem can be identified by two indices i and j, we can store the intermediate values in a two-dimensional array $\boldsymbol{LIS[0..n, 1..n]}$.[4] Since there are $O(n^2)$ entries in the

[4] In fact, we only need half of this array, because we always have $i < j$. But even if we cared about constant factors in this class (we don't), this would be the wrong time to worry about them. The first order of business is to find an algorithm that actually *works*; once we have that, then we can think about optimizing it.

table, our memoized algorithm uses ***O(n²) space***. Each entry in the table can be computed in $O(1)$ time once we know its predecessors, so our memoized algorithm runs in ***O(n²) time***.

It's not immediately clear what order the recursive algorithm fills the rest of the table; all we can tell from the recurrence is that each entry $LIS[i, j]$ is filled in *after* the entries $LIS[i, j+1]$ and $LIS[j, j+1]$ in the next columns. But just this partial information is enough to give us an explicit evaluation order. If we fill in our table one column at a time, from right to left, then whenever we reach an entry in the table, the entries it depends on are already available.



Dependencies in the memoization table for longest increasing subsequence, and a legal evaluation order

Finally, putting everything together, we obtain the following dynamic programming algorithm:

```
LIS(A[1..n]):
  A[0] ← −∞                          ⟨⟨Add a sentinel⟩⟩
  for i ← 0 to n                     ⟨⟨Base cases⟩⟩
      LIS[i, n+1] ← 0

  for j ← n downto 1
      for i ← 0 to j − 1
          if A[i] ≥ A[j]
              LIS[i, j] ← LIS[i, j+1]
          else
              LIS[i, j] ← max{LIS[i, j+1], 1 + LIS[j, j+1]}

  return LIS[0, 1]
```

As expected, the algorithm clearly uses ***O(n²) time and space***. However, we can reduce the space to $O(n)$ by only maintaining the two most recent columns of the table, $LIS[\cdot, j]$ and $LIS[\cdot, j+1]$.[5]

This is not the only recursive strategy we could use for computing longest increasing subsequences efficiently. Here is another recurrence that gives us the $O(n)$ space bound for free. Let $LIS'(i)$ denote the length of the longest increasing subsequence of $A[i..n]$ that starts with $A[i]$. Our goal is to compute $LIS'(0) - 1$; we subtract 1 to ignore the sentinel value $-\infty$. To define $LIS'(i)$ recursively, we only need to specify the *second* element in subsequence; the Recursion Fairy will do the rest.

$$LIS'(i) = 1 + \max\left\{LIS'(j) \mid j > i \text{ and } A[j] > A[i]\right\}$$

Here, I'm assuming that $\max\varnothing = 0$, so that the base case is $L'(n) = 1$ falls out of the recurrence automatically. Memoizing this recurrence requires only ***O(n) space***, and the resulting algorithm runs in ***O(n²) time***. To transform this memoized recurrence into a dynamic programming algorithm, we only need to guarantee that $LIS'(j)$ is computed before $LIS'(i)$ whenever $i < j$.

[5] See, I told you not to worry about constant factors yet!

```
LIS2(A[1..n]):
  A[0] = -∞                          ⟨⟨Add a sentinel⟩⟩
  for i ← n downto 0
      LIS'[i] ← 1
      for j ← i + 1 to n
          if A[j] > A[i] and 1 + LIS'[j] > LIS'[i]
              LIS'[i] ← 1 + LIS'[j]
  return LIS'[0] - 1                 ⟨⟨Don't count the sentinel⟩⟩
```

8.3 The Pattern: Smart Recursion

In a nutshell, dynamic programming is *recursion without repetition*. Dynamic programming algorithms store the solutions of intermediate subproblems, often *but not always* in some kind of array or table. Many algorithms students make the mistake of focusing on the table (because tables are easy and familiar) instead of the *much* more important (and difficult) task of finding a correct recurrence. As long as we memoize the correct recurrence, an explicit table isn't really necessary, but if the recursion is incorrect, nothing works.

Dynamic programming is *not* about filling in tables.
It's about smart recursion!

Dynamic programming algorithms are almost always developed in two distinct stages.

1. **Formulate the problem recursively.** Write down a recursive formula or algorithm for the whole problem in terms of the answers to smaller subproblems. This is the hard part. It generally helps to think in terms of a recursive definition of the object you're trying to construct. A complete recursive formulation has two parts:

   (a) Describe the precise function you want to evaluate, in coherent English. Without this specification, it is impossible, even in principle, to determine whether your solution is correct.

   (b) Give a formal recursive definition of that function.

2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution, by considering intermediate subproblems in the correct order. This stage can be broken down into several smaller, relatively mechanical steps:

   (a) **Identify the subproblems.** What are all the different ways can your recursive algorithm call itself, starting with some initial input? For example, the argument to RECFIBO is always an integer between 0 and n.

   (b) **Analyze space and running time.** The number of possible distinct subproblems determines the space complexity of your memoized algorithm. To compute the time complexity, add up the running times of all possible subproblems, *ignoring the recursive calls*. For example, if we already know F_{i-1} and F_{i-2}, we can compute F_i in $O(1)$ time, so computing the first n Fibonacci numbers takes $O(n)$ time.

   (c) **Choose a data structure to memoize intermediate results.** For most problems, each recursive subproblem can be identified by a few integers, so you can use a multidimensional array. For some problems, however, a more complicated data structure is required.

(d) **Identify dependencies between subproblems.** Except for the base cases, every recursive subproblem depends on other subproblems—which ones? Draw a picture of your data structure, pick a generic element, and draw arrows from each of the other elements it depends on. Then formalize your picture.

(e) **Find a good evaluation order.** Order the subproblems so that each subproblem comes *after* the subproblems it depends on. Typically, this means you should consider the base cases first, then the subproblems that depends only on base cases, and so on. More formally, the dependencies you identified in the previous step define a partial order over the subproblems; in this step, you need to find a linear extension of that partial order. ***Be careful!***

(f) **Write down the algorithm.** You know what order to consider the subproblems, and you know how to solve each subproblem. So do that! If your data structure is an array, this usually means writing a few nested for-loops around your original recurrence. ***You don't need to do this on homework or exams.***

Of course, you have to prove that each of these steps is correct. If your recurrence is wrong, or if you try to build up answers in the wrong order, your algorithm won't work!

8.4 Warning: Greed is Stupid

If we're very very very very lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a *greedy* algorithm. The general greedy strategy is look for the best first step, take it, and then continue. While this approach seems very natural, it almost never works; optimization problems that can be solved correctly by a greedy algorithm are *very* rare. Nevertheless, for many problems that should be solved by dynamic programming, many students' first intuition is to apply a greedy strategy.

For example, a greedy algorithm for the edit distance problem might look for the longest common substring of the two strings, match up those substrings (since those substitutions don't cost anything), and then recursively look for the edit distances between the left halves and right halves of the strings. If there is no common substring—that is, if the two strings have no characters in common—the edit distance is clearly the length of the larger string. If this sounds like a stupid hack to you, pat yourself on the back. It isn't even *close* to the correct solution.

Everyone should tattoo the following sentence on the back of their hands, right under all the rules about logarithms and big-Oh notation:

Greedy algorithms never work!
Use dynamic programming instead!

What, never?
No, never!
What, *never*?
Well... hardly ever.[6]

A different lecture note describes the effort required to prove that greedy algorithms are correct, in the rare instances when they are. **You will not receive *any* credit for *any* greedy algorithm for *any***

[6] Greedy methods hardly ever work!
So give three cheers, and one cheer more,
for the careful Captain of the *Pinafore*!
Then give three cheers, and one cheer more,
for the Captain of the *Pinafore*!

problem in this class without a *formal* proof of correctness. We'll push through the formal proofs for several greedy algorithms later in the semester.

8.5 Edit Distance

The *edit distance* between two words—sometimes also called the *Levenshtein distance*—is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word into another. For example, the edit distance between FOOD and MONEY is at most four:

$$\underline{F}\text{OOD} \rightarrow \text{MO}\underline{O}\text{D} \rightarrow \text{MON}\underset{\wedge}{\text{D}} \rightarrow \text{MONE}\underline{D} \rightarrow \text{MONEY}$$

A better way to display this editing process is to place the words one above the other, with a gap in the first word for every insertion, and a gap in the second word for every deletion. Columns with two *different* characters correspond to substitutions. Thus, the number of editing steps is just the number of columns that don't contain the same character twice.

```
F  O  O     D
M  O  N  E  Y
```

It's fairly obvious that you can't get from FOOD to MONEY in three steps, so their edit distance is exactly four. Unfortunately, this is not so easy in general. Here's a longer example, showing that the distance between ALGORITHM and ALTRUISTIC is at most six. Is this optimal?

```
A  L  G  O  R     I     T  H  M
A  L     T  R  U  I  S  T  I  C
```

To develop a dynamic programming algorithm to compute the edit distance between two strings, we first need to develop a recursive definition. Our gap representation for edit sequences has a crucial "optimal substructure" property. Suppose we have the gap representation for the shortest edit sequence for two strings. **If we remove the last column, the remaining columns must represent the shortest edit sequence for the remaining substrings.** We can easily prove this by contradiction. If the substrings had a shorter edit sequence, we could just glue the last column back on and get a shorter edit sequence for the original strings. Once we figure out what should go in the last column, the Recursion Fairy will magically give us the rest of the optimal gap representation.

So let's recursively define the edit distance between two strings $A[1..m]$ and $B[1..n]$, which we denote by $\mathit{Edit}(A[1..m], B[1..n])$. If neither string is empty, there are three possibilities for the last column in the shortest edit sequence:

- **Insertion:** The last entry in the bottom row is empty. In this case, the edit distance is equal to $\mathit{Edit}(A[1..m-1], B[1..n])+1$. The $+1$ is the cost of the final insertion, and the recursive expression gives the minimum cost for the other columns.
- **Deletion:** The last entry in the top row is empty. In this case, the edit distance is equal to $\mathit{Edit}(A[1..m], B[1..n-1])+1$. The $+1$ is the cost of the final deletion, and the recursive expression gives the minimum cost for the other columns.
- **Substitution:** Both rows have characters in the last column. If the characters are the same, the substitution is free, so the edit distance is equal to $\mathit{Edit}(A[1..m-1], B[1..n-1])$. If the characters are different, then the edit distance is equal to $\mathit{Edit}(A[1..m-1], B[1..n-1])+1$.

The edit distance between A and B is the smallest of these three possibilities:[7]

$$Edit(A[1..m], B[1..n]) = \min \left\{ \begin{array}{l} Edit(A[1..m-1], B[1..n]) + 1 \\ Edit(A[1..m], B[1..n-1]) + 1 \\ Edit(A[1..m-1], B[1..n-1]) + \big[A[m] \neq B[n]\big] \end{array} \right\}$$

This recurrence has two easy base cases. The only way to convert the empty string into a string of n characters is by performing n insertions. Similarly, the only way to convert a string of m characters into the empty string is with m deletions, Thus, if ε denotes the empty string, we have

$$Edit(A[1..m], \varepsilon) = m, \qquad Edit(\varepsilon, B[1..n]) = n.$$

Both of these expressions imply the trivial base case $Edit(\varepsilon, \varepsilon) = 0$.

Now notice that the arguments to our recursive subproblems are always *prefixes* of the original strings A and B. We can simplify our notation by using the lengths of the prefixes, instead of the prefixes themselves, as the arguments to our recursive function.

Let $Edit(i, j)$ denote the edit distance between the prefixes $A[1..i]$ and $B[1..j]$.

This function satisfies the following recurrence:

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{l} Edit(i-1, j) + 1, \\ Edit(i, j-1) + 1, \\ Edit(i-1, j-1) + \big[A[i] \neq B[j]\big] \end{array} \right\} & \text{otherwise} \end{cases}$$

The edit distance between the original strings A and B is just $Edit(m, n)$. This recurrence translates directly into a recursive algorithm; the precise running time is not obvious, but it's clearly exponential in m and n. ***Fortunately, we don't care about the precise running time of the recursive algorithm.*** The recursive running time wouldn't tell us anything about our eventual dynamic programming algorithm, so we're just not going to bother computing it.[8]

Because each recursive subproblem can be identified by two indices i and j, we can memoize intermediate values in a two-dimensional array $\boldsymbol{Edit[0..m, 0..n]}$. Note that the index ranges start at zero to accommodate the base cases. Since there are $\Theta(mn)$ entries in the table, our memoized algorithm uses $\boldsymbol{\Theta(mn)}$ ***space***. Since each entry in the table can be computed in $\Theta(1)$ time once we know its predecessors, our memoized algorithm runs in $\boldsymbol{\Theta(mn)}$ ***time***.

[7] Once again, I'm using Iverson's bracket notation $\big[P\big]$ to denote the *indicator variable* for the logical proposition P, which has value 1 if P is true and 0 if P is false.

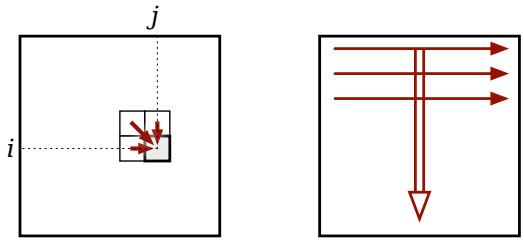
[8] In case you're curious, the running time of the unmemoized recursive algorithm obeys the following recurrence:

$$T(m,n) = \begin{cases} O(1) & \text{if } n = 0 \text{ or } m = 0, \\ T(m,n-1) + T(m-1,n) + T(n-1,m-1) + O(1) & \text{otherwise.} \end{cases}$$

I don't know of a general closed-form solution for this mess, but we can derive an upper bound by defining a new function

$$T'(N) = \max_{n+m=N} T(n,m) = \begin{cases} O(1) & \text{if } N = 0, \\ 2T(N-1) + T(N-2) + O(1) & \text{otherwise.} \end{cases}$$

The annihilator method implies that $T'(N) = O((1+\sqrt{2})^N)$. Thus, the running time of our recursive edit-distance algorithm is *at most* $T'(n+m) = O((1+\sqrt{2})^{n+m})$.



Dependencies in the memoization table for edit distance, and a legal evaluation order

Each entry $Edit[i,j]$ depends only on its three neighboring entries $Edit[i-1,j]+1$, $Edit[i,j-1]+1$, and $Edit[i-1,j-1]$. If we fill in our table in the standard row-major order—row by row from top down, each row from left to right—then whenever we reach an entry in the table, the entries it depends on are already available. Putting everything together, we obtain the following dynamic programming algorithm:

```
EDITDISTANCE(A[1..m], B[1..n]):
  for j ← 1 to n
      Edit[0, j] ← j

  for i ← 1 to m
      Edit[i, 0] ← i
      for j ← 1 to n
          if A[i] = B[j]
              Edit[i, j] ← min{Edit[i−1, j]+1, Edit[i, j−1]+1, Edit[i−1, j−1]}
          else
              Edit[i, j] ← min{Edit[i−1, j]+1, Edit[i, j−1]+1, Edit[i−1, j−1]+1}

  return Edit[m, n]
```

The resulting table for ALGORITHM → ALTRUISTIC is shown on the next page. Bold numbers indicate places where characters in the two strings are equal. The arrows represent the predecessor(s) that actually define each entry. Each direction of arrow corresponds to a different edit operation: horizontal=deletion, vertical=insertion, and diagonal=substitution. Bold diagonal arrows indicate "free" substitutions of a letter for itself. Any path of arrows from the top left corner to the bottom right corner of this table represents an optimal edit sequence between the two strings. (There can be many such paths.) Moreover, since we can compute these arrows in a post-processing phase from the values stored in the table, we can reconstruct the actual optimal editing sequence in $O(n+m)$ additional time.

The edit distance between ALGORITHM and ALTRUISTIC is indeed six. There are three paths through this table from the top left to the bottom right, so there are three optimal edit sequences:

```
  A  L  G  O  R  I     T  H  M
  A  L  T  R  U  I  S  T  I  C

A  L  G  O  R     I     T  H  M
A  L     T  R  U  I  S  T  I  C

A  L  G  O  R     I     T  H  M
A  L  T     R  U  I  S  T  I  C
```

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	**6**	5	5	6
C	10	9	8	8	8	8	7	6	6	6

The memoization table for *Edit*(ALGORITHM, ALTRUISTIC)

8.6 More Examples

In the previous note on backtracking algorithms, we saw two other examples of recursive algorithms that we can significantly speed up via dynamic programming.

8.6.1 Subset Sum

Recall that the *Subset Sum* problem asks, given a set X of positive integers (represented as an array $X[1..n]$ and an integer T, whether any subset of X sums to T. In that lecture, we developed a recursive algorithm which can be reformulated as follows. Fix the original input array $X[1..n]$ and the original target sum T, and define the boolean function

$$SS(i,t) = \text{some subset of } X[i..n] \text{ sums to } t.$$

Our goal is to compute $S(1,T)$, using the recurrence

$$SS(i,t) = \begin{cases} \textsc{True} & \text{if } t = 0, \\ \textsc{False} & \text{if } t < 0 \text{ or } i > n, \\ SS(i+1,t) \vee SS(i+1,t-X[i]) & \text{otherwise.} \end{cases}$$

There are only nT possible values for the input parameters that lead to the interesting case of this recurrence, and we can memoize all such values in an $n \times T$ array. If $S(i+1,t)$ and $S(i+1,t-X[i])$ are already known, we can compute $S(i,t)$ in constant time, so memoizing this recurrence gives us and algorithm that runs in ***O(nT) time***.[9] To turn this into an explicit dynamic programming algorithm, we only need to consider the subproblems $S(i,t)$ in the proper order:

[9]Even though *SubsetSum* is NP-complete, this bound does *not* imply that P=NP, because T is not necessary bounded by a polynomial function of the input size.

```
SUBSETSUM(X[1..n], T):
  S[n+1, 0] ← TRUE
  for t ← 1 to T
      S[n+1, t] ← FALSE

  for i ← n downto 1
      S[i, 0] = TRUE
      for t ← 1 to X[i]−1
          S[i, t] ← S[i+1, t]        ⟨⟨Avoid the case t < 0⟩⟩
      for t ← X[i] to T
          S[i, t] ← S[i+1, t] ∨ S[i+1, t−X[i]]

  return S[1, T]
```

This iterative algorithm clearly always uses ***O(nT) time and space***. In particular, if T is significantly larger than 2^n, this algorithm is actually slower than our naïve recursive algorithm. Dynamic programming isn't *always* an improvement!

8.6.2 NFA acceptance

The other problem we considered in the previous lecture note was determining whether a given NFA $M = (\Sigma, Q, s, A, \delta)$ accepts a given string $w \in \Sigma^*$. To make the problem concrete, we can assume without loss of generality that the alphabet is $\Sigma = \{1, 2, \ldots, |\Sigma|\}$, the state set is $Q = \{1, 2, \ldots, |Q|\}$, the start state is state 1, and our input consists of three arrays:

- A boolean array $A[1..|Q|]$, where $A[q] = \text{TRUE}$ if and only if $q \in A$.
- A boolean array $\delta[1..|Q|, 1..|\Sigma|, 1..|Q|]$, where $\delta[p, a, q] = \text{TRUE}$ if and only if $p \in \delta(q, a)$.
- An array $w[1..n]$ of symbols, representing the input string.

Now consider the boolean function

$$\textit{Accepts?}(q, i) = \text{TRUE} \text{ if and only if } M \text{ accepts the suffix } w[i..n] \text{ starting in state } q,$$

or equivalently,

$$\textit{Accepts?}(q, i) = \text{TRUE} \text{ if and only if } \delta^*(q, w[i..n]) \text{ contains at least one state in } A.$$

We need to compute $\textit{Accepts}(1, 1)$. The recursive definition of the string transition function δ^* implies the following recurrence for *Accepts?*:

$$\textit{Accepts?}(q, i) := \begin{cases} \text{TRUE} & \text{if } i > n \text{ and } q \in A \\ \text{FALSE} & \text{if } i > n \text{ and } q \in A \\ \displaystyle\bigvee_{r \in \delta(q,a)} \textit{Accepts?}(r, x) & \text{if } w = ax \end{cases}$$

Rewriting this recurrence in terms of our input representation gives us the following:

$$\textit{Accepts?}(q, i) := \begin{cases} \text{TRUE} & \text{if } i > n \text{ and } A[q] = \text{TRUE} \\ \text{FALSE} & \text{if } i > n \text{ and } A[q] = \text{FALSE} \\ \displaystyle\bigvee_{r=1}^{|Q|} \big(\delta[q, w[i], r] \wedge \textit{Accepts?}(r, i+1)\big) & \text{otherwise} \end{cases}$$

We can memoize this function into a two-dimensional array $Accepts?[1..|Q|, 1..n+1]$. Each entry $Accepts?[q,i]$ depends on some subset of entries of the form $Accepts?[r,i+1]$. So we can fill the memoization table by considering the possible indices i in decreasing order in the outer loop, and consider states q in arbitrary order in the inner loop. Evaluating each entry $Accepts?[q,i]$ requires $O(|Q|)$ time, using an even deeper loop over all states r, and there are $O(n|Q|)$ such entries. Thus, the entire dynamic programming algorithm requires $\boldsymbol{O(n|Q|^2)}$ ***time***.

```
NFAACCEPTS?(A[1..|Q|], δ[1..|Q|, 1..|Σ|, 1..|Q|], w[1..n]):
  for q ← 1 to |Q|
      Accepts?[q, n+1] ← A[q]
  for i ← n down to 1
      for q ← 1 to |Q|
          Accepts?[q, i] ← FALSE
          for r ← 1 to |Q|
              if δ[q, w[i], r] and Accepts?[r, i+1]
                  Accepts?[q, i] ← TRUE
  return Accepts?[1, 1]
```

8.7 Optimal Binary Search Trees

In an earlier lecture, we developed a recursive algorithm for the optimal binary search tree problem. We are given a sorted array $A[1..n]$ of search keys and an array $f[1..n]$ of frequency counts, where $f[i]$ is the number of searches to $A[i]$. Our task is to construct a binary search tree for that set such that the total cost of all the searches is as small as possible. We developed the following recurrence for this problem:

$$OptCost(f[1..n]) = \min_{1\le r\le n}\left\{OptCost(f[1..r-1]) \;+\; \sum_{i=1}^{n} f[i] \;+\; OptCost(f[r+1..n])\right\}$$

To put this recurrence in more standard form, fix the frequency array f, and let $OptCost(i,j)$ denote the total search time in the optimal search tree for the subarray $A[i..j]$. To simplify notation a bit, let $F(i,j)$ denote the total frequency count for all the keys in the interval $A[i..j]$:

$$F(i,j) := \sum_{k=i}^{j} f[k]$$

We can now write

$$OptCost(i,j) = \begin{cases} 0 & \text{if } j < i \\ F(i,j) + \min\limits_{i\le r\le j}\big(OptCost(i,r-1) + OptCost(r+1,j)\big) & \text{otherwise} \end{cases}$$

The base case might look a little weird, but all it means is that the total cost for searching an empty set of keys is zero.

The algorithm will be somewhat simpler and more efficient if we precompute all possible values of $F(i,j)$ and store them in an array. Computing each value $F(i,j)$ using a separate for-loop would $O(n^3)$ time. A better approach is to turn the recurrence

$$F(i,j) = \begin{cases} f[i] & \text{if } i = j \\ F(i,j-1) + f[j] & \text{otherwise} \end{cases}$$

into the following $O(n^2)$-time dynamic programming algorithm:

```
INITF(f[1..n]):
  for i ← 1 to n
      F[i, i−1] ← 0
      for j ← i to n
          F[i, j] ← F[i, j−1] + f[j]
```

This will be used as an initialization subroutine in our final algorithm.

So now let's compute the optimal search tree cost $OptCost(1, n)$ from the bottom up. We can store all intermediate results in a table $OptCost[1..n, 0..n]$. Only the entries $OptCost[i, j]$ with $j \geq i-1$ will actually be used. The base case of the recurrence tells us that any entry of the form $OptCost[i, i-1]$ can immediately be set to 0. For any other entry $OptCost[i, j]$, we can use the following algorithm fragment, which comes directly from the recurrence:

```
COMPUTEOPTCOST(i, j):
  OptCost[i, j] ← ∞
  for r ← i to j
      tmp ← OptCost[i, r−1] + OptCost[r+1, j]
      if OptCost[i, j] > tmp
          OptCost[i, j] ← tmp
  OptCost[i, j] ← OptCost[i, j] + F[i, j]
```

The only question left is what order to fill in the table.

Each entry $OptCost[i, j]$ depends on all entries $OptCost[i, r-1]$ and $OptCost[r+1, j]$ with $i \leq k \leq j$. In other words, every entry in the table depends on all the entries directly to the left or directly below. In order to fill the table efficiently, we must choose an order that computes all those entries before $OptCost[i, j]$. There are at least three different orders that satisfy this constraint. The one that occurs to most people first is to scan through the table one diagonal at a time, starting with the trivial base cases $OptCost[i, i-1]$. The complete algorithm looks like this:

```
OPTIMALSEARCHTREE(f[1..n]):
  INITF(f[1..n])
  for i ← 1 to n
      OptCost[i, i−1] ← 0
  for d ← 0 to n−1
      for i ← 1 to n−d
          COMPUTEOPTCOST(i, i+d)
  return OptCost[1, n]
```

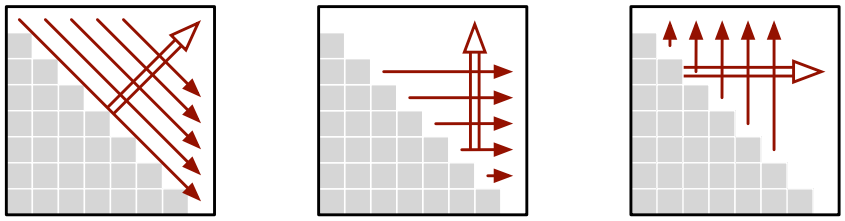
We could also traverse the array row by row from the bottom up, traversing each row from left to right, or column by column from left to right, traversing each columns from the bottom up.

```
OPTIMALSEARCHTREE2(f[1..n]):
  INITF(f[1..n])
  for i ← n downto 1
      OptCost[i, i−1] ← 0
      for j ← i to n
          COMPUTEOPTCOST(i, j)
  return OptCost[1, n]
```

```
OPTIMALSEARCHTREE3(f[1..n]):
  INITF(f[1..n])
  for j ← 0 to n
      OptCost[j+1, j] ← 0
      for i ← j downto 1
          COMPUTEOPTCOST(i, j)
  return OptCost[1, n]
```

No matter which of these orders we actually use, the resulting algorithm runs in $\boldsymbol{\Theta(n^3)}$ ***time*** and uses $\boldsymbol{\Theta(n^2)}$ ***space***. We could have predicted these space and time bounds directly from the original recurrence.

$$OptCost(i, j) = \begin{cases} 0 & \text{if } j = i - i \\ F(i, j) + \min\limits_{i \leq r \leq j} \big(OptCost(i, r-1) + OptCost(r+1, j) \big) & \text{otherwise} \end{cases}$$



Three different evaluation orders for the table $OptCost[i, j]$.

First, the function has two arguments, each of which can take on any value between 1 and n, so we probably need a table of size $O(n^2)$. Next, there are *three* variables in the recurrence (i, j, and r), each of which can take any value between 1 and n, so it should take us $O(n^3)$ time to fill the table.

8.8 The CYK Parsing Algorithm

In the same earlier lecture, we developed a recursive backtracking algorithm for parsing context-free languages. The input consists of a string w and a context-free grammar G in Chomsky normal form—meaning every production has the form $A \to a$, for some symbol a, or $A \to BC$, for some non-terminals B and C. Our task is to determine whether w is in the language generated by G.

Our backtracking algorithm recursively evaluates the boolean function $Generates?(A, x)$, which equals TRUE if and only if string x can be derived from non-terminal A, using the following recurrence:

$$Generates?(A,x) = \begin{cases} \text{TRUE} & \text{if } |x| = 1 \text{ and } A \to x \\ \text{FALSE} & \text{if } |x| = 1 \text{ and } A \not\to x \\ \displaystyle\bigvee_{A \to BC} \bigvee_{y \bullet z = x} Generates?(B,y) \wedge Generates?(C,z) & \text{otherwise} \end{cases}$$

This recurrence was transformed into a dynamic programming algorithm by Tadao Kasami in 1965, and again independently by Daniel Younger in 1967, and again independently by John Cocke in 1970, so naturally the resulting algorithm is known as "Cocke-Younger-Kasami", or more commonly ***the CYK algorithm***.

We can derive the CYK algorithm from the previous recurrence as follows. As usual for recurrences involving strings, we need to modify the function slightly to ease memoization. Fix the input string w, and then let $Generates?(A, i, j) = \text{TRUE}$ if and only if the substring $w[i..j]$ can be derived from non-terminal A. Now our earlier recurrence can be rewritten as follows:

$$Generates?(A,i,j) = \begin{cases} \text{TRUE} & \text{if } i = j \text{ and } A \to w[i] \\ \text{FALSE} & \text{if } i = j \text{ and } A \not\to w[i] \\ \displaystyle\bigvee_{A \to BC} \bigvee_{k=i}^{j-1} Generates?(B,i,k) \wedge Generates?(C,k+1,j) & \text{otherwise} \end{cases}$$

This recurrence can be memoized into a three-dimensional boolean array $Gen[1..|\Gamma|, 1..n, 1..n]$, where the first dimension is indexed by the non-terminals Γ in the input grammar. Each entry $Gen[A, i, j]$ in this array depends on entries of the form $Gen[\cdot, i, k]$ for some $k < j$, or $Gen[\cdot, k+1, j]$ for some $k \geq i$. Thus, we can fill the array by increasing j in the outer loop, decreasing i in the middle loop, and considering non-terminals A in arbitrary order in the inner loop. The resulting dynamic programming algorithm runs in $\boldsymbol{O(n^3 \cdot |\Gamma|)}$ ***time***.

```
CYK(w, G):
  for i ← 1 to n
      for all non-terminals A
          if G contains the production A → w[i]
              Gen[A, i, i] ← TRUE
          else
              Gen[A, i, i] ← FALSE
  for j ← 1 to n
      for i ← n down to j + 1
          for all non-terminals A
              Gen[A, i, j] ← FALSE
          for all production rules A → BC
              for k ← i to j − 1
                  if Gen[B, i, k] and Gen[C, k + 1, j]
                      Gen[A, i, j] ← TRUE
  return Gen[S, 1, n]
```

8.9 Dynamic Programming on Trees

So far, all of our dynamic programming example use a multidimensional array to store the results of recursive subproblems. However, as the next example shows, this is not always the most appropriate date structure to use.

A ***independent set*** in a graph is a subset of the vertices that have no edges between them. Finding the largest independent set in an arbitrary graph is extremely hard; in fact, this is one of the canonical NP-hard problems described in another lecture note. But from some special cases of graphs, we can find the largest independent set efficiently. In particular, when the input graph is a tree (a connected and acyclic graph) with n vertices, we can compute the largest independent set in $O(n)$ time.

In the recursion notes, we saw a recursive algorithm for computing the size of the largest independent set in an arbitrary graph:

```
MAXIMUMINDSETSIZE(G):
  if G = ∅
      return 0

  v ← any node in G
  withv ← 1 + MAXIMUMINDSETSIZE(G \ N(v))
  withoutv ← MAXIMUMINDSETSIZE(G \ {v})
  return max{withv, withoutv}.
```

Here, $N(v)$ denotes the *neighborhood* of v: the set containing v and all of its neighbors. As we observed in the other lecture notes, this algorithm has a worst-case running time of $O(2^n \operatorname{poly}(n))$, where n is the number of vertices in the input graph.

Now suppose we require that the input graph is a tree; we will call this tree T instead of G from now on. We need to make a slight change to the algorithm to make it truly recursive. The subgraphs $T \setminus \{v\}$ and $T \setminus N(v)$ are forests, which may have more than one component. But the largest independent set in a disconnected graph is just the union of the largest independent sets in its components, so we can separately consider each tree in these forests. Fortunately, this has the added benefit of making the recursive algorithm more efficient, especially if we can choose the node v such that the trees are all significantly smaller than T. Here is the modified algorithm:

```
MAXIMUMINDSETSIZE(T):
  if T = ∅
      return 0
  v ← any node in T
  withv ← 1
  for each tree T' in T \ N(v)
      withv ← withv + MAXIMUMINDSETSIZE(T')
  withoutv ← 0
  for each tree T' in T \ {v}
      withoutv ← withoutv + MAXIMUMINDSETSIZE(T')
  return max{withv, withoutv}.
```

Now let's try to memoize this algorithm. Each recursive subproblem considers a subtree (that is, a connected subgraph) of the original tree T. Unfortunately, a single tree T can have exponentially many subtrees, so we seem to be doomed from the start!

Fortunately, there's a degree of freedom that we have not yet exploited: *We get to choose the vertex v.* We need a recipe—an algorithm!—for choosing v in each subproblem that limits the number of different subproblems the algorithm considers. To make this work, we impose some additional structure on the original input tree. Specifically, we declare one of the vertices of T to be the *root*, and we orient all the edges of T away from that root. Then we let v be the root of the input tree; this choice guarantees that each recursive subproblem considers a *rooted* subtree of T. Each vertex in T is the root of exactly one subtree, so now the number of distinct subproblems is exactly n. We can further simplify the algorithm by only passing a single node instead of the entire subtree:

```
MAXIMUMINDSETSIZE(v):
  withv ← 1
  for each grandchild x of v
      withv ← withv + MAXIMUMINDSETSIZE(x)
  withoutv ← 0
  for each child w of v
      withoutv ← withoutv + MAXIMUMINDSETSIZE(w)
  return max{withv, withoutv}.
```

What data structure should we use to store intermediate results? The most natural choice is the tree itself! Specifically, for each node v, we store the result of MAXIMUMINDSETSIZE(v) in a new field $v.MIS$. (We *could* use an array, but then we'd have to add a new field to each node anyway, pointing to the corresponding array entry. Why bother?)

What's the running time of the algorithm? The non-recursive time associated with each node v is proportional to the number of children and grandchildren of v; this number can be very different from one vertex to the next. But we can turn the analysis around: Each vertex contributes a constant amount of time to its parent and its grandparent! Since each vertex has at most one parent and at most one grandparent, the total running time is $O(n)$.

What's a good order to consider the subproblems? The subproblem associated with any node v depends on the subproblems associated with the children and grandchildren of v. So we can visit the nodes in any order, provided that all children are visited before their parent. In particular, we can use a straightforward post-order traversal.

Here is the resulting dynamic programming algorithm. Yes, it's still recursive. I've swapped the evaluation of the with-v and without-v cases; we need to visit the kids first anyway, so why not consider the subproblem that depends directly on the kids first?

```
MAXIMUMINDSETSIZE(v):
  withoutv ← 0
  for each child w of v
      withoutv ← withoutv + MAXIMUMINDSETSIZE(w)
  withv ← 1
  for each grandchild x of v
      withv ← withv + x.MIS
  v.MIS ← max{withv, withoutv}
  return v.MIS
```

Another option is to store *two* values for each rooted subtree: the size of the largest independent set *that includes the root*, and the size of the largest independent set *that excludes the root*. This gives us an even simpler algorithm, with the same $O(n)$ running time.

```
MAXIMUMINDSETSIZE(v):
  v.MISno ← 0
  v.MISyes ← 1
  for each child w of v
      v.MISno ← v.MISno + MAXIMUMINDSETSIZE(w)
      v.MISyes ← v.MISyes + w.MISno
  return max{v.MISyes, v.MISno}
```

Exercises

Sequences/Arrays

1. In a previous life, you worked as a cashier in the lost Antarctican colony of Nadira, spending the better part of your day giving change to your customers. Because paper is a very rare and valuable resource in Antarctica, cashiers were required by law to use the fewest bills possible whenever they gave change. Thanks to the numerological predilections of one of its founders, the currency of Nadira, called Dream Dollars, was available in the following denominations: \$1, \$4, \$7, \$13, \$28, \$52, \$91, \$365.[10]

   (a) The greedy change algorithm repeatedly takes the largest bill that does not exceed the target amount. For example, to make \$122 using the greedy algorithm, we first take a \$91 bill, then a \$28 bill, and finally three \$1 bills. Give an example where this greedy algorithm uses more Dream Dollar bills than the minimum possible.

   (b) Describe and analyze a recursive algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream Dollars. (Don't worry about making your algorithm fast; just make sure it's correct.)

   (c) Describe a dynamic programming algorithm that computes, given an integer k, the minimum number of bills needed to make k Dream Dollars. (This one needs to be fast.)

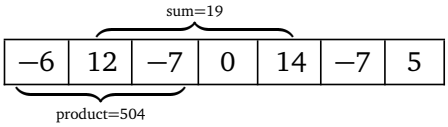
2. Suppose you are given an array $A[1..n]$ of numbers, which may be positive, negative, or zero, and which are ***not*** necessarily integers.

   (a) Describe and analyze an algorithm that finds the largest sum of of elements in a contiguous subarray $A[i..j]$.

[10] For more details on the history and culture of Nadira, including images of the various denominations of Dream Dollars, see http://moneyart.biz/dd/.

(b) Describe and analyze an algorithm that finds the largest *product* of of elements in a contiguous subarray $A[i..j]$.

For example, given the array $[-6, 12, -7, 0, 14, -7, 5]$ as input, your first algorithm should return the integer 19, and your second algorithm should return the integer 504.



For the sake of analysis, assume that comparing, adding, or multiplying any pair of numbers takes $O(1)$ time.

[Hint: Problem (a) has been a standard computer science interview question since at least the mid-1980s. You can find many correct solutions on the web; the problem even has its own Wikipedia page! But at least in 2013, the few solutions I found on the web for problem (b) were all either slower than necessary or incorrect.]

3. This series of exercises asks you to develop efficient algorithms to find optimal *subsequences* of various kinds. A subsequence is anything obtained from a sequence by extracting a subset of elements, but keeping them in the same order; the elements of the subsequence need not be contiguous in the original sequence. For example, the strings `C`, `DAMN`, `YAIOAI`, and `DYNAMICPROGRAMMING` are all subsequences of the string `DYNAMICPROGRAMMING`.

   (a) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common subsequence* of A and B is another sequence that is a subsequence of both A and B. Describe an efficient algorithm to compute the length of the *longest* common subsequence of A and B.

   (b) Let $A[1..m]$ and $B[1..n]$ be two arbitrary arrays. A *common supersequence* of A and B is another sequence that contains both A and B as subsequences. Describe an efficient algorithm to compute the length of the *shortest* common supersequence of A and B.

   (c) Call a sequence $X[1..n]$ of numbers *bitonic* if there is an index i with $1 < i < n$, such that the prefix $X[1..i]$ is increasing and the suffix $X[i..n]$ is decreasing. Describe an efficient algorithm to compute the length of the longest bitonic subsequence of an arbitrary array A of integers.

   (d) Call a sequence $X[1..n]$ of numbers *oscillating* if $X[i] < X[i+1]$ for all even i, and $X[i] > X[i+1]$ for all odd i. Describe an efficient algorithm to compute the length of the longest oscillating subsequence of an arbitrary array A of integers.

   (e) Describe an efficient algorithm to compute the length of the shortest oscillating supersequence of an arbitrary array A of integers.

   (f) Call a sequence $X[1..n]$ of numbers *convex* if $2 \cdot X[i] < X[i-1] + X[i+1]$ for all i. Describe an efficient algorithm to compute the length of the longest convex subsequence of an arbitrary array A of integers.

   *(g) Recall that a sequence $X[1..n]$ of numbers is *increasing* if $X[i] < X[i+1]$ for all i. Describe an efficient algorithm to compute the length of the *longest common increasing subsequence* of two given arrays of integers. For example, $\langle 1, 4, 5, 6, 7, 9\rangle$ is the longest common increasing subsequence of the sequences $\langle 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3\rangle$ and $\langle 1, 4, 1, 4, 2, 1, 3, 5, 6, 2, 3, 7, 3, 0, 9, 5\rangle$.

4. Describe an algorithm to compute the number of times that one given array $X[1..k]$ appears as a subsequence of another given array $Y[1..n]$. For example, if all characters in X and Y are equal, your algorithm should return $\binom{n}{k}$. For purposes of analysis, assume that adding two ℓ-bit integers requires $\Theta(\ell)$ time.

5. You and your eight-year-old nephew Elmo decide to play a simple card game. At the beginning of the game, the cards are dealt face up in a long row. Each card is worth a different number of points. After all the cards are dealt, you and Elmo take turns removing either the leftmost or rightmost card from the row, until all the cards are gone. At each turn, you can decide which of the two cards to take. The winner of the game is the player that has collected the most points when the game ends.

   Having never taken an algorithms class, Elmo follows the obvious greedy strategy—when it's his turn, Elmo *always* takes the card with the higher point value. Your task is to find a strategy that will beat Elmo whenever possible. (It might seem mean to beat up on a little kid like this, but Elmo absolutely *hates* it when grown-ups let him win.)

   (a) Prove that you should not also use the greedy strategy. That is, show that there is a game that you can win, but only if you do *not* follow the same greedy strategy as Elmo.

   (b) Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against Elmo.

   (c) Five years later, Elmo has become a *much* stronger player. Describe and analyze an algorithm to determine, given the initial sequence of cards, the maximum number of points that you can collect playing against a *perfect* opponent.

6. It's almost time to show off your flippin' sweet dancing skills! Tomorrow is the big dance contest you've been training for your entire life, except for that summer you spent with your uncle in Alaska hunting wolverines. You've obtained an advance copy of the the list of n songs that the judges will play during the contest, in chronological order.

   You know all the songs, all the judges, and your own dancing ability extremely well. For each integer k, you know that if you dance to the kth song on the schedule, you will be awarded exactly $Score[k]$ points, but then you will be physically unable to dance for the next $Wait[k]$ songs (that is, you cannot dance to songs $k+1$ through $k+Wait[k]$). The dancer with the highest total score at the end of the night wins the contest, so you want your total score to be as high as possible.

   Describe and analyze an efficient algorithm to compute the maximum total score you can achieve. The input to your sweet algorithm is the pair of arrays $Score[1..n]$ and $Wait[1..n]$.

7. You are driving a bus along a highway, full of rowdy, hyper, thirsty students and a soda fountain machine. Each minute that a student is on your bus, that student drinks one ounce of soda. Your goal is to drop the students off quickly, so that the total amount of soda consumed by all students is as small as possible.

   You know how many students will get off of the bus at each exit. Your bus begins somewhere along the highway (probably not at either end) and move s at a constant speed of 37.4 miles per hour. You must drive the bus along the highway; however, you may drive forward to one exit then backward to an exit in the opposite direction, switching as often as you like. (You can stop the bus, drop off students, and turn around instantaneously.)

Describe an efficient algorithm to drop the students off so that they drink as little soda as possible. Your input consists of the bus route (a list of the exits, together with the travel time between successive exits), the number of students you will drop off at each exit, and the current location of your bus (which you may assume is an exit).

8. A palindrome is any string that is exactly the same as its reversal, like `I`, or `DEED`, or `RACECAR`, or `AMANAPLANACATACANALPANAMA`.

   (a) Describe and analyze an algorithm to find the length of the *longest subsequence* of a given string that is also a palindrome. For example, the longest palindrome subsequence of `MAHDYNAMICPROGRAMZLETMESHOWYOUTHEM` is `MHYMRORMYHM`, so given that string as input, your algorithm should output the number 11.

   (b) Describe and analyze an algorithm to find the length of the *shortest supersequence* of a given string that is also a palindrome. For example, the shortest palindrome supersequence of `TWENTYONE` is `TWENTOYOTNEWT`, so given the string `TWENTYONE` as input, your algorithm should output the number 13.

   (c) Any string can be decomposed into a sequence of palindromes. For example, the string `BUBBASEESABANANA` ("Bubba sees a banana.") can be broken into palindromes in the following ways (and many others):

   `BUB` • `BASEESAB` • `ANANA`

   `B` • `U` • `BB` • `A` • `SEES` • `ABA` • `NAN` • `A`

   `B` • `U` • `BB` • `A` • `SEES` • `A` • `B` • `ANANA`

   `B` • `U` • `B` • `B` • `A` • `S` • `E` • `E` • `S` • `A` • `B` • `A` • `N` • `ANA`

   Describe and analyze an efficient algorithm to find the smallest number of palindromes that make up a given input string. For example, given the input string `BUBBASEESABANANA`, your algorithm would return the integer 3.

9. Suppose you have a black-box subroutine QUALITY that can compute the 'quality' of any given string $A[1..k]$ in $O(k)$ time. For example, the quality of a string might be 1 if the string is a Québecois curse word, and 0 otherwise.

   Given an arbitrary input string $T[1..n]$, we would like to break it into contiguous substrings, such that the total quality of all the substrings is as large as possible. For example, the string `SAINTCIBOIREDESACRAMENTDECRISSE` can be decomposed into the substrings `SAINT` • `CIBOIRE` • `DE` • `SACRAMENT` • `DE` • `CRISSE`, of which three (or possibly four) are *sacres*.

   Describe an algorithm that breaks a string into substrings of maximum total quality, using the QUALITY subroutine.

10. (a) Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

    (b) Suppose we are given a set L of n line segments in the plane, where each segment has one endpoint on the line $y = 0$ and one endpoint on the line $y = 1$, and all $2n$ endpoints are

distinct. Describe and analyze an algorithm to compute the largest subset of L in which ***every*** pair of segments intersects.

(c) Suppose we are given a set L of n line segments in the plane, where the endpoints of each segment lie on the unit circle $x^2 + y^2 = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which no pair of segments intersects.

(d) Suppose we are given a set L of n line segments in the plane, where the endpoints of each segment lie on the unit circle $x^2 + y^2 = 1$, and all $2n$ endpoints are distinct. Describe and analyze an algorithm to compute the largest subset of L in which ***every*** pair of segments intersects.

11. Let P be a set of n points evenly distributed on the unit circle, and let S be a set of m line segments with endpoints in P. The endpoints of the m segments are *not* necessarily distinct; n could be significantly smaller than $2m$.

(a) Describe an algorithm to find the size of the largest subset of segments in S such that every pair is disjoint. Two segments are disjoint if they do not intersect even at their endpoints.

(b) Describe an algorithm to find the size of the largest subset of segments in S such that every pair is interior-disjoint. Two segments are interior-disjoint if their intersection is either empty or an endpoint of both segments.

(c) Describe an algorithm to find the size of the largest subset of segments in S such that every pair intersects.

(d) Describe an algorithm to find the size of the largest subset of segments in S such that every pair crosses. Two segments cross if they intersect but not at their endpoints.

For full credit, all four algorithms should run in $O(mn)$ time.

12. A *shuffle* of two strings X and Y is formed by interspersing the characters into a new string, keeping the characters of X and Y in the same order. For example, the string BANANAANANAS is a shuffle of the strings BANANA and ANANAS in several different ways.

$$\texttt{BANANA}_{\texttt{ANANAS}} \qquad \texttt{BAN}_{\texttt{ANA}}\texttt{ANA}_{\texttt{NAS}} \qquad \texttt{B}_{\texttt{AN}}\texttt{AN}_{\texttt{A}}\texttt{A}_{\texttt{NA}}\texttt{NA}_{\texttt{S}}$$

Similarly, the strings PRODGYRNAMAMMIINCG and DYPRONGARMAMMICING are both shuffles of DYNAMIC and PROGRAMMING:

$$\texttt{PRO}^{\texttt{D}}\texttt{G}^{\texttt{Y}}\texttt{R}^{\texttt{NAM}}{}_{\texttt{AMMI}}{}^{\texttt{I}}\texttt{N}^{\texttt{C}}\texttt{G} \qquad \texttt{DY}_{\texttt{PRO}}{}^{\texttt{N}}\texttt{G}^{\texttt{A}}\texttt{R}^{\texttt{M}}{}_{\texttt{AMM}}{}^{\texttt{IC}}{}_{\texttt{ING}}$$

Given three strings $A[1..m]$, $B[1..n]$, and $C[1..m+n]$, describe and analyze an algorithm to determine whether C is a shuffle of A and B.

13. Describe and analyze an efficient algorithm to find the length of the longest contiguous substring that appears both forward and backward in an input string $T[1..n]$. The forward and backward substrings must not overlap. Here are several examples:

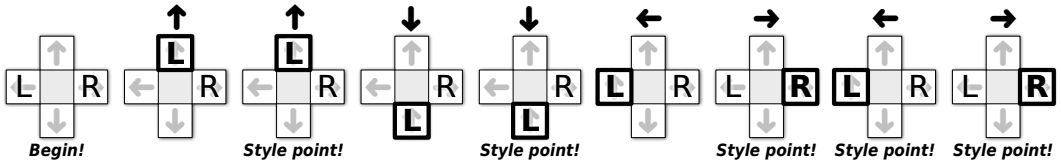
- Given the input string ALGORITHM, your algorithm should return 0.

- Given the input string RECURSION, your algorithm should return 1, for the substring R.
- Given the input string REDIVIDE, your algorithm should return 3, for the substring EDI. (The forward and backward substrings must not overlap!)
- Given the input string DYNAMICPROGRAMMINGMANYTIMES, your algorithm should return 4, for the substring YNAM. (In particular, it should *not* return 6, for the subsequence YNAMIR).

14. ***Dance Dance Revolution*** is a dance video game, first introduced in Japan by Konami in 1998. Players stand on a platform marked with four arrows, pointing forward, back, left, and right, arranged in a cross pattern. During play, the game plays a song and scrolls a sequence of n arrows (←, ↑, ↓, or →) from the bottom to the top of the screen. At the precise moment each arrow reaches the top of the screen, the player must step on the corresponding arrow on the dance platform. (The arrows are timed so that you'll step with the beat of the song.)

    You are playing a variant of this game called "Vogue Vogue Revolution", where the goal is to play perfectly but move as little as possible. When an arrow reaches the top of the screen, if one of your feet is already on the correct arrow, you are awarded one style point for maintaining your current pose. If neither foot is on the right arrow, you must move one (and *only* one) of your feet from its current location to the correct arrow on the platform. If you ever step on the wrong arrow, or fail to step on the correct arrow, or move more than one foot at a time, or move either foot when you are already standing on the correct arrow, all your style points are taken away and you lose the game.

    How should you move your feet to maximize your total number of style points? For purposes of this problem, assume you always start with you left foot on ← and you right foot on →, and that you've memorized the entire sequence of arrows. For example, if the sequence is ↑↑↓↓←→←→, you can earn 5 style points by moving you feet as shown below:



    (a) ***Prove*** that for *any* sequence of n arrows, it is possible to earn at least $n/4 - 1$ style points.

    (b) Describe an efficient algorithm to find the maximum number of style points you can earn during a given VVR routine. The input to your algorithm is an array $Arrow[1..n]$ containing the sequence of arrows.

15. Consider the following solitaire form of Scrabble. We begin with a fixed, finite sequence of tiles; each tile contains a letter and a numerical value. At the start of the game, we draw the seven tiles from the sequence and put them into our hand. In each turn, we form an English word from some or all of the tiles in our hand, place those tiles on the table, and receive the total value of those tiles as points. If no English word can be formed from the tiles in our hand, the game immediately ends. Then we repeatedly draw the next tile from the start of the sequence until either (a) we have seven tiles in our hand, or (b) the sequence is empty. (Sorry, no double/triple word/letter scores, bingos, blanks, or passing.) Our goal is to obtain as many points as possible.

    For example, suppose we are given the tile sequence

$\boxed{\texttt{I}_2}\boxed{\texttt{N}_2}\boxed{\texttt{X}_8}\boxed{\texttt{A}_1}\boxed{\texttt{N}_2}\boxed{\texttt{A}_1}\boxed{\texttt{D}_3}\boxed{\texttt{U}_5}\boxed{\texttt{D}_3}\boxed{\texttt{I}_2}\boxed{\texttt{D}_3}\boxed{\texttt{K}_8}\boxed{\texttt{U}_5}\boxed{\texttt{B}_4}\boxed{\texttt{L}_2}\boxed{\texttt{A}_1}\boxed{\texttt{K}_8}\boxed{\texttt{H}_5}\boxed{\texttt{A}_1}\boxed{\texttt{N}_2}$.

Then we can earn 68 points as follows:

- We initially draw $\boxed{\texttt{I}_2}\boxed{\texttt{N}_2}\boxed{\texttt{X}_8}\boxed{\texttt{A}_1}\boxed{\texttt{N}_2}\boxed{\texttt{A}_1}\boxed{\texttt{D}_3}$.
- Play the word $\boxed{\texttt{N}_2}\boxed{\texttt{A}_1}\boxed{\texttt{I}_2}\boxed{\texttt{A}_1}\boxed{\texttt{D}_3}$ for 9 points, leaving $\boxed{\texttt{N}_2}\boxed{\texttt{X}_8}$ in our hand.
- Draw the next five tiles $\boxed{\texttt{U}_5}\boxed{\texttt{D}_3}\boxed{\texttt{I}_2}\boxed{\texttt{D}_3}\boxed{\texttt{K}_8}$.
- Play the word $\boxed{\texttt{U}_5}\boxed{\texttt{N}_2}\boxed{\texttt{D}_3}\boxed{\texttt{I}_2}\boxed{\texttt{D}_3}$ for 15 points, leaving $\boxed{\texttt{K}_8}\boxed{\texttt{X}_8}$ in our hand.
- Draw the next five tiles $\boxed{\texttt{U}_5}\boxed{\texttt{B}_4}\boxed{\texttt{L}_2}\boxed{\texttt{A}_1}\boxed{\texttt{K}_8}$.
- Play the word $\boxed{\texttt{B}_4}\boxed{\texttt{U}_5}\boxed{\texttt{L}_2}\boxed{\texttt{K}_8}$ for 19 points, leaving $\boxed{\texttt{K}_8}\boxed{\texttt{X}_8}\boxed{\texttt{A}_1}$ in our hand.
- Draw the next three tiles $\boxed{\texttt{H}_5}\boxed{\texttt{A}_1}\boxed{\texttt{N}_2}$, emptying the list.
- Play the word $\boxed{\texttt{A}_1}\boxed{\texttt{N}_2}\boxed{\texttt{K}_8}\boxed{\texttt{H}_5}$ for 16 points, leaving $\boxed{\texttt{X}_8}\boxed{\texttt{A}_1}$ in our hand.
- Play the word $\boxed{\texttt{A}_1}\boxed{\texttt{X}_8}$ for 9 points, emptying our hand and ending the game.

(a) Suppose you are given as input two arrays $Letter[1..n]$, containing a sequence of letters between `A` and `Z`, and $Value[\texttt{A}..\texttt{Z}]$, where $Value[\ell]$ is the value of letter ℓ. Design and analyze an efficient algorithm to compute the maximum number of points that can be earned from the given sequence of tiles.

(b) Now suppose two tiles with the same letter can have different values; you are given two arrays $Letter[1..n]$ and $Value[1..n]$. Design and analyze an efficient algorithm to compute the maximum number of points that can be earned from the given sequence of tiles.

In both problems, the output is a single number: the maximum possible score. Assume that you can find all English words that can be made from any set of at most seven tiles, along with the point values of those words, in $O(1)$ time.

16. Suppose you are given a DFA $M = (\{\texttt{0},\texttt{1}\}, Q, s, A, \delta)$ and a binary string $w \in \{\texttt{0},\texttt{1}\}^*$.

(a) Describe and analyze an algorithm that computes the longest subsequence of w that is accepted by M, or correctly reports that M does not accept any subsequence of w.

*(b) Describe and analyze an algorithm that computes the *shortest supersequence* of w that is accepted by M, or correctly reports that M does not accept any supersequence of w. *[Hint: Careful!]*

Analyze both of your algorithms in terms of the parameters $n = |w|$ and $k = |Q|$.

17. *Vankin's Mile* is an American solitaire game played on an $n \times n$ square grid. The player starts by placing a token on any square of the grid. Then on each turn, the player moves the token either one square to the right or one square down. The game ends when player moves the token off the edge of the board. Each square of the grid has a numerical value, which could be positive, negative, or zero. The player starts with a score of zero; whenever the token lands on a square, the player adds its value to his score. The object of the game is to score as many points as possible.

For example, given the grid below, the player can score $8-6+7-3+4=10$ points by placing the initial token on the 8 in the second row, and then moving down, down, right, down, down. (This is *not* the best possible score for these values.)

−1	7	−8	10	−5
−4	−9	**8** ⇓	−6	0
5	−2	**−6** ⇓	−6	7
−7	4	**7** ⇒	**−3** ⇓	−3
7	1	−6	**4** ⇓	−9

(a) Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin's Mile, given the $n \times n$ array of values as input.

(b) In the European version of this game, appropriately called *Vankin's Kilometer*, the player can move the token either one square down, one square right, *or one square left* in each turn. However, to prevent infinite scores, the token cannot land on the same square more than once. Describe and analyze an efficient algorithm to compute the maximum possible score for a game of Vankin's Kilometer, given the $n \times n$ array of values as input.[11]

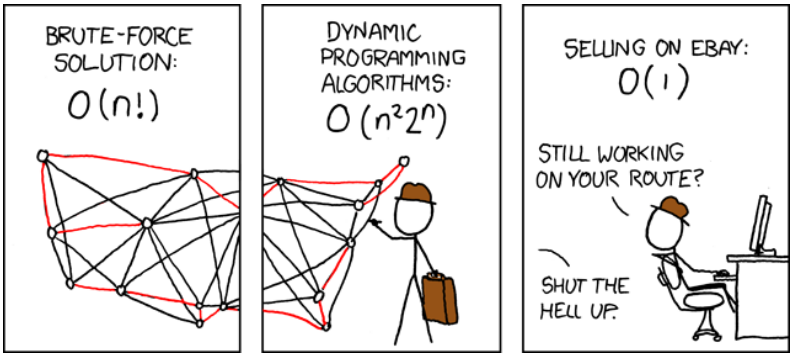
18. Suppose you are given an $m \times n$ bitmap, represented by an array $M[1..n, 1..n]$ of 0s and 1s. A *solid block* in M is a subarray of the form $M[i..i', j..j']$ containing only 1-bits. A solid block is square if it has the same number of rows and columns.

(a) Describe an algorithm to find the maximum area of a solid *square* block in M in $O(n^2)$ time.

(b) Describe an algorithm to find the maximum area of a solid block in M in $O(n^3)$ time.

*(c) Describe an algorithm to find the maximum area of a solid block in M in $O(n^2)$ time.

*19. Describe and analyze an algorithm to solve the traveling salesman problem in $O(2^n \operatorname{poly}(n))$ time. Given an undirected n-vertex graph G with weighted edges, your algorithm should return the weight of the lightest cycle in G that visits every vertex exactly once, or ∞ if G has no such cycles. *[Hint: The obvious recursive algorithm takes $O(n!)$ time.]*



— Randall Munroe, *xkcd* (http://xkcd.com/399/)

[11] If we also allowed upward movement, the resulting game (Vankin's Fathom?) would be Ebay-hard.

*20. Let $\mathscr{A} = \{A_1, A_2, \ldots, A_n\}$ be a finite set of strings over some fixed alphabet Σ. An *edit center* for $\mathscr{A}$ is a string $C \in \Sigma^*$ such that the maximum edit distance from C to any string in $\mathscr{A}$ is as small as possible. The *edit radius* of $\mathscr{A}$ is the maximum edit distance from an edit center to a string in $\mathscr{A}$. A set of strings may have several edit centers, but its edit radius is unique.

$$\mathit{EditRadius}(\mathscr{A}) = \min_{C \in \Sigma^*} \max_{A \in \mathscr{A}} \mathit{Edit}(A, C) \qquad \mathit{EditCenter}(\mathscr{A}) = \arg\min_{C \in \Sigma^*} \max_{A \in \mathscr{A}} \mathit{Edit}(A, C)$$

(a) Describe and analyze an efficient algorithm to compute the edit radius of three given strings.

(b) Describe and analyze an efficient algorithm to approximate the edit radius of an arbitrary set of strings within a factor of 2. (Computing the *exact* edit radius is NP-hard unless the number of strings is fixed.)

★21. Let $D[1..n]$ be an array of digits, each an integer between 0 and 9. An ***digital subsequence*** of D is a sequence of positive integers composed in the usual way from disjoint substrings of D. For example, $3, 4, 5, 6, 8, 9, 32, 38, 46, 64, 83, 279$ is a digital subsequence of the first several digits of π:

$$\underline{3}, 1, \underline{4}, 1, \underline{5}, 9, 2, \underline{6}, 5, 3, 5, \underline{8}, \underline{9}, 7, 9, \underline{3,2}, \underline{3,8}, \underline{4,6}, 2, \underline{6,4}, 3, 3, \underline{8,3}, \underline{2,7,9}$$

The *length* of a digital subsequence is the number of integers it contains, *not* the number of digits; the preceding example has length 12. As usual, a digital subsequence is ***increasing*** if each number is larger than its predecessor.

Describe and analyze an efficient algorithm to compute the longest increasing digital subsequence of D. *[Hint: Be careful about your computational assumptions. How long does it take to compare two k-digit numbers?]*

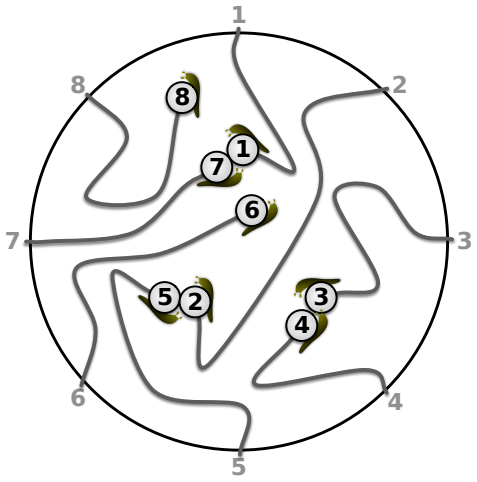
For full credit, your algorithm should run in $O(n^4)$ time; faster algorithms are worth extra credit. The fastest algorithm I know for this problem runs in $O(n^2 \log n)$ time; achieving this bound requires several tricks, both in the algorithm and in its analysis.

Splitting Sequences/Arrays

22. Every year, as part of its annual meeting, the Antarctican Snail Lovers of Upper Glacierville hold a Round Table Mating Race. Several high-quality breeding snails are placed at the edge of a round table. The snails are numbered in order around the table from 1 to n. During the race, each snail wanders around the table, leaving a trail of slime behind it. The snails have been specially trained never to fall off the edge of the table or to cross a slime trail, even their own. If two snails meet, they are declared a breeding pair, removed from the table, and whisked away to a romantic hole in the ground to make little baby snails. Note that some snails may never find a mate, even if the race goes on forever.

For every pair of snails, the Antarctican SLUG race organizers have posted a monetary reward, to be paid to the owners if that pair of snails meets during the Mating Race. Specifically, there is a two-dimensional array $M[1..n, 1..n]$ posted on the wall behind the Round Table, where $M[i, j] = M[j, i]$ is the reward to be paid if snails i and j meet.

Describe and analyze an algorithm to compute the maximum total reward that the organizers could be forced to pay, given the array M as input.



The end of a typical Antarctican SLUG race. Snails 6 and 8 never find mates.
The organizers must pay $M[3,4]+M[2,5]+M[1,7]$.

23. Suppose you are given a sequence of integers separated by $+$ and $\times$ signs; for example:

$$1+3\times 2\times 0+1\times 6+7$$

You can change the value of this expression by adding parentheses in different places. For example:

$$(1+(3\times 2))\times 0+(1\times 6)+7=13$$
$$((1+(3\times 2\times 0)+1)\times 6)+7=19$$
$$(1+3)\times 2\times(0+1)\times(6+7)=208$$

(a) Describe and analyze an algorithm to compute the maximum possible value the given expression can take by adding parentheses, assuming all integers in the input are *positive*. *[Hint: This is easy.]*

(b) Describe and analyze an algorithm to compute the maximum possible value the given expression can take by adding parentheses, assuming all integers in the input are *non-negative*.

(c) Describe and analyze an algorithm to compute the maximum possible value the given expression can take by adding parentheses, with no further restrictions on the input.

Assume any arithmetic operation takes $O(1)$ time.

24. Suppose you are given a sequence of integers separated by $+$ and $-$ signs; for example:

$$1+3-2-5+1-6+7$$

You can change the value of this expression by adding parentheses in different places. For example:

$$1+3-2-5+1-6+7=-1$$
$$(1+3-(2-5))+(1-6)+7=9$$
$$(1+(3-2))-(5+1)-(6+7)=-17$$

Describe and analyze an algorithm to compute, given a list of integers separated by $+$ and $-$ signs, the maximum possible value the expression can take by adding parentheses.

You may only use parentheses to group additions and subtractions; in particular, you are not allowed to create implicit multiplication as in $1+3(-2)(-5)+1-6+7=33$.

25. A ***basic arithmetic expression*** is composed of characters from the set $\{1, +, \times\}$ and parentheses. Almost every integer can be represented by more than one basic arithmetic expression. For example, all of the following basic arithmetic expression represent the integer 14:

$$1+1+1+1+1+1+1+1+1+1+1+1+1+1+1+1$$
$$((1+1)\times(1+1+1+1+1))+((1+1)\times(1+1))$$
$$(1+1)\times(1+1+1+1+1+1+1)$$
$$(1+1)\times(((1+1+1)\times(1+1))+1)$$

Describe and analyze an algorithm to compute, given an integer n as input, the minimum number of 1's in a basic arithmetic expression whose value is n. The number of parentheses doesn't matter, just the number of 1's. For example, when $n = 14$, your algorithm should return 8, for the final expression above. For full credit, the running time of your algorithm should be bounded by a small polynomial function of n.

26. After graduating from UIUC, you have decided to join the Wall Street Bank ***Long Live Boole***. The managing director of the bank, Eloob Egroeg, is a genius mathematician who worships George Boole (the inventor of Boolean Logic) every morning before leaving for the office. The first day of every hired employee is a 'solve-or-die' day where s/he has to solve one of the problems posed by Eloob within 24 hours. Those who fail to solve the problem are fired immediately!

Entering into the bank for the first time, you notice that the offices of the employees are organized in a straight row, with a large T or F printed on the door of each office. Furthermore, between each adjacent pair of offices, there is a board marked by one of the symbols $\wedge$, $\vee$, or $\oplus$. When you ask about these arcane symbols, Eloob confirms that T and F represent the boolean values TRUE and FALSE, and the symbols on the boards represent the standard boolean operators AND, OR, and XOR. He also explains that these letters and symbols describe whether certain combinations of employees can work together successfully. At the start of any new project, Eloob hierarchically clusters his employees by adding parentheses to the sequence of symbols, to obtain an unambiguous boolean expression. The project is successful if this parenthesized boolean expression evaluates to T.

For example, if the bank has three employees, and the sequence of symbols on and between their doors is $T \wedge F \oplus T$, there is exactly one successful parenthesization scheme: $(T \wedge (F \oplus T))$. However, if the list of door symbols is $F \wedge T \oplus F$, there is no way to add parentheses to make the project successful.

Eloob finally poses your solve-or-die question: Describe and algorithm to decide whether a given sequence of symbols can be parenthesized so that the resulting boolean expression evaluates to T. The input to your algorithm is an array $S[0..2n]$, where $S[i] \in \{T, F\}$ when i is even, and $S[i] \in \{\vee, \wedge, \oplus\}$ when i is odd.

27. Suppose we want to display a paragraph of text on a computer screen. The text consists of n words, where the ith word is p_i pixels wide. We want to break the paragraph into several lines, each exactly P pixels long. Depending on which words we put on each line, we must insert different amounts of white space between the words. The paragraph should be fully justified, meaning that the first word on each line starts at its leftmost pixel, and *except for the last line*, the last character on each line ends at its rightmost pixel. There must be at least one pixel of white-space between any two words on the same line. For example, the width of *the paragraph you are reading right*

now is exactly $6\frac{4}{33}$ inches or (assuming a display resolution of 600 pixels per inch) exactly $3672\frac{8}{11}$ pixels. (Sometimes TEX is weird. But thanks to anti-aliasing, fractional pixel widths are fine.)

Define the *slop* of a paragraph layout as the sum over all lines, *except the last*, of the cube of the number of extra white-space pixels in each line, not counting the one pixel required between every adjacent pair of words. Specifically, if a line contains words i through j, then the slop of that line is $\left(P - j + i - \sum_{k=i}^{j} p_k\right)^3$. Describe a dynamic programming algorithm to print the paragraph with minimum slop.

28. You have mined a large slab of marble from your quarry. For simplicity, suppose the marble slab is a rectangle measuring n inches in height and m inches in width. You want to cut the slab into smaller rectangles of various sizes—some for kitchen countertops, some for large sculpture projects, others for memorial headstones. You have a marble saw that can make either horizontal or vertical cuts across any rectangular slab. At any time, you can query the spot price $P[x, y]$ of an x-inch by y-inch marble rectangle, for any positive integers x and y. These prices will vary with demand, so do not make any assumptions about them; in particular, larger rectangles may have much smaller spot prices. Given the spot prices, describe an algorithm to compute how to subdivide an $n \times m$ marble slab to maximize your profit.

29. A string w of parentheses (and) and brackets [and] is ***balanced*** if it satisfies one of the following conditions:

    - w is the empty string.
    - $w = (x)$ for some balanced string x
    - $w = [x]$ for some balanced string x
    - $w = xy$ for some balanced strings x and y

    For example, the string

    $$w = ([()][]())[()()]()$$

    is balanced, because $w = xy$, where

    $$x = (\,[()]\,[]\,()\,) \qquad \text{and} \qquad y = [\,()\,()\,]\,().$$

    (a) Describe and analyze an algorithm to determine whether a given string of parentheses and brackets is balanced.

    (b) Describe and analyze an algorithm to compute the length of a longest balanced subsequence of a given string of parentheses and brackets.

    (c) Describe and analyze an algorithm to compute the length of a shortest balanced supersequence of a given string of parentheses and brackets.

    (d) Describe and analyze an algorithm to compute the minimum edit distance from a given string of parentheses and brackets to a balanced string of parentheses and brackets.

    For each problem, your input is an array $w[1..n]$, where $w[i] \in \{(,), [,]\}$ for every index i.

30. Congratulations! Your research team has just been awarded a $50M multi-year project, jointly funded by DARPA, Google, and McDonald's, to produce DWIM: The first compiler to read programmers' minds! Your proposal and your numerous press releases all promise that DWIM will

automatically correct errors in any given piece of code, while modifying that code as little as possible. Unfortunately, now it's time to start actually making the damn thing work.

As a warmup exercise, you decide to tackle the following necessary subproblem. Recall that the *edit distance* between two strings is the minimum number of single-character insertions, deletions, and replacements required to transform one string into the other. An *arithmetic expression* is a string w such that

- w is a string of one or more decimal digits,
- $w = (\,x\,)$ for some arithmetic expression x, or
- $w = x \diamond y$ for some arithmetic expressions x and y and some binary operator $\diamond$.

Suppose you are given a string of tokens from the alphabet $\{\#, \diamond, (,)\}$, where $\#$ represents a decimal digit and $\diamond$ represents a binary operator. Describe an algorithm to compute the minimum edit distance from the given string to an arithmetic expression.

31. Let P be a set of points in the plane in *convex position*. Intuitively, if a rubber band were wrapped around the points, then every point would touch the rubber band. More formally, for any point p in P, there is a line that separates p from the other points in P. Moreover, suppose the points are indexed $P[1], P[2], \ldots, P[n]$ in counterclockwise order around the 'rubber band', starting with the leftmost point $P[1]$.

    This problem asks you to solve a special case of the traveling salesman problem, where the salesman must visit every point in P, and the cost of moving from one point $p \in P$ to another point $q \in P$ is the Euclidean distance $|pq|$.

    (a) Describe a simple algorithm to compute the shortest *cyclic* tour of P.

    (b) A *simple* tour is one that never crosses itself. Prove that the shortest tour of P must be simple.

    (c) Describe and analyze an efficient algorithm to compute the shortest tour of P that starts at the leftmost point $P[1]$ and ends at the rightmost point $P[r]$.

    (d) Describe and analyze an efficient algorithm to compute the shortest tour of P, with no restrictions on the endpoints.

32. (a) Describe and analyze an efficient algorithm to determine, given a string w and a regular expression R, whether $w \in L(R)$.

    (b) *Generalized* regular expressions allow the binary operator $\cap$ (intersection) and the unary operator $\neg$ (complement), in addition to the usual concatenation, $+$ (or), and * (Kleene closure) operators. NFA constructions and Kleene's theorem imply that any generalized regular expression E represents a regular language $L(E)$.

    Describe and analyze an efficient algorithm to determine, given a string w and a generalized regular expression E, whether $w \in L(E)$.

    In both problems, assume that you are actually given a parse tree for the (generalized) regular expression, not just a string.

33. Ribonucleic acid (RNA) molecules are long chains of millions of nucleotides or *bases* of four different types: adenine (A), cytosine (C), guanine (G), and uracil (U). The *sequence* of an RNA molecule is a string $b[1..n]$, where each character $b[i] \in \{A, C, G, U\}$ corresponds to a base. In

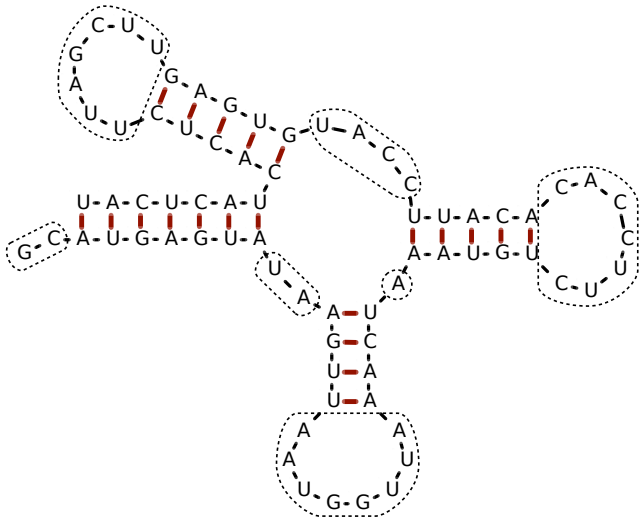
addition to the chemical bonds between adjacent bases in the sequence, hydrogen bonds can form between certain pairs of bases. The set of bonded base pairs is called the *secondary structure* of the RNA molecule.

We say that two base pairs (i, j) and (i', j') with $i < j$ and $i' < j'$ ***overlap*** if $i < i' < j < j'$ or $i' < i < j' < j$. In practice, most base pairs are non-overlapping. Overlapping base pairs create so-called *pseudoknots* in the secondary structure, which are essential for some RNA functions, but are more difficult to predict.

Suppose we want to predict the best possible secondary structure for a given RNA sequence. We will adopt a drastically simplified model of secondary structure:

- Each base can be paired with at most one other base.
- Only A-U pairs and C-G pairs can bond.
- Pairs of the form $(i, i+1)$ and $(i, i+2)$ cannot bond.
- Overlapping base pairs cannot bond.

The last restriction allows us to visualize RNA secondary structure as a sort of fat tree.



Example RNA secondary structure with 21 base pairs, indicated by heavy red lines.
Gaps are indicated by dotted curves. This structure has score $2^2 + 2^2 + 8^2 + 1^2 + 7^2 + 4^2 + 7^2 = 187$

(a) Describe and analyze an algorithm that computes the maximum possible *number* of bonded base pairs in a secondary structure for a given RNA sequence.

(b) A *gap* in a secondary structure is a maximal substring of unpaired bases. Large gaps lead to chemical instabilities, so secondary structures with smaller gaps are more likely. To account for this preference, let's define the *score* of a secondary structure to be the sum of the *squares* of the gap lengths. (This score function is utterly fictional; real RNA structure prediction requires *much* more complicated scoring functions.) Describe and analyze an algorithm that computes the minimum possible score of a secondary structure for a given RNA sequence.

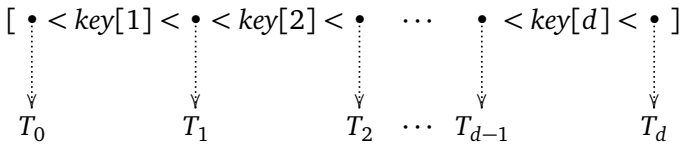
34. A standard method to improve the cache performance of search trees is to pack more search keys and subtrees into each node. A ***B-tree*** is a rooted tree in which each internal node stores up to B keys and pointers to up to $B + 1$ children, each the root of a smaller B-tree. Specifically, each node v stores three fields:

- a positive integer $v.d \le B$,
- a *sorted* array $v.key[1..v.d]$, and
- an array $v.child[0..v.d]$ of child pointers.

In particular, the number of child pointers is always exactly one more than the number of keys.

Each pointer $v.child[i]$ is either NULL or a pointer to the root of a B-tree whose keys are all larger than $v.key[i]$ and smaller than $v.key[i+1]$. In particular, all keys in the leftmost subtree $v.child[0]$ are smaller than $v.key[1]$, and all keys in the rightmost subtree $v.child[v.d]$ are larger than $v.key[v.d]$.

Intuitively, you should have the following picture in mind:



Here T_i is the subtree pointed to by $child[i]$.

The ***cost*** of searching for a key x in a B-tree is the number of nodes in the path from the root to the node containing x as one of its keys. A 1-tree is just a standard binary search tree.

Fix an arbitrary positive integer $B > 0$. (I suggest $B = 8$.) Suppose your are given a sorted array $A[1,\ldots,n]$ of search keys and a corresponding array $F[1,\ldots,n]$ of frequency counts, where $F[i]$ is the number of times that we will search for $A[i]$. Your task is to describe and analyze an efficient algorithm to find a B-tree that minimizes the total cost of searching for the given keys with the given frequencies.

(a) Describe a polynomial-time algorithm for the special case $B = 2$.

(b) Describe an algorithm for arbitrary B that runs in $O(n^{B+c})$ time for some fixed integer c.

(c) Describe an algorithm for arbitrary B that runs in $O(n^c)$ time for some fixed integer c that does *not* depend on B.

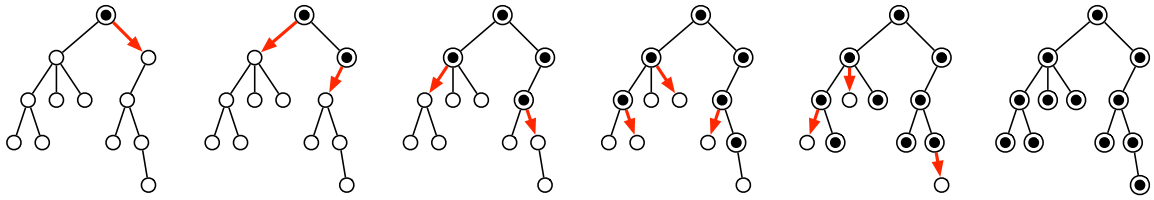
A few comments about *B*-trees. Normally, B-trees are required to satisfy two additional constraints, which guarantee a worst-case search cost of $O(\log_B n)$: Every leaf must have exactly the same depth, and every node except possibly the root must contain at least $B/2$ keys. However, in this problem, we are not interested in optimizing the *worst-case* search cost, but rather the *total* cost of a sequence of searches, so we will not impose these additional constraints.

In most large database systems, the parameter B is chosen so that each node exactly fits in a cache line. Since the entire cache line is loaded into cache anyway, and the cost of loading a cache line exceeds the cost of searching within the cache, the running time is dominated by the number of cache faults. This effect is even more noticeable if the data is too big to fit in RAM; then the cost is dominated by the number of *page faults*, and B should be roughly the size of a page. In extreme cases, the data is too large even to fit on disk (or flash-memory "disk") and is instead distributed on a bank of magnetic tape cartridges, in which case the cost is dominated by the number of *tape faults*. (I invite anyone who thinks tape is dead to visit a supercomputing center like Blue Waters.) In principle, your data might be so large that the cost of searching is actually dominated by the number of *FedEx faults*. (See https://what-if.xkcd.com/31/.)

Don't worry about the cache/disk/tape/FedEx performance in your solutions; just analyze the CPU time as usual. Designing algorithms with few cache misses or page faults is a interesting pastime; simultaneously optimizing CPU time *and* cache misses *and* page faults *and* FedEx faults is a topic of active research. Sadly, this kind of design and analysis requires tools we won't see in this class.

Trees and Subtrees

35. Suppose we need to distribute a message to all the nodes in a rooted tree. Initially, only the root node knows the message. In a single round, any node that knows the message can forward it to at most one of its children. Design an algorithm to compute the minimum number of rounds required for the message to be delivered to all nodes in a given tree.

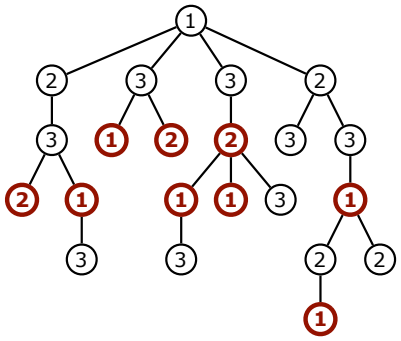


A message being distributed through a tree in five rounds.

36. Oh, no! You have been appointed as the organizer of Giggle, Inc.'s annual mandatory holiday party! The employees at Giggle are organized into a strict hierarchy, that is, a tree with the company president at the root. The all-knowing oracles in Human Resources have assigned a real number to each employee measuring how "fun" the employee is. In order to keep things social, there is one restriction on the guest list: an employee cannot attend the party if their immediate supervisor is also present. On the other hand, the president of the company *must* attend the party, even though she has a negative fun rating; it's her company, after all. Give an algorithm that makes a guest list for the party that maximizes the sum of the "fun" ratings of the guests.

37. Since so few people came to last year's holiday party, the president of Giggle, Inc. decides to give each employee a present instead this year. Specifically, each employee must receive on the three gifts: (1) an all-expenses-paid six-week vacation anywhere in the world, (2) an all-the-pancakes-you-can-eat breakfast for two at Jumping Jack Flash's Flapjack Stack Shack, or (3) a burning paper bag full of dog poop. Corporate regulations prohibit any employee from receiving exactly the same gift as his/her direct supervisor. Any employee who receives a better gift than his/her direct supervisor will almost certainly be fired in a fit of jealousy.

    As Giggle, Inc.'s official party czar, it's *your* job to decide which gift each employee receives. Describe an algorithm to distribute gifts so that the minimum number of people are fired. Yes, you may send the president a flaming bag of dog poop.



A tree labeling with cost 9.
Bold nodes have smaller labels than their parents.
This is *not* the optimal labeling for this tree.

More formally, you are given a rooted tree T, representing the company hierarchy, and you want to label each node in T with an integer 1, 2, or 3, so that every node has a different label from its parent. The *cost* of an labeling is the number of nodes that have smaller labels than their parents. Describe and analyze an algorithm to compute the minimum cost of any labeling of the given tree T.

38. After losing so many employees to last year's Flaming Dog Poop Holiday Debacle, the president of Giggle, Inc. has declared that once again there will be a holiday party this year. Recall that the employees are organized into a strict hierarchy, that is, a tree with the company president at the root. The president demands that you invite exactly k employees, including the president herself. Moreover, everyone who is invited is required to attend. Yeah, that'll be fun.

    The all-knowing oracles in Human Resources have assigned a real number to each employee indicating the *awkwardness* of inviting both that employee and their immediate supervisor; a negative value indicates that the employee and their supervisor actually like each other. Your goal is to choose a subset k employees to invite, so that the total awkwardness of the resulting party is as small as possible. For example, if the guest list does not include both an employee and their immediate supervisor, the total awkwardness is zero.

    (a) Describe an algorithm that computes the total awkwardness of the least awkward subset of k employees, assuming the company hierarchy is described by a *binary* tree. That is, assume that each employee directly supervises at most two others.

    *(b) Describe an algorithm that computes the total awkwardness of the least awkward subset of k employees, with no restrictions on the company hierarchy.

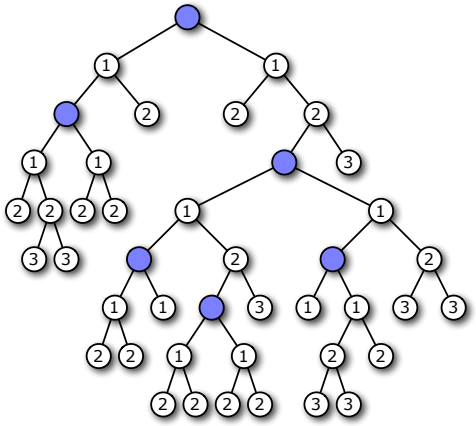
39. Let T be a rooted binary tree with n vertices, and let $k \le n$ be a positive integer. We would like to mark k vertices in T so that every vertex has a nearby marked ancestor. More formally, we define the *clustering cost* of any subset K of vertices as

$$cost(K) = \max_v cost(v, K),$$

where the maximum is taken over all vertices v in the tree, and

$$cost(v, K) = \begin{cases} 0 & \text{if } v \in K \\ \infty & \text{if } v \text{ is the root of } T \text{ and } v \notin K \\ 1 + cost(parent(v)) & \text{otherwise} \end{cases}$$

Describe and analyze a dynamic-programming algorithm to compute the minimum clustering cost of any subset of k vertices in T. For full credit, your algorithm should run in $O(n^2k^2)$ time.



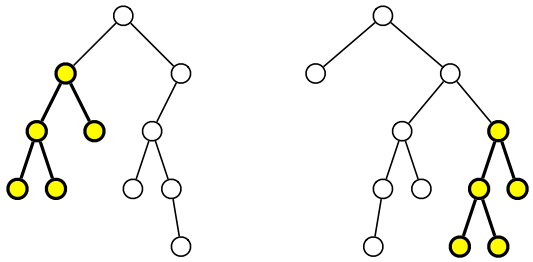
A subset of 5 vertices with clustering cost 3

The next several questions ask for algorithms to find various optimal subtrees in trees. To make the problem statements precise, we must distinguish between several different types of trees and subtrees:

- By default, a *tree* is just a connected, acyclic, undirected graph.
- A *rooted* tree has a distinguished vertex, called the *root*. A tree without a distinguished root vertex is called an *unrooted* tree or a *free* tree.
- In an *ordered* tree, the neighbors of every vertex have a well-defined cyclic order. A tree without these orders is called an *unordered* tree.
- A *binary* tree is a rooted tree in which every node has a (possibly empty) *left* subtree and a (possibly empty) *right* subtree. Two binary trees are isomorphic if they are both empty, or if their left subtrees are isomorphic and their right subtrees are isomorphic.
- A (rooted) subtree of a *rooted* tree consists of a node and all its descendants. A (free) subtree of an *unrooted* tree is any connected subgraph. Subtrees of ordered rooted trees are themselves ordered trees.

40. This question asks you to find efficient algorithms to compute the ***largest common rooted subtree*** of two given rooted trees. A rooted subtree consists of an arbitrary node and *all* its descendants. However, the precise definition of "common" depends on which rooted trees we consider to be isomorphic.

    (a) Describe an algorithm to find the largest common *binary* subtree of two given *binary* trees.



Two binary trees, with their largest common (rooted) subtree emphasized

    (b) An *ordered* tree is either empty or a node with a *sequence* of children, which are themselves the roots of (possibly empty) ordered trees. Two ordered trees are isomorphic if they are both empty, or if their ith subtrees are isomorphic for all i. Describe an algorithm to find the largest common ordered subtree of two ordered trees T_1 and T_2.

(c) An *unordered* tree is either empty or a node with a *set* of children, which are themselves the roots of (possibly empty) ordered trees. Two unordered trees are isomorphic if they are both empty, or the subtrees or each tree *can be ordered so that* their ith subtrees are isomorphic for all i. Describe an algorithm to find the largest common unordered subtree of two unordered trees T_1 and T_2.

41. This question asks you to find efficient algorithms to compute optimal subtrees in *unrooted* trees. A *subtree* of an unrooted tree is any connected subgraph.

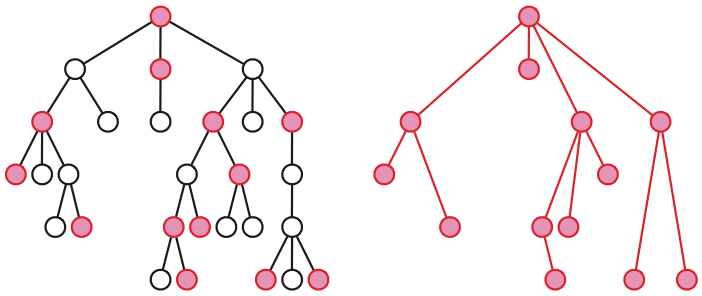
(a) Suppose you are given an unrooted tree T with weights on its *edges*, which may be positive, negative, or zero. Describe an algorithm to find a *path* in T with maximum total weight.

(b) Suppose you are given an unrooted tree T with weights on its vertices, which may be positive, negative, or zero. Describe an algorithm to find a *subtree* of T with maximum total weight.

(c) Let T_1 and T_2 be *ordered* trees, meaning that the neighbors of every node have a well-defined cyclic order. Describe an algorithm to find the largest common *ordered* subtree of T_1 and T_2.

*(d) Let T_1 and T_2 be *unordered* trees. Describe an algorithm to find the largest common *unordered* subtree of T_1 and T_2.

42. ***Sub-branchings*** of a rooted tree are a generalization of subsequences of a sequence. A sub-branching of a tree is a subset S of the nodes such that *exactly* one node in S that does not have a proper ancestor in S. Any sub-branching S implicitly defines a tree $T(S)$, in which the parent of a node $x \in S$ is the closest proper ancestor (in T) of x that is also in S.



A sub-branching S and its associated tree $T(S)$.

(a) Let T be a rooted tree with labeled nodes. We say that T is *boring* if, for each node x, all children of x have the same label; children of different nodes may have different labels. A sub-branching S of a labeled rooted tree T is boring if its associated tree $T(S)$ is boring; nodes in $T(S)$ inherit their labels from T. Describe an algorithm to find the largest boring sub-branching S of a given labeled rooted tree.

(b) Suppose we are given a rooted tree T whose nodes are labeled with numbers. Describe an algorithm to find the largest *heap-ordered sub-branching* of T. That is, your algorithm should return the largest sub-branching S such that every node in $T(S)$ has a smaller label than its children in $T(S)$.

(c) Suppose we are given a *binary* tree T whose nodes are labeled with numbers. Describe an algorithm to find the largest *binary-search-ordered sub-branching* of T. That is, your algorithm should return a sub-branching S such that every node in $T(S)$ has at most two children, and an inorder traversal of $T(S)$ is an increasing subsequence of an inorder traversal of T.

(d) Recall that a rooted tree is *ordered* if the children of each node have a well-defined left-to-right order. Describe an algorithm to find the largest binary-search-ordered sub-branching S of an *arbitrary* ordered tree T whose nodes are labeled with numbers. Again, the order of nodes in $T(S)$ should be consistent with their order in T.

*(e) Describe an algorithm to find the largest common *ordered* sub-branching of two *ordered* labeled rooted trees.

★(f) Describe an algorithm to find the largest common *unordered* sub-branching of two *unordered* labeled rooted trees. *[Hint: This problem will be much easier after you've seen flows.]*

The point is, ladies and gentleman, greed is good. Greed works, greed is right. Greed clarifies, cuts through, and captures the essence of the evolutionary spirit. Greed in all its forms, greed for life, money, love, knowledge has marked the upward surge in mankind. And greed—mark my words—will save not only Teldar Paper but the other malfunctioning corporation called the USA.

— Michael Douglas as Gordon Gekko, *Wall Street* (1987)

There is always an easy solution to every human problem—
neat, plausible, and wrong.

— H. L. Mencken, "The Divine Afflatus",
New York Evening Mail (November 16, 1917)

10 Greedy Algorithms

10.1 Storing Files on Tape

Suppose we have a set of n files that we want to store on a tape. In the future, users will want to read those files from the tape. Reading a file from tape isn't like reading a file from disk; first we have to fast-forward past all the other files, and that takes a significant amount of time. Let $L[1..n]$ be an array listing the lengths of each file; specifically, file i has length $L[i]$. If the files are stored in order from 1 to n, then the cost of accessing the kth file is

$$cost(k) = \sum_{i=1}^{k} L[i].$$

The cost reflects the fact that before we read file k we must first scan past all the earlier files on the tape. If we assume for the moment that each file is equally likely to be accessed, then the *expected* cost of searching for a random file is

$$\mathrm{E}[cost] = \sum_{k=1}^{n} \frac{cost(k)}{n} = \sum_{k=1}^{n} \sum_{i=1}^{k} \frac{L[i]}{n}.$$

If we change the order of the files on the tape, we change the cost of accessing the files; some files become more expensive to read, but others become cheaper. Different file orders are likely to result in different expected costs. Specifically, let $\pi(i)$ denote the index of the file stored at position i on the tape. Then the expected cost of the permutation π is

$$\mathrm{E}[cost(\pi)] = \sum_{k=1}^{n} \sum_{i=1}^{k} \frac{L[\pi(i)]}{n}.$$

Which order should we use if we want the expected cost to be as small as possible? The answer is intuitively clear; we should store the files in order from shortest to longest. So let's prove this.

Lemma 1. $\mathrm{E}[cost(\pi)]$ *is minimized when* $L[\pi(i)] \le L[\pi(i+1)]$ *for all* i.

Proof: Suppose $L[\pi(i)] > L[\pi(i+1)]$ for some i. To simplify notation, let $a = \pi(i)$ and $b = \pi(i+1)$. If we swap files a and b, then the cost of accessing a increases by $L[b]$, and the cost of accessing b decreases by $L[a]$. Overall, the swap changes the expected cost by $(L[b] - L[a])/n$. But this change is an improvement, because $L[b] < L[a]$. Thus, if the files are out of order, we can improve the expected cost by swapping some mis-ordered adjacent pair. □

This example gives us our first *greedy algorithm*. To minimize the *total* expected cost of accessing the files, we put the file that is cheapest to access first, and then recursively write everything else; no backtracking, no dynamic programming, just make the best local choice and blindly plow ahead. If we use an efficient sorting algorithm, the running time is clearly $O(n \log n)$, plus the time required to actually write the files. To prove the greedy algorithm is actually correct, we simply prove that the output of any other algorithm can be improved by some sort of swap.

Let's generalize this idea further. Suppose we are also given an array $F[1..n]$ of *access frequencies* for each file; file i will be accessed exactly $F[i]$ times over the lifetime of the tape. Now the *total* cost of accessing all the files on the tape is

$$\Sigma cost(\pi) = \sum_{k=1}^{n} \left(F[\pi(k)] \cdot \sum_{i=1}^{k} L[\pi(i)] \right) = \sum_{k=1}^{n} \sum_{i=1}^{k} \big(F[\pi(k)] \cdot L[\pi(i)] \big).$$

Now what order should store the files if we want to minimize the total cost?

We've already proved that if all the frequencies are equal, then we should sort the files by increasing size. If the frequencies are all different but the file lengths $L[i]$ are all equal, then intuitively, we should sort the files by *decreasing* access frequency, with the most-accessed file first. In fact, this is not hard to prove by modifying the proof of Lemma 1. But what if the sizes and the frequencies are both different? In this case, we should sort the files by the ratio L/F.

Lemma 2. *$\Sigma cost(\pi)$ is minimized when $\dfrac{L[\pi(i)]}{F[\pi(i)]} \le \dfrac{L[\pi(i+1)]}{F[\pi(i+1)]}$ for all i.*

Proof: Suppose $L[\pi(i)]/F[\pi(i)] > L[\pi(i+1)]/F[\pi(i+i)]$ for some i. To simplify notation, let $a = \pi(i)$ and $b = \pi(i+1)$. If we swap files a and b, then the cost of accessing a increases by $L[b]$, and the cost of accessing b decreases by $L[a]$. Overall, the swap changes the total cost by $L[b]F[a] - L[a]F[b]$. But this change is an improvement, since

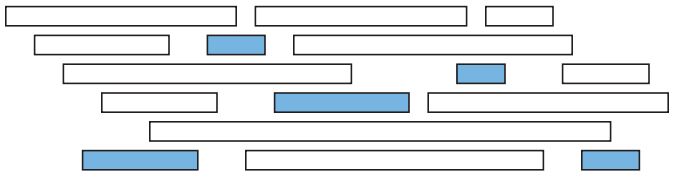
$$\frac{L[a]}{F[a]} > \frac{L[b]}{F[b]} \implies L[b]F[a] - L[a]F[b] < 0.$$

Thus, if two adjacent files are out of order, we can improve the total cost by swapping them. □

10.2 Scheduling Classes

The next example is slightly less trivial. Suppose you decide to drop out of computer science at the last minute and change your major to Applied Chaos. The Applied Chaos department offers all of its classes on the same day every week, called 'Soberday' by the students (but interestingly, *not* by the faculty). Every class has a different start time and a different ending time: AC 101 ('Toilet Paper Landscape Architecture') starts at 10:27pm and ends at 11:51pm; AC 666 ('Immanentizing the Eschaton') starts at 4:18pm and ends at 7:06pm, and so on. In the interest of graduating as quickly as possible, you want to register for as many classes as you can. (Applied Chaos classes don't require any actual *work*.) The university's registration computer won't let you register for overlapping classes, and no one in the department knows how to override this 'feature'. Which classes should you take?

More formally, suppose you are given two arrays $S[1..n]$ and $F[1..n]$ listing the start and finish times of each class; to be concrete, we can assume that $0 \le S[i] < F[i] \le M$ for each i, for some value M (for example, the number of picoseconds in Soberday). Your task is to choose the largest possible subset $X \in \{1, 2, \ldots, n\}$ so that for any pair $i, j \in X$, either $S[i] > F[j]$ or $S[j] > F[i]$. We can illustrate the problem by drawing each class as a rectangle whose left and right x-coordinates show the start and finish times. The goal is to find a largest subset of rectangles that do not overlap vertically.



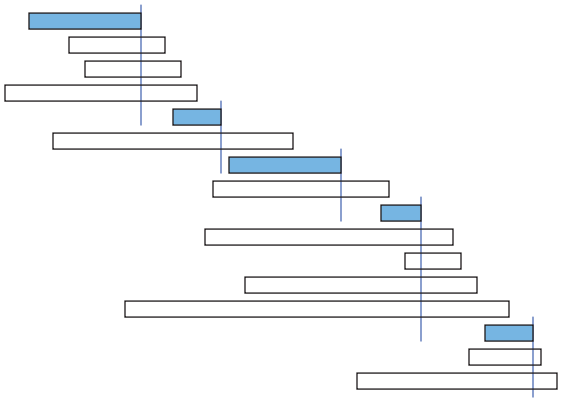
A maximal conflict-free schedule for a set of classes.

This problem has a fairly simple recursive solution, based on the observation that either you take class 1 or you don't. Let B_4 denote the set of classes that end *before* class 1 starts, and let L_8 denote the set of classes that start *later* than class 1 ends:

$$B_4 = \{i \mid 2 \le i \le n \text{ and } F[i] < S[1]\} \qquad L_8 = \{i \mid 2 \le i \le n \text{ and } S[i] > F[1]\}$$

If class 1 is in the optimal schedule, then so are the optimal schedules for B_4 and L_8, which we can find recursively. If not, we can find the optimal schedule for $\{2, 3, \ldots, n\}$ recursively. So we should try both choices and take whichever one gives the better schedule. Evaluating this recursive algorithm from the bottom up gives us a dynamic programming algorithm that runs in $O(n^2)$ time. I won't bother to go through the details, because we can do better.[1]

Intuitively, we'd like the first class to finish as early as possible, because that leaves us with the most remaining classes. If this greedy strategy works, it suggests the following very simple algorithm. Scan through the classes in order of finish time; whenever you encounter a class that doesn't conflict with your latest class so far, take it!



The same classes sorted by finish times and the greedy schedule.

We can write the greedy algorithm somewhat more formally as follows. (Hopefully the first line is understandable.) The algorithm clearly runs in $O(n \log n)$ time.

```
GREEDYSCHEDULE(S[1..n], F[1..n]):
  sort F and permute S to match
  count ← 1
  X[count] ← 1
  for i ← 2 to n
      if S[i] > F[X[count]]
          count ← count + 1
          X[count] ← i
  return X[1..count]
```

[1] But you should still work out the details yourself. The dynamic programming algorithm can be used to find the "best" schedule for several different definitions of "best", but the greedy algorithm I'm about to describe only works when "best" means "biggest". Also, you need the practice.

To prove that this algorithm actually gives us a maximal conflict-free schedule, we use an exchange argument, similar to the one we used for tape sorting. We are not claiming that the greedy schedule is the *only* maximal schedule; there could be others. (See the figures on the previous page.) All we can claim is that at least one of the maximal schedules is the one that the greedy algorithm produces.

Lemma 3. *At least one maximal conflict-free schedule includes the class that finishes first.*

Proof: Let f be the class that finishes first. Suppose we have a maximal conflict-free schedule X that does not include f. Let g be the first class in X to finish. Since f finishes before g does, f cannot conflict with any class in the set $S \setminus \{g\}$. Thus, the schedule $X' = X \cup \{f\} \setminus \{g\}$ is also conflict-free. Since X' has the same size as X, it is also maximal. □

To finish the proof, we call on our old friend, induction.

Theorem 4. *The greedy schedule is an optimal schedule.*

Proof: Let f be the class that finishes first, and let L be the subset of classes the start after f finishes. The previous lemma implies that some optimal schedule contains f, so the best schedule that contains f is an optimal schedule. The best schedule that includes f must contain an optimal schedule for the classes that do not conflict with f, that is, an optimal schedule for L. The greedy algorithm chooses f and then, by the inductive hypothesis, computes an optimal schedule of classes from L. □

The proof might be easier to understand if we unroll the induction slightly.

Proof: Let $\langle g_1, g_2, \ldots, g_k \rangle$ be the sequence of classes chosen by the greedy algorithm. Suppose we have a maximal conflict-free schedule of the form

$$\langle g_1, g_2, \ldots, g_{j-1}, c_j, c_{j+1}, \ldots, c_m \rangle,$$

where class c_j is different from the class g_j that would be chosen by the greedy algorithm. (We may have $j = 1$, in which case this schedule starts with a non-greedy choice c_1.) By construction, the jth greedy choice g_j does not conflict with any earlier class $g_1, g_2, \ldots, g_{j-1}$, and since our schedule is conflict-free, neither does c_j. Moreover, g_j has the *earliest* finish time among all classes that don't conflict with the earlier classes; in particular, g_j finishes before c_j. This implies that g_j does not conflict with any of the later classes $c_{j+1}, \ldots, c_m$. Thus, the schedule

$$\langle g_1, g_2, \ldots, g_{j-1}, g_j, c_{j+1}, \ldots, c_m \rangle,$$

is conflict-free. (This is just a generalization of Lemma 3, which considers the case $j = 1$.)

By induction, it now follows that there is an optimal schedule $\langle g_1, g_2, \ldots, g_k, c_{k+1}, \ldots, c_m \rangle$ that includes *every* class chosen by the greedy algorithm. But this is impossible unless $k = m$; if there were a class c_{k+1} that does not conflict with g_k, the greedy algorithm would choose more than k classes. □

10.3 General Structure

The basic structure of this correctness proof is exactly the same as for the tape-sorting problem: an inductive exchange argument.

- Assume that there is an optimal solution that is different from the greedy solution.
- Find the "first" difference between the two solutions.

- Argue that we can exchange the optimal choice for the greedy choice without degrading the solution.

This argument implies by induction that some optimal solution that *contains* the entire greedy solution, and therefore *equals* the greedy solution. Sometimes, as in the scheduling problem, an additional step is required to show no optimal solution *strictly* improves the greedy solution.

10.4 Huffman Codes

A *binary code* assigns a string of 0s and 1s to each character in the alphabet. A binary code is *prefix-free* if no code is a prefix of any other. 7-bit ASCII and Unicode's UTF-8 are both prefix-free binary codes. Morse code is a binary code, but it is not prefix-free; for example, the code for `S` (···) includes the code for `E` (·) as a prefix. Any prefix-free binary code can be visualized as a binary tree with the encoded characters stored at the leaves. The code word for any symbol is given by the path from the root to the corresponding leaf; 0 for left, 1 for right. The length of a codeword for a symbol is the depth of the corresponding leaf.

Let me emphasize that binary code trees are *not* binary search trees; we don't care at all about the order of symbols at the leaves.

Suppose we want to encode messages in an n-character alphabet so that the encoded message is as short as possible. Specifically, given an array frequency counts $f[1..n]$, we want to compute a prefix-free binary code that minimizes the total encoded length of the message:[2]

$$\sum_{i=1}^{n} f[i] \cdot depth(i).$$

In 1951, as a PhD student at MIT, David Huffman developed the following greedy algorithm to produce such an optimal code:[3]

> HUFFMAN: Merge the two least frequent letters and recurse.

For example, suppose we want to encode the following helpfully self-descriptive sentence, discovered by Lee Sallows:[4]

> This sentence contains three a's, three c's, two d's, twenty-six e's, five f's, three g's, eight h's, thirteen i's, two l's, sixteen n's, nine o's, six r's, twenty-seven s's, twenty-two t's, two u's, five v's, eight w's, four x's, five y's, and only one z.

To keep things simple, let's forget about the forty-four spaces, nineteen apostrophes, nineteen commas, three hyphens, and only one period, and just encode the letters. Here's the frequency table:

A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1

[2] This looks almost exactly like the cost of a binary search tree, but the optimization problem is very different: code trees are not required to keep the keys in any particular order.

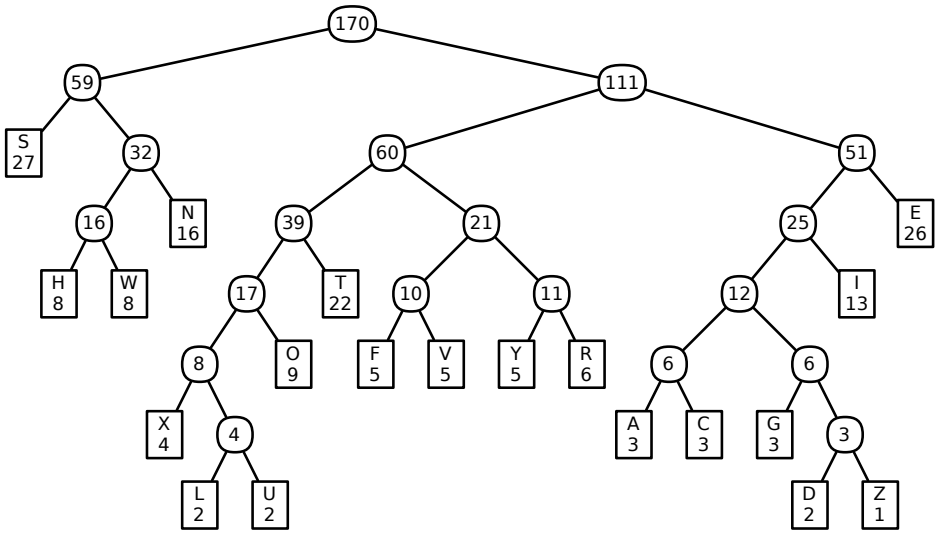
[3] Huffman was a student in an information theory class taught by Robert Fano, who was a close colleague of Claude Shannon, the father of information theory. Fano and Shannon had previously developed a different greedy algorithm for producing prefix codes—split the frequency array into two subarrays as evenly as possible, and then recursively build a code for each subarray—but these Fano-Shannon codes were known not to be optimal. Fano posed the (then open) problem of finding an optimal encoding to his class; Huffman solved the problem as a class project, in lieu of taking a final exam.

[4] A. K. Dewdney. Computer recreations. *Scientific American*, October 1984. Douglas Hofstadter published a few earlier examples of Lee Sallows' self-descriptive sentences in his *Scientific American* column in January 1982.

Huffman's algorithm picks out the two least frequent letters, breaking ties arbitrarily—in this case, say, Z and D—and merges them together into a single new character DZ with frequency 3. This new character becomes an internal node in the code tree we are constructing, with Z and D as its children; it doesn't matter which child is which. The algorithm then recursively constructs a Huffman code for the new frequency table

A	C	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	DZ
3	3	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	3

After 19 merges, all 20 characters have been merged together. The record of merges gives us our code tree. The algorithm makes a number of arbitrary choices; as a result, there are actually several different Huffman codes. One such code is shown below. For example, the code for A is 110000, and the code for S is 00.



A Huffman code for Lee Sallows' self-descriptive sentence; the numbers are frequencies for merged characters

If we use this code, the encoded message starts like this:

1001	0100	1101	00	00	111	011	1001	111	011	110001	111	110001	10001	011	1001	110000	1101	...
T	H	I	S	S	E	N	T	E	N	C	E	C	O	N	T	A	I	

Here is the list of costs for encoding each character in the example message, along with that character's contribution to the total length of the encoded message:

char.	A	C	D	E	F	G	H	I	L	N	O	R	S	T	U	V	W	X	Y	Z
freq.	3	3	2	26	5	3	8	13	2	16	9	6	27	22	2	5	8	4	5	1
depth	6	6	7	3	5	6	4	4	7	3	4	4	2	4	7	5	4	6	5	7
total	18	18	14	78	25	18	32	52	14	48	36	24	54	88	14	25	32	24	25	7

Altogether, the encoded message is 646 bits long. Different Huffman codes would assign different codes, possibly with different lengths, to various characters, but the overall length of the encoded message is the same for any Huffman code: 646 bits.

Given the simple structure of Huffman's algorithm, it's rather surprising that it produces an *optimal* prefix-free binary code. Encoding Lee Sallows' sentence using *any* prefix-free code requires at least 646 bits! Fortunately, the recursive structure makes this claim easy to prove using an exchange argument, similar to our earlier optimality proofs. We start by proving that the algorithm's very first choice is correct.

Lemma 5. *Let x and y be the two least frequent characters (breaking ties between equally frequent characters arbitrarily). There is an optimal code tree in which x and y are siblings.*

Proof: I'll actually prove a stronger statement: There is an optimal code in which x and y are siblings *and* have the largest depth of any leaf.

Let T be an optimal code tree, and suppose this tree has depth d. Since T is a full binary tree, it has at least two leaves at depth d that are siblings. (Verify this by induction!) Suppose those two leaves are *not* x and y, but some other characters a and b.

Let T' be the code tree obtained by swapping x and a. The depth of x increases by some amount Δ, and the depth of a decreases by the same amount. Thus,

$$cost(T') = cost(T) - (f[a] - f[x])\Delta.$$

By assumption, x is one of the two least frequent characters, but a is not, which implies that $f[a] \geq f[x]$. Thus, swapping x and a does not increase the total cost of the code. Since T was an optimal code tree, swapping x and a does not decrease the cost, either. Thus, T' is also an optimal code tree (and incidentally, $f[a]$ actually equals $f[x]$).

Similarly, swapping y and b must give yet another optimal code tree. In this final optimal code tree, x and y are maximum-depth siblings, as required. □

Now optimality is guaranteed by our dear friend the Recursion Fairy! Essentially we're relying on the following recursive definition for a full binary tree: either a single node, or a full binary tree where some leaf has been replaced by an internal node with two leaf children.

Theorem 6. *Huffman codes are optimal prefix-free binary codes.*

Proof: If the message has only one or two different characters, the theorem is trivial.

Otherwise, let $f[1..n]$ be the original input frequencies, where without loss of generality, $f[1]$ and $f[2]$ are the two smallest. To keep things simple, let $f[n+1] = f[1] + f[2]$. By the previous lemma, we know that some optimal code for $f[1..n]$ has characters 1 and 2 as siblings.

Let T' be the Huffman code tree for $f[3..n+1]$; the inductive hypothesis implies that T' is an optimal code tree for the smaller set of frequencies. To obtain the final code tree T, we replace the leaf labeled $n+1$ with an internal node with two children, labelled 1 and 2. I claim that T is optimal for the original frequency array $f[1..n]$.

To prove this claim, we can express the cost of T in terms of the cost of T' as follows. (In these equations, $depth(i)$ denotes the depth of the leaf labelled i in either T or T'; if the leaf appears in both T and T', it has the same depth in both trees.)

$$\begin{aligned}
cost(T) &= \sum_{i=1}^{n} f[i] \cdot depth(i) \\
&= \sum_{i=3}^{n+1} f[i] \cdot depth(i) + f[1] \cdot depth(1) + f[2] \cdot depth(2) - f[n+1] \cdot depth(n+1) \\
&= cost(T') + f[1] \cdot depth(1) + f[2] \cdot depth(2) - f[n+1] \cdot depth(n+1) \\
&= cost(T') + (f[1] + f[2]) \cdot depth(T) - f[n+1] \cdot (depth(T) - 1) \\
&= cost(T') + f[1] + f[2]
\end{aligned}$$

This equation implies that minimizing the cost of T is equivalent to minimizing the cost of T'; in particular, attaching leaves labeled 1 and 2 to the leaf in T' labeled $n+1$ gives an optimal code tree for the original frequencies. □

To actually implement Huffman codes efficiently, we keep the characters in a min-heap, where the priority of each character is its frequency. We can construct the code tree by keeping three arrays of indices, listing the left and right children and the parent of each node. The root of the tree is the node with index $2n-1$.

```
BUILDHUFFMAN(f[1..n]):
  for i ← 1 to n
      L[i] ← 0; R[i] ← 0
      INSERT(i, f[i])

  for i ← n to 2n−1
      x ← EXTRACTMIN( )
      y ← EXTRACTMIN( )
      f[i] ← f[x] + f[y]
      L[i] ← x; R[i] ← y
      P[x] ← i; P[y] ← i
      INSERT(i, f[i])

  P[2n−1] ← 0
```

The algorithm performs $O(n)$ min-heap operations. If we use a balanced binary tree as the heap, each operation requires $O(\log n)$ time, so the total running time of BUILDHUFFMAN is $O(n \log n)$.

Finally, here are simple algorithms to encode and decode messages:

```
HUFFMANENCODE(A[1..k]):
  m ← 1
  for i ← 1 to k
      HUFFMANENCODEONE(A[i])

HUFFMANENCODEONE(x):
  if x < 2n−1
      HUFFMANENCODEONE(P[x])
      if x = L[P[x]]
          B[m] ← 0
      else
          B[m] ← 1
      m ← m + 1
```

```
HUFFMANDECODE(B[1..m]):
  k ← 1
  v ← 2n−1
  for i ← 1 to m
      if B[i] = 0
          v ← L[v]
      else
          v ← R[v]
      if L[v] = 0
          A[k] ← v
          k ← k + 1
          v ← 2n−1
```

Exercises

1. For each of the following alternative greedy algorithms for the class scheduling problem, either prove that the algorithm always constructs an optimal schedule, or describe a small input example for which the algorithm does not produce an optimal schedule. Assume that all algorithms break ties arbitrarily (that is, in a manner that is completely out of your control).

   (a) Choose the course x that *ends last*, discard classes that conflict with x, and recurse.

   (b) Choose the course x that *starts first*, discard all classes that conflict with x, and recurse.

   (c) Choose the course x that *starts last*, discard all classes that conflict with x, and recurse.

   (d) Choose the course x with *shortest duration*, discard all classes that conflict with x, and recurse.

   (e) Choose a course x that *conflicts with the fewest other courses*, discard all classes that conflict with x, and recurse.

   (f) If no classes conflict, choose them all. Otherwise, discard the course with *longest duration* and recurse.

(g) If no classes conflict, choose them all. Otherwise, discard a course that *conflicts with the most other courses* and recurse.

(h) Let x be the class with the *earliest start time*, and let y be the class with the *second earliest start time*.

- If x and y are disjoint, choose x and recurse on everything but x.
- If x completely contains y, discard x and recurse.
- Otherwise, discard y and recurse.

(i) If any course x completely contains another course, discard x and recurse. Otherwise, choose the course y that *ends last*, discard all classes that conflict with y, and recurse.

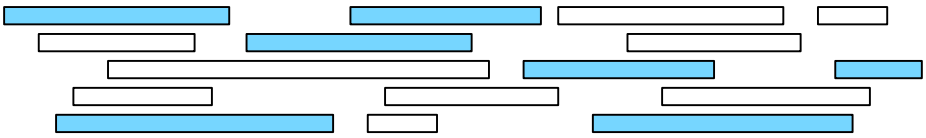
2. Now consider a weighted version of the class scheduling problem, where different classes offer different number of credit hours (totally unrelated to the duration of the class lectures). Your goal is now to choose a set of non-conflicting classes that give you the largest possible number of credit hours, given an array of start times, end times, and credit hours as input.

   (a) Prove that the greedy algorithm described in the notes — Choose the class that ends first and recurse — does *not* always return an optimal schedule.

   (b) Describe an algorithm to compute the optimal schedule in $O(n^2)$ time.

3. Let X be a set of n intervals on the real line. A subset of intervals $Y \subseteq X$ is called a *tiling path* if the intervals in Y cover the intervals in X, that is, any real value that is contained in some interval in X is also contained in some interval in Y. The *size* of a tiling cover is just the number of intervals.

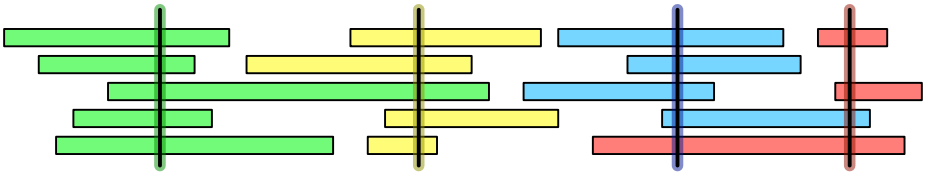
   Describe and analyze an algorithm to compute the smallest tiling path of X as quickly as possible. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X. If you use a greedy algorithm, you must prove that it is correct.



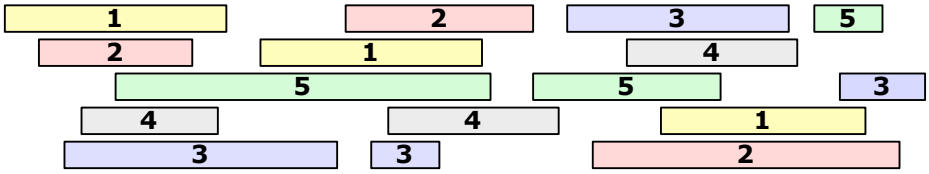
A set of intervals. The seven shaded intervals form a tiling path.

4. Let X be a set of n intervals on the real line. We say that a set P of points *stabs* X if every interval in X contains at least one point in P. Describe and analyze an efficient algorithm to compute the smallest set of points that stabs X. Assume that your input consists of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X. As usual, If you use a greedy algorithm, you must prove that it is correct.

5. Let X be a set of n intervals on the real line. A *proper coloring* of X assigns a color to each interval, so that any two overlapping intervals are assigned different colors. Describe and analyze an efficient algorithm to compute the minimum number of colors needed to properly color X. Assume that your input consists of two arrays $L[1..n]$ and $R[1..n]$, where $L[i]$ and $R[i]$ are the left and right endpoints of the ith interval. As usual, if you use a greedy algorithm, you must prove that it is correct.



A set of intervals stabbed by four points (shown here as vertical segments)



A proper coloring of a set of intervals using five colors.

6. Suppose you are a simple shopkeeper living in a country with n different types of coins, with values $1 = c[1] < c[2] < \cdots < c[n]$. (In the U.S., for example, $n = 6$ and the values are 1, 5, 10, 25, 50 and 100 cents.) Your beloved and benevolent dictator, El Generalissimo, has decreed that whenever you give a customer change, you must use the smallest possible number of coins, so as not to wear out the image of El Generalissimo lovingly engraved on each coin by servants of the Royal Treasury.

   (a) In the United States, there is a simple greedy algorithm that always results in the smallest number of coins: subtract the largest coin and recursively give change for the remainder. El Generalissimo does not approve of American capitalist greed. Show that there is a set of coin values for which the greedy algorithm does *not* always give the smallest possible of coins.

   (b) Now suppose El Generalissimo decides to impose a currency system where the coin denominations are consecutive powers $b^0, b^1, b^2, \ldots, b^k$ of some integer $b \geq 2$. Prove that despite El Generalissimo's disapproval, the greedy algorithm described in part (a) does make optimal change in this currency system.

   (c) Describe and analyze an efficient algorithm to determine, given a target amount A and a sorted array $c[1..n]$ of coin denominations, the smallest number of coins needed to make A cents in change. Assume that $c[1] = 1$, so that it is possible to make change for any amount A.

7. Suppose you have just purchased a new type of hybrid car that uses fuel extremely efficiently, but can only travel 100 miles on a single battery. The car's fuel is stored in a single-use battery, which must be replaced after at most 100 miles. The actual fuel is virtually free, but the batteries are expensive and can only be installed by licensed battery-replacement technicians. Thus, even if you decide to replace your battery early, you must still pay full price for the new battery to be installed. Moreover, because these batteries are in high demand, no one can afford to own more than one battery at a time.

   Suppose you are trying to get from San Francisco to New York City on the new Inter-Continental Super-Highway, which runs in a direct line between these two cities. There are several fueling stations along the way; each station charges a different price for installing a new battery. Before you start your trip, you carefully print the Wikipedia page listing the locations and prices of every

fueling station on the ICSH. Given this information, how do you decide the best places to stop for fuel?

More formally, suppose you are given two arrays $D[1..n]$ and $C[1..n]$, where $D[i]$ is the distance from the start of the highway to the ith station, and $C[i]$ is the cost to replace your battery at the ith station. Assume that your trip starts and ends at fueling stations (so $D[1] = 0$ and $D[n]$ is the total length of your trip), and that your car starts with an empty battery (so you must install a new battery at station 1).

(a) Describe and analyze a greedy algorithm to find the minimum number of refueling stops needed to complete your trip. Don't forget to prove that your algorithm is correct.

(b) But what you really want to minimize is the total *cost* of travel. Show that your greedy algorithm in part (a) does *not* produce an optimal solution when extended to this setting.

(c) Describe an efficient algorithm to compute the locations of the fuel stations you should stop at to minimize the total cost of travel.

8. Recall that a string w of parentheses (and) is ***balanced*** if it satisfies one of the following conditions:

   - w is the empty string.
   - $w = (x)$ for some balanced string x
   - $w = xy$ for some balanced strings x and y

   For example, the string

   $$w = ((())()())(()())()$$

   is balanced, because $w = xy$, where

   $$x = (\,(())\,()\,()\,) \qquad \text{and} \qquad y = (\,()\,()\,)\,().$$

   (a) Describe and analyze an algorithm to determine whether a given string of parentheses is balanced.

   (b) Describe and analyze a greedy algorithm to compute the length of a longest balanced subsequence of a given string of parentheses. As usual, don't forget to prove your algorithm is correct.

   For both problems, your input is an array $w[1..n]$, where for each i, either $w[i] = ($ or $w[i] =)$. Both of your algorithms should run in $O(n)$ time.

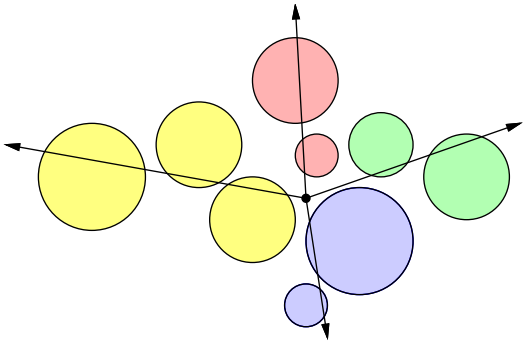
9. Congratulations! You have successfully conquered Camelot, transforming the former battle-scarred kingdom with an anarcho-syndicalist commune, where citizens take turns to act as a sort of executive-officer-for-the-week, but with all the decisions of that officer ratified at a special bi-weekly meeting, by a simple majority in the case of purely internal affairs, but by a two-thirds majority in the case of more major....

   As a final symbolic act, you order the Round Table (surprisingly, an actual circular table) to be split into pizza-like wedges and distributed to the citizens of Camelot as trophies. Each citizen has submitted a request for an angular wedge of the table, specified by two angles—for example: Sir Robin the Brave might request the wedge from 17.23° to 42°, and Sir Lancelot the Pure might

request the 2° wedge from 359° to 1°. Each citizen will be happy if and only if they receive *precisely* the wedge that they requested. Unfortunately, some of these ranges overlap, so satisfying *all* the citizens' requests is simply impossible. Welcome to politics.

Describe and analyze an algorithm to find the maximum number of requests that can be satisfied. *[Hint: Careful! The output of your algorithm must not change if you rotate the table. Do not assume that angles are integers.]*

10. Suppose you are standing in a field surrounded by several large balloons. You want to use your brand new Acme Brand Zap-O-Matic™ to pop all the balloons, without moving from your current location. The Zap-O-Matic™ shoots a high-powered laser beam, which pops all the balloons it hits. Since each shot requires enough energy to power a small country for a year, you want to fire as few shots as possible.



Nine balloons popped by 4 shots of the Zap-O-Matic™

The *minimum zap* problem can be stated more formally as follows. Given a set C of n circles in the plane, each specified by its radius and the (x, y) coordinates of its center, compute the minimum number of rays from the origin that intersect every circle in C. Your goal is to find an efficient algorithm for this problem.

(a) Suppose it is possible to shoot a ray that does not intersect any balloons. Describe and analyze a greedy algorithm that solves the minimum zap problem in this special case. *[Hint: See Exercise 2.]*

(b) Describe and analyze a greedy algorithm whose output is within 1 of optimal. That is, if m is the minimum number of rays required to hit every balloon, then your greedy algorithm must output either m or $m + 1$. (Of course, you must prove this fact.)

(c) Describe an algorithm that solves the minimum zap problem in $O(n^2)$ time.

*(d) Describe an algorithm that solves the minimum zap problem in $O(n \log n)$ time.

Assume you have a subroutine INTERSECTS(r, c) that determines whether an arbitrary ray r intersects an arbitrary circle c in $O(1)$ time. This subroutine is not difficult to write, but it's not the interesting part of the problem.

Obie looked at the seein' eye dog. Then at the twenty-seven 8 by 10 color glossy pictures with the circles and arrows and a paragraph on the back of each one. . . and then he looked at the seein' eye dog. And then at the twenty-seven 8 by 10 color glossy pictures with the circles and arrows and a paragraph on the back of each one and began to cry.

Because Obie came to the realization that it was a typical case of American blind justice, and there wasn't nothin' he could do about it, and the judge wasn't gonna look at the twenty-seven 8 by 10 color glossy pictures with the circles and arrows and a paragraph on the back of each one explainin' what each one was, to be used as evidence against us.

And we was fined fifty dollars and had to pick up the garbage. In the snow.

But that's not what I'm here to tell you about.

— Arlo Guthrie, "Alice's Restaurant" (1966)

I study my Bible as I gather apples.
First I shake the whole tree, that the ripest might fall.
Then I climb the tree and shake each limb,
and then each branch and then each twig,
and then I look under each leaf.

— Martin Luther

23 Basic Graph Algorithms

23.1 Definitions

A ***graph*** is normally defined as a pair of sets (V, E), where V is a set of arbitrary objects called ***vertices***[1] or ***nodes***. E is a set of pairs of vertices, which we call ***edges*** or (more rarely) ***arcs***. In an *undirected* graph, the edges are unordered pairs, or just sets of two vertices; I usually write $\boldsymbol{uv}$ instead of $\{u, v\}$ to denote the undirected edge between u and v. In a *directed* graph, the edges are ordered pairs of vertices; I usually write $\boldsymbol{u \to v}$ instead of (u, v) to denote the directed edge from u to v.

The definition of a graph as a pair of *sets* forbids graphs with loops (edges from a vertex to itself) and/or parallel edges (multiple edges with the same endpoints). Graphs *without* loops and parallel edges are often called ***simple*** graphs; non-simple graphs are sometimes called ***multigraphs***. Despite the formal definitional gap, most algorithms for simple graphs extend to non-simple graphs with little or no modification.

Following standard (but admittedly confusing) practice, I'll also use V to denote the *number* of vertices in a graph, and E to denote the *number* of edges. Thus, in any undirected graph we have $0 \le E \le \binom{V}{2}$, and in any directed graph we have $0 \le E \le V(V-1)$.

For any edge uv in an undirected graph, we call u a ***neighbor*** of v and vice versa. The ***degree*** of a node is its number of neighbors. In directed graphs, we have two kinds of neighbors. For any directed edge $u \to v$, we call u a ***predecessor*** of v and v a ***successor*** of u. The ***in-degree*** of a node is the number of predecessors, which is the same as the number of edges going into the node. The ***out-degree*** is the number of successors, or the number of edges going out of the node.

A graph $G' = (V', E')$ is a ***subgraph*** of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.

A ***walk*** in a graph is a sequence of edges, where each successive pair of edges shares one vertex; a walk is called a ***path*** if it visits each vertex at most once. An undirected graph is ***connected*** if there

[1] The singular of 'vertices' is **vertex**. The singular of 'matrices' is **matrix**. Unless you're speaking Italian, there is no such thing as a vertice, a matrice, an indice, an appendice, a helice, an apice, a vortice, a radice, a simplice, a codice, a directrice, a dominatrice, a Unice, a Kleenice, an Asterice, an Obelice, a Dogmatice, a Getafice, a Cacofonice, a Vitalstatistice, a Geriatrice, or Jimi Hendrice! You *will* lose points for using any of these so-called words.

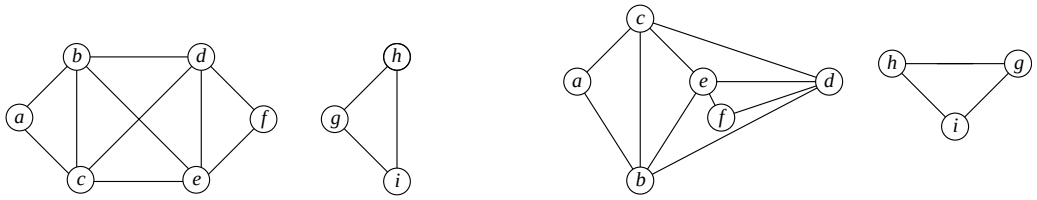
is a walk (and therefore a path) between any two vertices. A disconnected graph consists of several ***components***, which are its maximal connected subgraphs. Two vertices are in the same component if and only if there is a path between them. Components are sometimes called "connected components", but this usage is redundant; components are connected by definition.

A ***cycle*** is a path that starts and ends at the same vertex, and has at least one edge. An undirected graph is ***acyclic*** if no subgraph is a cycle; acyclic graphs are also called ***forests***. A ***tree*** is a connected acyclic graph, or equivalently, one component of a forest. A ***spanning tree*** of a graph G is a subgraph that is a tree and contains every vertex of G. A graph has a spanning tree if and only if it is connected. A ***spanning forest*** of G is a collection of spanning trees, one for each connected component of G.

Directed graphs can contain directed paths and directed cycles. A directed graph is ***strongly connected*** if there is a directed path from any vertex to any other. A directed graph is ***acyclic*** if it does not contain a directed cycle; directed acyclic graphs are often called ***dags***.

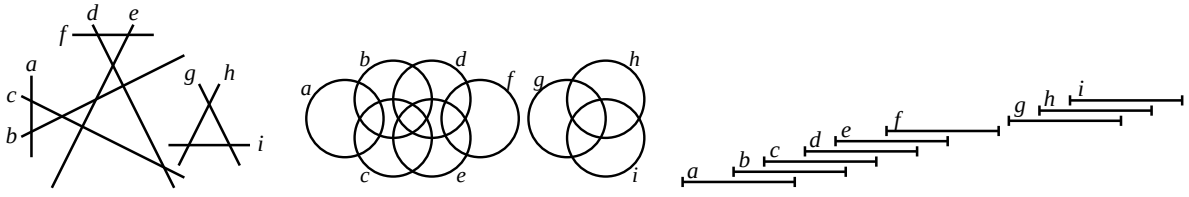
23.2 Abstract Representations and Examples

The most common way to visually represent graphs is with an ***embedding***. An embedding of a graph maps each vertex to a point in the plane (typically drawn as a small circle) and each edge to a curve or straight line segment between the two vertices. A graph is ***planar*** if it has an embedding where no two edges cross. The same graph can have many different embeddings, so it is important not to confuse a particular embedding with the graph itself. In particular, planar graphs can have non-planar embeddings!



A non-planar embedding of a planar graph with nine vertices, thirteen edges, and two components, and a planar embedding of the same graph.

However, embeddings are not the only useful representation of graphs. For example, the ***intersection graph*** of a collection of objects has a node for every object and an edge for every intersecting pair. Whether a particular graph can be represented as an intersection graph depends on what kind of object you want to use for the vertices. Different types of objects—line segments, rectangles, circles, etc.—define different classes of graphs. One particularly useful type of intersection graph is an *interval graph*, whose vertices are intervals on the real line, with an edge between any two intervals that overlap.



The example graph is also the intersection graph of (a) a set of line segments, (b) a set of circles, and (c) a set of intervals on the real line (stacked for visibility).

Another good example is the ***dependency graph*** of a recursive algorithm. Dependency graphs are directed acyclic graphs. The vertices are all the distinct recursive subproblems that arise when executing the algorithm on a particular input. There is an edge from one subproblem to another if evaluating the

second subproblem requires a recursive evaluation of the first. For example, for the Fibonacci recurrence

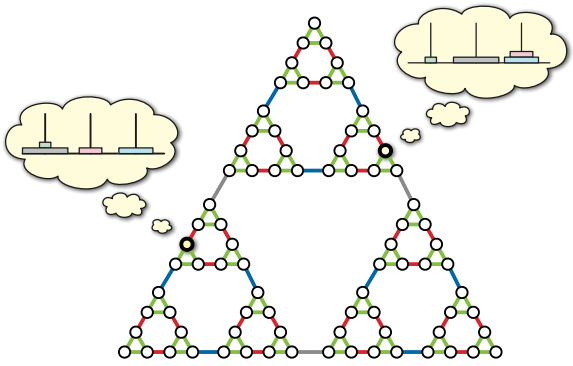
$$F_n = \begin{cases} 0 & \text{if } n = 0, \\ 1 & \text{if } n = 1, \\ F_{n-1} + F_{n-2} & \text{otherwise,} \end{cases}$$

the vertices of the dependency graph are the integers $0, 1, 2, \ldots, n$, and the edges are the pairs $(i-1)\to i$ and $(i-2)\to i$ for every integer i between 2 and n. As a more complex example, consider the following recurrence, which solves a certain sequence-alignment problem called *edit distance*; see the dynamic programming notes for details:

$$Edit(i,j) = \begin{cases} i & \text{if } j = 0 \\ j & \text{if } i = 0 \\ \min \left\{ \begin{array}{l} Edit(i-1,j)+1, \\ Edit(i,j-1)+1, \\ Edit(i-1,j-1)+\big[A[i] \neq B[j]\big] \end{array} \right\} & \text{otherwise} \end{cases}$$

The dependency graph of this recurrence is an $m \times n$ grid of vertices (i, j) connected by vertical edges $(i-1, j)\to(i, j)$, horizontal edges $(i, j-1)\to(i, j)$, and diagonal edges $(i-1, j-1)\to(i, j)$. Dynamic programming works efficiently for any recurrence that has a reasonably small dependency graph; a proper evaluation order ensures that each subproblem is visited *after* its predecessors.

Another interesting example is the ***configuration graph*** of a game, puzzle, or mechanism like tic-tac-toe, checkers, the Rubik's Cube, the Towers of Hanoi, or a Turing machine. The vertices of the configuration graph are all the valid configurations of the puzzle; there is an edge from one configuration to another if it is possible to transform one configuration into the other with a simple move. (Obviously, the precise definition depends on what moves are allowed.) Even for reasonably simple mechanisms, the configuration graph can be extremely complex, and we typically only have access to local information about the configuration graph.



The configuration graph of the 4-disk Tower of Hanoi.

Finite-state automata used in formal language theory can be modeled as labeled directed graphs. Recall that a deterministic finite-state automaton is formally defined as a 5-tuple $M = (\Sigma, Q, s, A, \delta)$, where Σ is a finite set called the *alphabet*, Q is a finite set of *states*, $s \in Q$ is the *start state*, $A \subseteq Q$ is the set of *accepting states*, and $\delta : Q \times \Sigma \to Q$ is a *transition function*. But it is often more useful to think of M as a directed graph G_M whose vertices are the states Q, and whose edges have the form $q\to\delta(q, a)$ for every state $q \in Q$ and symbol $a \in \Sigma$. Then basic questions about the language accepted by M can be phrased

as questions about the graph G_M. For example, the language accepted by M is empty if and only if there is no path in G_M from the start state/vertex q_0 to an accepting state/vertex.

Finally, sometimes one graph can be used to implicitly represent other larger graphs. A good example of this implicit representation is the subset construction used to convert NFAs into DFAs. The subset construction can be generalized to *arbitrary* directed graphs as follows. Given *any* directed graph $G = (V, E)$, we can define a new directed graph $G' = (2^V, E')$ whose vertices are all *subsets* of vertices in V, and whose edges E' are defined as follows:

$$E' := \left\{ A \to B \;\middle|\; u \to v \in E \text{ for some } u \in A \text{ and } v \in B \right\}$$

We can mechanically translate this definition into an algorithm to construct G' from G, but strictly speaking, this construction is unnecessary, because ***G is already an implicit representation of G'***. Viewed in this light, the *incremental* subset construction used to convert NFAs to DFAs without unreachable states is just a breadth-first search of the implicitly-represented DFA.

It's important not to confuse these examples/representations of graphs with the actual formal *definition*: A graph is a pair of sets (V, E), where V is an arbitrary non-empty finite set, and E is a set of pairs (either ordered or unordered) of elements of V.

23.3 Graph Data Structures

In practice, graphs are represented by two data structures: *adjacency matrices*[2] and *adjacency lists*.

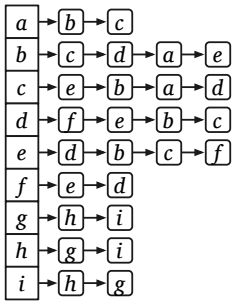
The ***adjacency matrix*** of a graph G is a $V \times V$ matrix, in which each entry indicates whether a particular edge is or is not in the graph:

$$A[i, j] := \left[(i, j) \in E \right].$$

For undirected graphs, the adjacency matrix is always *symmetric*: $A[i, j] = A[j, i]$. Since we don't allow edges from a vertex to itself, the diagonal elements $A[i, i]$ are all zeros.

Given an adjacency matrix, we can decide in $\Theta(1)$ time whether two vertices are connected by an edge just by looking in the appropriate slot in the matrix. We can also list all the neighbors of a vertex in $\Theta(V)$ time by scanning the corresponding row (or column). This is optimal in the worst case, since a vertex can have up to $V - 1$ neighbors; however, if a vertex has few neighbors, we may still have to examine every entry in the row to see them all. Similarly, adjacency matrices require $\Theta(V^2)$ space, regardless of how many edges the graph actually has, so it is only space-efficient for very *dense* graphs.

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*
a	0	1	1	0	0	0	0	0	0
b	1	0	1	1	1	0	0	0	0
c	1	1	0	1	1	0	0	0	0
d	0	1	1	0	1	1	0	0	0
e	0	1	1	1	0	1	0	0	0
f	0	0	0	1	1	0	0	0	0
g	0	0	0	0	0	0	0	1	0
h	0	0	0	0	0	0	1	0	1
i	0	0	0	0	0	0	1	1	0



Adjacency matrix and adjacency list representations for the example graph.

For *sparse* graphs—graphs with relatively few edges—adjacency lists are usually a better choice. An ***adjacency list*** is an array of linked lists, one list per vertex. Each linked list stores the neighbors of

[2] See footnote 1.

the corresponding vertex. For undirected graphs, each edge (u, v) is stored twice, once in u's neighbor list and once in v's neighbor list; for directed graphs, each edge is stored only once. Either way, the overall space required for an adjacency list is $O(V + E)$. Listing the neighbors of a node v takes $O(1 + \deg(v))$ time; just scan the neighbor list. Similarly, we can determine whether (u, v) is an edge in $O(1 + \deg(u))$ time by scanning the neighbor list of u. For undirected graphs, we can improve the time to $O(1 + \min\{\deg(u), \deg(v)\})$ by simultaneously scanning the neighbor lists of both u and v, stopping either we locate the edge or when we fall of the end of a list.

The adjacency list data structure should immediately remind you of hash tables with chaining; the two data structures are identical.[3] Just as with chained hash tables, we can make adjacency lists more efficient by using something other than a linked list to store the neighbors of each vertex. For example, if we use a hash table with constant load factor, when we can detect edges in $O(1)$ time, just as with an adjacency matrix. (Most hash give us only $O(1)$ *expected* time, but we can get $O(1)$ *worst-case* time using cuckoo hashing.)

The following table summarizes the performance of the various standard graph data structures. Stars* indicate expected amortized time bounds for maintaining dynamic hash tables.

	Adjacency matrix	Standard adjacency list (linked lists)	Adjacency list (hash tables)
Space	$\Theta(V^2)$	$\Theta(V + E)$	$\Theta(V + E)$
Time to test if $uv \in E$	$O(1)$	$O(1 + \min\{\deg(u), \deg(v)\}) = O(V)$	$O(1)$
Time to test if $u \to v \in E$	$O(1)$	$O(1 + \deg(u)) = O(V)$	$O(1)$
Time to list the neighbors of v	$O(V)$	$O(1 + \deg(v))$	$O(1 + \deg(v))$
Time to list all edges	$\Theta(V^2)$	$\Theta(V + E)$	$\Theta(V + E)$
Time to add edge uv	$O(1)$	$O(1)$	$O(1)^*$
Time to delete edge uv	$O(1)$	$O(\deg(u) + \deg(v)) = O(V)$	$O(1)^*$

At this point, one might reasonably wonder why anyone would ever use an adjacency matrix; after all, adjacency lists with hash tables support the same operations in the same time, using less space. Similarly, why would anyone use linked lists in an adjacency list structure to store neighbors, instead of hash tables? Although the main reason in practice is almost surely ***tradition***—If it was good enough for your grandfather's code, it should be good enough for yours!—there are some more principled arguments. One reason is that the standard adjacency lists are usually good enough; most graph algorithms never actually ask whether a given edge is present or absent! Another reason is that for sufficiently dense graphs, adjacency matrices are simpler and more efficient in practice, because they avoid the overhead of chasing pointers or computing hash functions.

But perhaps the most compelling reason is that many graphs are *implicitly* represented by adjacency matrices and standard adjacency lists. For example, intersection graphs are usually represented as a list of the underlying geometric objects. As long as we can test whether two objects intersect in constant time, we can apply any graph algorithm to an intersection graph by *pretending* that it is stored explicitly as an adjacency matrix.

Similarly, any data structure composed from records with pointers between them can be seen as a directed graph; graph algorithms can be applied to these data structures by *pretending* that the graph is stored in a standard adjacency list. Similarly, we can apply any graph algorithm to a configuration graph *as though* it were given to us as a standard adjacency list, provided we can enumerate all possible moves from a given configuration in constant time each. In both of these contexts, we can enumerate the edges leaving any vertex in time proportional to its degree, but we *cannot* necessarily determine in constant

[3]For some reason, adjacency lists are always drawn with *horizontal* lists, while chained hash tables are always drawn with *vertical* lists. Don't ask me; I just work here.

time if two vertices are connected. (Is there a pointer from this record to that record? Can we get from this configuration to that configuration in one move?) Thus, a standard adjacency list, with neighbors stored in linked lists, is the appropriate model data structure.

23.4 Traversing Connected Graphs

To keep things simple, we'll consider only undirected graphs for the rest of this lecture, although the algorithms also work for directed graphs with minimal changes.

Suppose we want to visit every node in a connected graph (represented either explicitly or implicitly). Perhaps the simplest graph-traversal algorithm is *depth-first search*. This algorithm can be written either recursively or iteratively. It's exactly the same algorithm either way; the only difference is that we can actually see the "recursion" stack in the non-recursive version. Both versions are initially passed a *source* vertex s.

```
RECURSIVEDFS(v):
  if v is unmarked
    mark v
    for each edge vw
      RECURSIVEDFS(w)
```

```
ITERATIVEDFS(s):
  PUSH(s)
  while the stack is not empty
    v ← POP
    if v is unmarked
      mark v
      for each edge vw
        PUSH(w)
```

Depth-first search is just one (perhaps the most common) species of a general family of graph traversal algorithms. The generic graph traversal algorithm stores a set of candidate edges in some data structure that I'll call a "bag". The only important properties of a "bag" are that we can put stuff into it and then later take stuff back out. (In C++ terms, think of the bag as a template for a real data structure.) A stack is a particular type of bag, but certainly not the only one. Here is the generic traversal algorithm:

```
TRAVERSE(s):
  put s into the bag
  while the bag is not empty
    take v from the bag
    if v is unmarked
      mark v
      for each edge vw
        put w into the bag
```

This traversal algorithm clearly marks each vertex in the graph *at most* once. In order to show that it visits every node in a connected graph *at least* once, we modify the algorithm slightly; the modifications are highlighted in red. Instead of keeping vertices in the bag, the modified algorithm stores pairs of vertices. This modification allows us to remember, whenever we visit a vertex v for the first time, which previously-visited neighbor vertex put v into the bag. We call this earlier vertex the *parent* of v.

```
TRAVERSE(s):
  put (∅,s) in bag
  while the bag is not empty
    take (p,v) from the bag                (⋆)
    if v is unmarked
      mark v
      parent(v) ← p
      for each edge vw                     (†)
        put (v,w) into the bag             (⋆⋆)
```

Lemma 1. TRAVERSE(s) *marks every vertex in any connected graph exactly once, and the set of pairs* $(v, parent(v))$ *with* $parent(v) \neq \emptyset$ *defines a spanning tree of the graph.*

Proof: The algorithm marks s. Let v be any vertex other than s, and let $(s, \ldots, u, v)$ be the path from s to v with the minimum number of edges. Since the graph is connected, such a path always exists. (If s and v are neighbors, then $u = s$, and the path has just one edge.) If the algorithm marks u, then it must put (u, v) into the bag, so it must later take (u, v) out of the bag, at which point v must be marked. Thus, by induction on the shortest-path distance from s, the algorithm marks every vertex in the graph, which implies that $parent(v)$ is well-defined for every vertex v.

The algorithm clearly marks every vertex at most once, so it must mark every vertex *exactly* once.

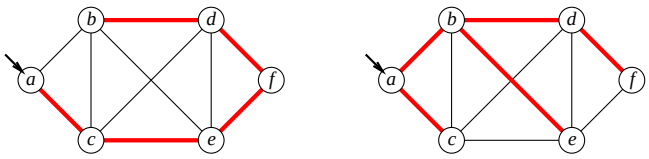
Call any pair $(v, parent(v))$ with $parent(v) \neq \emptyset$ a *parent edge*. For any node v, the path of parent edges $(v, parent(v), parent(parent(v)), \ldots)$ eventually leads back to s, so the set of parent edges form a connected graph. Clearly, both endpoints of every parent edge are marked, and the number of parent edges is exactly one less than the number of vertices. Thus, the parent edges form a spanning tree. □

The exact running time of the traversal algorithm depends on how the graph is represented and what data structure is used as the 'bag', but we can make a few general observations. Because each vertex is marked at most once, the for loop (†) is executed at most V times. Each edge uv is put into the bag exactly twice; once as the pair (u, v) and once as the pair (v, u), so line ($\star\star$) is executed at most $2E$ times. Finally, we can't take more things out of the bag than we put in, so line ($\star$) is executed at most $2E + 1$ times.

23.5 Examples

Let's first assume that the graph is represented by a standard adjacency list, so that the overhead of the for loop (†) is only constant time per edge.

- If we implement the 'bag' using a *stack*, we recover our original depth-first search algorithm. Each execution of ($\star$) or ($\star\star$) takes constant time, so the algorithms runs in $\boldsymbol{O(V+E)}$ ***time*** . If the graph is connected, we have $V \leq E + 1$, and so we can simplify the running time to $O(E)$. The spanning tree formed by the parent edges is called a ***depth-first spanning tree***. The exact shape of the tree depends on the start vertex and on the order that neighbors are visited in the for loop (†), but in general, depth-first spanning trees are long and skinny.
- If we use a *queue* instead of a stack, we get ***breadth-first search***. Again, each execution of ($\star$) or ($\star\star$) takes constant time, so the overall running time for connected graphs is still $\boldsymbol{O(E)}$. In this case, the ***breadth-first spanning tree*** formed by the parent edges contains ***shortest paths*** from the start vertex s to every other vertex in its connected component. We'll see shortest paths again in a future lecture. Again, exact shape of a breadth-first spanning tree depends on the start vertex and on the order that neighbors are visited in the for loop (†), but in general, breadth-first spanning trees are short and bushy.



A depth-first spanning tree and a breadth-first spanning tree of one component of the example graph, with start vertex a.

- Now suppose the edges of the graph are weighted. If we implement the 'bag' using a *priority queue*, always extracting the minimum-weight edge in line ($\star$), the resulting algorithm is reasonably called ***shortest-first search***. In this case, each execution of ($\star$) or ($\star\star$) takes $O(\log E)$ time, so the overall running time is $\boldsymbol{O(V + E \log E)}$, which simplifies to $\boldsymbol{O(E \log E)}$ if the graph is connected. For this algorithm, the set of parent edges form the ***minimum spanning tree*** of the connected component of s. Surprisingly, as long as all the edge weights are distinct, the resulting tree does *not* depend on the start vertex or the order that neighbors are visited; in this case, there is only one minimum spanning tree. We'll see minimum spanning trees again in the next lecture.

If the graph is represented using an adjacency matrix instead of an adjacency list, finding all the neighbors of each vertex in line (†) takes $O(V)$ time. Thus, depth- and breadth-first search each run in $O(V^2)$ time, and 'shortest-first search' runs in $O(V^2 + E \log E) = O(V^2 \log V)$ time.

23.6 Searching Disconnected Graphs

If the graph is disconnected, then TRAVERSE(s) only visits the nodes in the connected component of the start vertex s. If we want to visit all the nodes in every component, we can use the following 'wrapper' around our generic traversal algorithm. Since TRAVERSE computes a spanning tree of one component, TRAVERSEALL computes a spanning *forest* of the entire graph.

```
TRAVERSEALL(s):
  for all vertices v
    if v is unmarked
      TRAVERSE(v)
```

Surprisingly, a few well-known algorithms textbooks claim that this wrapper can only be used with depth-first search. They're wrong.

Exercises

1. Prove that the following definitions are all equivalent.

   - A tree is a connected acyclic graph.
   - A tree is one component of a forest.
   - A tree is a connected graph with *at most* $V - 1$ edges.
   - A tree is a minimal connected graph; removing any edge makes the graph disconnected.
   - A tree is an acyclic graph with *at least* $V - 1$ edges.
   - A tree is a maximal acyclic graph; adding an edge between any two vertices creates a cycle.

2. Prove that any connected acyclic graph with $n \geq 2$ vertices has at least two vertices with degree 1. Do not use the words "tree" or "leaf", or any well-known properties of trees; your proof should follow entirely from the definitions of "connected" and "acyclic".

3. Let G be a connected graph, and let T be a depth-first spanning tree of G rooted at some node v. Prove that if T is also a breadth-first spanning tree of G rooted at v, then $G = T$.

4. Whenever groups of pigeons gather, they instinctively establish a *pecking order*. For any pair of pigeons, one pigeon always pecks the other, driving it away from food or potential mates. The same pair of pigeons always chooses the same pecking order, even after years of separation, no matter what other pigeons are around. Surprisingly, the overall pecking order can contain cycles—for example, pigeon A pecks pigeon B, which pecks pigeon C, which pecks pigeon A.

   (a) Prove that any finite set of pigeons can be arranged in a row from left to right so that every pigeon pecks the pigeon immediately to its left. Pretty please.

   (b) Suppose you are given a directed graph representing the pecking relationships among a set of n pigeons. The graph contains one vertex per pigeon, and it contains an edge $i \to j$ if and only if pigeon i pecks pigeon j. Describe and analyze an algorithm to compute a pecking order for the pigeons, as guaranteed by part (a).

5. A graph (V, E) is *bipartite* if the vertices V can be partitioned into two subsets L and R, such that every edge has one vertex in L and the other in R.

   (a) Prove that every tree is a bipartite graph.

   (b) Describe and analyze an efficient algorithm that determines whether a given undirected graph is bipartite.

6. An ***Euler tour*** of a graph G is a closed walk through G that traverses every edge of G exactly once.

   (a) Prove that a connected graph G has an Euler tour if and only if every vertex has even degree.

   (b) Describe and analyze an algorithm to compute an Euler tour in a given graph, or correctly report that no such graph exists.

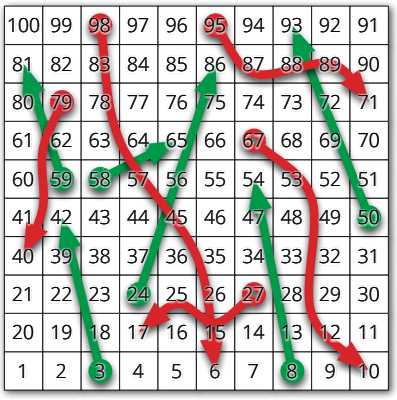
7. The d-dimensional hypercube is the graph defined as follows. There are $2d$ vertices, each labeled with a different string of d bits. Two vertices are joined by an edge if their labels differ in exactly one bit.

   (a) A Hamiltonian cycle in a graph G is a cycle of edges in G that visits every vertex of G exactly once. Prove that for all $d \geq 2$, the d-dimensional hypercube has a Hamiltonian cycle.

   (b) Which hypercubes have an Euler tour (a closed walk that traverses every edge exactly once)? *[Hint: This is very easy.]*

8. ***Snakes and Ladders*** is a classic board game, originating in India no later than the 16th century. The board consists of an $n \times n$ grid of squares, numbered consecutively from 1 to n^2, starting in the bottom left corner and proceeding row by row from bottom to top, with rows alternating to the left and right. Certain pairs of squares in this grid, always in different rows, are connected by either "snakes" (leading down) or "ladders" (leading up). Each square can be an endpoint of at most one snake or ladder.

   You start with a token in cell 1, in the bottom left corner. In each move, you advance your token up to k positions, for some fixed constant k. If the token ends the move at the *top* end of a snake, it slides down to the bottom of that snake. Similarly, if the token ends the move at the *bottom* end of a ladder, it climbs up to the top of that ladder.

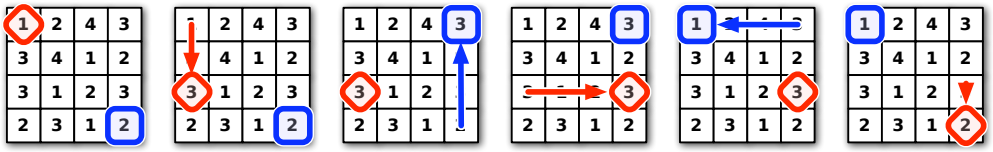


A typical Snakes and Ladders board.
Upward straight arrows are ladders; downward wavy arrows are snakes.

Describe and analyze an algorithm to compute the smallest number of moves required for the token to reach the last square of the grid.

9. The following puzzle was invented by the infamous Mongolian puzzle-warrior Vidrach Itky Leda in the year 1473. The puzzle consists of an $n \times n$ grid of squares, where each square is labeled with a positive integer, and two tokens, one red and the other blue. The tokens always lie on distinct squares of the grid. The tokens start in the top left and bottom right corners of the grid; the goal of the puzzle is to swap the tokens.

   In a single turn, you may move either token up, right, down, or left *by a distance determined by the **other** token*. For example, if the red token is on a square labeled 3, then you may move the blue token 3 steps up, 3 steps left, 3 steps right, or 3 steps down. However, you may not move a token off the grid or to the same square as the other token.



A five-move solution for a 4×4 Vidrach Itky Leda puzzle.

Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given Vidrach Itky Leda puzzle, or correctly reports that the puzzle has no solution. For example, given the puzzle above, your algorithm would return the number 5.

10. ***Racetrack*** (also known as *Graph Racers* and *Vector Rally*) is a two-player paper-and-pencil racing game that Jeff played on the bus in 5th grade.[4] The game is played with a track drawn on a sheet of graph paper. The players alternately choose a sequence of grid points that represent the motion of a car around the track, subject to certain constraints explained below.

    Each car has a *position* and a *velocity*, both with integer x- and y-coordinates. A subset of grid squares is marked as the *starting area*, and another subset is marked as the *finishing area*.

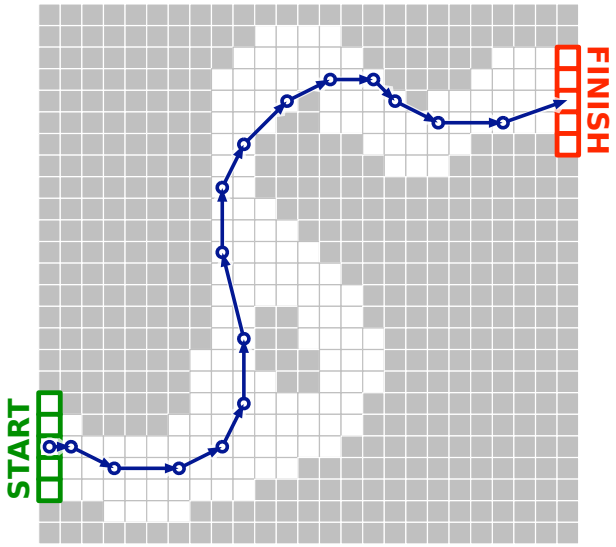
[4]The actual game is a bit more complicated than the version described here. See http://harmmade.com/vectorracer/ for an excellent online version.

The initial position of each car is chosen by the player somewhere in the starting area; the initial velocity of each car is always $(0,0)$. At each step, the player optionally increments or decrements either or both coordinates of the car's velocity; in other words, each component of the velocity can change by at most 1 in a single step. The car's new position is then determined by adding the new velocity to the car's previous position. The new position must be inside the track; otherwise, the car crashes and that player loses the race. The race ends when the first car reaches a position inside the finishing area.

Suppose the racetrack is represented by an $n \times n$ array of bits, where each 0 bit represents a grid point inside the track, each 1 bit represents a grid point outside the track, the 'starting area' is the first column, and the 'finishing area' is the last column.

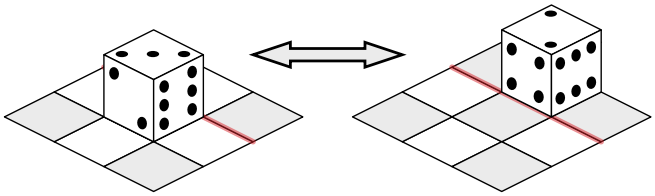
Describe and analyze an algorithm to find the minimum number of steps required to move a car from the starting line to the finish line of a given racetrack. *[Hint: Build a graph. What are the vertices? What are the edges? What problem is this?]*

velocity	position
$(0,0)$	$(1,5)$
$(1,0)$	$(2,5)$
$(2,-1)$	$(4,4)$
$(3,0)$	$(7,4)$
$(2,1)$	$(9,5)$
$(1,2)$	$(10,7)$
$(0,3)$	$(10,10)$
$(-1,4)$	$(9,14)$
$(0,3)$	$(9,17)$
$(1,2)$	$(10,19)$
$(2,2)$	$(12,21)$
$(2,1)$	$(14,22)$
$(2,0)$	$(16,22)$
$(1,-1)$	$(17,21)$
$(2,-1)$	$(19,20)$
$(3,0)$	$(22,20)$
$(3,1)$	$(25,21)$



A 16-step Racetrack run, on a 25×25 track. This is *not* the shortest run on this track.

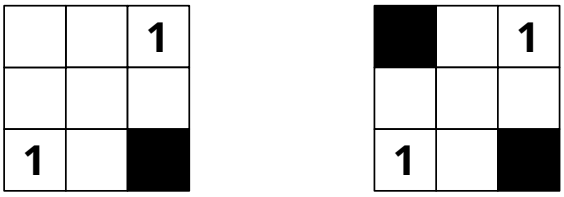
11. A *rolling die maze* is a puzzle involving a standard six-sided die (a cube with numbers on each side) and a grid of squares. You should imagine the grid lying on top of a table; the die always rests on and exactly covers one square. In a single step, you can *roll* the die 90 degrees around one of its bottom edges, moving it to an adjacent square one step north, south, east, or west.



Rolling a die.

Some squares in the grid may be *blocked*; the die can never rest on a blocked square. Other squares may be *labeled* with a number; whenever the die rests on a labeled square, the number of pips on the *top* face of the die must equal the label. Squares that are neither labeled nor marked are *free*. You may not roll the die off the edges of the grid. A rolling die maze is *solvable* if it is possible to place a die on the lower left square and roll it to the upper right square under these constraints.

For example, here are two rolling die mazes. Black squares are blocked. The maze on the left can be solved by placing the die on the lower left square with 1 pip on the top face, and then rolling it north, then north, then east, then east. The maze on the right is not solvable.



Two rolling die mazes. Only the maze on the left is solvable.

(a) Suppose the input is a two-dimensional array $L[1..n][1..n]$, where each entry $L[i][j]$ stores the label of the square in the ith row and jth column, where 0 means the square is free and -1 means the square is blocked. Describe and analyze a polynomial-time algorithm to determine whether the given rolling die maze is solvable.

*(b) Now suppose the maze is specified *implicitly* by a list of labeled and blocked squares. Specifically, suppose the input consists of an integer M, specifying the height and width of the maze, and an array $S[1..n]$, where each entry $S[i]$ is a triple (x, y, L) indicating that square (x, y) has label L. As in the explicit encoding, label -1 indicates that the square is blocked; free squares are not listed in S at all. Describe and analyze an efficient algorithm to determine whether the given rolling die maze is solvable. For full credit, the running time of your algorithm should be polynomial in the input size n.

[Hint: You have some freedom in how to place the initial die. There are rolling die mazes that can only be solved if the initial position is chosen correctly.]

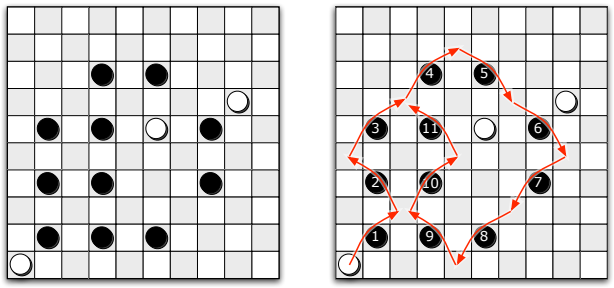
*12. Draughts (also known as checkers) is a game played on an $m \times m$ grid of squares, alternately colored light and dark. (The game is usually played on an 8×8 or 10×10 board, but the rules easily generalize to any board size.) Each dark square is occupied by at most one game piece (usually called a *checker* in the U.S.), which is either black or white; light squares are always empty. One player ('White') moves the white pieces; the other ('Black') moves the black pieces.

Consider the following simple version of the game, essentially American checkers or British draughts, but where every piece is a king.[5] Pieces can be moved in any of the four diagonal directions, either one or two steps at a time. On each turn, a player either *moves* one of her pieces one step diagonally into an empty square, or makes a series of *jumps* with one of her checkers. In a single jump, a piece moves to an empty square two steps away in any diagonal direction, but only

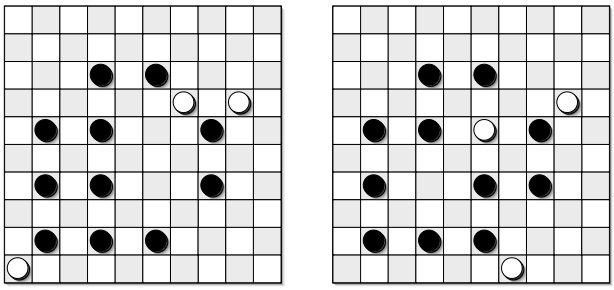
[5]Most other variants of draughts have 'flying kings', which behave *very* differently than what's described here. In particular, if we allow flying kings, it is actually NP-hard to determine which move captures the most enemy pieces. The most common international version of draughts also has a forced-capture rule, which *requires* each player to capture the maximum possible number of enemy pieces in each move. Thus, just following the rules is NP-hard!

if the intermediate square is occupied by a piece of the opposite color; this enemy piece is *captured* and immediately removed from the board. Multiple jumps are allowed in a single turn as long as they are made by the same piece. A player wins if her opponent has no pieces left on the board.

Describe an algorithm that correctly determines whether White can capture every black piece, thereby winning the game, *in a single turn*. The input consists of the width of the board (m), a list of positions of white pieces, and a list of positions of black pieces. For full credit, your algorithm should run in $O(n)$ time, where n is the total number of pieces. *[Hint: The greedy strategy—make arbitrary jumps until you get stuck—does* ***not*** *always find a winning sequence of jumps even when one exists. See problem 6. Parity, parity, parity.]*



White wins in one turn.



White cannot win in one turn from either of these positions.

Ts'ui Pe must have said once: I am withdrawing to write a book.
And another time: I am withdrawing to construct a labyrinth.
Every one imagined two works;
to no one did it occur that the book and the maze were one and the same thing.

— Jorge Luis Borges, "El jardín de senderos que se bifurcan" (1942)
English translation ("The Garden of Forking Paths") by Donald A. Yates (1958)

"Com'è bello il mondo e come sono brutti i labirinti!" dissi sollevato.
"Come sarebbe bello il mondo se ci fosse una regola per girare nei labirinti,"
rispose il mio maestro.

["How beautiful the world is, and how ugly labyrinths are," I said, relieved.
"How beautiful the world would be if there were a procedure for moving through labyrinths,"
my master replied.]

— Umberto Eco, *Il nome della rosa* (1980)
English translation (*The Name of the Rose*) by William Weaver (1983)

At some point, the learning stops and the pain begins.

— Rao Kosaraju

19 Depth-First Search

Recall from the previous lecture the recursive formulation of depth-first search in undirected graphs.

```
DFS(v):
  if v is unmarked
      mark v
      for each edge vw
          DFS(w)
```

We can make this algorithm slightly faster (in practice) by checking whether a node is marked *before* we recursively explore it. This modification ensures that we call DFS(v) only once for each vertex v. We can further modify the algorithm to define parent pointers and other useful information about the vertices. This additional information is computed by two black-box subroutines PREVISIT and POSTVISIT, which we leave unspecified for now.

```
DFS(v):
  mark v
  PREVISIT(v)
  for each edge vw
      if w is unmarked
          parent(w) ← v
          DFS(w)
  POSTVISIT(v)
```

We can search any *connected* graph by unmarking all vertices and then calling DFS(s) for an arbitrary start vertex s. As we argued in the previous lecture, the subgraph of all parent edges $v \to parent(v)$ defines a spanning tree of the graph, which we consider to be rooted at the start vertex s.

Lemma 1. *Let T be a depth-first spanning tree of a connected undirected graph G, computed by calling* DFS(s)*. For any node v, the vertices that are marked during the execution of* DFS(v) *are the proper descendants of v in T.*

Proof: T is also the recursion tree for DFS(s). □

Lemma 2. *Let T be a depth-first spanning tree of a connected undirected graph G. For every edge vw in G, either v is an ancestor of w in T, or v is a descendant of w in T.*

Proof: Assume without loss of generality that v is marked before w. Then w is unmarked when DFS(v) is invoked, but marked when DFS(v) returns, so the previous lemma implies that w is a proper descendant of v in T. □

Lemma 2 implies that any depth-first spanning tree T divides the edges of G into two classes: *tree* edges, which appear in T, and *back* edges, which connect some node in T to one of its ancestors.

19.1 Counting and Labeling Components

For graphs that might be disconnected, we can compute a depth-first spanning *forest* by calling the following wrapper function; again, we introduce a generic black-box subroutine PREPROCESS to perform any necessary preprocessing for the POSTVISIT and POSTVISIT functions.

```
DFSALL(G):
  PREPROCESS(G)
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          DFS(v)
```

With very little additional effort, we can count the components of a graph; we simply increment a counter inside the wrapper function. Moreover, we can also record which component contains each vertex in the graph by passing this counter to DFS. The single line $comp(v) \leftarrow count$ is a trivial example of PREVISIT. (And the absence of code after the for loop is a vacuous example of POSTVISIT.)

```
COUNTANDLABEL(G):
  count ← 0
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          count ← count + 1
          LABELCOMPONENT(v, count)
  return count
```

```
LABELCOMPONENT(v, count):
  mark v
  comp(v) ← count
  for each edge vw
      if w is unmarked
          LABELCOMPONENT(w, count)
```

It should be emphasized that depth-first search is not specifically required here; any other instantiation of our earlier generic traversal algorithm ("whatever-first search") can be used to count components in the same asymptotic running time. However, most of the other algorithms we consider in this note *do* specifically require *depth*-first search.

19.2 Preorder and Postorder Labeling

You should already be familiar with preorder and postorder traversals of rooted trees, both of which can be computed using from depth-first search. Similar traversal orders can be defined for arbitrary graphs by passing around a counter as follows:

```
PREPOSTLABEL(G):
  for all vertices v
      unmark v
  clock ← 0
  for all vertices v
      if v is unmarked
          clock ← LABELCOMPONENT(v, clock)
```

```
LABELCOMPONENT(v, clock):
  mark v
  pre(v) ← clock
  clock ← clock + 1
  for each edge vw
      if w is unmarked
          clock ← LABELCOMPONENT(w, clock)
  post(v) ← clock
  clock ← clock + 1
  return clock
```

Equivalently, if we're willing to use (shudder) global variables, we can use our generic depth-first-search algorithm with the following subroutines PREPROCESS, PREVISIT, and POSTVISIT.

```
PREPROCESS(G):
  clock ← 0
```

```
PREVISIT(v):
  pre(v) ← clock
  clock ← clock + 1
```

```
POSTVISIT(v):
  post(v) ← clock
  clock ← clock + 1
```

Consider two vertices u and v, where u is marked after v. Then we must have $pre(u) < pre(v)$. Moreover, Lemma 1 implies that if v is a descendant of u, then $post(u) > post(v)$, and otherwise, $pre(v) > post(u)$. Thus, for any two vertices u and v, the intervals $[pre(u), post(u)]$ and $[pre(v), post(v)]$ are either disjoint or nested; in particular, if uv is an edge, Lemma 2 implies that the intervals must be nested.

19.3 Directed Graphs and Reachability

The recursive algorithm requires only one minor change to handle directed graphs:

```
DFSALL(G):
  for all vertices v
      unmark v
  for all vertices v
      if v is unmarked
          DFS(v)
```

```
DFS(v):
  mark v
  PREVISIT(v)
  for each edge v→w
      if w is unmarked
          DFS(w)
  POSTVISIT(v)
```

However, we can no longer use this modified algorithm to count components. Suppose G is a single directed path. Depending on the order that we choose to visit the nodes in DFSALL, we may discover any number of "components" between 1 and n. All that we can guarantee is that the "component" numbers computed by DFSALL do not increase as we traverse the path. In fact, the real problem is that the *definition* of "component" is only suitable for *undirected* graphs.

Instead, for directed graphs we rely on a more subtle notion of ***reachability***. We say that a node v is *reachable* from another node u in a directed graph G—or more simply, that u can reach v—if and only if there is a directed path in G from u to v. Let ***Reach(u)*** denote the set of vertices that are reachable from u (including u itself). A simple inductive argument proves that $Reach(u)$ is precisely the subset of nodes that are marked by calling DFS(u).

19.4 Directed Acyclic Graphs

A ***directed acyclic graph*** or ***dag*** is a directed graph with no directed cycles. Any vertex in a dag that has no incoming vertices is called a ***source***; any vertex with no outgoing edges is called a ***sink***. Every

dag has at least one source and one sink (Do you see why?), but may have more than one of each. For example, in the graph with n vertices but no edges, every vertex is a source and every vertex is a sink.

We can check whether a given directed graph G is a dag in $O(V+E)$ time as follows. First, to simplify the algorithm, we add a single artificial source s, with edges from s to every other vertex. Let $G+s$ denote the resulting augmented graph. Because s has no outgoing edges, no directed cycle in $G+s$ goes through s, which implies that $G+s$ is a dag if and only if G is a dag. Then we preform a depth-first search of $G+s$ starting at the new source vertex s; by construction every other vertex is reachable from s, so this search visits every node in the graph.

Instead of vertices being merely marked or unmarked, each vertex has one of three statuses—NEW, ACTIVE, or DONE—which depend on whether we have started or finished the recursive depth-first search at that vertex. (Since this algorithm never uses parent pointers, I've removed the line "$parent(w) \leftarrow v$".)

```
IsAcyclic(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      status(v) ← NEW
  return IsAcyclicDFS(s)
```

```
IsAcyclicDFS(v):
  status(v) ← ACTIVE
  for each edge v→w
      if status(w) = ACTIVE
          return FALSE
      else if status(w) = NEW
          if IsAcyclicDFS(w) = FALSE
              return FALSE
  status(v) ← DONE
  return TRUE
```

Suppose the algorithm returns FALSE. Then the algorithm must discover an edge $v \to w$ such that $status(w) = \text{ACTIVE}$. The active vertices are precisely the vertices currently on the recursion stack, which are all ancestors of the current vertex v. Thus, there is a directed path from w to v, and so the graph has a directed cycle.

On the other hand, suppose G has a directed cycle. Let w be the first vertex in this cycle that we visit, and let $v \to w$ be the edge leading into v in the same cycle. Because there is a directed path from w to v, we must call IsAcyclicDFS(v) during the execution of IsAcyclicDFS(w), unless we discover some other cycle first. During the execution of IsAcyclicDFS(v), we consider the edge $v \to w$, discover that $status(w) = \text{ACTIVE}$. The return value FALSE bubbles up through all the recursive calls to the top level.

We conclude that IsAcyclic(G) returns TRUE if and only if G is a dag.

19.5 Topological Sort

A ***topological ordering*** of a directed graph G is a total order $\prec$ on the vertices such that $u \prec v$ for every edge $u \to v$. Less formally, a topological ordering arranges the vertices along a horizontal line so that all edges point from left to right. A topological ordering is clearly impossible if the graph G has a directed cycle—the rightmost vertex of the cycle would have an edge pointing to the left! On the other hand, every dag has a topological order, which can be computed by either of the following algorithms.

```
TopologicalSort(G):
  n ← |V|
  for i ← 1 to n
      v ← any source in G
      S[i] ← v
      delete v and all edges leaving v
  return S[1..n]
```

```
TopologicalSort(G):
  n ← |V|
  for i ← n down to 1
      v ← any sink in G
      S[1] ← v
      delete v and all edges entering v
  return S[1..n]
```

The correctness of these algorithms follow inductively from the observation that *deleting* a vertex cannot *create* a cycle.

This simple algorithm has two major disadvantages. First, the algorithm actually destroys the input graph. This destruction can be avoided by simply marking the "deleted" vertices, instead of actually deleting them, and defining a vertex to be a source (sink) if none of its incoming (outgoing) edges come from (lead to) an unmarked vertex. The more serious problem is that finding a source vertex seems to require $\Theta(V)$ time in the worst case, which makes the running time of this algorithm $\Theta(V^2)$. In fact, a careful implementation of this algorithm computes a topological ordering in $O(V + E)$ time without removing any edges.

But there is a simpler linear-time algorithm based on our earlier algorithm for deciding whether a directed graph is acyclic. The new algorithm is based on the following observation:

Lemma 3. *For any directed acyclic graph G, the first vertex marked* DONE *by* ISACYCLIC(G) *must be a sink.*

Proof: Let v be the first vertex marked DONE during an execution of ISACYCLIC. For the sake of argument, suppose v has an outgoing edge $v \to w$. When ISACYCLICDFS first considers the edge $v \to w$, there are three cases to consider.

- If $status(w) =$ DONE, then w is marked DONE before v, which contradicts the definition of v.
- If $status(w) =$ NEW, the algorithm calls TOPOSORTDFS(w), which (among other computation) marks w DONE. Thus, w is marked DONE before v, which contradicts the definition of v.
- If $status(w) =$ ACTIVE, then G has a directed cycle, contradicting our assumption that G is acyclic.

In all three cases, we have a contradiction, so v must be a sink. □

Thus, to topologically sort a dag G, it suffice to list the vertices in the *reverse* order of being marked DONE. For example, we could push each vertex onto a stack when we mark it DONE, and then pop every vertex off the stack.

```
TOPOLOGICALSORT(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      status(v) ← NEW
  TOPOSORTDFS(s)
  for i ← 1 to V
      S[i] ← POP
  return S[1..V]
```

```
TOPOSORTDFS(v):
  status(v) ← ACTIVE
  for each edge v→w
      if status(w) = NEW
          PROCESSBACKWARDDFS(w)
      else if status(w) = ACTIVE
          fail gracefully
  status(v) ← DONE
  PUSH(v)
  return TRUE
```

But maintaining a separate data structure is actually overkill. In most applications of topological sort, an explicit sorted list of the vertices is not our actual goal; instead, we want to performing some fixed computation at each vertex of the graph, either in topological order or in reverse topological order. In this case, it is not necessary to *record* the topological order. To process the graph in *reverse* topological order, we can just process each vertex at the end of its recursive depth-first search.

```
ProcessBackward(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      status(v) ← New
  ProcessBackwardDFS(s)
```

```
ProcessBackwardDFS(v):
  status(v) ← Active
  for each edge v→w
      if status(w) = New
          ProcessBackwardDFS(w)
      else if status(w) = Active
          fail gracefully
  status(v) ← Done
  Process(v)
```

If we already *know* that the input graph is acyclic, we can simplify the algorithm by simply marking vertices instead of labeling them Active or Done.

```
ProcessDagBackward(G):
  add vertex s
  for all vertices v ≠ s
      add edge s→v
      unmark v
  ProcessDagBackwardDFS(s)
```

```
ProcessDagBackwardDFS(v):
  mark v
  for each edge v→w
      if w is unmarked
          ProcessDagBackwardDFS(w)
  Process(v)
```

Except for the addition of the artificial source vertex s, which we need to ensure that every vertex is visited, this is just the standard depth-first search algorithm, with PostVisit renamed to Process!

The simplest way to process a dag in *forward* topological order is to construct the ***reversal*** of the input graph, which is obtained by replacing each each $v \to w$ with its reversal $w \to v$. Reversing a directed cycle gives us another directed cycle with the opposite orientation, so the reversal of a dag is another dag. Every source in G becomes a sink in the reversal of G and vice versa; it follows inductively that every topological ordering for the reversal of G is the reversal of a topological ordering of G. The reversal of any directed graph can be computed in $O(V + E)$ time; the details of this construction are left as an easy exercise.

19.6 Every Dynamic Programming Algorithm?

Our topological sort algorithm is arguably the model fora wide class of dynamic programming algorithms. Recall that the ***dependency graph*** of a recurrence has a vertex for every recursive subproblem and an edge from one subproblem to another if evaluating the first subproblem requires a recursive evaluation of the second. The dependency graph must be acyclic, or the naïve recursive algorithm would never halt. Evaluating any recurrence with memoization is exactly the same as performing a depth-first search of the dependency graph. In particular, a vertex of the dependency graph is 'marked' if the value of the corresponding subproblem has already been computed, and the black-box subroutine Process is a placeholder for the actual value computation.

However, there are some minor differences between most dynamic programming algorithms and topological sort.

- First, in most dynamic programming algorithms, the dependency graph is *implicit*—the nodes and edges are not given as part of the input. But this difference really is minor; as long as we can enumerate recursive subproblems in constant time each, we can traverse the dependency graph exactly as if it were explicitly stored in an adjacency list.

- More significantly, most dynamic programming recurrences have highly structured dependency graphs. For example, the dependency graph for edit distance is a regular grid with diagonals, and the dependency graph for optimal binary search trees is an upper triangular grid with all possible

rightward and upward edges. This regular structure lets us hard-wire a topological order directly into the algorithm, so we don't have to compute it at run time.

Conversely, we can use depth-first search to build dynamic programming algorithms for problems with less structured dependency graphs. For example, consider the ***longest path*** problem, which asks for the path of *maximum* total weight from one node s to another node t in a directed graph G with weighted edges. The longest path problem is NP-hard in general directed graphs, by an easy reduction from the traveling salesman problem, but it is easy to solve in linear time if the input graph G is acyclic, as follows. For any node s, let $LLP(s,t)$ denote the Length of the Longest Path in G from s to t. If G is a dag, this function satisfies the recurrence

$$LLP(s,t) = \begin{cases} 0 & \text{if } s = t, \\ \max_{s\to v} (\ell(s\to v) + LLP(v,t)) & \text{otherwise,} \end{cases}$$

where $\ell(v\to w)$ is the given weight ("length") of edge $v\to w$. In particular, if s is a *sink* but not equal to t, then $LLP(s,t) = \infty$. The dependency graph for this recurrence is the input graph G itself: subproblem $LLP(u,t)$ depends on subproblem $LLP(v,t)$ if and only if $u\to v$ is an edge in G. Thus, we can evaluate this recursive function in $O(V+E)$ time by performing a depth-first search of G, starting at s.

```
LongestPath(s, t):
  if s = t
      return 0

  if LLP(s) is undefined
      LLP(s) ← ∞
      for each edge s→v
          LLP(s) ← max {LLP(v), ℓ(s→v) + LongestPath(v, t)}
  return LLP(s)
```

A surprisingly large number of dynamic programming problems (but *not* all) can be recast as optimal path problems in the associated dependency graph.

19.7 Strong Connectivity

Let's go back to the proper definition of connectivity in directed graphs. Recall that one vertex u can *reach* another vertex v in a graph G if there is a directed path in G from u to v, and that $Reach(u)$ denotes the set of all vertices that u can reach. Two vertices u and v are ***strongly connected*** if u can reach v and v can reach u. Tedious definition-chasing implies that strong connectivity is an equivalence relation over the set of vertices of any directed graph, just as connectivity is for undirected graphs. The equivalence classes of this relation are called the ***strongly connected components*** (or more simply, the ***strong components***) of G. If G has a single strong component, we call it ***strongly connected***. G is a directed acyclic graph if and only if every strong component of G is a single vertex.

It is straightforward to compute the strong component containing a single vertex v in $O(V+E)$ time. First we compute $Reach(v)$ by calling $DFS(v)$. Then we compute $Reach^{-1}(v) = \{u \mid v \in Reach(u)\}$ by searching the reversal of G. Finally, the strong component of v is the intersection $Reach(v) \cap Reach^{-1}(v)$. In particular, we can determine whether the entire graph is strongly connected in $O(V+E)$ time.

We can compute *all* the strong components in a directed graph by wrapping the single-strong-component algorithm in a wrapper function, just as we did for depth-first search in undirected graphs. However, the resulting algorithm runs in $O(VE)$ time; there are at most V strong components, and each requires $O(E)$ time to discover. Surely we can do better! After all, we only need $O(V+E)$ time to decide whether every strong component is a single vertex.

19.8 Strong Components in Linear Time

For any directed graph G, the ***strong component graph scc(G)*** is another directed graph obtained by contracting each strong component of G to a single (meta-)vertex and collapsing parallel edges. The strong component graph is sometimes also called the *meta-graph* or *condensation* of G. It's not hard to prove (hint, hint) that $scc(G)$ is always a dag. Thus, in principle, it is possible to topologically order the strong components of G; that is, the vertices can be ordered so that every *backward* edge joins two edges in the same strong component.

Let C be any strong component of G that is a sink in $scc(G)$; we call C a *sink component*. Every vertex in C can reach every other vertex in C, so a depth-first search from any vertex in C visits every vertex in C. On the other hand, because C is a sink component, there is no edge from C to any other strong component, so a depth-first search starting in C visits *only* vertices in C. So if we can compute all the strong components as follows:

```
StrongComponents(G):
  count ← 0
  while G is non-empty
      count ← count + 1
      v ← any vertex in a sink component of G
      C ← OneComponent(v, count)
      remove C and incoming edges from G
```

At first glance, finding a vertex in a sink component *quickly* seems quite hard. However, we can quickly find a vertex in a *source* component using the standard depth-first search. A source component is a strong component of G that corresponds to a source in $scc(G)$. Specifically, we compute *finishing times* (otherwise known as post-order labeling) for the vertices of G as follows.

```
DFSAll(G):
  for all vertices v
      unmark v
  clock ← 0
  for all vertices v
      if v is unmarked
          clock ← DFS(v, clock)
```

```
DFS(v, clock):
  mark v
  for each edge v→w
      if w is unmarked
          clock ← DFS(w, clock)
  clock ← clock + 1
  finish(v) ← clock
  return clock
```

Lemma 4. *The vertex with largest finishing time lies in a source component of G.*

Proof: Let v be the vertex with largest finishing time. Then DFS$(v, clock)$ must be the last direct call to DFS made by the wrapper algorithm DFSALL.

Let C be the strong component of G that contains v. For the sake of argument, suppose there is an edge $x \to y$ such that $x \notin C$ and $y \in C$. Because v and y are strongly connected, y can reach v, and therefore x can reach v. There are two cases to consider.

- If x is already marked when DFS(v) begins, then v must have been marked during the execution of DFS(x), because x can reach v. But then v was already marked when DFS(v) was called, which is impossible.
- If x is not marked when DFS(v) begins, then x must be marked during the execution of DFS(v), which implies that v can reach x. Since x can also reach v, we must have $x \in C$, contradicting the definition of x.

We conclude that C is a source component of G. □

Essentially the same argument implies the following more general result.

Lemma 5. *For any edge $v \to w$ in G, if $finish(v) < finish(w)$, then v and w are strongly connected in G.*

Proof: Let $v \to w$ be an arbitrary edge of G. There are three cases to consider.If w is unmarked when DFS(v) begins, then the recursive call to DFS(w) finishes w, which implies that $finish(w) < finish(v)$. If w is still active when DFS(v) begins, there must be a path from w to v, which implies that v and w are strongly connected. Finally, if w is finished when DFS(v) begins, then clearly $finish(w) < finish(v)$. □

This observation is consistent with our earlier topological sorting algorithm; for *every* edge $v \to w$ in a directed acyclic graph, we have $finish(v) > finish(w)$.

It is easy to check (hint, hint) that any directed G has exactly the same strong components as its reversal $rev(G)$; in fact, we have $rev(scc(G)) = scc(rev(G))$. Thus, if we order the vertices of G by their finishing times in DFSALL($rev(G)$), the *last* vertex in this order lies in a sink component of G. Thus, if we run DFSALL(G), visiting vertices in reverse order of their finishing times in DFSALL($rev(G)$), then each call to DFS visits exactly one strong component of G.

Putting everything together, we obtain the following algorithm to count and label the strong components of a directed graph in $O(V+E)$ time, first discovered (but never published) by Rao Kosaraju in 1978, and then independently rediscovered by Micha Sharir in 1981. The Kosaraju-Sharir algorithm has two phases. The first phase performs a depth-first search of the reversal of G, pushing each vertex onto a stack when it is finished. In the second phase, we perform another depth-first search of the original graph G, considering vertices in the order they appear on the stack.

```
KosarajuSharir(G):
  ⟨⟨Phase 1: Push in finishing order⟩⟩
  unmark all vertices
  for all vertices v
      if v is unmarked
          clock ← RevPushDFS(v)
  ⟨⟨Phase 2: DFS in stack order⟩⟩
  unmark all vertices
  count ← 0
  while the stack is non-empty
      v ← Pop
      if v is unmarked
          count ← count + 1
          LabelOneDFS(v, count)
```

```
RevPushDFS(v):
  mark v
  for each edge v→u in rev(G)
      if u is unmarked
          RevPushDFS(u)
  Push(v)
```

```
LabelOneDFS(v, count):
  mark v
  label(v) ← count
  for each edge v→w in G
      if w is unmarked
          LabelOneDFS(w, count)
```

With further minor modifications, we can also compute the strongly connected component graph $scc(G)$ in $O(V+E)$ time.

Exercises

0. (a) Describe an algorithm to compute the reversal $rev(G)$ of a directed graph in $O(V+E)$ time.
   (b) Prove that for any directed graph G, the strong component graph $scc(G)$ is a dag.
   (c) Prove that for any directed graph G, we have $scc(rev(G)) = rev(scc(G))$.
   (d) Suppose S and T are two strongly connected components in a directed graph G. Prove that $finish(u) < finish(v)$ for all vertices $u \in S$ and $v \in T$.

1. One of the oldest[1] algorithms for exploring graphs was proposed by Gaston Tarry in 1895. The input to Tarry's algorithm is a directed graph G that contains both directions of every edge; that is, for every edge $u \to v$ in G, its reversal $v \to u$ is also an edge in G.

```
TARRY(G):
  unmark all vertices of G
  color all edges of G white
  s ← any vertex in G
  RECTARRY(s)
```

```
RECTARRY(v):
  mark v                                  ⟨⟨"visit v"⟩⟩
  if there is a white arc v→w
      if w is unmarked
          color w→v green
      color v→w red                       } ⟨⟨"traverse v→w"⟩⟩
      RECTARRY(w)
  else if there is a green arc v→w
      color v→w red                       } ⟨⟨"traverse v→w"⟩⟩
      RECTARRY(w)
```

We informally say that Tarry's algorithm "visits" vertex v every time it marks v, and it "traverses" edge $v \to w$ when it colors that edge red and recursively calls RECTARRY(w).

   (a) Describe how to implement Tarry's algorithm so that it runs in $O(V + E)$ time.

   (b) Prove that no directed edge in G is traversed more than once.

   (c) When the algorithm visits a vertex v for the kth time, exactly how many edges into v are red, and exactly how many edges out of v are red? *[Hint: Consider the starting vertex s separately from the other vertices.]*

   (d) Prove each vertex v is visited at most $\deg(v)$ times, except the starting vertex s, which is visited at most $\deg(s) + 1$ times. This claim immediately implies that TARRY(G) terminates.

   (e) Prove that when TARRY(G) ends, the last visited vertex is the starting vertex s.

   (f) For every vertex v that TARRY(G) visits, prove that all edges into v and out of v are red when TARRY(G) halts. *[Hint: Consider the vertices in the order that they are marked for the first time, starting with s, and prove the claim by induction.]*

   (g) Prove that TARRY(G) visits every vertex of G. This claim and the previous claim imply that TARRY(G) traverses every edge of G exactly once.

2. Consider the following variant of Tarry's graph-traversal algorithm; this variant traverses green edges without recoloring them red and assigns two numerical labels to every vertex:

```
TARRY2(G):
  unmark all vertices of G
  color all edges of G white
  s ← any vertex in G
  RECTARRY(s, 1)
```

```
RECTARRY2(v, clock):
  if v is unmarked
      pre(v) ← clock;  clock ← clock + 1
      mark v
  if there is a white arc v→w
      if w is unmarked
          color w→v green
      color v→w red
      RECTARRY2(w, clock)
  else if there is a green arc v→w
      post(v) ← clock;  clock ← clock + 1
      RECTARRY2(w, clock)
```

[1] Even older graph-traversal algorithms were described by Charles Trémaux in 1882, Christian Wiener in 1873, and (implicitly) Leonhard Euler in 1736. Wiener's algorithm is equivalent to depth-first search in a connected undirected graph.

Prove or disprove the following claim: When TARRY2(G) halts, the green edges define a spanning tree and the labels $pre(v)$ and $post(v)$ define a preorder and postorder labeling that are all consistent with a single depth-first search of G. In other words, prove or disprove that TARRY2 produces the same output as depth-first search.

3. Let $G = (V, E)$ be a given directed graph.

   (a) The *transitive closure* G^T is a directed graph with the same vertices as G, that contains any edge $u \to v$ if and only if there is a directed path from u to v in G. Describe an efficient algorithm to compute the transitive closure of G.

   (b) A *transitive reduction* G^{TR} is a graph with the smallest possible number of edges whose transitive closure is G^T. (The same graph may have several transitive reductions.) Describe an efficient algorithm to compute the transitive reduction of G.

4. For any two nodes u and v in a directed acyclic graph G, the ***interval*** $\boldsymbol{G[u, v]}$ is the union of all directed paths in G from u to v. Equivalently, $G[u, v]$ consists of all vertices x such that $x \in Reach(u)$ and $v \in Reach(x)$, together with all the edges in G connecting those vertices.

   Suppose we are given a directed acyclic graph G, in which every edge has a numerical weight, which may be positive, negative, or zero. Describe an efficient algorithm to find the maximum-weight interval in G, where the weight of any interval is the sum of the weights of its vertices. *[Hint: Don't try to be clever.]*

5. Let G be a directed acyclic graph with a unique source s and a unique sink t.

   (a) A *Hamiltonian path* in G is a directed path in G that contains every vertex in G. Describe an algorithm to determine whether G has a Hamiltonian path.

   (b) Suppose the *vertices* of G have weights. Describe an efficient algorithm to find the path from s to t with maximum total weight.

   (c) Suppose we are also given an integer ℓ. Describe an efficient algorithm to find the maximum-weight path from s to t, such that the path contains at most ℓ edges. (Assume there is at least one such path.)

   (d) Suppose the vertices of G have integer labels, where $label(s) = -\infty$ and $label(t) = \infty$. Describe an algorithm to find the path from s to t with the maximum number of edges, such that the vertex labels define an increasing sequence.

   (e) Describe an algorithm to compute the number of distinct paths from s to t in G. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

6. Let G and H be directed acyclic graphs, whose vertices have labels from some fixed alphabet, and let $A[1..\ell]$ be a string over the same alphabet. Any directed path in G has a label, which is a string obtained by concatenating the labels of its vertices.

   (a) Describe an algorithm that either finds a path in G whose label is A or correctly reports that there is no such path.

   (b) Describe an algorithm to find the *number* of paths in G whose label is A. (Assume that you can add arbitrarily large integers in $O(1)$ time.)

(c) Describe an algorithm to find the longest path in G whose label is a subsequence of A.

(d) Describe an algorithm to find the *shortest* path in G whose label is a *supersequence* of A.

(e) Describe an algorithm to find a path in G whose label has minimum edit distance from A.

(f) Describe an algorithm to find the longest string that is both a label of a directed path in G and the label of a directed path in H.

(g) Describe an algorithm to find the longest string that is both a *subsequence* of the label of a directed path in G and a *subsequence* of the label of a directed path in H.

(h) Describe an algorithm to find the shortest string that is both a *supersequence* of the label of a directed path in G and a *supersequence* of the label of a directed path in H.

(i) Describe an algorithm to find the longest path in G whose label is a palindrome.

(j) Describe an algorithm to find the longest palindrome that is a subsequence of the label of a path in G.

(k) Describe an algorithm to find the shortest palindrome that is a supersequence of the label of a path in G.

7. Suppose two players are playing a turn-based game on a directed acyclic graph G with a unique source s. Each vertex v of G is labeled with a real number $\ell(v)$, which could be positive, negative, or zero. The game starts with three tokens at s. In each turn, the current player moves one of the tokens along a directed edge from its current node to another node, and the current player's score is increased by $\ell(u) \cdot \ell(v)$, where u and v are the locations of the two tokens that did *not* move. At most one token is allowed on any node except s at any time. The game ends when the current player is unable to move (for example, when all three tokens lie on sinks); at that point, the player with the higher score is the winner.

   Describe an efficient algorithm to determine who wins this game on a given labeled graph, assuming both players play optimally.

*8. Let $x = x_1 x_2 \ldots x_n$ be a given n-character string over some finite alphabet Σ, and let A be a deterministic finite-state machine with m states over the same alphabet.

(a) Describe and analyze an algorithm to compute the longest subsequence of x that is accepted by A. For example, if A accepts the language $(\texttt{ar})^*$ and $x = \texttt{abracadabra}$, your algorithm should output `arar`.

(b) Describe and analyze an algorithm to compute the shortest (not necessarily contiguous) supersequence of x that is accepted by A. For example, if A accepts the language $(\texttt{abcdr})^*$ and $x = \texttt{abracadabra}$, your algorithm should output **ab**cd**ra**bcdra**b**c**d**r**ab**cd**ra**bcdr.

9. Not every dynamic programming algorithm can be modeled as finding an optimal path through a directed acyclic graph; the most obvious counterexample is the optimal binary search tree problem. But every dynamic programming problem does traverse a dependency graph in reverse topological order, performing some additional computation at every vertex.

(a) Suppose we are given a directed acyclic graph G where every node stores a numerical search key. Describe and analyze an algorithm to find the largest binary search tree that is a subgraph of G.

(b) Let G be a directed acyclic graph with the following features:

- G has a single source s and several sinks $t_1, t_2, \ldots, t_k$.
- Each edge $v \to w$ has an associated numerical value $p(v \to w)$ between 0 and 1.
- For each non-sink vertex v, we have $\sum_w p(v \to w) = 1$.

The values $p(v \to w)$ define a random walk in G from the source s to some sink t_i; after reaching any non-sink vertex v, the walk follows edge $v \to w$ with probability $p(v \to w)$. Describe and analyze an algorithm to compute the probability that this random walk reaches sink t_i, for every index i. (Assume that any arithmetic operation requires $O(1)$ time.)

We must all hang together, gentlemen, or else we shall most assuredly hang separately.

— Benjamin Franklin, at the signing of the Declaration of Independence (July 4, 1776)

It is a very sad thing that nowadays there is so little useless information.

— Oscar Wilde

A ship in port is safe, but that is not what ships are for.

— Rear Admiral Grace Murray Hopper

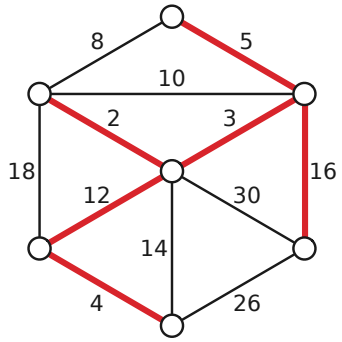
25 Minimum Spanning Trees

25.1 Introduction

Suppose we are given a connected, undirected, *weighted* graph. This is a graph $G = (V, E)$ together with a function $w : E \to \mathbb{R}$ that assigns a real *weight* $w(e)$ to each edge e, which may be positive, negative, or zero. Our task is to find the ***minimum spanning tree*** of G, that is, the spanning tree T that minimizes the function

$$w(T) = \sum_{e \in T} w(e).$$

To keep things simple, I'll assume that all the edge weights are distinct: $w(e) \neq w(e')$ for any pair of edges e and e'. Distinct weights guarantee that the minimum spanning tree of the graph is unique. Without this condition, there may be several different minimum spanning trees. For example, if all the edges have weight 1, then *every* spanning tree is a minimum spanning tree with weight $V - 1$.



A weighted graph and its minimum spanning tree.

If we have an algorithm that assumes the edge weights are unique, we can still use it on graphs where multiple edges have the same weight, as long as we have a consistent method for breaking ties. One way to break ties consistently is to use the following algorithm in place of a simple comparison. SHORTEREDGE takes as input four integers i, j, k, l, and decides which of the two edges (i, j) and (k, l) has "smaller" weight.

```
SHORTEREDGE(i,j,k,l)
  if w(i,j) < w(k,l)              then return (i,j)
  if w(i,j) > w(k,l)              then return (k,l)
  if min(i,j) < min(k,l)          then return (i,j)
  if min(i,j) > min(k,l)          then return (k,l)
  if max(i,j) < max(k,l)          then return (i,j)
  ⟨⟨if max(i,j) < max(k,l)⟩⟩       return (k,l)
```

25.2 The Only Minimum Spanning Tree Algorithm

There are several different methods for computing minimum spanning trees, but really they are all instances of the following generic algorithm. The situation is similar to the previous lecture, where we saw that depth-first search and breadth-first search were both instances of a single generic traversal algorithm.

The generic minimum spanning tree algorithm maintains an acyclic subgraph F of the input graph G, which we will call an *intermediate spanning forest*. F is a subgraph of the minimum spanning tree of G, and every component of F is a minimum spanning tree of its vertices. Initially, F consists of n one-node trees. The generic algorithm merges trees together by adding certain edges between them. When the algorithm halts, F consists of a single n-node tree, which must be the minimum spanning tree. Obviously, we have to be careful about *which* edges we add to the evolving forest, since not every edge is in the minimum spanning tree.

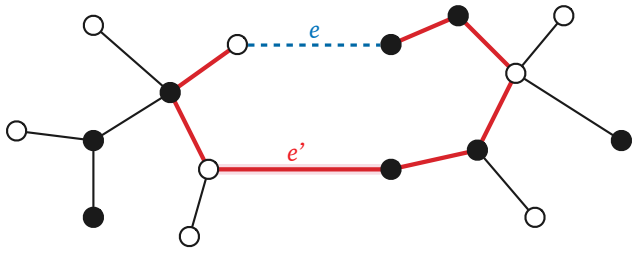
The intermediate spanning forest F induces two special types of edges. An edge is ***useless*** if it is not an edge of F, but both its endpoints are in the same component of F. For each component of F, we associate a ***safe*** edge—the minimum-weight edge with exactly one endpoint in that component. Different components might or might not have different safe edges. Some edges are neither safe nor useless—we call these edges *undecided*.

All minimum spanning tree algorithms are based on two simple observations.

Lemma 1. *The minimum spanning tree contains every safe edge.*

Proof: In fact we prove the following stronger statement: For *any* subset S of the vertices of G, the minimum spanning tree of G contains the minimum-weight edge with exactly one endpoint in S. We prove this claim using a greedy exchange argument.

Let S be an arbitrary subset of vertices of G; let e be the lightest edge with exactly one endpoint in S; and let T be an arbitrary spanning tree that does *not* contain e. Because T is connected, it contains a path from one endpoint of e to the other. Because this path starts at a vertex of S and ends at a vertex not in S, it must contain at least one edge with exactly one endpoint in S; let e' be *any* such edge. Because T is acyclic, removing e' from T yields a spanning *forest* with exactly two components, one containing each endpoint of e. Thus, adding e to this forest gives us a new spanning tree $T' = T - e' + e$. The definition of e implies $w(e') > w(e)$, which implies that T' has smaller total weight than T. We conclude that T is not the minimum spanning tree, which completes the proof. □



Proving that every safe edge is in the minimum spanning tree. Black vertices are in the subset S.

Lemma 2. *The minimum spanning tree contains no useless edge.*

Proof: Adding any useless edge to F would introduce a cycle. □

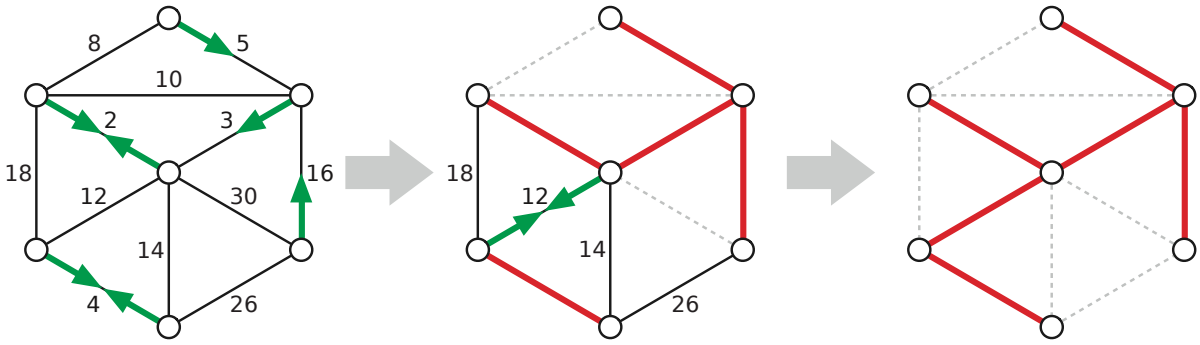
Our generic minimum spanning tree algorithm repeatedly adds one or more safe edges to the evolving forest F. Whenever we add new edges to F, some undecided edges become safe, and others become useless. To specify a particular algorithm, we must decide which safe edges to add, and we must describe how to identify new safe and new useless edges, at each iteration of our generic template.

25.3 Borůvka's Algorithm

The oldest and arguably simplest minimum spanning tree algorithm was discovered by Borůvka in 1926, long before computers even existed, and practically before the invention of graph theory![1] The algorithm was rediscovered by Choquet in 1938; again by Florek, Łukaziewicz, Perkal, Stienhaus, and Zubrzycki in 1951; and again by Sollin some time in the early 1960s. Because Sollin was the only Western computer scientist in this list—Choquet was a civil engineer; Florek and his co-authors were anthropologists—this is often called "Sollin's algorithm", especially in the parallel computing literature.

The Borůvka/Choquet/Florek/Łukaziewicz/Perkal/Stienhaus/Zubrzycki/Sollin algorithm can be summarized in one line:

BORŮVKA: Add ***ALL*** the safe edges[2] and recurse.



Borůvka's algorithm run on the example graph. Thick edges are in F.
Arrows point along each component's safe edge. Dashed (gray) edges are useless.

We can find all the safe edge in the graph in $O(E)$ time as follows. First, we count the components of F using whatever-first search, using the standard wrapper function. As we count, we label every vertex with its component number; that is, every vertex in the first traversed component gets label 1, every vertex in the second component gets label 2, and so on.

If F has only one component, we're done. Otherwise, we compute an array $S[1..V]$ of edges, where $S[i]$ is the minimum-weight edge with one endpoint in the ith component (or a sentinel value NULL if

[1] Leonard Euler published the first graph theory result, his famous theorem about the bridges of Königsburg, in 1736. However, the first textbook on graph theory, written by Dénes König, was not published until 1936.

[2] See also: Allie Brosh, "This is Why I'll Never be an Adult", *Hyperbole and a Half*, June 17, 2010. Actually, just go see everything in *Hyperbole and a Half*. And then go buy the book. And an extra copy for your cat.

there are less than i components). To compute this array, we consider each edge uv in the input graph G. If the endpoints u and v have the same label, then uv is useless. Otherwise, we compare the weight of uv to the weights of $S[label(u)]$ and $S[label(v)]$ and update the array entries if necessary.

```
BORŮVKA(V, E):
  F = (V, ∅)
  count ← COUNTANDLABEL(F)
  while count > 1
      ADDALLSAFEEDGES(E, F, count)
      count ← COUNTANDLABEL(F)
  return F
```

```
ADDALLSAFEEDGES(E, F, count):
  for i ← 1 to count
      S[i] ← NULL        ⟨⟨sentinel: w(NULL) := ∞⟩⟩
  for each edge uv ∈ E
      if label(u) ≠ label(v)
          if w(uv) < w(S[label(u)])
              S[label(u)] ← uv
          if w(uv) < w(S[label(v)])
              S[label(v)] ← uv
  for i ← 1 to count
      if S[i] ≠ NULL
          add S[i] to F
```

Each call to TRAVERSEALL requires $O(V)$ time, because the forest F has at most $V-1$ edges. Assuming the graph is represented by an adjacency list, the rest of each iteration of the main while loop requires $O(E)$ time, because we spend constant time on each edge. Because the graph is connected, we have $V \le E+1$, so each iteration of the while loop takes $O(E)$ time.

Each iteration reduces the number of components of F by at least a factor of two—the worst case occurs when the components coalesce in pairs. Since F initially has V components, the while loop iterates at most $O(\log V)$ times. Thus, the overall running time of Borůvka's algorithm is $\boldsymbol{O(E \log V)}$.

Despite its relatively obscure origin, early algorithms researchers were aware of Borůvka's algorithm, but dismissed it as being "too complicated"! As a result, despite its simplicity and efficiency, Borůvka's algorithm is rarely mentioned in algorithms and data structures textbooks. On the other hand, Borůvka's algorithm has several distinct advantages over other classical MST algorithms.

- Borůvka's algorithm often runs faster than the $O(E \log V)$ worst-case running time. In arbitrary graphs, the number of components in F can drop by significantly more than a factor of 2 in a single iteration, reducing the number of iterations below the worst-case $\lceil \log_2 V \rceil$. A slight reformulation of Borůvka's algorithm (actually closer to Borůvka's original presentation) actually runs in $O(E)$ time for a broad class of interesting graphs, including graphs that can be drawn in the plane without edge crossings. In contrast, the time analysis for the other two algorithms applies to *all* graphs.

- Borůvka's algorithm allows for significant parallelism; in each iteration, each component of F can be handled in a separate independent thread. This implicit parallelism allows for even faster performance on multicore or distributed systems. In contrast, the other two classical MST algorithms are intrinsically serial.

- There are several more recent minimum-spanning-tree algorithms that are faster even in the worst case than the classical algorithms described here. *All* of these faster algorithms are generalizations of Borůvka's algorithm.

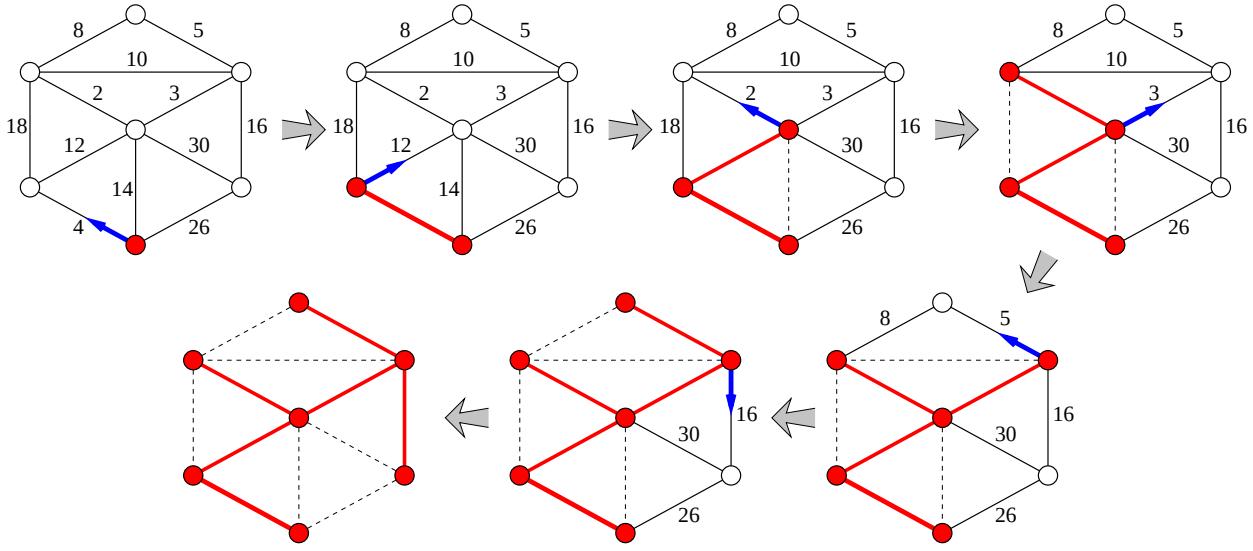
In short, if you ever need to implement a minimum-spanning-tree algorithm, use Borůvka. On the other hand, if you want to *prove things about* minimum spanning trees effectively, you really need to know the next two algorithms as well.

25.4 Jarník's ("Prim's") Algorithm

The next oldest minimum spanning tree algorithm was first described by the Czech mathematician Vojtěch Jarník in a 1929 letter to Borůvka; Jarník published his discovery the following year. The algorithm was independently rediscovered by Kruskal in 1956, by Prim in 1957, by Loberman and Weinberger in 1957, and finally by Dijkstra in 1958. Prim, Loberman, Weinberger, and Dijkstra all (eventually) knew of and even cited Kruskal's paper, but since Kruskal also described two other minimum-spanning-tree algorithms in the same paper, *this* algorithm is usually called "Prim's algorithm", or sometimes "the Prim/Dijkstra algorithm", even though by 1958 Dijkstra already had another algorithm (inappropriately) named after him.

In Jarník's algorithm, the forest F contains only one nontrivial component T; all the other components are isolated vertices. Initially, T consists of an arbitrary vertex of the graph. The algorithm repeats the following step until T spans the whole graph:

> JARNÍK: Repeatedly add T's safe edge to T.



Jarník's algorithm run on the example graph, starting with the bottom vertex.
At each stage, thick edges are in T, an arrow points along T's safe edge, and dashed edges are useless.

To implement Jarník's algorithm, we keep all the edges adjacent to T in a priority queue. When we pull the minimum-weight edge out of the priority queue, we first check whether both of its endpoints are in T. If not, we add the edge to T and then add the new neighboring edges to the priority queue. In other words, Jarník's algorithm is another instance of the generic graph traversal algorithm we saw last time, using a priority queue as the "bag"! If we implement the algorithm this way, the algorithm runs in $O(E \log E) = \boldsymbol{O(E \log V)}$ time.

*25.5 Improving Jarník's Algorithm

We can improve Jarník's algorithm using a more advanced priority queue data structure called a *Fibonacci heap*, first described by Michael Fredman and Robert Tarjan in 1984. Fibonacci heaps support the standard priority queue operations INSERT, EXTRACTMIN, and DECREASEKEY. However, unlike standard binary heaps, which require $O(\log n)$ time for every operation, Fibonacci heaps support INSERT and DECREASEKEY in constant *amortized* time. The amortized cost of EXTRACTMIN is still $O(\log n)$.

To apply this faster data structure, we keep *vertices* in the priority queue instead of edge, where the key for each vertex v is either the minimum-weight edge between v and the evolving tree T, or ∞ if there is no such edge. We can INSERT all the vertices into the priority queue at the beginning of the algorithm; then, whenever we add a new edge to T, we may need to decrease the keys of some neighboring vertices.

To make the description easier, we break the algorithm into two parts. JARNÍK INIT initializes the priority queue; JARNÍK LOOP is the main algorithm. The input consists of the vertices and edges of the graph, plus the start vertex s. For each vertex v, we maintain both its key $key(v)$ and the incident edge $edge(v)$ such that $w(edge(v)) = key(v)$.

```
JARNÍK(V, E, s):
  JARNÍK INIT(V, E, s)
  JARNÍK LOOP(V, E, s)
```

```
JARNÍK INIT(V, E, s):
  for each vertex v ∈ V \ {s}
      if (v, s) ∈ E
          edge(v) ← (v, s)
          key(v) ← w(v, s)
      else
          edge(v) ← NULL
          key(v) ← ∞
      INSERT(v)
```

```
JARNÍK LOOP(V, E, s):
  T ← ({s}, ∅)
  for i ← 1 to |V| − 1
      v ← EXTRACTMIN
      add v and edge(v) to T
      for each neighbor u of v
          if u ∉ T and key(u) > w(uv)
              edge(u) ← uv
              DECREASEKEY(u, w(uv))
```

The operations INSERT and EXTRACTMIN are each called $O(V)$ times once for each vertex except s, and DECREASEKEY is called $O(E)$ times, at most twice for each edge. Thus, if we use a Fibonacci heap, the improved algorithm runs in $\boldsymbol{O(E + V \log V)}$ ***time***, which is faster than Borůvka's algorithm unless $E = O(V)$.

In practice, however, this improvement is rarely faster than the naive implementation using a binary heap, unless the graph is extremely large and dense. The Fibonacci heap algorithms are quite complex, and the hidden constants in both the running time and space are significant—not outrageous, but certainly bigger than the hidden constant 1 in the $O(\log n)$ time bound for binary heap operations.

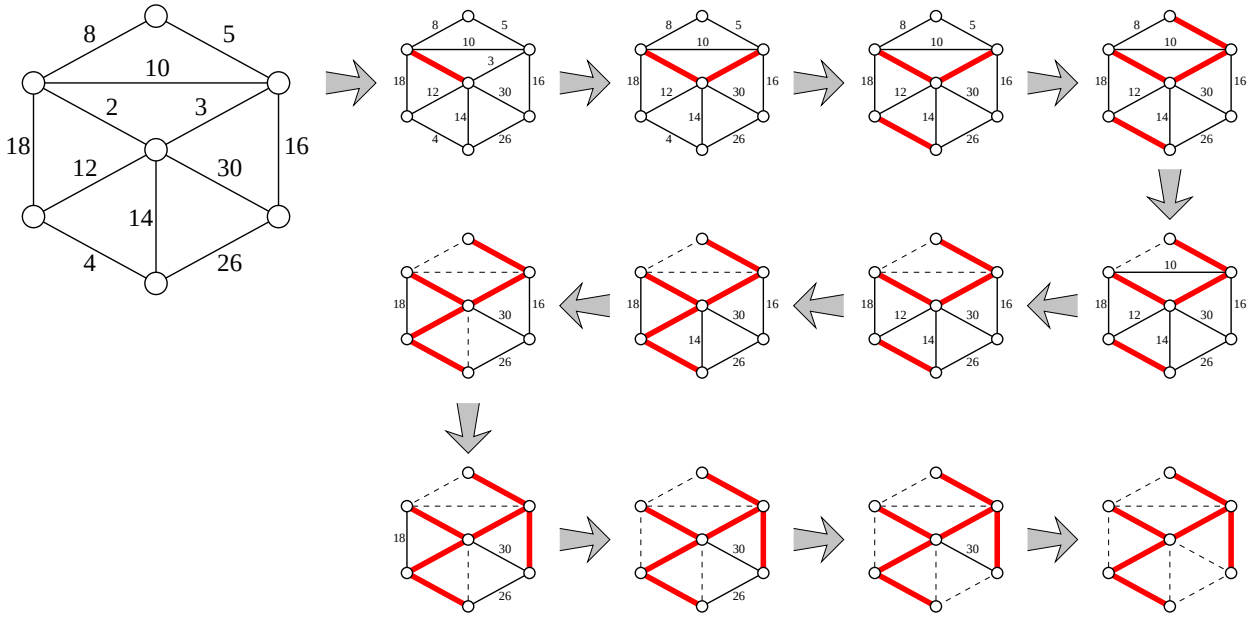
25.6 Kruskal's Algorithm

The last minimum spanning tree algorithm I'll discuss was first described by Kruskal in 1956, in the same paper where he rediscovered Jarnik's algorithm. Kruskal was motivated by "a typewritten translation (of obscure origin)" of Borůvka's original paper, claiming that Borůvka's algorithm was "unnecessarily elaborate".[3] This algorithm was also rediscovered in 1957 by Loberman and Weinberger, but somehow avoided being renamed after them.

KRUSKAL: Scan all edges in increasing weight order; if an edge is safe, add it to F.

Since we examine the edges in order from lightest to heaviest, any edge we examine is safe if and only if its endpoints are in different components of the forest F. To prove this, suppose the edge e joins two components A and B but is not safe. Then there would be a lighter edge e' with exactly one endpoint

[3]To be fair, Borůvka's original paper *was* unnecessarily elaborate, but in his followup paper, also published in 1927, simplified his algorithm essentially to its current modern form. Kruskal was apparently unaware of Borůvka's second paper. Stupid Iron Curtain.



Kruskal's algorithm run on the example graph. Thick edges are in F. Dashed edges are useless.

in A. But this is impossible, because (inductively) any previously examined edge has both endpoints in the same component of F.

Just as in Borůvka's algorithm, each component of F has a "leader" node. An edge joins two components of F if and only if the two endpoints have different leaders. But unlike Borůvka's algorithm, we do not recompute leaders from scratch every time we add an edge. Instead, when two components are joined, the two leaders duke it out in a nationally-televised no-holds-barred steel-cage grudge match.[4] One of the two emerges victorious as the leader of the new larger component. More formally, we will use our earlier algorithms for the UNION-FIND problem, where the vertices are the elements and the components of F are the sets. Here's a more formal description of the algorithm:

```
KRUSKAL(V, E):
  sort E by increasing weight
  F ← (V, ∅)
  for each vertex v ∈ V
      MAKESET(v)
  for i ← 1 to |E|
      uv ← ith lightest edge in E
      if FIND(u) ≠ FIND(v)
          UNION(u, v)
          add uv to F
  return F
```

In our case, the sets are components of F, and $n = V$. Kruskal's algorithm performs $O(E)$ FIND operations, two for each edge in the graph, and $O(V)$ UNION operations, one for each edge in the minimum spanning tree. Using union-by-rank and path compression allows us to perform each UNION or FIND in $O(\alpha(E, V))$ time, where α is the not-quite-constant inverse-Ackerman function. So ignoring the cost of sorting the edges, the running time of this algorithm is $O(E\,\alpha(E, V))$.

[4] Live at the Assembly Hall! Only $49.95 on Pay-Per-View![5]

[5] Is Pay-Per-View still a thing?

We need $O(E \log E) = O(E \log V)$ additional time just to sort the edges. Since this is bigger than the time for the UNION-FIND data structure, the overall running time of Kruskal's algorithm is $\boldsymbol{O(E \log V)}$, exactly the same as Borůvka's algorithm, or Jarník's algorithm with a normal (non-Fibonacci) heap.

Exercises

1. Most classical minimum-spanning-tree algorithms use the notions of "safe" and "useless" edges described in the lecture notes, but there is an alternate formulation. Let G be a weighted undirected graph, where the edge weights are distinct. We say that an edge e is ***dangerous*** if it is the longest edge in some cycle in G, and ***useful*** if it does not lie in any cycle in G.

   (a) Prove that the minimum spanning tree of G contains every useful edge.

   (b) Prove that the minimum spanning tree of G does not contain any dangerous edge.

   (c) Describe and analyze an efficient implementation of the "anti-Kruskal" MST algorithm: Examine the edges of G in *decreasing* order; if an edge is dangerous, remove it from G. *[Hint: It won't be as fast as Kruskal's algorithm.]*

2. Let $G = (V, E)$ be an arbitrary connected graph with weighted edges.

   (a) Prove that for any partition of the vertices V into two disjoint subsets, the minimum spanning tree of G includes the minimum-weight edge with one endpoint in each subset.

   (b) Prove that for any cycle in G, the minimum spanning tree of G *excludes* the maximum-weight edge in that cycle.

   (c) Prove or disprove: The minimum spanning tree of G includes the minimum-weight edge in *every* cycle in G.

3. Throughout this lecture note, we assumed that no two edges in the input graph have equal weights, which implies that the minimum spanning tree is unique. In fact, a weaker condition on the edge weights implies MST uniqueness.

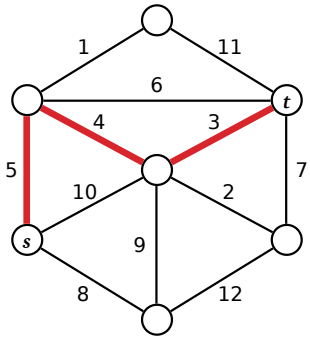
   (a) Describe an edge-weighted graph that has a unique minimum spanning tree, even though two edges have equal weights.

   (b) Prove that an edge-weighted graph G has a *unique* minimum spanning tree if and only if the following conditions hold:
   - For any partition of the vertices of G into two subsets, the minimum-weight edge with one endpoint in each subset is unique.
   - The maximum-weight edge in any cycle of G is unique.

   (c) Describe and analyze an algorithm to determine whether or not a graph has a unique minimum spanning tree.

4. Consider a path between two vertices s and t in an undirected weighted graph G. The *bottleneck length* of this path is the maximum weight of any edge in the path. The *bottleneck distance* between s and t is the minimum bottleneck length of any path from s to t. (If there are no paths from s to t, the bottleneck distance between s and t is ∞.)

   Describe an algorithm to compute the bottleneck distance between *every* pair of vertices in an arbitrary undirected weighted graph. Assume that no two edges have the same weight.



The bottleneck distance between s and t is 5.

5. (a) Describe and analyze an algorithm to compute the *maximum*-weight spanning tree of a given edge-weighted graph.

   (b) A *feedback edge set* of an undirected graph G is a subset F of the edges such that every cycle in G contains at least one edge in F. In other words, removing every edge in F makes the graph G acyclic. Describe and analyze a fast algorithm to compute the minimum weight feedback edge set of of a given edge-weighted graph.

6. Suppose we are given both an undirected graph G with weighted edges and a minimum spanning tree T of G.

   (a) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is decreased.

   (b) Describe an algorithm to update the minimum spanning tree when the weight of a single edge e is increased.

   In both cases, the input to your algorithm is the edge e and its new weight; your algorithms should modify T so that it is still a minimum spanning tree. *[Hint: Consider the cases $e \in T$ and $e \notin T$ separately.]*

7. (a) Describe and analyze and algorithm to find the *second smallest spanning tree* of a given graph G, that is, the spanning tree of G with smallest total weight except for the minimum spanning tree.

   *(b) Describe and analyze an efficient algorithm to compute, given a weighted undirected graph G and an integer k, the k spanning trees of G with smallest weight.

8. We say that a graph $G = (V, E)$ is *dense* if $E = \Theta(V^2)$. Describe a modification of Jarník's minimum-spanning tree algorithm that runs in $O(V^2)$ time (independent of E) when the input graph is dense, using only simple data structures (and in particular, *without* using a Fibonacci heap).

9. (a) Prove that the minimum spanning tree of a graph is also a spanning tree whose maximum-weight edge is minimal.

   *(b) Describe an algorithm to compute a spanning tree whose maximum-weight edge is minimal, in $O(V + E)$ time. *[Hint: Start by computing the median of the edge weights.]*

10. Consider the following variant of Borůvka's algorithm. Instead of counting and labeling components of F to find safe edges, we use a standard disjoint set data structure. Each component of F is represented by an up-tree; each vertex v stores a pointer $parent(v)$ to its parent in the up-tree containing v. Each leader vertex $\bar{v}$ also maintains an edge $safe(\bar{v})$, which is (eventually) the lightest edge with one endpoint in $\bar{v}$'s component of F.

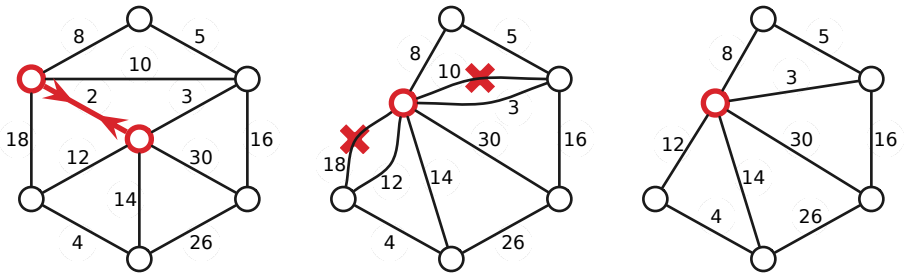
```
BORŮVKA(V, E):
  F = ∅
  for each vertex v ∈ V
      parent(v) ← v
  while FINDSAFEEDGES(V, E)
      ADDSAFEEDGES(V, E, F)
  return F
```

```
FINDSAFEEDGES(V, E):
  for each vertex v ∈ V
      safe(v) ← NULL
  found ← FALSE
  for each edge uv ∈ E
      ū ← FIND(u); v̄ ← FIND(v)
      if ū ≠ v̄
          if w(uv) < w(safe(ū))
              safe(ū) ← uv
          if w(uv) < w(safe(v̄))
              safe(v̄) ← uv
          found ← TRUE
  return done
```

```
ADDSAFEEDGES(V, E, F):
  for each vertex v ∈ V
      if safe(v) ≠ NULL
          xy ← safe(v)
          if FIND(x) ≠ FIND(y)
              UNION(x, y)
              add xy to F
```

Prove that if FIND uses path compression, then each call to FINDSAFEEDGES and ADDSAFEEDGES requires only $O(V + E)$ time. *[Hint: It doesn't matter how* UNION *is implemented! What is the depth of the up-trees when* FINDSAFEEDGES *ends?]*

11. Minimum-spanning tree algorithms are often formulated using an operation called *edge contraction*. To contract the edge uv, we insert a new node, redirect any edge incident to u or v (except uv) to this new node, and then delete u and v. After contraction, there may be multiple parallel edges between the new node and other nodes in the graph; we remove all but the lightest edge between any two nodes.



Contracting an edge and removing redundant parallel edges.

The three classical minimum-spanning tree algorithms can be expressed cleanly in terms of contraction as follows. All three algorithms start by making a clean copy G' of the input graph G

and then repeatedly contract safe edges in G; the minimum spanning tree consists of the contracted edges.

- BORŮVKA: Mark the lightest edge leaving each vertex, contract all marked edges, and recurse.
- JARNÍK: Repeatedly contract the lightest edge incident to some fixed root vertex.
- KRUSKAL: Repeatedly contract the lightest edge in the graph.

(a) Describe an algorithm to execute a single pass of Borůvka's contraction algorithm in $O(V + E)$ time. The input graph is represented in an adjacency list.

(b) Consider an algorithm that first performs k passes of Borůvka's contraction algorithm, and then runs Jarník's algorithm (*with* a Fibonacci heap) on the resulting contracted graph.

  i. What is the running time of this hybrid algorithm, as a function of V, E, and k?

  ii. For which value of k is this running time minimized? What is the resulting running time?

(c) Call a family of graphs *nice* if it has the following properties:

- A nice graph with n vertices has only $O(n)$ edges.
- Contracting an edge of a nice graph yields another nice graph.

For example, graphs that can be drawn in the plane without crossing edges are nice; Euler's formula implies that any planar graph with n vertices has at most $3n - 6$ edges.

Prove that Borüvka's contraction algorithm computes the minimum spanning tree of any nice n-vertex graph in $O(n)$ time.

Well, ya turn left by the fire station in the village and take the old post road by the reservoir and. . . no, that won't do.

Best to continue straight on by the tar road until you reach the schoolhouse and then turn left on the road to Bennett's Lake until. . . no, that won't work either.

East Millinocket, ya say? Come to think of it, you can't get there from here.

— Robert Bryan and Marshall Dodge,
Bert and I and Other Stories from Down East (1961)

Hey farmer! Where does this road go?
Been livin' here all my life, it ain't gone nowhere yet.
Hey farmer! How do you get to Little Rock?
Listen stranger, you can't get there from here.
Hey farmer! You don't know very much do you?
No, but I ain't lost.

— Michelle Shocked, "Arkansas Traveler" (1992)

26 Shortest Paths

26.1 Introduction

Suppose we are given a weighted *directed* graph $G = (V, E, w)$ with two special vertices, and we want to find the shortest path from a *source* vertex s to a *target* vertex t. That is, we want to find the directed path p starting at s and ending at t that minimizes the function

$$w(p) := \sum_{u \to v \in p} w(u \to v).$$

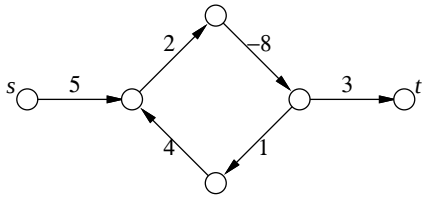
For example, if I want to answer the question "What's the fastest way to drive from my old apartment in Champaign, Illinois to my wife's old apartment in Columbus, Ohio?", I might use a graph whose vertices are cities, edges are roads, weights are driving times, s is Champaign, and t is Columbus.[1] The graph is directed, because driving times along the same road might be different in different directions. (At one time, there was a speed trap on I-70 just east of the Indiana/Ohio border, but only for eastbound traffic.)

Perhaps counter to intuition, we will allow the weights on the edges to be negative. Negative edges make our lives complicated, because the presence of a negative *cycle* might imply that there is no shortest path. In general, a shortest path from s to t exists if and only if there is *at least one* path from s to t, but there is no path from s to t that touches a negative cycle. If any negative cycle is reachable from s and can reach t, we can always find a shorter path by going around the cycle one more time.

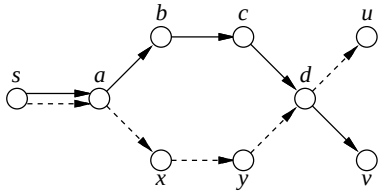
Almost every algorithm known for solving this problem actually solves (large portions of) the following more general ***single source shortest path*** or ***SSSP*** problem: Find the shortest path from the source vertex s to *every* other vertex in the graph. This problem is usually solved by finding a ***shortest path tree*** rooted at s that contains all the desired shortest paths.

It's not hard to see that if shortest paths are unique, then they form a tree, because any subpath of a shortest path is itself a shortest path. If there are multiple shortest paths to some vertices, we can always choose one shortest path to each vertex so that the union of the paths is a tree. If there are shortest paths to two vertices u and v that diverge, then meet, then diverge again, we can modify one of the paths without changing its length so that the two paths only diverge once.

[1] West on Church, north on Prospect, east on I-74, south on I-465, east on Airport Expressway, north on I-65, east on I-70, north on Grandview, east on 5th, north on Olentangy River, east on Dodridge, north on High, west on Kelso, south on Neil. Depending on traffic. We both live in Urbana now.

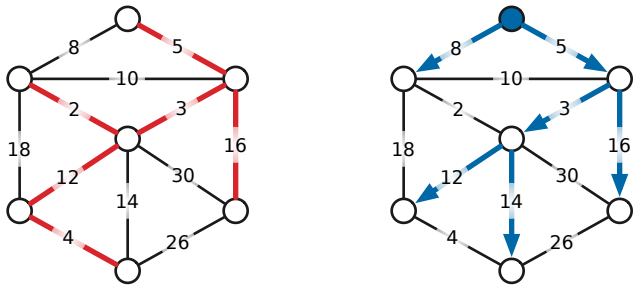


There is no shortest path from s to t.



If $s{\to}a{\to}b{\to}c{\to}d{\to}v$ and $s{\to}a{\to}x{\to}y{\to}d{\to}u$ are shortest paths, then $s{\to}a{\to}b{\to}c{\to}d{\to}u$ is also a shortest path.

Although they are both optimal spanning trees, shortest-path trees and minimum spanning trees are very different creatures. Shortest-path trees are rooted and directed; minimum spanning trees are unrooted and undirected. Shortest-path trees are most naturally defined for directed graphs; only undirected graphs have minimum spanning trees. If edge weights are distinct, there is only one minimum spanning tree, but every source vertex induces a different shortest-path tree; moreover, it is possible for *every* shortest path tree to use a different set of edges from the minimum spanning tree.

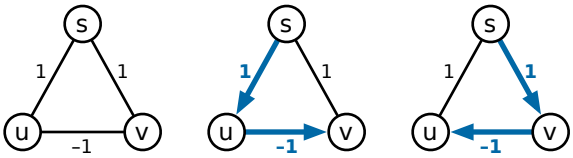


A minimum spanning tree and a shortest path tree (rooted at the top vertex) of the same undirected graph.

26.2 Warning!

Throughout this lecture, we will explicitly consider *only* directed graphs. All of the algorithms described in this lecture also work for undirected graphs with some minor modifications, *but only if negative edges are prohibited*. Dealing with negative edges in undirected graphs is considerably more subtle. We cannot simply replace every undirected edge with a pair of directed edges, because this would transform any negative edge into a short negative cycle. Subpaths of an undirected shortest path that contains a negative edge are *not* necessarily shortest paths; consequently, the set of all undirected shortest paths from a single source vertex may not define a tree, even if shortest paths are unique.

A complete treatment of undirected graphs with negative edges is beyond the scope of this lecture (if not the entire course). I will only mention that a *single* shortest path in an undirected graph with negative edges can be computed in $O(VE + V^2 \log V)$ time, by a reduction to maximum weighted matching.



An undirected graph where shortest paths from s are unique but do not define a tree.

26.3 The Only SSSP Algorithm

Just like graph traversal and minimum spanning trees, there are several different SSSP algorithms, but they are all special cases of the a single generic algorithm, first proposed by Lester Ford in 1956, and independently by George Dantzig in 1957.[2] Each vertex v in the graph stores two values, which (inductively) describe a *tentative* shortest path from s to v.

- $dist(v)$ is the length of the tentative shortest $s \rightsquigarrow v$ path, or ∞ if there is no such path.
- $pred(v)$ is the predecessor of v in the tentative shortest $s \rightsquigarrow v$ path, or NULL if there is no such vertex.

In fact, the predecessor pointers automatically define a tentative shortest path *tree*; they play exactly the same role as the parent pointers in our generic graph traversal algorithm. At the beginning of the algorithm, we already know that $dist(s) = 0$ and $pred(s) = \text{NULL}$. For every vertex $v \neq s$, we initially set $dist(v) = \infty$ and $pred(v) = \text{NULL}$ to indicate that we do not know of *any* path from s to v.

During the execution of the algorithm, we call an edge $u \to v$ ***tense*** if $dist(u) + w(u \to v) < dist(v)$. If $u \to v$ is tense, the tentative shortest path $s \rightsquigarrow v$ is clearly incorrect, because the path $s \rightsquigarrow u \to v$ is shorter. Our generic algorithm repeatedly finds a tense edge in the graph and *relaxes* it:

```
RELAX(u→v):
  dist(v) ← dist(u) + w(u→v)
  pred(v) ← u
```

When there are no tense edges, the algorithm halts, and we have our desired shortest path tree.

The correctness of Ford's generic relaxation algorithm follows from the following series of claims:

1. For every vertex v, the distance $dist(v)$ is either ∞ or the length of some walk from s to v. This claim can be proved by induction on the number of relaxations.

2. If the graph has no negative cycles, then $dist(v)$ is either ∞ or the length of some *simple path* from s to v. Specifically, if $dist(v)$ is the length of a walk from s to v that contains a directed cycle, that cycle must have negative weight. This claim implies that if G has no negative cycles, the relaxation algorithm eventually halts, because there are only a finite number of simple paths in G.

3. If no edge in G is tense, then for every vertex v, the distance $dist(v)$ is the length of the predecessor path $s \to \cdots pred(pred(v)) \to pred(v) \to v$. Specifically, if v violates this condition but its predecessor $pred(v)$ does not, the edge $pred(v) \to v$ is tense.

4. If no edge in G is tense, then for every vertex v, the path $s \to \cdots pred(pred(v)) \to pred(v) \to v$ is a shortest path from s to v. Specifically, if v violates this condition but its predecessor u *in some shortest path* does not, the edge $u \to v$ is tense. This claim also implies that if the G has a negative cycle, then some edge is *always* tense, so the generic algorithm never halts.

[2] Specifically, Dantzig showed that the shortest path problem can be phrased as a linear programming problem, and then described an interpretation of his simplex method in terms of the original graph. His description is equivalent to Ford's relaxation strategy.

So far I haven't said anything about how we detect which edges can be relaxed, or in what order we relax them. To make this easier, we refine the relaxation algorithm slightly, into something closely resembling the generic graph traversal algorithm. We maintain a "bag" of vertices, initially containing just the source vertex s. Whenever we take a vertex u from the bag, we scan all of its outgoing edges, looking for something to relax. Finally, whenever we successfully relax an edge $u \to v$, we put v into the bag. Unlike our generic graph traversal algorithm, we do not mark vertices when we visit them; the same vertex could be visited many times, and the same edge could be relaxed many times.

```
INITSSSP(s):
  dist(s) ← 0
  pred(s) ← NULL
  for all vertices v ≠ s
      dist(v) ← ∞
      pred(v) ← NULL
```

```
GENERICSSSP(s):
  INITSSSP(s)
  put s in the bag
  while the bag is not empty
      take u from the bag
      for all edges u→v
          if u→v is tense
              RELAX(u→v)
              put v in the bag
```

Just as with graph traversal, different "bag" data structures for the give us different algorithms. There are three obvious choices to try: a stack, a queue, and a priority queue. Unfortunately, if we use a stack, the resulting algorithm performs $\Theta(2^V)$ relaxation steps in the worst case! (Proving this is a good homework problem.) The other two possibilities are much more efficient.

26.4 Dijkstra's Algorithm

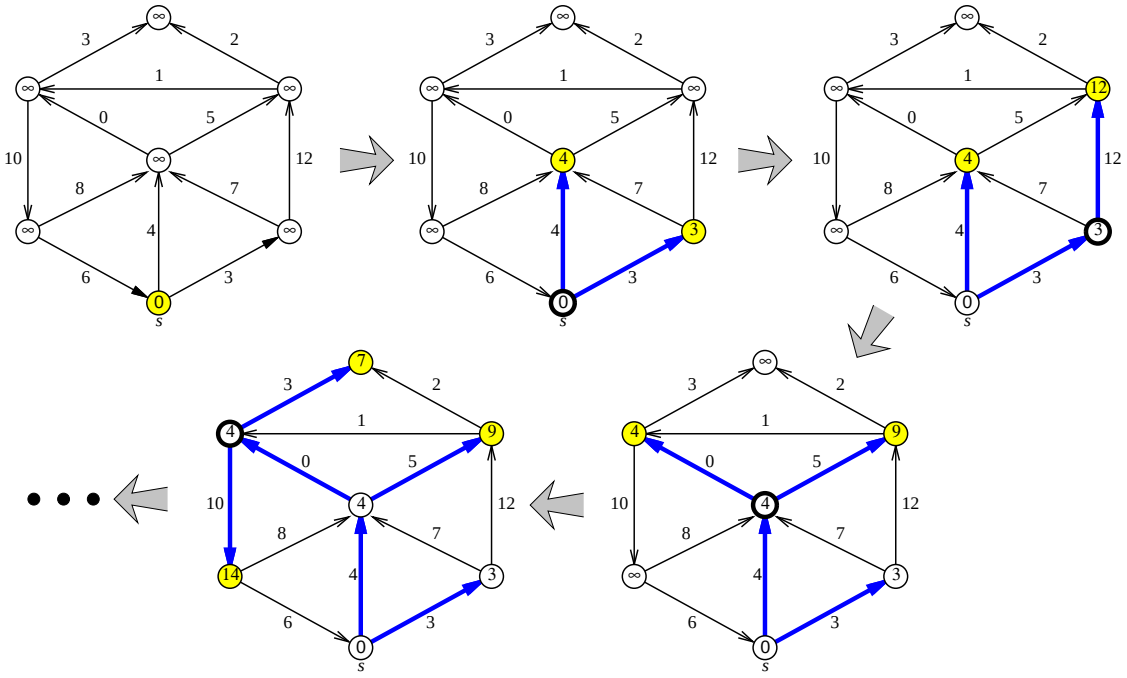
If we implement the bag using a priority queue, where the key of a vertex v is its tentative distance $dist(v)$, we obtain an algorithm first "published" in 1957 by a team of researchers at the Case Institute of Technology, in an annual project report for the Combat Development Department of the US Army Electronic Proving Ground. The same algorithm was later independently rediscovered and actually publicly published by Edsger Dijkstra in 1959. A nearly identical algorithm was also described by George Dantzig in 1958.

Dijkstra's algorithm, as it is universally known[3], is particularly well-behaved if the graph has no negative-weight edges. In this case, it's not hard to show (by induction, of course) that the vertices are scanned in increasing order of their shortest-path distance from s. It follows that each vertex is scanned at most once, and thus that each edge is relaxed at most once. Since the key of each vertex in the heap is its tentative distance from s, the algorithm performs a DECREASEKEY operation every time an edge is relaxed. Thus, the algorithm performs at most E DECREASEKEYs. Similarly, there are at most V INSERT and EXTRACTMIN operations. Thus, if we store the vertices in a Fibonacci heap, the total running time of Dijkstra's algorithm is $\boldsymbol{O(E + V \log V)}$; if we use a regular binary heap, the running time is $\boldsymbol{O(E \log V)}$.

This analysis assumes that no edge has negative weight. Dijkstra's algorithm (in the form I'm presenting here[4]) is still *correct* if there are negative edges, but the worst-case running time could be exponential. (Proving this unfortunate fact is a good homework problem.) On the other hand, in practice, Dijkstra's algorithm is usually quite fast even for graphs with negative edges.

[3] I will follow this common convention, despite the historical inaccuracy, partly because I don't think anybody wants to read about the "Leyzorek-Gray-Johnson-Ladew-Meaker-Petry-Seitz algorithm", and partly because papers that aren't actually *publically* published don't count.

[4] Most algorithms textbooks, Wikipedia, and even Dijkstra's original paper present a version of Dijkstra's algorithm that gives incorrect results for graphs with negative edges, because it *never* visits the same vertex more than once. I've taken the liberty of correcting Dijkstra's mistake. Even Dijkstra would agree that a correct algorithm that is sometimes slow (and in practice, *rarely* slow) is better than a fast algorithm that doesn't always work.



Four phases of Dijkstra's algorithm run on a graph with no negative edges.
At each phase, the shaded vertices are in the heap, and the bold vertex has just been scanned.
The bold edges describe the evolving shortest path tree.

26.5 The A^* Heuristic

A slight generalization of Dijkstra's algorithm, commonly known as the A^* algorithm, is frequently used to find a shortest path from a single source node s to a single target node t. This heuristic was first described in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael. A^* uses a black-box function GUESSDISTANCE(v, t) that returns an estimate of the distance from v to t. The only difference between Dijkstra and A^* is that the key of a vertex v is $dist(v) +$ GUESSDISTANCE(v, t).

The function GUESSDISTANCE is called *admissible* if GUESSDISTANCE(v, t) never overestimates the actual shortest path distance from v to t. If GUESSDISTANCE is admissible and the actual edge weights are all non-negative, the A^* algorithm computes the actual shortest path from s to t at least as quickly as Dijkstra's algorithm. In practice, the closer GUESSDISTANCE(v, t) is to the real distance from v to t, the faster the algorithm. However, in the worst case, the running time is still $O(E + V \log V)$.

The heuristic is especially useful in situations where the actual graph is not known. For example, A^* can be used to find optimal solutions to many puzzles (15-puzzle, Freecell, Shanghai, Sokoban, Atomix, Rush Hour, Rubik's Cube, Racetrack, ...) and other path planning problems where the starting and goal configurations are given, but the graph of all possible configurations and their connections is not given explicitly.

26.6 Shimbel's Algorithm

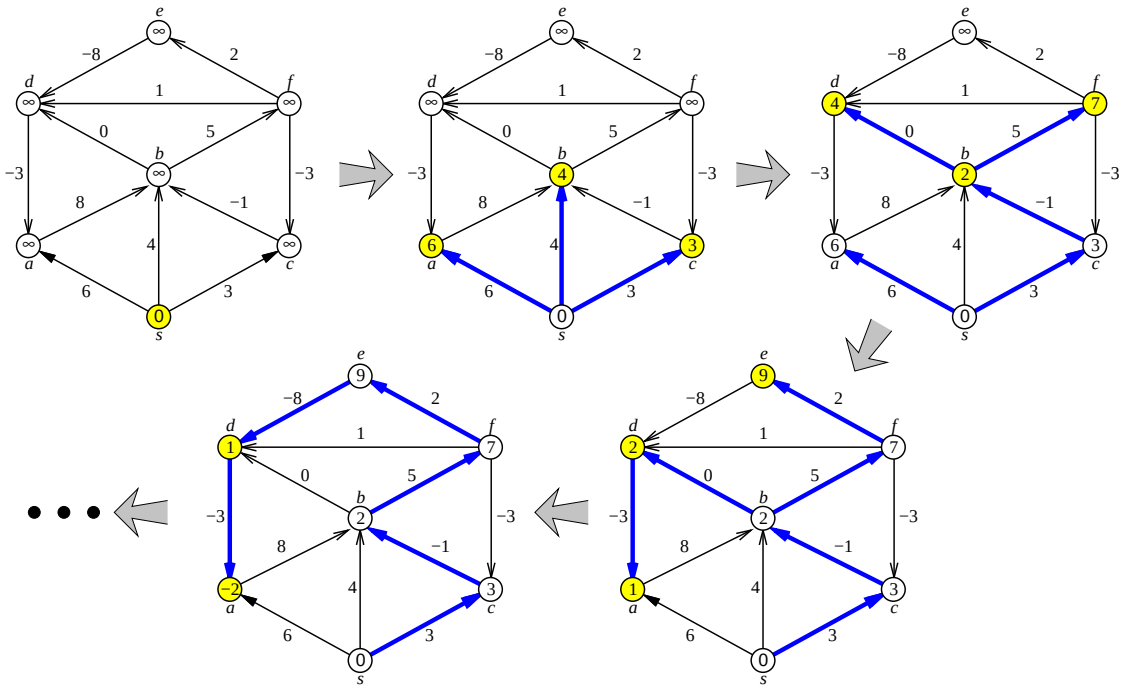
If we replace the heap in Dijkstra's algorithm with a FIFO queue, we obtain an algorithm first sketched by Shimbel in 1954, described in more detail by Moore in 1957, then independently rediscovered by Woodbury and Dantzig in 1957 and again by Bellman in 1958. Because Bellman explicitly used Ford's formulation of relaxing edges, this algorithm is almost universally called "Bellman-Ford", although some early sources refer to "Bellman-Shimbel". Shimbel's algorithm is efficient even if there are negative edges, and it can be used to quickly detect the presence of negative cycles. If there are no negative edges,

however, Dijkstra's algorithm is faster. (In fact, in practice, Dijkstra's algorithm is often faster even for graphs with negative edges.)

The easiest way to analyze the algorithm is to break the execution into *phases*, by introducing an imaginary *token*. Before we even begin, we insert the token into the queue. The current phase ends when we take the token out of the queue; we begin the next phase by reinserting the token into the queue. The 0th phase consists entirely of scanning the source vertex s. The algorithm ends when the queue contains *only* the token. A simple inductive argument (hint, hint) implies the following invariant for every integer i and vertex v:

> **After i phases of the algorithm, $dist(v)$ is *at most* the length of the shortest walk from s to v consisting of *at most* i edges.**

Since a shortest path can only pass through each vertex once, either the algorithm halts before the Vth phase, or the graph contains a negative cycle. In each phase, we scan each vertex at most once, so we relax each edge at most once, so the running time of a single phase is $O(E)$. Thus, the overall running time of Shimbel's algorithm is $\boldsymbol{O(VE)}$.



Four phases of Shimbel's algorithm run on a directed graph with negative edges.
Nodes are taken from the queue in the order $s \diamond a\ b\ c \diamond d\ f\ b \diamond a\ e\ d \diamond d\ a \diamond\diamond$, where $\diamond$ is the end-of-phase token.
Shaded vertices are in the queue at the end of each phase. The bold edges describe the evolving shortest path tree.

Once we understand how the phases of Shimbel's algorithm behave, we can simplify the algorithm considerably by producing the same behavior on purpose. Instead of performing a partial breadth-first search of the graph in each phase, we can simply scan through the adjacency list directly, relaxing every tense edge we find in the graph.

SHIMBEL: Relax ***ALL*** the tense edges and recurse.

```
SHIMBELSSSP(s)
  INITSSSP(s)
  repeat V times:
      for every edge u→v
          if u→v is tense
              RELAX(u→v)
  for every edge u→v
      if u→v is tense
          return "Negative cycle!"
```

This is how most textbooks present "Bellman-Ford".[5] The $O(VE)$ running time of this formulation of the algorithm should be obvious, but it may be less clear that the algorithm is still correct. In fact, correctness follows from exactly the same invariant as before:

> **After i phases of the algorithm, $dist(v)$ is *at most* the length of the shortest walk from s to v consisting of *at most* i edges.**

As before, it is straightforward to prove by induction (hint, hint) that this invariant holds for every integer i and vertex v.

26.7 Shimbel's Algorithm as Dynamic Programming

Shimbel's algorithm can also be recast as a dynamic programming algorithm. Let $dist_i(v)$ denote the length of the shortest path $s \rightsquigarrow v$ consisting of at most i edges. It's not hard to see that this function obeys the following recurrence:

$$dist_i(v) = \begin{cases} 0 & \text{if } i = 0 \text{ and } v = s \\ \infty & \text{if } i = 0 \text{ and } v \neq s \\ \min \left\{ \begin{array}{c} dist_{i-1}(v), \\ \min\limits_{u \to v \in E} (dist_{i-1}(u) + w(u \to v)) \end{array} \right\} & \text{otherwise} \end{cases}$$

For the moment, let's assume the graph has no negative cycles; our goal is to compute $dist_{V-1}(t)$. We can clearly memoize this two-parameter function into a two-dimensional array. A straightforward dynamic programming evaluation of this recurrence looks like this:

[5] In fact, this is essentially the formulation proposed by both Shimbel and Bellman. Bob Tarjan recognized in the early 1980s that Shimbel's algorithm is equivalent to Dijkstra's algorithm with a queue instead of a heap.

```
SHIMBELDP(s)
  dist[0,s] ← 0
  for every vertex v ≠ s
      dist[0,v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i,v] ← dist[i−1,v]
          for every edge u→v
              if dist[i,v] > dist[i−1,u] + w(u→v)
                  dist[i,v] ← dist[i−1,u] + w(u→v)
```

Now let us make two minor changes to this algorithm. First, we remove one level of indentation from the last three lines. This may change the order in which we examine edges, but the modified algorithm still computes $dist_i(v)$ for all i and v. Second, we change the indices in the last two lines from $i-1$ to i. This change may cause the distances $dist[i,v]$ to approach the true shortest-path distances more quickly than before, but the algorithm is still correct.

```
SHIMBELDP2(s)
  dist[0,s] ← 0
  for every vertex v ≠ s
      dist[0,v] ← ∞
  for i ← 1 to V − 1
      for every vertex v
          dist[i,v] ← dist[i−1,v]
      for every edge u→v
          if dist[i,v] > dist[i,u] + w(u→v)
              dist[i,v] ← dist[i,u] + w(u→v)
```

Now notice that the iteration index i is completely redundant! We really only need to keep a one-dimensional array of distances, which means we don't need to scan the vertices in each iteration of the main loop.

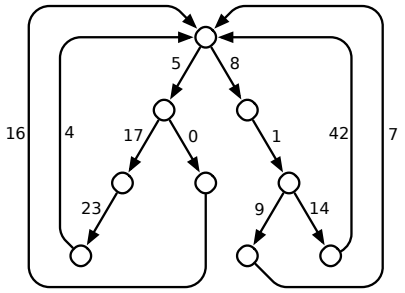
```
SHIMBELDP3(s)
  dist[s] ← 0
  for every vertex v ≠ s
      dist[v] ← ∞
  for i ← 1 to V − 1
      for every edge u→v
          if dist[v] > dist[u] + w(u→v)
              dist[v] ← dist[u] + w(u→v)
```

The resulting algorithm is almost identical to our earlier algorithm SHIMBELSSSP! The first three lines initialize the shortest path distances, and the last two lines check whether an edge is tense, and if so, relaxes it. The only feature missing from the new algorithm is explicit maintenance of predecessors, but that's easy to add.

Exercises

0. Prove that the following invariant holds for every integer i and every vertex v: After i phases of Shimbel's algorithm (in either formulation), $dist(v)$ is at most the length of the shortest path $s \rightsquigarrow v$ consisting of at most i edges.

1. A *looped tree* is a weighted, directed graph built from a binary tree by adding an edge from every leaf back to the root. Every edge has a non-negative weight.



A looped tree.

   (a) How much time would Dijkstra's algorithm require to compute the shortest path between two vertices u and v in a looped tree with n nodes?

   (b) Describe and analyze a faster algorithm.

2. For any edge e in any graph G, let $G \setminus e$ denote the graph obtained by deleting e from G.

   (a) Suppose we are given a directed graph G in which the shortest path σ from vertex s to vertex t passes through *every* vertex of G. Describe an algorithm to compute the shortest-path distance from s to t in $G \setminus e$, for *every* edge e of G, in $O(E \log V)$ time. Your algorithm should output a set of E shortest-path distances, one for each edge of the input graph. You may assume that all edge weights are non-negative. *[Hint: If we delete an edge of the original shortest path, how do the old and new shortest paths overlap?]*

   *(b) Let s and t be *arbitrary* vertices in an arbitrary *undirected* graph G. Describe an algorithm to compute the shortest-path distance from s to t in $G \setminus e$, for *every* edge e of G, in $O(E \log V)$ time. Again, you may assume that all edge weights are non-negative.

3. Let $G = (V, E)$ be a connected directed graph with non-negative edge weights, let s and t be vertices of G, and let H be a subgraph of G obtained by deleting some edges. Suppose we want to reinsert exactly one edge from G back into H, so that the shortest path from s to t in the resulting graph is as short as possible. Describe and analyze an algorithm that chooses the best edge to reinsert, in $O(E \log V)$ time.

4. When there is more than one shortest path from one node s to another node t, it is often convenient to choose a shortest path with the fewest edges; call this the ***best path*** from s to t. Suppose we are given a directed graph G with positive edge weights and a source vertex s in G. Describe and analyze an algorithm to compute *best* paths in G from s to every other vertex.

*5. (a) Prove that Ford's generic shortest-path algorithm (while the graph contains a tense edge, relax it) can take exponential time in the worst case when implemented with a stack instead of a priority queue (like Dijkstra) or a queue (like Shimbel). Specifically, for every positive integer n, construct a weighted directed n-vertex graph G_n, such that the stack-based shortest-path algorithm call RELAX $\Omega(2^n)$ times when G_n is the input graph. *[Hint: Towers of Hanoi.]*

(b) Prove that Dijkstra's shortest-path algorithm can require exponential time in the worst case when edges are allowed to have negative weight. Specifically, for every positive integer n, construct a weighted directed n-vertex graph G_n, such that Dijkstra's algorithm calls RELAX $\Omega(2^n)$ times when G_n is the input graph. *[Hint: This is relatively easy if you've already solved part (a).]*

6. (a) Describe and analyze a modification of Shimbel's shortest-path algorithm that actually returns a negative cycle if any such cycle is reachable from s, or a shortest-path tree if there is no such cycle. The modified algorithm should still run in $O(VE)$ time.

   (b) Describe and analyze a modification of Shimbel's shortest-path algorithm that computes the correct shortest path distances from s to every other vertex of the input graph, even if the graph contains negative cycles. Specifically, if any walk from s to v contains a negative cycle, your algorithm should end with $dist(v) = -\infty$; otherwise, $dist(v)$ should contain the length of the shortest path from s to v. The modified algorithm should still run in $O(VE)$ time.

   *(c) Repeat parts (a) and (b), but for Ford's generic shortest-path algorithm. You may assume that the unmodified algorithm halts in $O(2^V)$ steps if there is no negative cycle; your modified algorithms should also run in $O(2^V)$ time.

*7. Describe and analyze an efficient algorithm to compute the *number* of shortest paths between two specified vertices s and t in a directed graph G whose edges have positive weights. *[Hint: Which edges of G can lie on a shortest path from s to t?]*

8. After a grueling algorithms midterm, you decide to take the bus home. Since you planned ahead, you have a schedule that lists the times and locations of every stop of every bus in Champaign-Urbana. Unfortunately, there isn't a single bus that visits both your exam building and your home; you must transfer between bus lines at least once.

   Describe and analyze an algorithm to determine the sequence of bus rides that will get you home as early as possible, assuming there are b different bus lines, and each bus stops n times per day. Your goal is to minimize your *arrival time*, not the time you actually spend traveling. Assume that the buses run exactly on schedule, that you have an accurate watch, and that you are too tired to walk between bus stops.

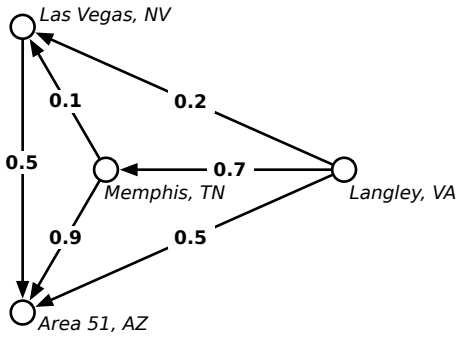
9. After graduating you accept a job with Aerophobes-Я-Us, the leading traveling agency for people who hate to fly. Your job is to build a system to help customers plan airplane trips from one city to another. All of your customers are afraid of flying (and by extension, airports), so any trip you plan needs to be as short as possible. You know all the departure and arrival times of all the flights on the planet.

   Suppose one of your customers wants to fly from city X to city Y. Describe an algorithm to find a sequence of flights that minimizes the *total time in transit*—the length of time from the initial departure to the final arrival, including time at intermediate airports waiting for connecting flights. *[Hint: Modify the input data and apply Dijkstra's algorithm.]*

10. Mulder and Scully have computed, for every road in the United States, the exact probability that someone driving on that road *won't* be abducted by aliens. Agent Mulder needs to drive from

Langley, Virginia to Area 51, Nevada. What route should he take so that he has the least chance of being abducted?

More formally, you are given a directed graph $G = (V, E)$, where every edge e has an independent safety probability $p(e)$. The *safety* of a path is the product of the safety probabilities of its edges. Design and analyze an algorithm to determine the safest path from a given start vertex s to a given target vertex t.



For example, with the probabilities shown above, if Mulder tries to drive directly from Langley to Area 51, he has a 50% chance of getting there without being abducted. If he stops in Memphis, he has a $0.7 \times 0.9 = 63\%$ chance of arriving safely. If he stops first in Memphis and then in Las Vegas, he has a $1 - 0.7 \times 0.1 \times 0.5 = 96.5\%$ chance of being abducted! (That's how they got Elvis, you know.) Although this example is a dag, your algorithm must handle *arbitrary* directed graphs.

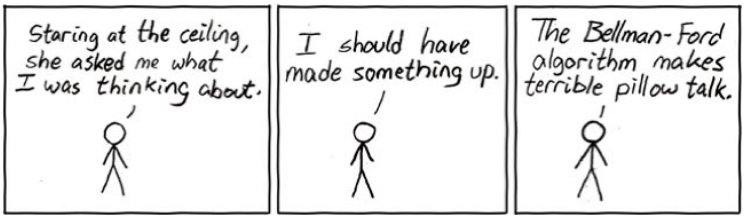
11. On an overnight camping trip in Sunnydale National Park, you are woken from a restless sleep by a scream. As you crawl out of your tent to investigate, a terrified park ranger runs out of the woods, covered in blood and clutching a crumpled piece of paper to his chest. As he reaches your tent, he gasps, "Get out…while…you…", thrusts the paper into your hands, and falls to the ground. Checking his pulse, you discover that the ranger is stone dead.

    You look down at the paper and recognize a map of the park, drawn as an undirected graph, where vertices represent landmarks in the park, and edges represent trails between those landmarks. (Trails start and end at landmarks and do not cross.) You recognize one of the vertices as your current location; several vertices on the boundary of the map are labeled EXIT.

    On closer examination, you notice that someone (perhaps the poor dead park ranger) has written a real number between 0 and 1 next to each vertex and each edge. A scrawled note on the back of the map indicates that a number next to an edge is the probability of encountering a vampire along the corresponding trail, and a number next to a vertex is the probability of encountering a vampire at the corresponding landmark. (Vampires can't stand each other's company, so you'll never see more than one vampire on the same trail or at the same landmark.) The note warns you that stepping off the marked trails will result in a slow and painful death.

    You glance down at the corpse at your feet. Yes, his death certainly looked painful. Wait, was that a twitch? Are his teeth getting longer? After driving a tent stake through the undead ranger's heart, you wisely decide to leave the park immediately.

    Describe and analyze an efficient algorithm to find a path from your current location to an arbitrary EXIT node, such that the total *expected number* of vampires encountered along the path is as small as possible. *Be sure to account for* ***both*** *the vertex probabilities* ***and*** *the edge probabilities!*



— Randall Munroe, *xkcd* (http://xkcd.com/69/)

The tree which fills the arms grew from the tiniest sprout;
the tower of nine storeys rose from a (small) heap of earth;
the journey of a thousand li commenced with a single step.

— Lao-Tzu, *Tao Te Ching*, chapter 64 (6th century BC), translated by J. Legge (1891)

And I would walk five hundred miles,
And I would walk five hundred more,
Just to be the man who walks a thousand miles
To fall down at your door.

— The Proclaimers, "Five Hundred Miles (I'm Gonna Be)", *Sunshine on Leith* (2001)

Almost there. . . Almost there. . .

— Red Leader [Drewe Henley], *Star Wars* (1977)

27 All-Pairs Shortest Paths

In the previous lecture, we saw algorithms to find the shortest path from a source vertex s to a target vertex t in a directed graph. As it turns out, the best algorithms for this problem actually find the shortest path from s to every possible target (or from every possible source to t) by constructing a shortest path tree. The shortest path tree specifies two pieces of information for each node v in the graph:

- $dist(v)$ is the length of the shortest path (if any) from s to v;
- $pred(v)$ is the second-to-last vertex (if any) the shortest path (if any) from s to v.

In this lecture, we want to generalize the shortest path problem even further. In the *all pairs shortest path* problem, we want to find the shortest path from *every* possible source to *every* possible destination. Specifically, for every pair of vertices u and v, we need to compute the following information:

- $dist(u, v)$ is the length of the shortest path (if any) from u to v;
- $pred(u, v)$ is the second-to-last vertex (if any) on the shortest path (if any) from u to v.

For example, for any vertex v, we have $dist(v, v) = 0$ and $pred(v, v) = \text{NULL}$. If the shortest path from u to v is only one edge long, then $dist(u, v) = w(u \to v)$ and $pred(u, v) = u$. If there is *no* shortest path from u to v—either because there's no path at all, or because there's a negative cycle—then $dist(u, v) = \infty$ and $pred(v, v) = \text{NULL}$.

The output of our shortest path algorithms will be a pair of $V \times V$ arrays encoding all V^2 distances and predecessors. Many maps include a distance matrix—to find the distance from (say) Champaign to (say) Columbus, you would look in the row labeled 'Champaign' and the column labeled 'Columbus'. In these notes, I'll focus almost exclusively on computing the distance array. The predecessor array, from which you would compute the actual shortest paths, can be computed with only minor additions to the algorithms I'll describe (hint, hint).

27.1 Lots of Single Sources

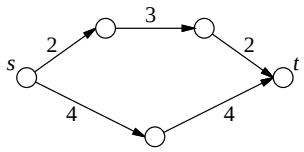
The obvious solution to the all-pairs shortest path problem is just to run a single-source shortest path algorithm V times, once for every possible source vertex! Specifically, to fill in the one-dimensional subarray $dist[s, \cdot]$, we invoke either Dijkstra's or Shimbel's algorithm starting at the source vertex s.

```
OBVIOUSAPSP(V, E, w):
  for every vertex s
      dist[s, ·] ← SSSP(V, E, w, s)
```

The running time of this algorithm depends on which single-source shortest path algorithm we use. If we use Shimbel's algorithm, the overall running time is $\Theta(V^2E) = O(V^4)$. If all the edge weights are non-negative, we can use Dijkstra's algorithm instead, which decreases the running time to $\Theta(VE + V^2 \log V) = O(V^3)$. For graphs with negative edge weights, Dijkstra's algorithm can take exponential time, so we can't get this improvement directly.

27.2 Reweighting

One idea that occurs to most people is increasing the weights of all the edges by the same amount so that all the weights become positive, and then applying Dijkstra's algorithm. Unfortunately, this simple idea doesn't work. Different paths change by different amounts, which means the shortest paths in the reweighted graph may not be the same as in the original graph.



Increasing all the edge weights by 2 changes the shortest path s to t.

However, there is a more complicated method for reweighting the edges in a graph. Suppose each vertex v has some associated *cost* $c(v)$, which might be positive, negative, or zero. We can define a new weight function w' as follows:

$$w'(u \to v) = c(u) + w(u \to v) - c(v)$$

To give some intuition, imagine that when we leave vertex u, we have to pay an exit tax of $c(u)$, and when we enter v, we get $c(v)$ as an entrance gift.

Now it's not too hard to show that the shortest paths with the new weight function w' are exactly the same as the shortest paths with the original weight function w. In fact, for *any* path $u \rightsquigarrow v$ from one vertex u to another vertex v, we have

$$w'(u \rightsquigarrow v) = c(u) + w(u \rightsquigarrow v) - c(v).$$

We pay $c(u)$ in exit fees, plus the original weight of of the path, minus the $c(v)$ entrance gift. At every intermediate vertex x on the path, we get $c(x)$ as an entrance gift, but then immediately pay it back as an exit tax!

27.3 Johnson's Algorithm

Johnson's all-pairs shortest path algorithm finds a cost $c(v)$ for each vertex, so that when the graph is reweighted, every edge has non-negative weight.

Suppose the graph has a vertex s that has a path to every other vertex. Johnson's algorithm computes the shortest paths from s to every other vertex, using Shimbel's algorithm (which doesn't care if the edge weights are negative), and then sets $c(v) \leftarrow dist(s, v)$, so the new weight of every edge is

$$w'(u \to v) = dist(s, u) + w(u \to v) - dist(s, v).$$

Why are all these new weights non-negative? Because otherwise, Shimbel's algorithm wouldn't be finished! Recall that an edge $u \to v$ is *tense* if $dist(s, u) + w(u \to v) < dist(s, v)$, and that single-source

shortest path algorithms eliminate all tense edges. The only exception is if the graph has a negative cycle, but then shortest paths aren't defined, and Johnson's algorithm simply aborts.

But what if the graph *doesn't* have a vertex s that can reach everything? No matter where we start Shimbel's algorithm, some of those vertex costs will be infinite. Johnson's algorithm avoids this problem by adding a new vertex s to the graph, with zero-weight edges going from s to every other vertex, but *no* edges going back into s. This addition doesn't change the shortest paths between any other pair of vertices, because there are no paths into s.

So here's Johnson's algorithm in all its glory.

```
JOHNSONAPSP(V, E, w) :
  create a new vertex s
  for every vertex v
      w(s→v) ← 0
      w(v→s) ← ∞
  dist[s, ·] ← SHIMBEL(V, E, w, s)
  if SHIMBEL found a negative cycle
      fail gracefully
  for every edge (u, v) ∈ E
      w'(u→v) ← dist[s, u] + w(u→v) − dist[s, v]
  for every vertex u
      dist[u, ·] ← DIJKSTRA(V, E, w', u)
      for every vertex v
          dist[u, v] ← dist[u, v] − dist[s, u] + dist[s, v]
```

The algorithm spends $\Theta(V)$ time adding the artificial start vertex s, $\Theta(VE)$ time running SHIMBEL, $O(E)$ time reweighting the graph, and then $\Theta(VE + V^2 \log V)$ running V passes of Dijkstra's algorithm. Thus, the overall running time is $\boldsymbol{\Theta(VE + V^2 \log V)}$.

27.4 Dynamic Programming

There's a completely different solution to the all-pairs shortest path problem that uses dynamic programming instead of a single-source algorithm. For *dense* graphs where $E = \Omega(V^2)$, the dynamic programming approach eventually leads to the same $O(V^3)$ running time as Johnson's algorithm, but with a much simpler algorithm. In particular, the new algorithm avoids Dijkstra's algorithm, which gets its efficiency from Fibonacci heaps, which are rather easy to screw up in the implementation. **In the rest of this lecture, I will assume that the input graph contains no negative cycles.**

As usual for dynamic programming algorithms, we first need to come up with a recursive formulation of the problem. Here is an "obvious" recursive definition for $dist(u, v)$:

$$dist(u,v) = \begin{cases} 0 & \text{if } u = v \\ \min\limits_{x \to v} \big(dist(u,x) + w(x \to v) \big) & \text{otherwise} \end{cases}$$

In other words, to find the shortest path from u to v, we consider all possible last edges $x \to v$ and recursively compute the shortest path from u to x. **Unfortunately, this recurrence doesn't work!** To compute $dist(u, v)$, we may need to compute $dist(u, x)$ for every other vertex x. But to compute $dist(u, x)$, we may need to compute $dist(u, v)$. We're stuck in an infinite loop!

To avoid this circular dependency, we need an additional parameter that decreases at each recursion, eventually reaching zero at the base case. One possibility is to include the number of edges in the shortest path as this third magic parameter, just as we did in the dynamic programming formulation of Shimbel's

algorithm. Let $dist(u, v, k)$ denote the length of the shortest path from u to v that uses *at most k edges*. Since we know that the shortest path between any two vertices has at most $V-1$ vertices, $dist(u, v, V-1)$ is the actual shortest-path distance. As in the single-source setting, we have the following recurrence:

$$dist(u, v, k) = \begin{cases} 0 & \text{if } u = v \\ \infty & \text{if } k = 0 \text{ and } u \neq v \\ \min_{x \to v} \big(dist(u, x, k-1) + w(x \to v) \big) & \text{otherwise} \end{cases}$$

Turning this recurrence into a dynamic programming algorithm is straightforward. To make the algorithm a little shorter, let's assume that $w(v \to v) = 0$ for every vertex v. Assuming the graph is stored in an adjacency list, the resulting algorithm runs in $\boldsymbol{\Theta(V^2E)}$ ***time***.

```
DynamicProgrammingAPSP(V, E, w):
  for all vertices u
      for all vertices v
          if u = v
              dist[u, v, 0] ← 0
          else
              dist[u, v, 0] ← ∞

  for k ← 1 to V − 1
      for all vertices u
          for all vertices v
              dist[u, v, k] ← ∞
              for all edges x→v
                  if dist[u, v, k] > dist[u, x, k − 1] + w(x→v)
                      dist[u, v, k] ← dist[u, x, k − 1] + w(x→v)
```

This algorithm was first sketched by Shimbel in 1955; in fact, this algorithm is just running V different instances of Shimbel's single-source algorithm, one for each possible source vertex. Just as in the dynamic programming development of Shimbel's single-source algorithm, we don't actually need the inner loop over vertices v, and we only need a two-dimensional table. After the kth iteration of the main loop in the following algorithm, $dist[u, v]$ lies between the true shortest path distance from u to v and the value $dist[u, v, k]$ computed in the previous algorithm.

```
ShimbelAPSP(V, E, w):
  for all vertices u
      for all vertices v
          if u = v
              dist[u, v] ← 0
          else
              dist[u, v] ← ∞

  for k ← 1 to V − 1
      for all vertices u
          for all edges x→v
              if dist[u, v] > dist[u, x] + w(x→v)
                  dist[u, v] ← dist[u, x] + w(x→v)
```

27.5 Divide and Conquer

But we can make a more significant improvement. The recurrence we just used broke the shortest path into a slightly shorter path and a single edge, by considering all predecessors. Instead, let's break it

into two shorter paths at the *middle* vertex of the path. This idea gives us a different recurrence for $dist(u, v, k)$. Once again, to simplify things, let's assume $w(v \to v) = 0$.

$$dist(u, v, k) = \begin{cases} w(u \to v) & \text{if } k = 1 \\ \min_x \big(dist(u, x, k/2) + dist(x, v, k/2) \big) & \text{otherwise} \end{cases}$$

This recurrence only works when k is a power of two, since otherwise we might try to find the shortest path with a fractional number of edges! But that's not really a problem, since $dist(u, v, 2^{\lceil \lg V \rceil})$ gives us the overall shortest distance from u to v. Notice that we use the base case $k = 1$ instead of $k = 0$, since we can't use half an edge.

Once again, a dynamic programming solution is straightforward. Even before we write down the algorithm, we can tell the running time is $\mathbf{\Theta(V^3 \log V)}$—we consider V possible values of u, v, and x, but only $\lceil \lg V \rceil$ possible values of k.

```
FastDynamicProgrammingAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)
  for i ← 1 to ⌈lg V⌉                ⟨⟨k = 2^i⟩⟩
      for all vertices u
          for all vertices v
              dist[u, v, i] ← ∞
              for all vertices x
                  if dist[u, v, i] > dist[u, x, i−1] + dist[x, v, i−1]
                      dist[u, v, i] ← dist[u, x, i−1] + dist[x, v, i−1]
```

This algorithm is ***not*** the same as V invocations of any single-source algorithm; in particular, the innermost loop does not simply relax tense edges. However, we can remove the last dimension of the table, using $dist[u, v]$ everywhere in place of $dist[u, v, i]$, just as in Shimbel's single-source algorithm, thereby reducing the space from $O(V^3)$ to $O(V^2)$.

```
FastShimbelAPSP(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)
  for i ← 1 to ⌈lg V⌉
      for all vertices u
          for all vertices v
              for all vertices x
                  if dist[u, v] > dist[u, x] + dist[x, v]
                      dist[u, v] ← dist[u, x] + dist[x, v]
```

This faster algorithm was discovered by Leyzorek *et al.* in 1957, in the same paper where they describe Dijkstra's algorithm.

27.6 Aside: 'Funny' Matrix Multiplication

There is a very close connection (first observed by Shimbel, and later independently by Bellman) between computing shortest paths in a directed graph and computing powers of a square matrix. Compare the following algorithm for multiplying two $n \times n$ matrices A and B with the inner loop of our first dynamic programming algorithm. (I've changed the variable names in the second algorithm slightly to make the similarity clearer.)

```
MATRIXMULTIPLY(A, B):
  for i ← 1 to n
      for j ← 1 to n
          C[i, j] ← 0
          for k ← 1 to n
              C[i, j] ← C[i, j] + A[i, k] · B[k, j]
```

```
APSPINNERLOOP:
  for all vertices u
      for all vertices v
          D'[u, v] ← ∞
          for all vertices x
              D'[u, v] ← min {D'[u, v], D[u, x] + w[x, v]}
```

The *only* difference between these two algorithms is that we use addition instead of multiplication and minimization instead of addition. For this reason, the shortest path inner loop is often referred to as 'funny' matrix multiplication.

DYNAMICPROGRAMMINGAPSP is the standard iterative algorithm for computing the $(V-1)$th 'funny power' of the weight matrix w. The first set of for loops sets up the 'funny identity matrix', with zeros on the main diagonal and infinity everywhere else. Then each iteration of the second main for loop computes the next 'funny power'. FASTDYNAMICPROGRAMMINGAPSP replaces this iterative method for computing powers with repeated squaring, exactly like we saw at the beginning of the semester. The fast algorithm is simplified slightly by the fact that unless there are negative cycles, every 'funny power' after the Vth is the same.

There are faster methods for multiplying matrices, similar to Karatsuba's divide-and-conquer algorithm for multiplying integers. (Google for 'Strassen's algorithm'.) Unfortunately, these algorithms us subtraction, and there's no 'funny' equivalent of subtraction. (What's the inverse operation for min?) So at least for general graphs, there seems to be no way to speed up the inner loop of our dynamic programming algorithms.

Fortunately, this isn't true. There a beautiful randomized algorithm, discovered by Alon, Galil, Margalit, and Noar[1], that computes all-pairs shortest paths in undirected graphs in $O(M(V)\log^2 V)$ expected time, where $M(V)$ is the time to multiply two $V \times V$ integer matrices. A simplified version of this algorithm for *unweighted* graphs was discovered by Seidel.[2]

27.7 Floyd-(Roy-Kleene-)Warshall

Our fast dynamic programming algorithm is still a factor of $O(\log V)$ slower than Johnson's algorithm. A different formulation that removes this logarithmic factor was proposed in 1962 by Robert Floyd, slightly generalizing an algorithm of Stephen Warshall published earlier in the same year. (In fact, Warshall's algorithm was independently discovered by Bernard Roy in 1959, but the underlying technique was used even earlier by Stephen Kleene[3] in 1951.) Warshall's (and Roy's and Kleene's) insight was to use a different third parameter in the dynamic programming recurrence.

Number the vertices arbitrarily from 1 to V. For every pair of vertices u and v and every integer r, we define a path $\pi(u, v, r)$ as follows:

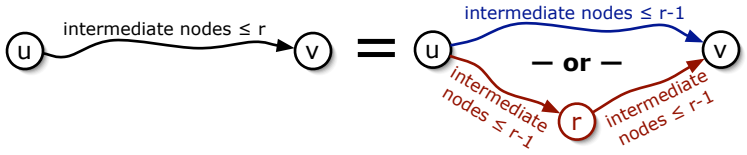
[1]Noga Alon, Zvi Galil, Oded Margalit*, and Moni Naor. Witnesses for Boolean matrix multiplication and for shortest paths. *Proc. 33rd FOCS* 417-426, 1992. See also Noga Alon, Zvi Galil, Oded Margalit*. On the exponent of the all pairs shortest path problem. *Journal of Computer and System Sciences* 54(2):255–262, 1997.

[2]Raimund Seidel. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of Computer and System Sciences*, 51(3):400-403, 1995. This is one of the few algorithms papers where (in the conference version at least) the algorithm is completely described and analyzed *in the abstract* of the paper.

[3]Pronounced "clay knee", not "clean" or "clean-ee" or "clay-nuh" or "dimaggio".

$\pi(u, v, r)$:= the shortest path from u to v where every intermediate vertex (that is, every vertex except u and v) is numbered at most r.

If $r = 0$, we aren't allowed to use any intermediate vertices, so $\pi(u, v, 0)$ is just the edge (if any) from u to v. If $r > 0$, then either $\pi(u, v, r)$ goes through the vertex numbered r, or it doesn't. If $\pi(u, v, r)$ does contain vertex r, it splits into a subpath from u to r and a subpath from r to v, where every intermediate vertex in these two subpaths is numbered at most $r - 1$. Moreover, the subpaths are as short as possible with this restriction, so they must be $\pi(u, r, r-1)$ and $\pi(r, v, r-1)$. On the other hand, if $\pi(u, v, r)$ does not go through vertex r, then every intermediate vertex in $\pi(u, v, r)$ is numbered at most $r - 1$; since $\pi(u, v, r)$ must be the *shortest* such path, we have $\pi(u, v, r) = \pi(u, v, r-1)$.



Recursive structure of the restricted shortest path $\pi(u, v, r)$.

This recursive structure implies the following recurrence for the length of $\pi(u, v, r)$, which we will denote by $dist(u, v, r)$:

$$dist(u, v, r) = \begin{cases} w(u \to v) & \text{if } r = 0 \\ \min\left\{dist(u, v, r-1),\ dist(u, r, r-1) + dist(r, v, r-1)\right\} & \text{otherwise} \end{cases}$$

We need to compute the shortest path distance from u to v with no restrictions, which is just $dist(u, v, V)$. Once again, we should immediately see that a dynamic programming algorithm will implement this recurrence in $\boldsymbol{\Theta(V^3)}$ time.

```
FLOYDWARSHALL(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v, 0] ← w(u→v)

  for r ← 1 to V
      for all vertices u
          for all vertices v
              if dist[u, v, r-1] < dist[u, r, r-1] + dist[r, v, r-1]
                  dist[u, v, r] ← dist[u, v, r-1]
              else
                  dist[u, v, r] ← dist[u, r, r-1] + dist[r, v, r-1]
```

Just like our earlier algorithms, we can simplify the algorithm by removing the third dimension of the memoization table. Also, because the vertex numbering was chosen arbitrarily, there's no reason to refer to it explicitly in the pseudocode.

```
FLOYDWARSHALL2(V, E, w):
  for all vertices u
      for all vertices v
          dist[u, v] ← w(u→v)

  for all vertices r
      for all vertices u
          for all vertices v
              if dist[u, v] > dist[u, r] + dist[r, v]
                  dist[u, v] ← dist[u, r] + dist[r, v]
```

Now compare this algorithm with FASTSHIMBELAPSP. Instead of $O(\log V)$ passes through all triples of vertices, FLOYDWARSHALL2 only requires a single pass, but only because it uses a different nesting order for the three for-loops!

27.8 Converting DFAs to regular expressions

Floyd's algorithm is a special case of a more general method for solving problems involving paths between vertices in graphs. The earliest example (that I know of) of this technique is an 1951 algorithm of Stephen Kleene to convert a deterministic finite automaton into an equivalent regular expression.

Recall that a deterministic finite automaton (DFA) formally consists of the following components:

- A finite set Σ, called the ***alphabet***, and whose elements we call ***symbols***.
- A finite set Q, whose elements are called ***states***.
- An ***initial state*** $s \in Q$.
- A subset $A \subseteq Q$ of ***accepting states***.
- A ***transition function*** $\delta : Q \times \Sigma \to Q$.

The ***extended transition function*** $\delta^* : Q \times \Sigma^* \to Q$ is recursively defined as follows:

$$\delta^*(q, w) := \begin{cases} q & \text{if } w = \varepsilon, \\ \delta^*(\delta(q, a), x) & \text{if } w = ax \text{ for some } a \in \Sigma \text{ and } x \in \Sigma^*. \end{cases}$$

Finally, a DFA ***accepts*** a string $w \in \Sigma^*$ if and only if $\delta^*(s, w) \in A$.

Equivalently, a DFA is a directed (multi-)graph with labeled edges whose vertices are the states, such that each vertex (state) has exactly one outgoing edge (transition) labeled with each symbol in Σ. There is a special "start" vertex s, and a subset A of the vertices are marked as "accepting". For any string $w \in \Sigma^*$, there is a unique walk starting at s whose sequence of edge labels is w. The DFA accepts w if and only if this walk ends at a state in A.

Kleene described the following algorithm to convert DFAs into equivalent regular expressions. Suppose we are given a DFA M with n states, where (without loss of generality) each state is identified by an integer between 1 and n. Let $\boldsymbol{L(i, j, r)}$ denote the set of all strings that describe walks in M that start at state i and end at state j, such that every intermediate state has index at most r. Thus, the language accepted by M is precisely

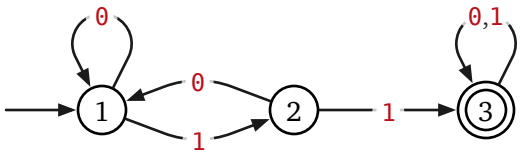
$$L(M) = \bigcup_{q \in A} L(s, q, n).$$

We prove inductively that every language $L(i, j, r)$ is regular, by recursively constructing a regular expression $R(i, j, r)$ that represents $L(i, j, r)$. There are two cases to consider.

- First, suppose $r = 0$. The language $L(i, j, 0)$ contains the labels walks from state i to state j that do not pass through *any* intermediate states. Thus, every string in $L(i, j, 0)$ has length at most 1. Specifically, for any symbol $a \in \Sigma$, we have $a \in L(i, j, 0)$ if and only if $\delta(i, a) = j$, and we have $\varepsilon \in L(i, j, 0)$ if and only if $i = j$. Thus, $L(i, j, 0)$ is always finite, and therefore regular.

  For example, the DFA sown on the next page defines the following regular languages $L(i, j, 0)$.

$$\begin{array}{lll} R[1,1,0] = \varepsilon + \texttt{0} & R[2,1,0] = \texttt{0} & R[3,1,0] = \varnothing \\ R[1,2,0] = \texttt{1} & R[2,2,0] = \varepsilon & R[3,2,0] = \varnothing \\ R[1,3,0] = \varnothing & R[2,3,0] = \texttt{1} & R[3,3,0] = \varepsilon + \texttt{0} + \texttt{1} \end{array}$$

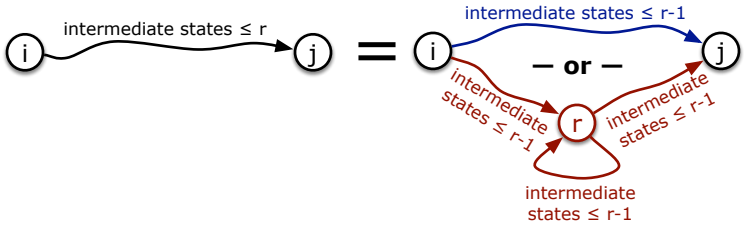


An example DFA

- Now suppose $r > 0$. Each string $w \in L(i,j,r)$ describes a walk from state i to state j where every intermediate state has index at most r. If this walk does not pass through state r, then $w \in L(i,j,r-1)$ by definition. Otherwise, we can split w into a sequence of substrings $w = w_1 \cdot w_2 \cdots w_\ell$ at the points where the walk visits state r. These substrings have the following properties:
  - The prefix w_1 describes a walk from state i to state r and thus belongs to $L(i,r,r-1)$.
  - The suffix w_ℓ describes a walk from state r to state j and thus belongs to $L(r,j,r-1)$.
  - For every other index k, the substring w_k describes a walk from state r to state r and thus belongs to $L(r,r,r-1)$.

  We conclude that

$$L(i,j,r) \;=\; L(i,j,r-1) \;\cup\; L(i,r,r-1) \bullet L(r,r,r-1)^* \bullet L(r,j,r-1).$$



Recursive structure of the regular language $L(i,j,r)$.

Putting these pieces together, we can recursively define a regular ***expression*** $R(i,j,r)$ that describes the language $L(i,j,r)$, as follows:

$$R(i,j,r) := \begin{cases} \varepsilon + \sum_{\delta(i,a)=j} a & \text{if } r = 0 \text{ and } i = j \\ \sum_{\delta(i,a)=j} a & \text{if } r = 0 \text{ and } i \neq j \\ R(i,j,r-1) + R(i,r,r-1) \bullet R(r,r,r-1)^* \bullet R(r,j,r-1) & \text{otherwise} \end{cases}$$

Kleene's algorithm evaluates this recurrence bottom-up using the natural dynamic programming algorithm. We memoize the previous recurrence into a three-dimensional array $R[1..n, 1..n, 0..n]$, which we traverse by increasing r in the outer loop, and in arbitrary order in the inner two loops.

```
KLEENE(Σ, n, δ, F):
  ⟨⟨Base cases⟩⟩
  for i ← 1 to n
      for j ← 1 to n
          if i = j then R[i, j, 0] ← ε else R[i, j, 0] ← ∅
          for all symbols a ∈ Σ
              if δ[i, a] = j
                  R[i, j, 0] ← R[i, j, 0] + a
  ⟨⟨Recursive cases⟩⟩
  for r ← 1 to n
      for i ← 1 to n
          for j ← 1 to n
              R[i, j, r] ← R[i, j, r − 1] + R[i, r, r − 1] • R[r, r, r − 1]* • R[r, j, r − 1]
  ⟨⟨Assemble the final result⟩⟩
  R ← ∅
  for q ← 0 to n − 1
      if q ∈ F
          R ← R + R[1, q, n − 1]
      return R
```

For purposes of analysis, let's assume the alphabet Σ has constant size. Assuming each alternation ($+$), concatenation ($\bullet$), and Kleene closure (*) operation requires constant time, the entire algorithm runs in $\boldsymbol{O(n^3)}$ ***time***.

However, regular expressions over an alphabet Σ are normally represented either as standard strings (arrays) over the larger alphabet $\Sigma \cup \{+, \bullet, *, (,), \varepsilon\}$, or as regular expression *trees*, whose internal nodes are $+$, $\bullet$, and $*$ operators and whose leaves are symbols and εs. In either representation, the regular expressions in Kleene's algorithm grow in size by roughly a factor of 4 in each iteration of the outer loop, at least in the worst case. Thus, in the worst case, each regular expression $R[i, j, r]$ has size $O(4^r)$, the size of the final output expression is $O(4^n n)$, and entire algorithm runs in $\boldsymbol{O(4^n n^2)}$ ***time***.

So we shouldn't do this. After all, the running time is exponential, and exponential time is bad. Right? Moreover, this exponential dependence is unavoidable; Hermann Gruber and Markus Holzer proved in 2008[4] that there are n-state DFAs over the binary alphabet $\{0, 1\}$ such that any equivalent regular expression has length $2^{\Omega(n)}$.

Well, maybe it's not so bad. The output regular expression has exponential size *because it contains multiple copies of the same subexpressions*; similarly, the regular expression tree has exponential size *because it contains multiples copies of several subtrees*. But it's precisely this exponential behavior that we use dynamic programming to avoid! In fact, it's not hard to modify Kleene's algorithm to compute a ***regular expression dag*** of size $O(n^3)$, ***in*** $\boldsymbol{O(n^3)}$ ***time***, that (intuitively) contains each subexpression $R[i, j, r]$ only once. This regular expression dag has exactly the same relationship to the regular expression *tree* as the dependency graph of Kleene's algorithm has to the recursion tree of its underlying recurrence.

Exercises

1. All of the algorithms discussed in this lecture fail if the graph contains a negative cycle. Johnson's algorithm detects the negative cycle in the initialization phase (via Shimbel's algorithm) and aborts; the dynamic programming algorithms just return incorrect results. However, all of these algorithms can be modified to return correct shortest-path distances, even in the presence of negative cycles.

[4] Hermann Gruber and Markus Holzer. Finite automata, digraph connectivity, and regular expression size. *Proc. 35th ICALP*, 39–50, 2008.

Specifically, if there is a path from vertex u to a negative cycle and a path from that negative cycle to vertex v, the algorithm should report that $dist[u, v] = -\infty$. If there is no directed path from u to v, the algorithm should return $dist[u, v] = \infty$. Otherwise, $dist[u, v]$ should equal the length of the shortest directed path from u to v.

(a) Describe how to modify Johnson's algorithm to return the correct shortest-path distances, even if the graph has negative cycles.

(b) Describe how to modify the Floyd-Warshall algorithm (FLOYDWARSHALL2) to return the correct shortest-path distances, even if the graph has negative cycles.

2. All of the shortest-path algorithms described in this note can also be modified to return an explicit description of some negative cycle, instead of simply reporting that a negative cycle exists.

   (a) Describe how to modify Johnson's algorithm to return either the matrix of shortest-path distances or a negative cycle.

   (b) Describe how to modify the Floyd-Warshall algorithm (FLOYDWARSHALL2) to return either the matrix of shortest-path distances or a negative cycle.

   If the graph contains more than one negative cycle, your algorithms may choose one arbitrarily.

3. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero. Suppose the vertices of G are partitioned into k disjoint subsets $V_1, V_2, \ldots, V_k$; that is, every vertex of G belongs to exactly one subset V_i. For each i and j, let $\delta(i, j)$ denote the minimum shortest-path distance between vertices in V_i and vertices in V_j:

   $$\delta(i, j) = \min \left\{ dist(u, v) \;\middle|\; u \in V_i \text{ and } v \in V_j \right\}.$$

   Describe an algorithm to compute $\delta(i, j)$ for all i and j in time $O(V^2 + kE \log E)$.

4. Let $G = (V, E)$ be a directed graph with weighted edges; edge weights could be positive, negative, or zero.

   (a) How could we delete an arbitrary vertex v from this graph, without changing the shortest-path distance between any other pair of vertices? Describe an algorithm that constructs a directed graph $G' = (V', E')$ with weighted edges, where $V' = V \setminus \{v\}$, and the shortest-path distance between any two nodes in H is equal to the shortest-path distance between the same two nodes in G, in $O(V^2)$ time.

   (b) Now suppose we have already computed all shortest-path distances in G'. Describe an algorithm to compute the shortest-path distances from v to every other vertex, and from every other vertex to v, in the original graph G, in $O(V^2)$ time.

   (c) Combine parts (a) and (b) into another all-pairs shortest path algorithm that runs in $O(V^3)$ time. (The resulting algorithm is *not* the same as Floyd-Warshall!)

5. In this problem we will discover how you, too, can be employed by Wall Street and cause a major economic collapse! The *arbitrage* business is a money-making scheme that takes advantage of differences in currency exchange. In particular, suppose that 1 US dollar buys 120 Japanese yen; 1 yen buys 0.01 euros; and 1 euro buys 1.2 US dollars. Then, a trader starting with \$1 can convert his money from dollars to yen, then from yen to euros, and finally from euros back to dollars, ending with \$1.44! The cycle of currencies $\$ \to ¥ \to € \to \$$ is called an ***arbitrage cycle***. Of course, finding and exploiting arbitrage cycles before the prices are corrected requires extremely fast algorithms.

   Suppose n different currencies are traded in your currency market. You are given the matrix $R[1..n, 1..n]$ of exchange rates between every pair of currencies; for each i and j, one unit of currency i can be traded for $R[i, j]$ units of currency j. (Do *not* assume that $R[i, j] \cdot R[j, i] = 1$.)

   (a) Describe an algorithm that returns an array $V[1..n]$, where $V[i]$ is the maximum amount of currency i that you can obtain by trading, starting with one unit of currency 1, assuming there are no arbitrage cycles.

(b) Describe an algorithm to determine whether the given matrix of currency exchange rates creates an arbitrage cycle.

(c) Modify your algorithm from part (b) to actually return an arbitrage cycle, if it exists.

*6. Let $G = (V, E)$ be an undirected, unweighted, connected, n-vertex graph, represented by the adjacency matrix $A[1..n, 1..n]$. In this problem, we will derive Seidel's sub-cubic algorithm to compute the $n \times n$ matrix $D[1..n, 1..n]$ of shortest-path distances using fast matrix multiplication. Assume that we have a subroutine MATRIXMULTIPLY that multiplies two $n \times n$ matrices in $\Theta(n^\omega)$ time, for some unknown constant $\omega \geq 2$.[5]

(a) Let G^2 denote the graph with the same vertices as G, where two vertices are connected by a edge if and only if they are connected by a path of length at most 2 in G. Describe an algorithm to compute the adjacency matrix of G^2 using a single call to MATRIXMULTIPLY and $O(n^2)$ additional time.

(b) Suppose we discover that G^2 is a complete graph. Describe an algorithm to compute the matrix D of shortest path distances in $O(n^2)$ additional time.

(c) Let D^2 denote the (recursively computed) matrix of shortest-path distances in G^2. Prove that the shortest-path distance from node i to node j is either $2 \cdot D^2[i, j]$ or $2 \cdot D^2[i, j] - 1$.

(d) Suppose G^2 is not a complete graph. Let $X = D^2 \cdot A$, and let $\deg(i)$ denote the degree of vertex i in the original graph G. Prove that the shortest-path distance from node i to node j is $2 \cdot D^2[i, j]$ if and only if $X[i, j] \geq D^2[i, j] \cdot \deg(i)$.

(e) Describe an algorithm to compute the matrix of shortest-path distances in G in $O(n^\omega \log n)$ time.

[5] The matrix multiplication algorithm you already know runs in $\Theta(n^3)$ time, but this is not the fastest algorithm known. The current record is $\omega \approx 2.3727$, due to Virginia Vassilevska Williams. Determining the smallest possible value of ω is a long-standing open problem; many people believe there is an undiscovered $O(n^2)$-time algorithm for matrix multiplication.

Caveat lector: This is the zeroth (draft) edition of this lecture note. In particular, some topics still need to be written. Please send bug reports and suggestions to jeffe@illinois.edu.

Think globally, act locally.

— Attributed to Patrick Geddes (c.1915), among many others.

We can only see a short distance ahead,
but we can see plenty there that needs to be done.

— Alan Turing, "Computing Machinery and Intelligence" (1950)

Never worry about theory as long as the machinery does what it's supposed to do.

— Robert Anson Heinlein, *Waldo & Magic, Inc.* (1950)

36 Turing Machines

In 1936, a few months before his 24th birthday, Alan Turing launched computer science as a modern intellectual discipline. In a single remarkable paper, Turing provided the following results:

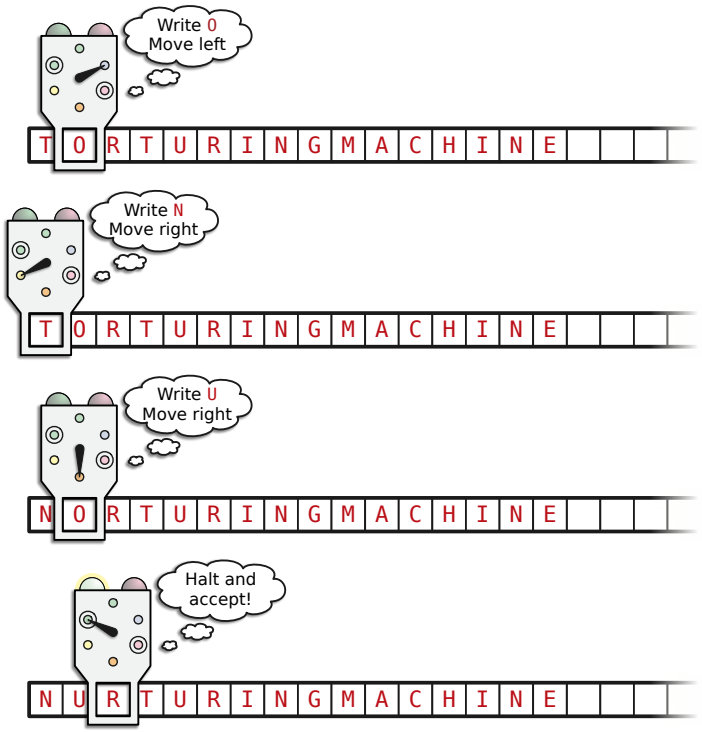
- A simple formal model of mechanical computation now known as *Turing machines*.
- A description of a single *universal* machine that can be used to compute *any* function computable by *any* other Turing machine.
- A proof that no Turing machine can solve the *halting problem*—Given the formal description of an arbitrary Turing machine M, does M halt or run forever?
- A proof that no Turing machine can determine whether an arbitrary given proposition is provable from the axioms of first-order logic. This Hilbert and Ackermann's famous *Entscheidungsproblem* ("decision problem")
- Compelling arguments[1] that his machines can execute *arbitrary* "calculation by finite means".

Turing's paper was not the first to prove that the *Entscheidungsproblem* had no algorithmic solution. Alonzo Church published the first proof just a new months earlier, using a very different model of computation, now called the ***untyped λ-calculus***. Turing and Church developed their results independently; indeed, Turing rushed the submission of his own paper immediately after receiving a copy of Church's paper, pausing only long enough to prove that any function computable via λ-calculus can also be computed by a Turing machine and vice versa. Church was the referee for Turing's paper; between the paper's submission and its acceptance, Turing was admitted to Princeton, where he became Church's PhD student. He finished his PhD two years later.

Informally, Turing described a device with a finite number of ***internal states*** that has access to memory in the form of a ***tape***. The tape consists of a semi-infinite sequence of ***cells***, each containing a single symbol from some arbitrary finite alphabet. The Turing machine can access the tape only through its ***head***, which is positioned over a single cell. Initially, the tape contains an arbitrary finite ***input string*** followed by an infinite sequence of ***blanks***, and the head is positioned over the first cell on the tape. In a single iteration, the machine reads the symbol in that cell, possibly write a new symbol into that cell,

[1] As Turing put it, "All arguments which can be given are bound to be, fundamentally, appeals to intuition, and for this reason rather unsatisfactory mathematically." The claim that anything that can be computed can be computing using Turing machines is now known as the ***Church-Turing thesis***.

possibly changes its internal state, possibly moves the head to a neighboring cell, and possibly halts. The precise behavior of the machine at each iteration is entirely determined by its internal state and the symbol that it reads. When the machine halts, it indicates whether it has ***accepted*** or ***rejected*** the original input string.



A few iterations of a six-state Turing machine.

36.1 Why Bother?

Students used to thinking of computation in terms of higher-level operations like random memory accesses, function calls, and recursion may wonder why we should even consider a model as simple and constrained as Turing machines. Admittedly, Turing machines are a terrible model for thinking about ***fast*** computation; simple operations that take constant time in the standard random-access model can require *arbitrarily* many steps on a Turing machine. Worse, seemingly minor variations in the precise definition of "Turing machine" can have significant impact on problem complexity. As a simple example (which will make more sense later), we can reverse a string of n bits in $O(n)$ time using a two-tape Turing machine, but the same task provably requires $\Omega(n^2)$ time on a single-tape machine.

But here we are not interested in finding *fast* algorithms, or indeed in finding algorithms at all, but rather in proving that some problems cannot be solved by *any* computational means. Such a bold claim requires a formal definition of "computation" that is simple enough to support formal argument, but still powerful enough to describe arbitrary algorithms. Turing machines are ideal for this purpose. In particular, Turing machines are powerful enough to *simulate other Turing machines*, while still simple enough to let us build up this self-simulation from scratch, unlike more complex but efficient models like the standard random-access machine

(Arguably, self-simulation is even simpler in Church's λ-calculus, or in Schönfinkel and Curry's combinator calculus, which is one of many reasons those models are more common in the design and

analysis of programming languages than Turing machines. Those models much more abstract; in particular, they are harder to show equivalent to standard iterative models of computation.)

36.2 Formal Definitions

Formally, a Turing machine consists of the following components. (Hang on; it's a long list.)

- An arbitrary finite set Γ with at least two elements, called the ***tape alphabet***.
- An arbitrary symbol $\square \in \Gamma$, called the ***blank symbol*** or just the ***blank***.
- An arbitrary nonempty subset $\Sigma \subseteq (\Gamma \setminus \{\square\})$, called the ***input alphabet***.
- Another arbitrary finite set Q whose elements are called ***states***.
- Three distinct special states $\mathsf{start}, \mathsf{accept}, \mathsf{reject} \in Q$.
- A ***transition*** function $\delta : (Q \setminus \{\mathsf{accept}, \mathsf{reject}\}) \times \Gamma \to Q \times \Gamma \times \{-1, +1\}$.

A ***configuration*** or ***global state*** of a Turing machine is represented by a triple $(q, x, i) \in Q \times \Gamma^* \times \mathbb{N}$, indicating that the machine's internal state is q, the tape contains the string x followed by an infinite sequence of blanks, and the head is located at position i. Trailing blanks in the tape string are ignored; the triples (q, x, i) and $(q, x\square, i)$ describe exactly the same configuration.

The transition function δ describes the evolution of the machine. For example, $\delta(q, a) = (p, b, -1)$ means that when the machine reads symbol a in state q, it changes its internal state to p, writes symbol b onto the tape at its current location (replacing a), and then decreases its position by 1 (or more intuitively, moves one step to the left). If the position of the head becomes negative, no further transitions are possible, and the machine ***crashes***.

We write $\boldsymbol{(p, x, i) \Rightarrow_M (q, y, j)}$ to indicate that Turing machine M transitions from the first configuration to the second in one step. (The symbol $\Rightarrow$ is often pronounced "yields"; I will omit the subscript M if the machine is clear from context.) For example, $\delta(p, a) = (q, b, \pm 1)$ means that

$$(p, xay, i) \;\Rightarrow\; (q, xby, i \pm 1)$$

for any non-negative integer i, any string x of length i, and any string y. The evolution of any Turing machine is ***deterministic***; each configuration C yields a *unique* configuration C'. We write $\boldsymbol{C \Rightarrow^* C'}$ to indicate that there is a (possibly empty) sequence of transitions from configuration C to configuration C'. (The symbol $\Rightarrow^*$ can be pronounced "eventually yields".)

The initial configuration is $(w, \mathsf{start}, 0)$ for some arbitrary (and possibly empty) ***input string*** $w \in \Sigma^*$. If M eventually reaches the accept state—more formally, if $(w, \mathsf{start}, 0) \Rightarrow^* (x, \mathsf{accept}, i)$ for some string $x \in \Gamma^*$ and some integer i—we say that M ***accepts*** the original input string w. Similarly, if M eventually reaches the reject state, we say that M ***rejects*** w. We must emphasize that "rejects" and "does not accept" are *not* synonyms; if M crashes or runs forever, then M neither accepts nor rejects w.

We distinguish between two different senses in which a Turing machine can "accept" a language. Let M be a Turing machine with input alphabet Σ, and let $L \subseteq \Sigma^*$ be an arbitrary language over Σ.

- M ***recognizes*** or ***accepts*** L if and only if M accepts every string in L but nothing else. A language is ***recognizable*** (or *semi-computable* or *recursively enumerable*) if it is recognized by some Turing machine.
- M ***decides*** L if and only if M accepts every string in L and rejects every string in $\Sigma^* \setminus L$. Equivalently, M decides L if and only if M recognizes L and halts (without crashing) on all inputs. A language is ***decidable*** (or *computable* or *recursive*) if it is decided by some Turing machine.

Trivially, every decidable language is recognizable, but (as we will see later), not every recognizable language is decidable.

36.3 A First Example

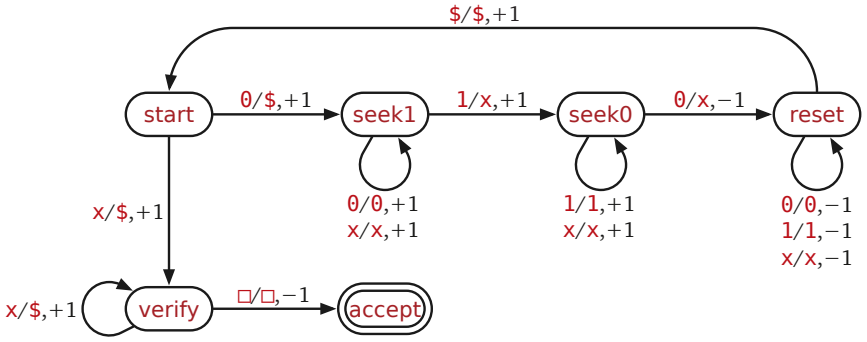
Consider the language $L = \{0^n1^n0^n \mid n \geq 0\}$. This language is neither regular nor context-free, but it can be decided by the following six-state Turing machine. The alphabets and states of the machine are defined as follows:

$$\Gamma = \{0, 1, \$, x, \square\}$$
$$\Sigma = \{0, 1\}$$
$$Q = \{\text{start}, \text{seek1}, \text{seek0}, \text{reset}, \text{verify}, \text{accept}, \text{reject}\}$$

The transition function is described in the following table; all unspecified transitions lead to the reject state. We also give a graphical representation of the same machine, which resembles a drawing of a DFA, but with output symbols and actions specified on each edge. For example, we indicate the transition $\delta(p, 0) = (q, 1, +1)$ by writing $0/1, +1$ next to the arrow from state p to state q.

$\delta(p, a) = (q, b, \Delta)$	explanation
$\delta($start$, 0) = ($seek1$, \$, +1)$	mark first 0 and scan right
$\delta($start$, x) = ($verify$, \$, +1)$	looks like we're done, but let's make sure
$\delta($seek1$, 0) = ($seek1$, 0, +1)$	scan rightward for 1
$\delta($seek1$, x) = ($seek1$, x, +1)$	
$\delta($seek1$, 1) = ($seek0$, x, +1)$	mark 1 and continue right
$\delta($seek0$, 1) = ($seek0$, 1, +1)$	scan rightward for 0
$\delta($seek0$, x) = ($seek0$, x, +1)$	
$\delta($seek0$, 0) = ($reset$, x, +1)$	mark 0 and scan left
$\delta($reset$, 0) = ($reset$, 0, -1)$	scan leftward for $
$\delta($reset$, 1) = ($reset$, 1, -1)$	
$\delta($reset$, x) = ($reset$, x, -1)$	
$\delta($reset$, \$) = ($start$, \$, +1)$	step right and start over
$\delta($verify$, x) = ($verify$, \$, +1)$	scan right for any unmarked symbol
$\delta($verify$, \square) = ($accept$, \square, -1)$	success!

The transition function for a Turing machine that decides the language $\{0^n1^n0^n \mid n \geq 0\}$.



A graphical representation of the example Turing machine

Finally, we trace the execution of this machine on two input strings: $001100 \in L$ and $00100 \notin L$. In each configuration, we indicate the position of the head using a small triangle instead of listing the position explicitly. Notice that we automatically add blanks to the tape string as necessary. Proving

that this machine actually decides L—and in particular, that it never crashes or infinite-loops—is a straightforward but tedious exercise in induction.

$$\begin{aligned}
&(\text{start}, \underset{\blacktriangle}{0}01100) \Rightarrow (\text{seek1}, \$\underset{\blacktriangle}{0}1100) \Rightarrow (\text{seek1}, \$\underset{\blacktriangle}{0}1100) \Rightarrow (\text{seek0}, \$0\underset{\blacktriangle}{1}00) \Rightarrow (\text{seek0}, \$0\underset{\blacktriangle}{1}00) \\
&\quad \Rightarrow (\text{reset}, \$0\underset{\blacktriangle}{1}x0) \Rightarrow (\text{reset}, \$0\underset{\blacktriangle}{1}x0) \Rightarrow (\text{reset}, \$\underset{\blacktriangle}{0}1x0) \Rightarrow (\text{reset}, \$\underset{\blacktriangle}{0}1x0) \\
&\quad \Rightarrow (\text{start}, \$\underset{\blacktriangle}{0}1x0) \\
&\quad \Rightarrow (\text{seek1}, \$\$\underset{\blacktriangle}{x}1x0) \Rightarrow (\text{seek1}, \$\$\underset{\blacktriangle}{x}1x0) \Rightarrow (\text{seek0}, \$\$xx\underset{\blacktriangle}{0}) \Rightarrow (\text{seek0}, \$\$xx\underset{\blacktriangle}{0}) \\
&\quad \Rightarrow (\text{reset}, \$\$x\underset{\blacktriangle}{x}xx) \Rightarrow (\text{reset}, \$\$x\underset{\blacktriangle}{x}xx) \Rightarrow (\text{reset}, \$\$\underset{\blacktriangle}{x}xxx) \Rightarrow (\text{reset}, \$\$\underset{\blacktriangle}{x}xxx) \\
&\quad \Rightarrow (\text{verify}, \$\$\underset{\blacktriangle}{x}xxx) \Rightarrow (\text{verify}, \$\$\$\underset{\blacktriangle}{x}xx) \Rightarrow (\text{verify}, \$\$\$\$\underset{\blacktriangle}{x}x) \\
&\quad \Rightarrow (\text{verify}, \$\$\$\$\$\underset{\blacktriangle}{x}) \Rightarrow (\text{verify}, \$\$\$\$\$\$\underset{\blacktriangle}{\square}) \Rightarrow (\text{accept}, \$\$\$\$\$\$\underset{\blacktriangle}{\$}) \Rightarrow \textbf{accept!}
\end{aligned}$$

The evolution of the example Turing machine on the input string `001100` $\in L$

$$\begin{aligned}
&(\text{start}, \underset{\blacktriangle}{0}0100) \Rightarrow (\text{seek1}, \$\underset{\blacktriangle}{0}100) \Rightarrow (\text{seek1}, \$\underset{\blacktriangle}{0}100) \Rightarrow (\text{seek0}, \$0\underset{\blacktriangle}{0}0) \\
&\quad \Rightarrow (\text{reset}, \$0\underset{\blacktriangle}{x}x0) \Rightarrow (\text{reset}, \$\underset{\blacktriangle}{0}xx0) \Rightarrow (\text{reset}, \$\underset{\blacktriangle}{0}xx0) \\
&\quad \Rightarrow (\text{start}, \$\underset{\blacktriangle}{0}xx0) \\
&\quad \Rightarrow (\text{seek1}, \$\$\underset{\blacktriangle}{x}x0) \Rightarrow (\text{seek1}, \$\$x\underset{\blacktriangle}{x}0) \Rightarrow (\text{seek1}, \$\$xx\underset{\blacktriangle}{0}) \Rightarrow \textbf{reject!}
\end{aligned}$$

The evolution of the example Turing machine on the input string `00100` $\notin L$

36.4 Variations

There are actually several formal models that all fall under the name "Turing machine", each with small variations on the definition we've given. Although we do need to be explicit about *which* variant we want to use for any particular problem, the differences between the variants are relatively unimportant. For any machine defined in one model, there is an equivalent machine in each of the other models; in particular, all of these variants recognize the same languages and decide the same languages. For example:

- **Halting conditions.** Some models allow multiple accept and reject states, which (depending on the precise model) trigger acceptance or rejection either when the machine enters the state, or when the machine has no valid transitions out of such a state. Others include only explicit accept states, and either equate crashing with rejection or do not define a rejection mechanism at all. Still other models include halting as one of the possible *actions* of the machine, in addition to moving left or moving right; in these models, the machine accepts/rejects its input if and only if it halts in an accepting/non-accepting state.

- **Actions.** Some Turing machine models allow transitions that do not move the head, or that move the head by more than one cell in a single step. Others insist that a single step of the machine *either* writes a new symbol onto the tape *or* moves the head one step. Finally, as mentioned above, some models include halting as one of the available actions.

- **Transition function.** Some models of Turing machines, including Turing's original definition, allow the transition function to be undefined on some state-symbol pairs. In this formulation, the transition function is given by a set $\delta \subset Q \times \Gamma \times Q \times \Gamma \times \{+1, -1\}$, such that for each state q and symbol a, there is at most one transition $(q, a, \cdot, \cdot, \cdot) \in \delta$. If the machine enters a configuration from which there is no transition, it halts and (depending on the precise model) either crashes or rejects. Others define the transition function as $\delta : Q \times \Gamma \to Q \times (\Gamma \cup \{-1, +1\})$, allowing the machine to *either* write a symbol to the tape *or* move the head in each step.
- **Beginning of the tape.** Some models forbid the head to move past the beginning of the tape, either by starting the tape with a special symbol that cannot be overwritten and that forces a rightward transition, or by declaring that a leftward transition at position 0 leaves the head in position 0, or even by pure fiat—declaring any machine that performs a leftward move at position 0 to be invalid.

To prove that any two of these variant "species" of Turing machine are equivalent, we must show how to transform a machine of one species into a machine of the other species that accepts and rejects the same strings. For example, let $M = (\Gamma, \square, \Sigma, Q, s, \mathsf{accept}, \mathsf{reject}, \delta)$ be a Turing machine with explicit accept and reject states. We can define an equivalent Turing machine M' that halts only when it moves left from position 0, and accepts only by halting while in an accepting state, as follows. We define the set of accepting states for M' as $A = \{\mathsf{accept}\}$ and define a new transition function

$$\delta'(q, a) := \begin{cases} (\mathsf{accept}, a, -1) & \text{if } q = \mathsf{accept} \\ (\mathsf{reject}, a, -1) & \text{if } q = \mathsf{reject} \\ \delta(q, a) & \text{otherwise} \end{cases}$$

Similarly, suppose someone gives us a Turing machine $M = (\Gamma, \square, \Sigma, Q, s, \mathsf{accept}, \mathsf{reject}, \delta)$ whose transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{-1, 0, +1\}$ allows the machine to transition without moving its head. We can construct an equivalent Turing machine $M' = (\Gamma, \square, \Sigma, Q', s, \mathsf{accept}, \mathsf{reject}, \delta')$ that moves its head at every transition by defining $Q' := Q \times \{0, 1\}$ and

$$\delta'((p, 0), a) := \begin{cases} ((q, 1), b, +1) & \text{if } \delta(p, a) = (q, b, 0), \\ ((q, 0), b, \Delta) & \text{if } \delta(p, a) = (q, b, \Delta) \text{ and } \Delta \neq 0, \end{cases}$$

$$\delta'((p, 1), a) := ((p, 0), a, -1).$$

36.5 Computing Functions

Turing machines can also be used to compute functions from strings to strings, instead of just accepting or rejecting strings. Since we don't care about acceptance or rejection, we replace the explicit accept and reject states with a single halt state, and we define the ***output*** of the Turing machine to be the contents of the tape when the machine halts, after removing the infinite sequence of trailing blanks. More formally, for any Turing machine M, any string $w \in \Sigma^*$, and any string $x \in \Gamma^*$ that does not end with a blank, we write $\boldsymbol{M(w) = x}$ if and only if $(w, s, 0) \Rightarrow_M^* (x, \mathsf{halt}, i)$ for some integer i. If M does not halt on input w, then we write $\boldsymbol{M(w)} \nearrow$, which can be read either "M diverges on w" or "$M(w)$ is undefined." We say that M ***computes*** the function $f : \Sigma^* \to \Sigma^*$ if and only if $M(w) = f(w)$ for every string w.

36.5.1 Shifting

One basic operation that is used in many Turing machine constructions is ***shifting*** the input string a constant number of steps to the right or to the left. For example, given any input string $w \in \{0, 1\}^*$, we can

compute the string $\texttt{0}w$ using a Turing machine with tape alphabet $\Gamma = \{\texttt{0}, \texttt{1}, \square\}$, state set $Q = \{0, 1, \text{halt}\}$, start state 0, and the following transition function:

$$\begin{array}{l}
\delta(p,\ a) = (\ q\ ,\ b,\ \Delta\ ) \\
\hline
\delta(0,\ \texttt{0}) = (\ 0\ ,\ \texttt{0},\ +1) \\
\delta(0,\ \texttt{1}) = (\ 1\ ,\ \texttt{0},\ +1) \\
\delta(0,\ \square) = (\text{halt},\ \texttt{0},\ +1) \\
\hline
\delta(1,\ \texttt{0}) = (\ 0\ ,\ \texttt{1},\ +1) \\
\delta(1,\ \texttt{1}) = (\ 1\ ,\ \texttt{1},\ +1) \\
\delta(1,\ \square) = (\text{halt},\ \texttt{1},\ +1)
\end{array}$$

By increasing the number of states, we can build a Turing machine that shifts the input string any fixed number of steps in either direction. For example, a machine that shifts its input to the left by five steps might read the string from right to left, storing the five most recently read symbols in its internal state. A typical transition for such a machine would be $\delta(12345, 0) = (01234, \texttt{5}, -1)$.

36.5.2 Binary Addition

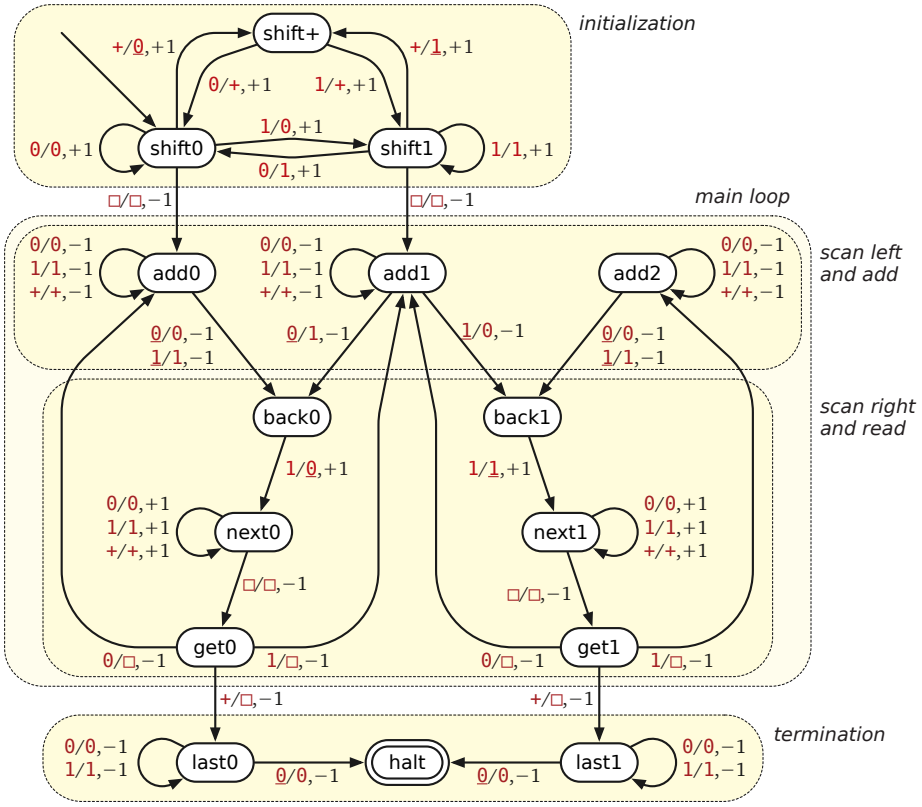
With a more complex Turing machine, we can implement binary addition. The input is a string of the form $w\texttt{+}x$, where $w, x \in \{\texttt{0}, \texttt{1}\}^n$, representing two numbers in binary; the output is the binary representation of $w + x$. To simplify our presentation, we assume that $|w| = |x| > 0$; however, this restrictions can be removed with the addition of a few more states. The following figure shows the entire Turing machine at a glance. The machine uses the tape alphabet $\Gamma = \{\square, \texttt{0}, \texttt{1}, \texttt{+}, \underline{\texttt{0}}, \underline{\texttt{1}}\}$; the start state is shift0. All missing transitions go to a fail state, indicating that the input was badly formed.

Execution of this Turing machine proceeds in several phases, each with its own subset of states, as indicated in the figure. The initialization phase scans the entire input, shifting it to the right to make room for the output string, marking the rightmost bit of w, and reading and erasing the last bit of x.

$$\begin{array}{l}
\delta(\ \ p\ \ ,\ a) = (\ \ q\ \ ,\ b,\ \Delta\ ) \\
\hline
\delta(\text{shift0},\ \texttt{0}) = (\text{shift0},\ \texttt{0},\ +1) \\
\delta(\text{shift0},\ \texttt{1}) = (\text{shift1},\ \texttt{0},\ +1) \\
\delta(\text{shift0},\ \texttt{+}) = (\text{shift+},\ \underline{\texttt{0}},\ +1) \\
\delta(\text{shift0},\ \square) = (\ \text{add0}\ ,\ \square,\ -1) \\
\hline
\delta(\text{shift1},\ \texttt{0}) = (\text{shift0},\ \texttt{1},\ +1) \\
\delta(\text{shift1},\ \texttt{1}) = (\text{shift1},\ \texttt{1},\ +1) \\
\delta(\text{shift1},\ \texttt{+}) = (\text{shift+},\ \underline{\texttt{1}},\ +1) \\
\delta(\text{shift1},\ \square) = (\ \text{add1}\ ,\ \square,\ -1) \\
\hline
\delta(\text{shift+},\ \texttt{0}) = (\text{shift0},\ \texttt{+},\ +1) \\
\delta(\text{shift+},\ \texttt{1}) = (\text{shift1},\ \texttt{+},\ +1)
\end{array}$$

The first part of the main loop scans left to the marked bit of w, adds the bit of x that was just erased plus the carry bit from the previous iteration, and records the carry bit for the next iteration in the machines internal state.

$$\begin{array}{l}
\delta(\ \ p\ \ ,\ a) = (\ \ q\ \ ,\ b,\ \Delta\ ) \\
\hline
\delta(\text{add0},\ \texttt{0}) = (\ \text{add0}\ ,\ \texttt{0},\ -1) \\
\delta(\text{add0},\ \texttt{1}) = (\ \text{add0}\ ,\ \texttt{0},\ -1) \\
\delta(\text{add0},\ \texttt{+}) = (\ \text{add0}\ ,\ \texttt{0},\ -1) \\
\delta(\text{add0},\ \underline{\texttt{0}}) = (\text{back0},\ \texttt{0},\ -1) \\
\delta(\text{add0},\ \underline{\texttt{1}}) = (\text{back0},\ \texttt{1},\ -1)
\end{array}
\qquad
\begin{array}{l}
\delta(\ \ p\ \ ,\ a) = (\ \ q\ \ ,\ b,\ \Delta\ ) \\
\hline
\delta(\text{add1},\ \texttt{0}) = (\ \text{add1}\ ,\ \texttt{0},\ -1) \\
\delta(\text{add1},\ \texttt{1}) = (\ \text{add1}\ ,\ \texttt{0},\ -1) \\
\delta(\text{add1},\ \texttt{+}) = (\ \text{add1}\ ,\ \texttt{0},\ -1) \\
\delta(\text{add1},\ \underline{\texttt{0}}) = (\text{back0},\ \texttt{1},\ -1) \\
\delta(\text{add1},\ \underline{\texttt{1}}) = (\text{back1},\ \texttt{0},\ -1)
\end{array}
\qquad
\begin{array}{l}
\delta(\ \ p\ \ ,\ a) = (\ \ q\ \ ,\ b,\ \Delta\ ) \\
\hline
\delta(\text{add2},\ \texttt{0}) = (\ \text{add2}\ ,\ \texttt{0},\ -1) \\
\delta(\text{add2},\ \texttt{1}) = (\ \text{add2}\ ,\ \texttt{0},\ -1) \\
\delta(\text{add2},\ \texttt{+}) = (\ \text{add2}\ ,\ \texttt{0},\ -1) \\
\delta(\text{add2},\ \underline{\texttt{0}}) = (\text{back1},\ \texttt{0},\ -1) \\
\delta(\text{add2},\ \underline{\texttt{1}}) = (\text{back1},\ \texttt{1},\ -1)
\end{array}$$



A Turing machine that adds two binary numbers of the same length.

The second part of the main loop marks the previous bit of w, scans right to the end of x, and then reads and erases the last bit of x, all while maintaining the carry bit.

$$\begin{array}{l|l}
\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta) & \delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta) \\
\hline
\delta(\text{back0},\ 0) = (\text{next0},\ \underline{0},\ +1) & \delta(\text{back1},\ 0) = (\text{next1},\ \underline{0},\ +1) \\
\delta(\text{back0},\ 1) = (\text{next0},\ \underline{1},\ +1) & \delta(\text{back1},\ 1) = (\text{next1},\ \underline{1},\ +1) \\
\hdashline
\delta(\text{next0},\ 0) = (\text{next0},\ 0,\ +1) & \delta(\text{next1},\ 0) = (\text{next1},\ 0,\ +1) \\
\delta(\text{next0},\ 1) = (\text{next0},\ 0,\ +1) & \delta(\text{next1},\ 1) = (\text{next1},\ 0,\ +1) \\
\delta(\text{next0},\ +) = (\text{next0},\ 0,\ +1) & \delta(\text{next1},\ +) = (\text{next1},\ 0,\ +1) \\
\delta(\text{next0},\ \square) = (\ \text{get0}\ ,\ \square,\ -1) & \delta(\text{next1},\ \square) = (\ \text{get1}\ ,\ \square,\ -1) \\
\hdashline
\delta(\ \text{get0}\ ,\ 0) = (\text{add0},\ \square,\ -1) & \delta(\ \text{get1}\ ,\ 0) = (\text{add1},\ \square,\ -1) \\
\delta(\ \text{get0}\ ,\ 1) = (\text{add1},\ \square,\ -1) & \delta(\ \text{get1}\ ,\ 1) = (\text{add2},\ \square,\ -1) \\
\delta(\ \text{get0}\ ,\ +) = (\text{last0},\ \square,\ -1) & \delta(\ \text{get1}\ ,\ +) = (\text{last1},\ \square,\ -1)
\end{array}$$

Finally, after erasing the + in the last iteration of the main loop, the termination phase adds the last carry bit to the leftmost output bit and halts.

$$\begin{array}{l}
\delta(\ p\ ,\ a) = (\ q\ ,\ b,\ \Delta) \\
\hline
\delta(\text{last0},\ 0) = (\text{last0},\ 0,\ -1) \\
\delta(\text{last0},\ 1) = (\text{last0},\ 0,\ -1) \\
\delta(\text{last0},\ \underline{0}) = (\ \text{halt}\ ,\ 0,\ \ \ ) \\
\hdashline
\delta(\text{last1},\ 0) = (\text{last1},\ 0,\ -1) \\
\delta(\text{last1},\ 1) = (\text{last1},\ 0,\ -1) \\
\delta(\text{last1},\ \underline{0}) = (\ \text{halt}\ ,\ 1,\ \ \ )
\end{array}$$

36.6 Variations on Tracks, Heads, and Tapes

Multiple Tracks

It is sometimes convenient to endow the Turing machine tape with multiple ***tracks***, each with its own tape alphabet, and allow the machine to read from and write to the same position on all tracks simultaneously. For example, to define a Turing machine with three tracks, we need three tape alphabets Γ_1, Γ_2, and Γ_3, each with its own blank symbol, where (say) Γ_1 contains the input alphabet Σ as a subset; we also need a transition function of the form

$$\delta : Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \to Q \times \Gamma_1 \times \Gamma_2 \times \Gamma_3 \times \{-1, +1\}$$

Describing a configuration of this machine requires a quintuple (q, x_1, x_2, x_3, i), indicating that each track i contains the string x_i followed by an infinite sequence of blanks. The initial configuration is $(\mathsf{start}, w, \varepsilon, \varepsilon, 0)$, with the input string written on the first track, and the other two tracks completely blank.

But any such machine is equivalent (if not *identical*) to a single-track Turing machine with the (still finite!) tape alphabet $\Gamma := \Gamma_1 \times \Gamma_2 \times \Gamma_3$. Instead of thinking of the tape as three infinite sequences of symbols, we think of it as a single infinite sequence of "records", each containing three symbols. Moreover, there's nothing special about the number 3 in this construction; a Turing machine with *any* constant number of tracks is equivalent to a single-track machine.

Doubly-Infinite Tape

It is also sometimes convenient to allow the tape to be infinite in both directions, for example, to avoid boundary conditions. There are several ways to simulate a doubly-infinite tape on a machine with only a semi-infinite tape. Perhaps the simplest method is to use a semi-infinite tape with two tracks, one containing the cells with positive index and the other containing the cells with negative index in reverse order, with a special marker symbol at position zero to indicate the transition.

0	+1	+2	+3	+4	⋯
►	−1	−2	−3	−4	⋯

Another method is to shuffle the positive-index and negative-index cells onto a single track, and add additional states to allow the Turing machine to move two steps in a single transition. Again, we need a special symbol at the left end of the tape to indicate the transition:

►	0	−1	+1	−2	+2	−3	+3	⋯

A third method maintains two sentinel symbols ► and ◄ that surround all other non-blank symbols on the tape. Whenever the machine reads the right sentinel ◄, we write a blank, move right, write ◄, move left, and then proceed as if we had just read a blank. On the other hand, when the machine reads the left sentinel ►, we shift the entire contents of the tape (up to and including the right sentinel) one step to the right, then move back to the left sentinel, move right, write a blank, and finally proceed as if we had just read a blank. Since the Turing machine does not actually have access to the position of the head *as an integer*, shifting the head and the tape contents one step right has no effect on its future evolution.

►	−3	−2	−1	0	+1	+2	+3	+4	+5	◄	⋯

Using either of the first two methods, we can simulate t steps of an arbitrary Turing machine with a doubly-infinite tape using only $O(t)$ steps on a standard Turing machine. The third method, unfortunately, requires $\Theta(t^2)$ steps in the worst case.

Insertion and Deletion

We can also allow Turing machines to insert and delete cells on the tape, in addition to simply overwriting existing symbols. We've already seen how to insert a new cell: Leave a special mark on the tape (perhaps in a second track), shift everything to the right of this mark one cell to the right, scan left to the mark, erase the mark, and finally write the correct character into the new cell. Deletion is similar: Mark the cell to be deleted, shift everything to the right of the mark one step to the left, scan left to the mark, and erase the mark. We may also need to maintain a mark in some cell to the right every non-blank symbol, indicating that all cells further to the right are blank, so that we know when to stop shifting left or right.

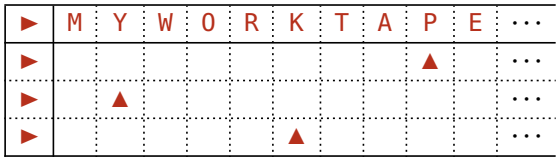
Multiple Heads

Another convenient extension is to allow machines simultaneous access to more than one position on the tape. For example, to define a Turing machine with *three* heads, we need a transition function of the form

$$\delta : Q \times \Gamma^3 \to Q \times \Gamma^3 \times \{-1, +1\}^3.$$

Describing a configuration of such a machine requires a quintuple (q, x, i, j, k), indicating that the machine is in state q, the tape contains string x, and the three heads are at positions i, j, k. The transition function tells us, given q and the three symbols $x[i], x[j], x[k]$, which three symbols to write on the tape and which direction to move each of the heads.

We can simulate this behavior with a single head by adding additional tracks to the tape that record the positions of each head. To simulate a machine M with three heads, we use a tape with four tracks: track 0 is the actual work tape; each of the remaining tracks has a single non-blank symbol recording the position of one of the heads. We also insert a special marker symbols at the left end of the tape.



We can simulate any single transition of M, starting with our single head at the left end of the tape, as follows. Throughout the simulation, we maintain the internal state of M as one of the components of our current state. First, for each i, we read the symbol under the ith head of M as follows:

> Scan to the right to find the mark on track i, read the corresponding symbol from track 0 into our internal state, and then return to the left end of the tape.

At this point, our internal state records M's current internal state and the three symbols under M's heads. After one more transition (using M's transition function), our internal state records M's *next* state, the symbol to be written by each head, and the direction to move each head. Then, for each i, we write with and move the ith head of M as follows:

> Scan to the right to find the mark on track i, write the correct symbol onto on track 0, move the mark on track i one step left or right, and then return to the left end of the tape.

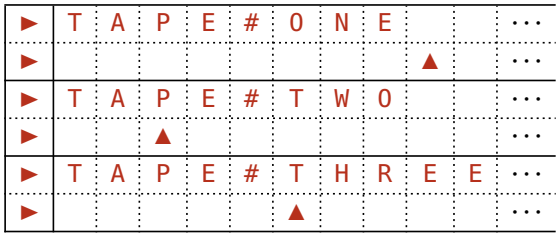
Again, there is nothing special about the number 3 here; we can simulate machines with *any* fixed number of heads.

Careful analysis of this technique implies that for any integer k, we can simulate t steps of an arbitrary Turing machine with k independent heads in $\Theta(t^2)$ time on a standard Turing machine with only one head. Unfortunately, this quadratic blowup is unavoidable. It is relatively easy to recognize the language

of *marked palindromes* $\{w \bullet w^R \mid w \in \{0,1\}^*\}$ in $O(n)$ time using a Turing machine with two heads, but recognizing this language provably requires $\Omega(n^2)$ time on a standard machine with only one head. On the other hand, with much more sophisticated techniques, it is possible to simulate t steps of a Turing machine with k head, for any fixed integer k, using only $O(t \log t)$ steps on a Turing machine with just *two* heads.

Multiple Tapes

We can also allow machines with multiple independent tapes, each with its own head. To simulate such a machine with a single tape, we simply maintain each tape as an independent track with its own head. Equivalently, we can simulate a machine with k tapes using a single tape with $2k$ tracks, half storing the contents of the k tapes and half storing the positions of the k heads.



Just as for multiple tracks, for any constant k, we can simulate t steps of an arbitrary Turing machine with k independent tapes in $\Theta(t^2)$ steps on a standard Turing machine with one tape, and this quadratic blowup is unavoidable. Moreover, it is possible to simulate t steps on a k-tape Turing machine using only $O(t \log t)$ steps on a *two*-tape Turing machine using more sophisticated techniques. (This faster simulation is easier to obtain for multiple independent tapes than for multiple heads on the same tape.)

By combining these tricks, we can simulate a Turing machine with any fixed number of tapes, each of which may be infinite in one or both directions, each with any fixed number of heads and any fixed number of tracks, with at most a quadratic blowup in the running time.

36.7 Simulating a Real Computer

36.7.1 Subroutines and Recursion

★★★

Use a second tape/track as a "call stack". Add save and restore actions. In the simplest formulation, subroutines do not have local memory. To call a subroutine, save the current state onto the call stack and jump to the first state of the subroutine. To return, restore (and remove) the return state from the call stack. We can simulate t steps of any recursive Turing machine with $O(t)$ steps on a multitape standard Turing machine, or in $O(t^2)$ steps on a standard Turing machine.

More complex versions of this simulation can adapt to

36.7.2 Random-Access Memory

★★★

Keep [address•data] pairs on a separate "memory" tape. Write address to an "address" tape; read data from or write data to a "data" tape. Add new or changed [address•data] pairs at the end of the memory tape. (Semantics of reading from an address that has never been written to?)

Suppose all memory accesses require at most ℓ address and data bits. Then we can simulate the kth memory access in $O(k\ell)$ steps on a multitape Turing machine or in $O(k^2\ell^2)$ steps on a single-tape machine. Thus, simulating t memory accesses in a random-access machine with ℓ-bit words requires $O(t^2\ell)$ time on a multitape Turing machine, or $O(t^3\ell^2)$ time on a single-tape machine.

36.8 Universal Turing Machines

With all these tools in hand, we can now describe the pinnacle of Turing machine constructions: the ***universal*** Turing machine. For modern computer scientists, it's useful to think of a universal Turing machine as a "Turing machine *interpreter* written in Turing machine". Just as the input to a Python interpreter is a string of Python source code, the input to our universal Turing machine U is a string $\langle M, w \rangle$ that encodes both an arbitrary Turing machine M and a string w in the input alphabet of M. Given these encodings, U simulates the execution of M on input w; in particular,

- U accepts $\langle M, w \rangle$ if and only if M accepts w.
- U rejects $\langle M, w \rangle$ if and only if M rejects w.

In the next few pages, I will sketch a universal Turing machine U that uses the input alphabet $\{0, 1, [,], \bullet, |\}$ and a somewhat larger tape alphabet (via marks on additional tracks). However, I do *not* require that the Turing machines that U simulates have similarly small alphabets, so we first need a method to encode *arbitrary* input and tape alphabets.

Encodings

Let $M = (\Gamma, \square, \Sigma, Q, \mathit{start}, \mathit{accept}, \mathit{reject}, \delta)$ be an arbitrary Turing machine, with a single half-infinite tape and a single read-write head. (I will consistently indicate the states and tape symbols of M in *slanted green* to distinguish them from the upright red states and tape symbols of U.)

We encode each symbol $a \in \Gamma$ as a unique string $|a|$ of $\lceil \lg(|\Gamma|) \rceil$ bits. Thus, if $\Gamma = \{0, 1, \$, x, \square\}$, we might use the following encoding:

$$\langle 0 \rangle = 001, \qquad \langle 1 \rangle = 010, \qquad \langle \$ \rangle = 011, \qquad \langle x \rangle = 100, \qquad \langle \square \rangle = 000.$$

The input string w is encoded by its sequence of symbol encodings, with separators $\bullet$ between every pair of symbols and with brackets $[$ and $]$ around the whole string. For example, with this encoding, the input string 001100 would be encoded on the input tape as

$$\langle 001100 \rangle = [001 \bullet 001 \bullet 010 \bullet 010 \bullet 001 \bullet 001]$$

Similarly, we encode each state $q \in Q$ as a distinct string $\langle q \rangle$ of $\lceil \lg |Q| \rceil$ bits. Without loss of generality, we encode the start state with all 1s and the reject state with all 0s. For example, if $Q = \{\mathit{start}, \mathit{seek1}, \mathit{seek0}, \mathit{reset}, \mathit{verify}, \mathit{accept}, \mathit{reject}\}$, we might use the following encoding:

$$\langle \mathit{start} \rangle = 111 \qquad \langle \mathit{seek1} \rangle = 010 \qquad \langle \mathit{seek0} \rangle = 011 \qquad \langle \mathit{reset} \rangle = 100$$
$$\langle \mathit{verify} \rangle = 101 \qquad \langle \mathit{accept} \rangle = 110 \qquad \langle \mathit{reject} \rangle = 000$$

We encode the machine M itself as the string $\langle M \rangle = [\langle \mathit{reject} \rangle \bullet \langle \square \rangle] \langle \delta \rangle$, where $\langle \delta \rangle$ is the concatenation of substrings $[\langle p \rangle \bullet \langle a \rangle | \langle q \rangle \bullet \langle b \rangle \bullet \langle \Delta \rangle]$ encoding each transition $\delta(p, a) = (q, b, \Delta)$ such that $q \neq \mathrm{reject}$. We encode the actions $\Delta = \pm 1$ by defining $\langle -1 \rangle := 0$ and $\langle +1 \rangle := 1$. Conveniently, every transition string has exactly the same length. For example, with the symbol and state encodings described above, the transition $\delta(\mathit{reset}, \$) = (\mathit{start}, \$, +1)$ would be encoded as

$$[100 \bullet 011 | 001 \bullet 011 \bullet 1].$$

Our first example Turing machine for recognizing $\{0^n1^n0^n \mid n \geq 0\}$ would be represented by the following string (here broken into multiple lines for readability):

```
[000•000][[001•001|010•011•1][001•100|101•011•1]
          [010•001|010•001•1][010•100|010•100•1]
          [010•010|011•100•1][011•010|011•010•1]
          [011•100|011•100•1][011•001|100•100•1]
          [100•001|100•001•0][100•010|100•010•0]
          [100•100|100•100•0][100•011|001•011•1]
          [101•100|101•011•1][101•000|110•000•0]]
```

Finally, we encode any *configuration* of M on U's work tape by alternating between encodings of states and encodings of tape symbols. Thus, each tape cell is represented by the string $[\langle q \rangle \bullet \langle a \rangle]$ indicating that (1) the cell contains symbol a; (2) if $q \neq \mathit{reject}$, then M's head is located at this cell, and M is in state q; and (3) if $q = \mathit{reject}$, then M's head is located somewhere else. Conveniently, each cell encoding uses exactly the same number of bits. We also surround the entire tape encoding with brackets `[` and `]`.

For example, with the encodings described above, the initial configuration $(\mathit{start}, \underset{\blacktriangle}{001100}, 0)$ for our first example Turing machine would be encoded on U's tape as follows.

$$\langle \mathit{start}, \underset{\blacktriangle}{001100}, 0 \rangle = \underbrace{[[111\bullet001]}_{\textbf{start } 0}\underbrace{[000\bullet001]}_{\text{reject } 0}\underbrace{[000\bullet010]}_{\text{reject } 1}\underbrace{[000\bullet010]}_{\text{reject } 1}\underbrace{[000\bullet001]}_{\text{reject } 0}\underbrace{[000\bullet001]]}_{\text{reject } 0}$$

Similarly, the intermediate configuration $(\mathit{reset}, \$0\underset{\blacktriangle}{x}1x0, 3)$ would be encoded as follows:

$$\langle \mathit{reset}, \$\$\underset{\blacktriangle}{x}1x0, 3 \rangle = \underbrace{[[000\bullet011]}_{\text{reject \$}}\underbrace{[000\bullet011]}_{\text{reject } 0}\underbrace{[000\bullet100]}_{\text{reject } x}\underbrace{[010\bullet010]}_{\textbf{reset } 1}\underbrace{[000\bullet100]}_{\text{reject } x}\underbrace{[000\bullet001]]}_{\text{reject } 0}$$

Input and Execution

Without loss of generality, we assume that the input to our universal Turing machine U is given on a separate read-only ***input tape***, as the encoding of an arbitrary Turing machine M followed by an encoding of its input string x. Notice the substrings `[[` and `]]` each appear only only once on the input tape, immediately before and after the encoded transition table, respectively. U also has a read-write ***work tape***, which is initially blank.

We start by initializing the work tape with the encoding $\langle \mathit{start}, x, 0 \rangle$ of the initial configuration of M with input x. First, we write $[[\langle \mathit{start} \rangle \bullet$. Then we copy the encoded input string $\langle x \rangle$ onto the work tape, but we change the punctuation as follows:

- Instead of copying the left bracket `[`, write $[[\langle \mathit{start} \rangle \bullet$.
- Instead of copying each separator `•`, write $][\langle \mathit{reject} \rangle \bullet$
- Instead of copying the right bracket `]`, write two right brackets `]]`.

The state encodings $\langle \mathit{start} \rangle$ and $\langle \mathit{reject} \rangle$ can be copied directly from the beginning of $\langle M \rangle$ (replacing `0`s for `1`s for $\langle \mathit{start} \rangle$). Finally, we move the head back to the start of U's tape.

At the start of each step of the simulation, U's head is located at the start of the work tape. We scan through the work tape to the unique encoded cell $[\langle p \rangle \bullet \langle a \rangle]$ such that $p \neq \mathit{reject}$. Then we scan through the encoded transition function $\langle \delta \rangle$ to find the unique encoded tuple $[\langle p \rangle \bullet \langle a \rangle | \langle q \rangle \bullet \langle b \rangle \bullet \langle \Delta \rangle]$ whose left half matches our the encoded tape cell. If there is no such tuple, then U immediately halts and rejects. Otherwise, we copy the right half $\langle q \rangle \bullet \langle b \rangle$ of the tuple to the work tape. Now if $q = \mathit{accept}$, then

U immediately halts and accepts. (We don't bother to encode *reject* transformations, so we know that $q \neq$ *reject*.) Otherwise, we transfer the state encoding to either the next or previous encoded cell, as indicated by M's transition function, and then continue with the next step of the simulation.

During the final state-copying phase, we ever read two right brackets `]]`, indicating that we have reached the right end of the tape encoding, we replace the second right bracket with `[`⟨*reject*⟩`•`⟨□⟩`]]` (mostly copied from the beginning of the machine encoding ⟨M⟩) and then scan back to the left bracket we just wrote. This trick allows our universal machine to *pretend* that its tape contains an infinite sequence of *encoded* blanks `[`⟨*reject*⟩`•`⟨□⟩`]` instead of *actual* blanks □.

Example

As an illustrative example, suppose U is simulating our first example Turing machine M on the input string `001100`. The execution of M on input w eventually reaches the configuration (*seek1*, `$$x1x0`, 3). At the start of the corresponding step in U's simulation, U is in the following configuration:

```
[[000•011][000•011][000•100][010•010][000•100][000•001]]
```

First U scans for the first encoded tape cell whose state is not *reject*. That is, U repeatedly compares the first half of each encoded state cell on the work tape with the prefix `[`⟨*reject*⟩`•` of the machine encoding ⟨M⟩ on the input tape. U finds a match in the fourth encoded cell.

```
[[000•011][000•011][000•100][010•010][000•100][000•001]]
```

Next, U scans the machine encoding ⟨M⟩ for the substring `[010•010` matching the current encoded cell. U eventually finds a match in the left size of the the encoded transition `[010•010|011•100•1]`. U copies the state-symbol pair `011•100` from the right half of this encoded transition into the current encoded cell. (The underline indicates which symbols are changed.)

```
[[000•011][000•011][000•100][011•100][000•100][000•001]]
```

The encoded transition instructs U to move the current state encoding one cell to the right. (The underline indicates which symbols are changed.)

```
[[000•011][000•011][000•100][000•100][011•100][000•001]]
```

Finally, U scans left until it reads two left brackets `[[`; this returns the head to the left end of the work tape to start the next step in the simulation. U's tape now holds the encoding of M's configuration (*seek0*, `$$xxx0`, 4), as required.

```
[[000•011][000•011][000•100][000•100][011•100][000•001]]
```

Exercises

1. Describe Turing machines that decide each of the following languages:

   (a) Palindromes over the alphabet {`0`, `1`}

   (b) $\{ww \mid w \in \{0,1\}^*\}$

   (c) $\left\{0^a 1^b 0^{ab} \mid a, b \in \mathbb{N}\right\}$

2. Let $\langle n \rangle_2$ denote the binary representation of the non-negative integer n. For example, $\langle 17 \rangle_2 = \texttt{10001}$ and $\langle 42 \rangle_2 = \texttt{101010}$. Describe Turing machines that compute the following functions from $\{\texttt{0}, \texttt{1}\}^*$ to $\{\texttt{0}, \texttt{1}\}^*$:

   (a) $w \mapsto www$

   (b) $\texttt{1}^n\texttt{01}^m \mapsto \texttt{1}^{mn}$

   (c) $\texttt{1}^n \mapsto \texttt{1}^{2^n}$

   (d) $\texttt{1}^n \mapsto \langle n \rangle_2$

   (e) $\texttt{0}^*\langle n \rangle_2 \mapsto \texttt{1}^n$

   (f) $\langle n \rangle_2 \mapsto \langle n^2 \rangle_2$

3. Describe Turing machines that write each of the following infinite streams of bits onto their tape. Specifically, for each integer n, there must be a finite time after which the first n symbols on the tape always match the first n symbols in the target stream.

   (a) An infinite stream of 1s

   (b) $\texttt{0101101110111101111101111110}\ldots$, where the nth block of 1s has length n.

   (c) The stream of bits whose nth bit is 1 if and only if n is prime.

   (d) The ***Thue-Morse sequence*** $T_0 \bullet T_1 \bullet T_2 \bullet T_3 \cdots$, where

$$T_n := \begin{cases} \texttt{0} & \text{if } n = 0 \\ \texttt{1} & \text{if } n = 1 \\ T_{n-1} \bullet \overline{T_{n-1}} & \text{otherwise} \end{cases}$$

   where $\overline{w}$ indicates the binary string obtained from w by flipping every bit. Equivalently, the nth bit of the Thue Morse sequence if 0 if the binary representation of n has an even number of 1s and 1 otherwise.

$$\texttt{0 1 10 1001 10010110 1001011001101001 10010110011010010110100110}\ldots$$

   (e) The ***Fibonacci*** sequence $F_0 \bullet F_1 \bullet F_2 \bullet F_3 \cdots$, where

$$F_n := \begin{cases} \texttt{0} & \text{if } n = 0 \\ \texttt{1} & \text{if } n = 1 \\ F_{n-2} \bullet F_{n-1} & \text{otherwise} \end{cases}$$

$$\texttt{0 1 01 101 01101 10101101 0110110101101 1010110101101101011010}\ldots$$

4. A ***two-stack machine*** is a Turing machine with two tapes with the following restricted behavior. At all times, on each tape, every cell to the right of the head is blank, and every cell at or to the left of the head is non-blank. Thus, a head can only move right by writing a non-blank symbol into a blank cell; symmetrically, a head can only move left by erasing the rightmost non-blank cell. Thus, each tape behaves like a stack. To avoid underflow, there is a special symbol at the start of each tape that cannot be overwritten. Initially, one tape contains the input string, with the head at its *last* symbol, and the other tape is empty (except for the start-of-tape symbol).

   Prove formally that any standard Turing machine can be simulated by a two-stack machine. That is, given any standard Turing machine M, describe a two-stack machine M' that accepts and rejects exactly the same input strings as M.

★★★

Counter machines. Configuration consists of k rational numbers and an internal state (from some finite set Q). Transition function $\delta : Q \times \{=0, >0\}^k \to Q \times \{-1, 0, +1\}^k$ takes internal state and signs of counters as input, and produces new internal state and changes to counters as output.

- Prove that any Turing machine can be simulated by a three-counter machine. One counter holds the binary representation of the tape after the head; another counter holds the reversed binary representation of the tape before the head. Implement transitions via halving, doubling, and parity, using the third counter for scratch work.
- Prove that two counters can simulate three. Store $2^a 3^b 5^c$ in one counter, use the other for scratch work.
- Prove that a three-counter machine can compute any computable function: Given input $(n, 0, 0)$, we can compute $(f(n), 0, 0)$ for *any* computable function f. First transform $(n, 0, 0)$ to $(2^n, 0, 0)$ using all three counters; then run two- (or three-)counter TM simulation to obtain $(2^{f(n)}, 0, 0)$; and finally transform $(2^{f(n)}, 0, 0)$ to $(f(n), 0, 0)$ using all three counters.
- **HARD:** Prove that a two-counter machine cannot transform $(n, 0)$ to $(2^n, 0)$. [Barzdin' 1963, Yao 1971, Schröpel 1972, Ibarra+Trân 1993]

★★★

FRACTRAN [Conway 1987]: One-counter machine whose "program" is a sequence of rational numbers. The counter is initially 1. At each iteration, multiply the counter by the first rational number that yields an integer; if there is no such number, halt.

- Prove that for any computable function $f : \mathbb{N} \to \mathbb{N}$, there is a FRACTRAN program that transforms 2^{n+1} into $3^{f(n)+1}$, for all natural numbers n.
- Prove that every FRACTRAN program, given the integer 1 as input, either outputs 1 or loops forever. It follows that there is no FRACTRAN program for the increment function $n \mapsto n + 1$.

5. A ***tag***-Turing machine has two heads: one can only read, the other can only write. Initially, the read head is located at the left end of the tape, and the write head is located at the first blank after the input string. At each transition, the read head can either move one cell to the right or stay put, but the write head *must* write a symbol to its current cell and move one cell to the right. Neither head can ever move to the left.

   Prove that any standard Turing machine can be simulated by a tag-Turing machine. That is, given any standard Turing machine M, describe a tag-Turing machine M' that accepts and rejects exactly the same input strings as M.

6. (a) Prove that any standard Turing machine can be simulated by a Turing machine with only three states. *[Hint: Use the tape to store an encoding of the state of the machine yours is simulating.]*

   ★(b) Prove that any standard Turing machine can be simulated by a Turing machine with only *two* states.

7. A ***two-dimensional*** Turing machine uses an infinite two-dimensional grid of cells as the tape; at each transition, the head can move from its current cell to any of its four neighbors on the grid. The transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\uparrow, \leftarrow, \downarrow, \rightarrow\}$, where the arrows indicate which direction the head should move.

   (a) Prove that any two-dimensional Turing machine can be simulated by a standard Turing machine.

   (b) Suppose further that we endow our two-dimensional Turing machine with the following additional actions, in addition to moving the head:

- Insert row: Move all symbols on or above the row containing the head up one row, leaving the head's row blank.
- Insert column: Move all symbols on or to the right of the column containing the head one column to the right, leaving the head's column blank.
- Delete row: Move all symbols above the row containing the head down one row, deleting the head's row of symbols.
- Delete column: Move all symbols the right of the column containing the head one column to the right, deleting the head's column of symbols.

Show that any two-dimensional Turing machine that can add an delete rows can be simulated by a standard Turing machine.

8. A ***binary-tree*** Turing machine uses an infinite binary tree as its tape; that is, *every* cell in the tape has a left child and a right child. At each step, the head moves from its current cell to its Parent, its Left child, or to its Right child. Thus, the transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathsf{P}, \mathsf{L}, \mathsf{R}\}$. The input string is initially given along the left spine of the tape.

   Show that any binary-tree Turing machine can be simulated by a standard Turing machine.

9. A ***stack-tape*** Turing machine uses an semi-infinite tape, where every cell is actually the top of an independent stack. The behavior of the machine at each iteration is governed by its internal state and the symbol *at the top* of the current cell's stack. At each transition, the head can optionally push a new symbol onto the stack, or pop the top symbol off the stack. (If a stack is empty, its "top symbol" is a blank and popping has no effect.)

   Show that any stack-tape Turing machine can be simulated by a standard Turing machine. (Compare with Problem 4!)

10. A ***tape-stack*** Turing machine has two actions that modify its work tape, in addition to simply writing individual cells: it can ***save*** the entire tape by pushing in onto a stack, and it can ***restore*** the entire tape by popping it off the stack. Restoring a tape returns the content of every cell to its content when the tape was saved. Saving and restoring the tape do not change the machine's state or the position of its head. If the machine attempts to "restore" the tape when the stack is empty, the machine crashes.

    Show that any tape-stack Turing machine can be simulated by a standard Turing machine.

★★★

- Tape alphabet = $\mathbb{N}$.
  - Read: zero or positive. Write: $+1, -1$
  - Read: even or odd. Write: $+1, -1, \times 2, \div 2$
  - Read: positive, negative, or zero. Write: $x + y$ (merge), $x - y$ (merge), 1, 0
- Never three times in a row in the same direction
- Hole-punch TM: tape alphabet $\{\square, \blacksquare\}$, and only $\square \mapsto \blacksquare$ transitions allowed.

Caveat lector: This is the zeroth (draft) edition of this lecture note. Please send bug reports and suggestions to jeffe@illinois.edu.

I said in my haste, All men are liars.

— Psalms 116:11 (King James Version)

yields falsehood when preceded by its quotation.

— William V. Quine, "Paradox", *Scientific American* (1962)

Some problems are so complex that you have to be highly intelligent and well informed just to be undecided about them.

— Laurence Johnston Peter, *Peter's Almanac* (September 24, 1982)

"Proving or disproving a formula—once you've encrypted the formula into numbers, that is—is just a calculation on that number. So it means that the answer to the question is, no! Some formulas cannot be proved or disproved by any mechanical process! So I guess there's some point in being human after all!"

Alan looked pleased until Lawrence said this last thing, and then his face collapsed. "Now there you go making unwarranted assumptions."

— Neal Stephenson, *Cryptonomicon* (1999)

No matter how P might perform, Q will scoop it:
Q uses P's output to make P look stupid.
Whatever P says, it cannot predict Q:
P is right when it's wrong, and is false when it's true!

— Geoffrey S. Pullum, "Scooping the Loop Sniffer" (2000)

This castle is in unacceptable condition! ***UNACCEPTABLE!!***

— Earl of Lemongrab [Justin Poiland], "Too Young"
Adventure Time (August 8, 2011)

37 Undecidability

Perhaps the single most important result in Turing's remarkable 1936 paper is his solution to Hilbert's *Entscheidungsproblem*, which asked for a general automatic procedure to determine whether a given statement of first-order logic is *provable*. Turing proved that no such procedure exists; there is no systematic way to distinguish between statements that cannot be proved even in principle and statements whose proofs we just haven't found yet.

37.1 Acceptable versus Decidable

Recall that there are three possible outcomes for a Turing machine M running on any particular input string w: acceptance, rejection, and divergence. Every Turing machine M immediately defines four different languages (over the input alphabet Σ of M):

- The *accepting* language $\textsc{Accept}(M) := \{w \in \Sigma^* \mid M \text{ accepts } w\}$
- The *rejecting* language $\textsc{Reject}(M) := \{w \in \Sigma^* \mid M \text{ rejects } w\}$
- The *halting* language $\textsc{Halt}(M) := \textsc{Accept}(M) \cup \textsc{Reject}(M)$
- The *diverging* language $\textsc{Diverge}(M) := \Sigma^* \setminus \textsc{Halt}(M)$

For any language L, the sentence "***M accepts L***" means $\textsc{Accept}(M) = L$, and the sentence "***M decides L***" means $\textsc{Accept}(M) = L$ and $\textsc{Diverge}(M) = \emptyset$.

Now let L be an arbitrary language. We say that L is ***acceptable*** (or *semi-computable*, or *semi-decidable*, or *recognizable*, or *listable*, or *recursively enumerable*) if some Turing machine accepts L, and ***unacceptable*** otherwise. Similarly, L is ***decidable*** (or *computable*, or *recursive*) if some Turing machine decides L, and ***undecidable*** otherwise.

37.2 Lo, I Have Become Death, Stealer of Pie

There is a subtlety in the definitions of "acceptable" and "decidable" that many beginners miss: A language can be decidable even if we can't exhibit a specific Turing machine decides it. As a canonical example, consider the language $\Pi = \{w \mid 1^{|w|} \text{ appears in the binary expansion of } \pi\}$. Despite appearances, this language *is* decidable! There are only two cases to consider:

- Suppose there is an integer N such that the binary expansion of π contains the substring 1^N but does not contain the substring 1^{N+1}. Let M_N be the Turing machine with $N+3$ states $\{0, 1, \ldots, N, \text{accept}, \text{reject}\}$, start state 0, and the following transition function:

$$\delta(q,a) = \begin{cases} \text{accept} & \text{if } a = \square \\ \text{reject} & \text{if } a \neq \square \text{ and } q = n \\ (q+1, a, +1) & \text{otherwise} \end{cases}$$

  This machine correctly decides Π.

- Suppose the binary expansion of π contains arbitrarily long substrings of 1s. Then any Turing machine that accepts all inputs correctly decides Π.

We have no idea which of these machines correctly decides Π, but one of them does, and that's enough!

37.3 Useful Lemmas

This subsection contains several lemmas that are useful for proving that languages are (un)decidable or (un)acceptable. For almost all of these lemmas, the proofs are straightforward; readers are strongly encouraged to try to prove each lemma themselves before reading ahead .

One might reasonably ask why we don't also define "rejectable" and "haltable" languages. The following lemma, whose proof is an easy exercise (hint, hint), implies that these are both identical to the acceptable languages.

Lemma 1. *Let M be an arbitrary Turing machine.*
(a) There is a Turing machine M^R such that $\textsc{Accept}(M^R) = \textsc{Reject}(M)$ and $\textsc{Reject}(M^R) = \textsc{Accept}(M)$.
(b) There is a Turing machine M^A such that $\textsc{Accept}(M^A) = \textsc{Accept}(M)$ and $\textsc{Reject}(M^A) = \emptyset$.
(c) There is a Turing machine M^H such that $\textsc{Accept}(M^H) = \textsc{Halt}(M)$ and $\textsc{Reject}(M^H) = \emptyset$.

The decidable languages have several fairly obvious useful properties.

Lemma 2. *If L and L' are decidable, then $L \cup L'$, $L \cap L'$, $L \setminus L'$, and $L \setminus L'$ are also decidable.*

Proof: Let M and M' be Turing machines that decide L and L', respectively. We can build a Turing machine $M_\cup$ that decides $L \cup L'$ as follows. First, $M_\cup$ copies its input string w onto a second tape. Then $M_\cup$ runs M on input w (on the first tape), and then runs M' on input w (on the second tape). If either M or M' accepts, then $M_\cup$ accepts; if both M and M' reject, then $M_\cup$ rejects.

The other three languages are similar. □

Corollary 3. *The following hold for all languages L and L'.*
(a) If $L \cap L'$ is undecidable and L' is decidable, then L is undecidable.
(b) If $L \cup L'$ is undecidable and L' is decidable, then L is undecidable.
(c) If $L \setminus L'$ is undecidable and L' is decidable, then L is undecidable.
(d) If $L' \setminus L$ is undecidable and L' is decidable, then L is undecidable.

The asymmetry between acceptance and rejection implies that merely acceptable languages are not quite as well-behaved as decidable languages.

Lemma 4. *For all acceptable languages L and L', the languages $L \cup L'$ and $L \cap L'$ are also acceptable.*

Proof: Let M and M' be Turing machines that decide L and L', respectively. We can build a Turing machine $M_\cap$ that decides $L \cap L'$ as follows. First, $M_\cap$ copies its input string w onto a second tape. Then $M_\cap$ runs M on input w using the first tape, and then runs M' on input w using the second tape. If both M and M' accept, then $M_\cap$ accepts; if either M or M' reject, then $M_\cap$ rejects; if either M or M' diverge, then $M_\cap$ diverges (automatically).

The construction for $L \cup L'$ is more subtle; instead of running M and M' in series, we must run them in parallel. Like $M_\cap$, the new machine $M_\cup$ starts by copying its input string w onto a second tape. But then $M_\cup$ runs M and M' simultaneously; with each step of $M_\cup$ simulating both one step of M on the first tape and one step of M' on the second. Ignoring the states and transitions needed for initialization, the state set of $M_\cup$ is the product of the state sets of M and M', and the transition function is

$$\delta_\cup(q, a, q', a') = \begin{cases} \mathsf{accept}_\cup & \text{if } q = \mathsf{accept} \text{ or } q' = \mathsf{accept}' \\ \mathsf{reject}_\cup & \text{if } q = \mathsf{reject} \text{ and } q' = \mathsf{reject}' \\ (\delta(q, a), \delta'(q', a')) & \text{otherwise} \end{cases}$$

Thus, $M_\cup$ accepts as soon as either M or M' accepts, and rejects only after both M or M' reject. □

Lemma 5. *An acceptable language L is decidable if and only if $\Sigma^* \setminus L$ is also acceptable.*

Proof: Let M and $\overline{M}$ be Turing machines that accept L and $\Sigma^* \setminus L$, respectively. Following the previous proof, we construct a new Turing machine M^* that copies its input onto a second tape, and then simulates M and M' in parallel on the two tapes. If M accepts, then M^* accepts; if $\overline{M}$ accepts, then M^* rejects. Since every string is accepted by either M or $\overline{M}$, we conclude that M^* decides L.

The other direction follows immediately from Lemma 1. □

37.4 Self-Haters Gonna Self-Hate

Let U be an arbitrary fixed universal Turing machine. Any Turing machine M can be encoded as a string $\langle M \rangle$ of symbols from U's input alphabet, so that U can simulate the execution of M on any suitably encoded input string. Different universal Turing machines require different encodings.[1]

A Turing machine encoding is just a string, and any string (over the correct alphabet) can be used as the input to a Turing machine. Thus, we can use the encoding $\langle M \rangle$ of any Turing machine M as the input to another Turing machine. We've already seen an example of this ability in our universal Turing

[1] In fact, these undecidability proofs never actually use the universal Turing machine; all we really need is an encoding function that associates a unique string $\langle M \rangle$ with every Turing machine M. However, we *do* need the encoding to be compatible with a universal Turing machine for the results in Section 37.7.

machine U, but more significantly, we can use $\langle M \rangle$ as the input to *the same Turing machine* M. Thus, each of the following languages is well-defined:

$$\begin{aligned}
\textsc{SelfAccept} &:= \{\langle M \rangle \mid M \text{ accepts } \langle M \rangle\} \\
\textsc{SelfReject} &:= \{\langle M \rangle \mid M \text{ rejects } \langle M \rangle\} \\
\textsc{SelfHalt} &:= \{\langle M \rangle \mid M \text{ halts on } \langle M \rangle\} \\
\textsc{SelfDiverge} &:= \{\langle M \rangle \mid M \text{ diverges on } \langle M \rangle\}
\end{aligned}$$

One of Turing's key observations is that ***SelfReject is undecidable***; Turing proved this theorem by contradiction as follows:

Suppose to the contrary that there is a Turing machine SR such that $\textsc{Accept}(SR) = \textsc{SelfReject}$ and $\textsc{Diverge}(SR) = \emptyset$. More explicitly, for ***any*** Turing machine M,

- SR accepts $\langle M \rangle \iff M$ rejects $\langle M \rangle$, and
- SR rejects $\langle M \rangle \iff M$ does not reject $\langle M \rangle$.

In particular, these equivalences must hold when M is equal to SR. Thus,

- SR accepts $\langle SR \rangle \iff SR$ rejects $\langle SR \rangle$, and
- SR rejects $\langle SR \rangle \iff SR$ does not reject $\langle SR \rangle$.

In short, SR accepts $\langle SR \rangle$ if and only if SR rejects $\langle SR \rangle$, which is impossible! The only logical conclusion is that the Turing machine SR does not exist!

37.5 Aside: Uncountable Barbers

Turing's proof by contradiction is nearly identical to the famous ***diagonalization argument*** that uncountable sets exist, published by Georg Cantor in 1891. Indeed, SelfReject is sometimes called "the diagonal language". Recall that a function $f : A \to B$ is a ***surjection***[2] if $f(A) = \{f(a) \mid a \in A\} = B$.

Cantor's Theorem. *Let $f : X \to 2^X$ be an **arbitrary** function from an **arbitrary** set X to its power set. This function f is not a surjection.*

Proof: Fix an arbitrary function $f : X \to 2^X$. Call an element $x \in X$ ***happy*** if $x \in f(x)$ and ***sad*** if $x \notin f(x)$. Let Y be the set of all sad elements of X; that is, for *every* element $x \in X$, we have

$$x \in Y \iff x \notin f(x).$$

For the sake of argument, suppose f is a surjection. Then (by definition of surjection) there must be an element $y \in X$ such that $f(y) = Y$. Then for *every* element $x \in X$, we have

$$x \in f(y) \iff x \notin f(x).$$

In particular, the previous equivalence must hold when $x = y$:

$$y \in f(y) \iff y \notin f(y).$$

We have a contradiction! We conclude that f is not a surjection after all. □

[2] more commonly, flouting all reasonable standards of grammatical English, "an onto function"

Now let $X = \Sigma^*$, and define the function $f : X \to 2^X$ as follows:

$$f(w) := \begin{cases} \textsc{Accept}(M) & \text{if } w = \langle M \rangle \text{ for some Turing machine } M \\ \emptyset & \text{if } w \text{ is not the encoding of a Turing machine} \end{cases}$$

Cantor's theorem immediately implies that not all languages are acceptable.

Alternatively, let X be the set of all Turing machines that halt on all inputs. For any Turing machine $M \in X$, let $f(M)$ be the set of all Turing machines $N \in X$ such that M accepts the encoding $\langle N \rangle$. Then a Turing machine M is *sad* if it rejects its own encoding $\langle M \rangle$; thus, Y is essentially the set SELFREJECT. Cantor's argument now immediately implies that no Turing machine decides the language SELFREJECT.

The core of Cantor's diagonalization argument also appears in the "barber paradox" popularized by Bertrand Russell in the 1910s. In a certain small town, every resident has a haircut on Haircut Day. Some residents cut their own hair; others have their hair cut by another resident of the same town. To obtain an official barber's license, a resident must cut the hair of all residents who don't cut their own hair, and no one else. Given these assumptions, we can immediately conclude that there are no licensed barbers. After all, who would cut the barber's hair?

To map Russell's barber paradox back to Cantor's theorem, let X be the set of residents, and let $f(x)$ be the set of residents who have their hair cut by x; then a resident is *sad* if they do not cut their own hair. To prove that SELFREJECT is undecidable, replace "resident" with "a Turing machine that halts on all inputs", and replace "A cuts B's hair" with "A accepts $\langle B \rangle$".

37.6 Just Don't Know What to Do with Myself

Similar diagonal arguments imply that the other three languages SELFACCEPT, SELFHALT, and SELFDIVERGE are also undecidable. The proofs are not quite as direct for these three languages as the proof for SELFREJECT; each fictional deciding machine requires a small modification to create the contradiction.

Theorem 6. *SELFACCEPT is undecidable.*

Proof: For the sake of argument, suppose there is a Turing machine SA such that $\textsc{Accept}(SA) = \textsc{SelfAccept}$ and $\textsc{Diverge}(M) = \emptyset$. Let SA^R be the Turing machine obtained from SA by swapping its accept and reject states (as in the proof of Lemma 1). Then $\textsc{Reject}(SA^R) = \textsc{SelfAccept}$ and $\textsc{Diverge}(SA^R) = \emptyset$. It follows that SA^R rejects $\langle SA^R \rangle$ if and only if SA^R accepts $\langle SA^R \rangle$, which is impossible. □

Theorem 7. *SELFHALT is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine SH such that $\textsc{Accept}(SH) = \textsc{SelfHalt}$ and $\textsc{Diverge}(SH) = \emptyset$. Let SH^X be the Turing machine obtained from SH by redirecting every transition to accept to a new hanging state hang, and then redirecting every transition to reject to accept. Then $\textsc{Accept}(SH^X) = \Sigma^* \setminus \textsc{SelfHalt}$ and $\textsc{Reject}(SH^X) = \emptyset$. It follows that SH^X accepts $\langle SH^X \rangle$ if and only if SH^X does not halt on $\langle SH^X \rangle$, and we have a contradiction. □

Theorem 8. *SELFDIVERGE is* ***unacceptable and therefore*** *undecidable.*

Proof: Suppose to the contrary that there is a Turing machine SD such that $\textsc{Accept}(M) = \textsc{SelfDiverge}$. Let SD^A be the Turing machine obtained from M by redirecting every transition to reject to a new hanging state hang such that $\delta(\text{hang}, a) = (\text{hang}, a, +1)$ for every symbol a. Then $\textsc{Accept}(SD^A) = \textsc{SelfDiverge}$ and $\textsc{Reject}(SD^A) = \emptyset$. It follows that SD^A accepts $\langle SD^A \rangle$ if and only if SD^A does not halt on $\langle SD^A \rangle$, which is impossible. □

*37.7 Nevertheless, Acceptable

Our undecidability argument for SELFDIVERGE actually implies the stronger result that SELFDIVERGE is unacceptable; we never assumed that the hypothetical accepting machine SD halts on all inputs. However, we can use or modify our universal Turing machine to accept the other three languages.

Theorem 9. *SELFACCEPT is acceptable.*

Proof: We describe a Turing machine SA that accepts the language SELFACCEPT. Given any string w as input, SA first verifies that w is the encoding of a Turing machine. If w is not the encoding of a Turing machine, then SA diverges. Otherwise, $w = \langle M \rangle$ for some Turing machine M; in this case, SA writes the string $ww = \langle M \rangle \langle M \rangle$ onto its tape and passes control to the universal Turing machine U. U then simulates M (the machine encoded by the first half of its input) on the string $\langle M \rangle$ (the second half of its input).[3] In particular, U accepts $\langle M, M \rangle$ if and only if M accepts $\langle M \rangle$. We conclude that SR accepts $\langle M \rangle$ if and only if M accepts $\langle M \rangle$. □

Theorem 10. *SELFREJECT is acceptable.*

Proof: Let U^R be the Turing machine obtained from our universal machine U by swapping the accept and reject states. We describe a Turing machine SR that accepts the language SELFREJECT as follows. SR first verifies that its input string w is the encoding of a Turing machine and diverges if not. Otherwise, SR writes the string $ww = \langle M, M \rangle$ onto its tape and passes control to the reversed universal Turing machine U^R. Then U^R accepts $\langle M, M \rangle$ if and only if M rejects $\langle M \rangle$. We conclude that SR accepts $\langle M \rangle$ if and only if M rejects $\langle M \rangle$. □

Finally, because SELFHALT is the union of two acceptable languages, SELFHALT is also acceptable.

37.8 The Halting Problem via Reduction

Consider the following related languages:[4]

$$\begin{aligned}
\text{ACCEPT} &:= \{\langle M, w \rangle \mid M \text{ accepts } w\} \\
\text{REJECT} &:= \{\langle M, w \rangle \mid M \text{ rejects } w\} \\
\text{HALT} &:= \{\langle M, w \rangle \mid M \text{ halts on } w\} \\
\text{DIVERGE} &:= \{\langle M, w \rangle \mid M \text{ diverges on } w\}
\end{aligned}$$

Deciding the language HALT is what is usually meant by the *halting problem*: Given a program M and an input w to that program, does the program halt? This problem may seem trivial; why not just run the program and see? More formally, why not just pass the input string $\langle M, x \rangle$ to our universal Turing machine U? That strategy works perfectly if we just want to *accept* HALT, but we actually want to *decide* HALT; if M is not going to halt on w, we still want an answer in a finite amount of time. Sadly, we can't always get what we want.

[3] To simplify the presentation, I am implicitly assuming here that $\langle M \rangle = \langle \langle M \rangle \rangle$. Without this assumption, we need a Turing machine that transforms an arbitrary string $w \in \Sigma_M^*$ into its encoding $\langle w \rangle$ for U; building such a Turing machine is straightforward.

[4] Sipser uses the shorter name A_{TM} instead of ACCEPT, but uses $HALT_{TM}$ instead of HALT. I have no idea why he thought four-letter names are okay, but six-letter names are not. His subscript TM is just a reminder that these are languages of *Turing machine* encodings, as opposed to encodings of DFAs or some other machine model.

Theorem 11. *HALT is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine H that decides HALT. Then we can use H to build another Turing machine SH that decides the language SELFHALT. Given any string w, the machine SH first verifies that $w = \langle M \rangle$ for some Turing machine M (rejecting if not), then writes the string $ww = \langle M, M \rangle$ onto the tape, and finally passes control to H. But SELFHALT is undecidable, so no such machine SH exists. We conclude that H does not exist either. □

Nearly identical arguments imply that the languages ACCEPT, REJECT, and DIVERGE are undecidable.

Here we have our first example of an undecidability proof by ***reduction***. Specifically, we ***reduced*** the language SELFHALT to the language HALT. More generally, to reduce one language X to another language Y, we assume (for the sake of argument) that there is a program P_Y that decides Y, and we write another program that decides X, using P_Y as a black-box subroutine. If later we discover that Y is decidable, we can immediately conclude that X is decidable. Equivalently, if we later discover that X is undecidable, we can immediately conclude that Y is undecidable.

> **To prove that a language L is undecidable, reduce a known undecidable language to L.**

Perhaps the most confusing aspect of reduction arguments is that the *languages* we want to prove undecidable nearly (but not quite) always involve encodings of Turing machines, while at the same time, the *programs* that we build to prove them undecidable are also Turing machines. Our proof that HALT is undecidable involved three different machines:

- The hypothetical Turing machine H that decides HALT.
- The new Turing machine SH that decides SELFHALT, using H as a subroutine.
- The Turing machine M whose encoding is the input to H.

It is *incredibly* easy to get confused about which machines are playing each in the proof. Therefore, it is absolutely vital that we give each machine in a reduction proof a unique and mnemonic name, and then ***always*** refer to each machine ***by name***. Never write, say, or even *think* "the machine" or "that machine" or (gods forbid) "it". You also may find it useful to think of the working ***programs*** we are trying to construct (H and SH in this proof) as being written in a different language than the arbitrary ***source code*** that we want those programs to analyze ($\langle M \rangle$ in this proof).

37.9 One Million Years Dungeon!

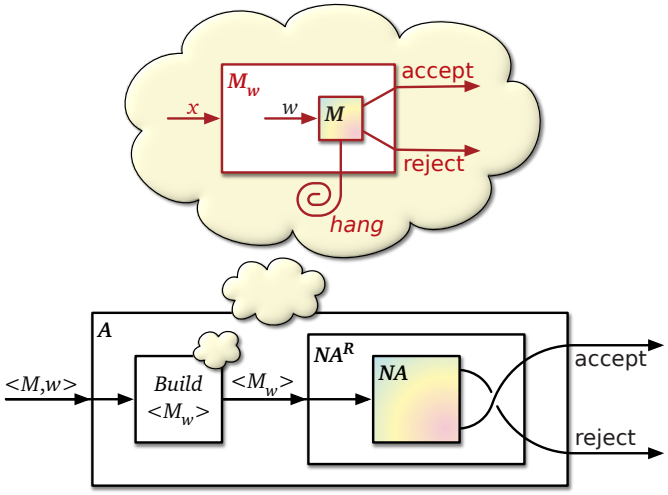
As a more complex set of examples, consider the following languages:

$$\begin{aligned}
\text{NEVERACCEPT} &:= \{\langle M \rangle \mid \text{ACCEPT}(M) = \varnothing\} \\
\text{NEVERREJECT} &:= \{\langle M \rangle \mid \text{REJECT}(M) = \varnothing\} \\
\text{NEVERHALT} &:= \{\langle M \rangle \mid \text{HALT}(M) = \varnothing\} \\
\text{NEVERDIVERGE} &:= \{\langle M \rangle \mid \text{DIVERGE}(M) = \varnothing\}
\end{aligned}$$

Theorem 12. *NEVERACCEPT is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine NA that decides NEVERACCEPT. Then by swapping the accept and reject states, we obtain a Turing machine NA^R that decides the complementary language $\Sigma^* \setminus$ NEVERACCEPT.

To reach a contradiction, we construct a Turing machine A that decides ACCEPT as follows. Given the encoding $\langle M, w\rangle$ of an arbitrary machine M and an arbitrary string w as input, A writes the encoding $\langle M_w\rangle$ of a new Turing machine M_w that ignores its input, writes w onto the tape, and then passes control to M. Finally, A passes the new encoding $\langle M_w\rangle$ as input to NA^R. The following cartoon tries to illustrate the overall construction.



A reduction from from ACCEPT to NEVERACCEPT, which proves NEVERACCEPT undecidable.

Before going any further, it may be helpful to list the various Turing machines that appear in this construction.

- The hypothetical Turing machine NA that decides NEVERACCEPT.
- The Turing machine NA^R that decides $\Sigma^* \setminus$ NEVERACCEPT, which we constructed by modifying NA.
- The Turing machine A that we are building, which decides ACCEPT using NA^R as a black-box subroutine.
- The Turing machine M, whose encoding is part of the input to A.
- The Turing machine M_w whose encoding A constructs from $\langle M, w\rangle$ and then passes to NA^R as input.

Now let M be an arbitrary Turing machine and w be an arbitrary string, and suppose we run our new Turing machine A on the encoding $\langle M, w\rangle$. To complete the proof, we need to consider two cases: Either M accepts w or M does not accept w.

- First, suppose M accepts w.
  - Then for all strings x, the machine M_w accepts x.
  - So ACCEPT$(M_w) = \Sigma^*$, by the definition of ACCEPT(M_w).
  - So $\langle M_w\rangle \notin$ NEVERACCEPT, by definition of NEVERACCEPT.
  - So NA rejects $\langle M_w\rangle$, because NA decides NEVERACCEPT.
  - So NA^R accepts $\langle M_w\rangle$, buy construction of NA^R.
  - We conclude that A accepts $\langle M, w\rangle$, by construction of A.

- On the other hand, suppose M does not accept w, either rejecting or diverging instead.
  - Then for all strings x, the machine M_w does not accept x.
  - So $\textsc{Accept}(M_w) = \emptyset$, by the definition of $\textsc{Accept}(M_w)$.
  - So $\langle M_w \rangle \in \textsc{NeverAccept}$, by definition of NeverAccept.
  - So NA accepts $\langle M_w \rangle$, because NA decides NeverAccept.
  - So NA^R rejects $\langle M_w \rangle$, buy construction of NA^R.
  - We conclude that A rejects $\langle M, w \rangle$, by construction of A.

In short, A decides the language Accept, which is impossible. We conclude that NA does not exist. □

Again, similar arguments imply that the languages NeverReject, NeverHalt, and NeverDiverge are undecidable. In each case, the core of the argument is describing how to transform the incoming machine-and-input encoding $\langle M, w \rangle$ into the encoding of an appropriate new Turing machine $\langle M_w \rangle$.

Now that we know that NeverAccept and its relatives are undecidable, we can use them as the basis of further reduction proofs. Here is a typical example:

Theorem 13. *The language* $\textsc{DivergeSame} := \left\{ \langle M_1 \rangle \langle M_2 \rangle \;\middle|\; \textsc{Diverge}(M_1) = \textsc{Diverge}(M_2) \right\}$ *is undecidable.*

Proof: Suppose for the sake of argument that there is a Turing machine DS that decides DivergeSame. Then we can build a Turing machine ND that decides NeverDiverge as follows. Fix a Turing machine Y that accepts Σ^* (for example, by defining $\delta(\mathsf{start}, a) = (\mathsf{accept}, \cdot, \cdot)$ for all $a \in \Gamma$). Given an arbitrary Turing machine encoding $\langle M \rangle$ as input, ND writes the string $\langle M \rangle \langle Y \rangle$ onto the tape and then passes control to DS. There are two cases to consider:

- If DS accepts $\langle M \rangle \langle Y \rangle$, then $\textsc{Diverge}(M) = \textsc{Diverge}(Y) = \emptyset$, so $\langle M \rangle \in \textsc{NeverDiverge}$.
- If DS rejects $\langle M \rangle \langle Y \rangle$, then $\textsc{Diverge}(M) \neq \textsc{Diverge}(Y) = \emptyset$, so $\langle M \rangle \notin \textsc{NeverDiverge}$.

In short, ND accepts $\langle M \rangle$ if and only if $\langle M \rangle \in \textsc{NeverDiverge}$, which is impossible. We conclude that DS does not exist. □

37.10 Rice's Theorem

In 1953, Henry Rice proved the following extremely powerful theorem, which essentially states that ***every*** interesting question about the language accepted by a Turing machine is undecidable.

Rice's Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\textsc{Accept}(Y) \in \mathcal{L}$.*
- *There is a Turing machine N such that $\textsc{Accept}(N) \notin \mathcal{L}$.*

The language $\textsc{AcceptIn}(\mathcal{L}) := \left\{ \langle M \rangle \;\middle|\; \textsc{Accept}(M) \in \mathcal{L} \right\}$ *is undecidable.*

Proof: Without loss of generality, suppose $\emptyset \notin \mathcal{L}$. (A symmetric argument establishes the theorem in the opposite case $\emptyset \in \mathcal{L}$.) Fix an arbitrary Turing machine Y such that $\textsc{Accept}(Y) \in \mathcal{L}$.

Suppose to the contrary that there is a Turing machine $A_{\mathcal{L}}$ that decides $\textsc{AcceptIn}(\mathcal{L})$. To derive a contradiction, we describe a Turing machine H that decides the halting language Halt, using $A_{\mathcal{L}}$ as a black-box subroutine. Given the encoding $\langle M, w \rangle$ of an arbitrary Turing machine M and an arbitrary string w as input, H writes the encoding $\langle WTF \rangle$ of a new Turing machine WTF that executes the following algorithm:

```
WTF(x):
  run M on input w (and discard the result)
  run Y on input x
```

H then passes the new encoding $\langle WTF \rangle$ to $A_{\mathcal{L}}$.

Now let M be an arbitrary Turing machine and w be an arbitrary string, and suppose we run our new Turing machine H on the encoding $\langle M, w\rangle$. There are two cases to consider.

- Suppose M halts on input w.
  - Then for all strings x, the machine *WTF* accepts x if and only if Y accepts x.
  - So $\textsc{Accept}(WTF) = \textsc{Accept}(Y)$, by definition of $\textsc{Accept}(\cdot)$.
  - So $\textsc{Accept}(WTF) \in \mathcal{L}$, by definition of Y.
  - So $A_{\mathcal{L}}$ accepts $\langle WTF\rangle$, because $A_{\mathcal{L}}$ decides $\textsc{AcceptIn}(\mathcal{L})$.
  - So H accepts $\langle M, w\rangle$, by definition of H.
- Suppose M does not halt on input w.
  - Then for all strings x, the machine *WTF* does not halt on input x, and therefore does not accept x.
  - So $\textsc{Accept}(WTF) = \emptyset$, by definition of $\textsc{Accept}(WTF)$.
  - So $\textsc{Accept}(WTF) \notin \mathcal{L}$, by our assumption that $\emptyset \notin \mathcal{L}$.
  - So $A_{\mathcal{L}}$ rejects $\langle WTF\rangle$, because $A_{\mathcal{L}}$ decides $\textsc{AcceptIn}(\mathcal{L})$.
  - So H rejects $\langle M, w\rangle$, by definition of H.

In short, H decides the language $\textsc{Halt}$, which is impossible. We conclude that $A_{\mathcal{L}}$ does not exist. □

The set $\mathcal{L}$ in the statement of Rice's Theorem is often called a ***property*** of languages, rather than a *set*, to avoid the inevitable confusion about sets of sets. We can also think of $\mathcal{L}$ as a ***decision problem*** about languages, where the languages are represented by Turing machines that accept or decide them. Rice's theorem states that the ***only*** properties of languages that are decidable are the trivial properties "Does this Turing machine accept an acceptable language?" (Answer: Yes, by definition.) and "Does this Turing machine accept Discover?" (Answer: No, because Discover is a credit card, not a language.)

Rice's Theorem makes it incredibly easy to prove that language properties are undecidable; we only need to exhibit one acceptable language that has the property and another acceptable language that does not. In fact, most proofs using Rice's theorem can use at least one of the following Turing machines:

- $M_{\textsc{Accept}}$ accepts every string, by defining $\delta(\mathsf{start}, a) = \mathsf{accept}$ for every tape symbol a.
- $M_{\textsc{Reject}}$ rejects every string, by defining $\delta(\mathsf{start}, a) = \mathsf{reject}$ for every tape symbol a.
- $M_{\textsc{Diverge}}$ diverges on every string, by defining $\delta(\mathsf{start}, a) = (\mathsf{start}, a, +1)$ for every tape symbol a.

Corollary 14. *Each of the following languages is undecidable.*
(a) $\{\langle M\rangle \mid M$ *accepts given an empty initial tape*$\}$
(b) $\{\langle M\rangle \mid M$ *accepts the string* `UIUC`$\}$
(c) $\{\langle M\rangle \mid M$ *accepts exactly three strings*$\}$
(d) $\{\langle M\rangle \mid M$ *accepts all palindromes*$\}$
(e) $\{\langle M\rangle \mid \textsc{Accept}(M)$ *is regular*$\}$
(f) $\{\langle M\rangle \mid \textsc{Accept}(M)$ *is not regular*$\}$
(g) $\{\langle M\rangle \mid \textsc{Accept}(M)$ *is undecidable*$\}$
(h) $\{\langle M\rangle \mid \textsc{Accept}(M) = \textsc{Accept}(N)\}$*, for some arbitrary fixed Turing machine* N.

Proof: In all cases, undecidability follows from Rice's theorem.

(a) Let $\mathcal{L}$ be the set of all languages that contain the empty string. Then $\textsc{AcceptIn}(\mathcal{L}) = \{\langle M\rangle \mid M$ accept given an empty initial tape$\}$.

- Given an empty initial tape, M_{ACCEPT} accepts, so $\text{HALT}(M_{\text{ACCEPT}}) \in \mathcal{L}$.
- Given an empty initial tape, M_{DIVERGE} does not accept, so $\text{HALT}(M_{\text{DIVERGE}}) \notin \mathcal{L}$.

Therefore, Rice's Theorem implies that $\text{ACCEPTIN}(\mathcal{L})$ is undecidable.

(b) Let $\mathcal{L}$ be the set of all languages that contain the string `UIUC`.

- M_{ACCEPT} accepts `UIUC`, so $\text{HALT}(M_{\text{ACCEPT}}) \in \mathcal{L}$.
- M_{DIVERGE} does not accept `UIUC`, so $\text{HALT}(M_{\text{DIVERGE}}) \notin \mathcal{L}$.

Therefore, $\text{ACCEPTIN}(\mathcal{L}) = \{\langle M \rangle \mid M \text{ accepts the string } \texttt{UIUC}\}$ is undecidable by Rice's Theorem.

(c) There is a Turing machine that accepts the language $\{\texttt{larry}, \texttt{curly}, \texttt{moe}\}$. On the other hand, M_{REJECT} does not accept exactly three strings.

(d) M_{ACCEPT} accepts all palindromes, and M_{REJECT} does not accept all palindromes.

(e) M_{REJECT} accepts the regular language $\emptyset$, and there is a Turing machine $M_{\texttt{0}^n\texttt{1}^n}$ that accepts the non-regular language $\{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$.

(f) M_{REJECT} accepts the regular language $\emptyset$, and there is a Turing machine $M_{\texttt{0}^n\texttt{1}^n}$ that accepts the non-regular language $\{\texttt{0}^n\texttt{1}^n \mid n \geq 0\}$.[5]

(g) M_{REJECT} accepts the decidable language $\emptyset$, and there is a Turing machine that accepts the undecidable language SELFREJECT.

(h) The Turing machine N accepts $\text{ACCEPT}(N)$ by definition. The Turing machine N^R, obtained by swapping the `accept` and `reject` states of N, accepts the language $\text{HALT}(L) \setminus \text{ACCEPT}(N) \neq \text{ACCEPT}(N)$. □

We can also use Rice's theorem as a component in more complex undecidability proofs, where the target language consists of more than just a single Turing machine encoding.

Theorem 15. *The language* $L := \left\{\langle M, w\rangle \;\middle|\; M \text{ accepts } w^k \text{ for every integer } k \geq 0\right\}$ *is undecidable.*

Proof: Fix an arbitrary string w, and let $\mathcal{L}$ be the set of all languages that contain w^k for all k. Then $\text{ACCEPT}(M_{\text{ACCEPT}}) = \Sigma^* \in \mathcal{L}$ and $\text{ACCEPT}(M_{\text{REJECT}}) = \emptyset \notin \mathcal{L}$. Thus, *even if the string w is fixed in advance*, no Turing machine can decide L. □

Nearly identical reduction arguments imply the following variants of Rice's theorem. (The names of these theorems are not standard.)

Rice's Rejection Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\text{REJECT}(Y) \in \mathcal{L}$*
- *There is a Turing machine N such that $\text{REJECT}(N) \notin \mathcal{L}$.*

The language $\text{REJECTIN}(\mathcal{L}) := \left\{\langle M\rangle \;\middle|\; \text{REJECT}(M) \in \mathcal{L}\right\}$ *is undecidable.*

Rice's Halting Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that $\text{HALT}(Y) \in \mathcal{L}$*
- *There is a Turing machine N such that $\text{HALT}(N) \notin \mathcal{L}$.*

The language $\text{HALTIN}(\mathcal{L}) := \left\{\langle M\rangle \;\middle|\; \text{HALT}(M) \in \mathcal{L}\right\}$ *is undecidable.*

Rice's Divergence Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

[5] Yes, parts (e) and (f) have exactly the same proof.

- *There is a Turing machine Y such that $\textsc{Diverge}(Y) \in \mathcal{L}$*
- *There is a Turing machine N such that $\textsc{Diverge}(N) \notin \mathcal{L}$.*

The language $\textsc{DivergeIn}(\mathcal{L}) := \{\langle M \rangle \mid \textsc{Diverge}(M) \in \mathcal{L}\}$ is undecidable.

Rice's Decision Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*

- *There is a Turing machine Y such that **decides** an language in $\mathcal{L}$.*
- *There is a Turing machine N such that **decides** an language not in $\mathcal{L}$.*

*The language $\textsc{DecideIn}(\mathcal{L}) := \{\langle M \rangle \mid M$ **decides** a language in $\mathcal{L}\}$ is undecidable.*

As a final sanity check, always be careful to distinguish the following objects:

- The string ε
- The language $\emptyset$
- The language $\{\varepsilon\}$
- The language property $\emptyset$
- The language property $\{\emptyset\}$
- The language property $\{\{\varepsilon\}\}$
- The Turing machine $M_{\textsc{Reject}}$ that rejects every string and therefore ***decides*** the language $\emptyset$.
- The Turing machine $M_{\textsc{Diverge}}$ that diverges on every string and therefore ***accepts*** the language $\emptyset$.

*37.11 The Rice-McNaughton-Myhill-Shapiro Theorem

The following subtle generalization of Rice's theorem precisely characterizes which properties of acceptable languages are *acceptable*. This result was partially proved by Henry Rice in 1953, in the same paper that proved Rice's Theorem; Robert McNaughton, John Myhill, and Norman Shapiro completed the proof a few years later, each independently from the other two.[6]

The Rice-McNaughton-Myhill-Shapiro Theorem. *Let $\mathcal{L}$ be an arbitrary set of acceptable languages. The language $\textsc{AcceptIn}(\mathcal{L}) := \{\langle M \rangle \mid \textsc{Accept}(M) \in \mathcal{L}\}$ is **acceptable** if and only if $\mathcal{L}$ satisfies the following conditions:*

*(a) $\mathcal{L}$ is **monotone**: For any language $L \in \mathcal{L}$, every superset of L is also in $\mathcal{L}$.*

*(b) $\mathcal{L}$ is **compact**: Every language in $\mathcal{L}$ has a finite subset that is also in $\mathcal{L}$.*

*(c) $\mathcal{L}$ is **finitely acceptable**: The language $\{\langle L \rangle \mid L \in \mathcal{L}$ and L is finite$\}$ is acceptable.*[7]

I won't give a complete proof of this theorem (in part because it requires techniques I haven't introduced), but the following lemma is arguably the most interesting component:

Lemma 16. *Let $\mathcal{L}$ be a set of acceptable languages. If $\mathcal{L}$ is not monotone, then $\textsc{AcceptIn}(\mathcal{L})$ is unacceptable.*

[6] McNaughton never published his proof (although he did announce the result); consequently, this theorem is sometimes called "The Rice-Myhill-Shapiro Theorem". Even more confusingly, Myhill published his proof twice, once in a paper with John Shepherdson and again in a later paper with Jacob Dekker. So maybe it should be called the Rice–Dekker-Myhill–McNaughton–Myhill-Shepherdson–Shapiro Theorem.

[7] Here the encoding $\langle L \rangle$ of a finite language $L \subseteq \Sigma^*$ is exactly the string that you would write down to explicitly describe L. Formally, $\langle L \rangle$ is the unique string over the alphabet $\Sigma \cup \{\texttt{\{}, \texttt{,}, \texttt{\}}, \texttt{ε}\}$ that contains the strings in L in lexicographic order, separated by commas `,` and surrounded by braces `{}`, with `ε` representing the empty string. For example, $\langle\{\varepsilon, \texttt{0}, \texttt{01}, \texttt{0110}, \texttt{01101001}\}\rangle =$ `{ε,0,01,0110,01101001}`.

Proof: Suppose to the contrary that there is a Turing machine $AI_{\mathcal{L}}$ that accepts AcceptIn($\mathcal{L}$). Using this Turing machine as a black box, we describe a Turing machine SD that accepts the unacceptable language SelfDiverge. Fix two Turing machines Y and N such that

$$\begin{aligned} &\textsc{Accept}(Y) \in \mathcal{L}, \\ &\textsc{Accept}(N) \notin \mathcal{L}, \\ \text{and} \quad &\textsc{Accept}(Y) \subseteq \textsc{Accept}(N). \end{aligned}$$

Let w be the input to SD. After verifying that $w = \langle M \rangle$ for some Turing machine M (and rejecting otherwise), SD writes the encoding $\langle WTF \rangle$ or a new Turing machine WTF that implements the following algorithm:

```
WTF(x):
  write x to second tape
  write ⟨M⟩ to third tape
  in parallel:
      run Y on the first tape
      run N on the second tape
      run M on the third tape
  if Y accepts x
      accept
  if N accepts x and M halts on ⟨M⟩
      accept
```

Finally, SD passes the new encoding $\langle WTF \rangle$ to $AI_{\mathcal{L}}$. There are two cases to consider:

- If M halts on $\langle M \rangle$, then $\textsc{Accept}(WTF) = \textsc{Accept}(N) \notin \mathcal{L}$, and therefore $AI_{\mathcal{L}}$ does not accept $\langle WTF \rangle$.
- If M does not halt on $\langle M \rangle$, then $\textsc{Accept}(WTF) = \textsc{Accept}(Y) \in \mathcal{L}$, and therefore $AI_{\mathcal{L}}$ accepts $\langle WTF \rangle$.

In short, SD accepts SelfDiverge, which is impossible. We conclude that SD does not exist. □

Corollary 17. *Each of the following languages is unacceptable.*

(a) $\{\langle M \rangle \mid \textsc{Accept}(M)$ *is finite*$\}$
(b) $\{\langle M \rangle \mid \textsc{Accept}(M)$ *is infinite*$\}$
(c) $\{\langle M \rangle \mid \textsc{Accept}(M)$ *is regular*$\}$
(d) $\{\langle M \rangle \mid \textsc{Accept}(M)$ *is not regular*$\}$
(e) $\{\langle M \rangle \mid \textsc{Accept}(M)$ *is decidable*$\}$
(f) $\{\langle M \rangle \mid \textsc{Accept}(M)$ *is undecidable*$\}$
(g) $\{\langle M \rangle \mid M$ *accepts at least one string in* SelfDiverge$\}$
(h) $\{\langle M \rangle \mid \textsc{Accept}(M) = \textsc{Accept}(N)\}$*, for some arbitrary fixed Turing machine* N.

Proof: (a) The set of finite languages is not monotone: $\emptyset$ is finite; Σ^* is not finite; both $\emptyset$ and Σ^* are acceptable (in fact decidable); and $\emptyset \subset \Sigma^*$.

(b) The set of infinite acceptable languages is not compact: No finite subset of the infinite acceptable language Σ^* is infinite!

(c) The set of regular languages is not monotone: Consider the languages $\emptyset$ and $\{0^n 1^n \mid n \geq 0\}$.

(d) The set of non-regular acceptable languages is not monotone: Consider the languages $\{0^n 1^n \mid n \geq 0\}$ and Σ^*.

(e) The set of decidable languages is not monotone: Consider the languages $\emptyset$ and SELFREJECT.

(f) The set of undecidable acceptable languages is not monotone: Consider the languages SELFREJECT and Σ^*.

(g) The set $\mathcal{L} = \{L \mid L \cap \text{SELFDIVERGE} \neq \emptyset\}$ is not finitely acceptable. For any string w, deciding whether $\{w\} \in \mathcal{L}$ is equivalent to deciding whether $w \in$ SELFDIVERGE, which is impossible.

(h) If $\text{ACCEPT}(N) \neq \Sigma^*$, then the set $\{\text{ACCEPT}(N)\}$ is not monotone. On the other hand, if $\text{ACCEPT}(N) = \Sigma^*$, then the set $\{\text{ACCEPT}(N)\}$ is not compact: No finite subset of Σ^* is equal to Σ^*! □

37.12 Turing Machine Behavior: It's Complicated

Rice's theorems imply that every interesting question about the language that a Turing machine accepts—or more generally, the function that a program computes—is undecidable. A more subtle question is whether we can recognize Turing machines that exhibit certain *internal behavior*. Some behaviors we can recognize; others we can't.

Theorem 18. *The language* NEVERLEFT $:= \{\langle M, w\rangle \mid$ *Given w as input, M never moves left*$\}$ *is decidable.*

Proof: Given the encoding $\langle M, w\rangle$, we simulate M with input w using our universal Turing machine U, but with the following termination conditions. If M ever moves its head to the left, then we reject. If M halts without moving its head to the left, then we accept. Finally, if M reads more than $|Q|$ blanks, where Q is the state set of M, then we accept. If the first two cases do not apply, M only moves to the right; moreover, after reading the entire input string, M only reads blanks. Thus, after reading $|Q|$ blanks, it must repeat some state, and therefore loop forever without moving to the left. The three cases are exhaustive. □

Theorem 19. *The language* LEFTTHREE $:= \{\langle M, w\rangle \mid$ *Given w as input, M eventually moves left three times in a row*$\}$ *is undecidable.*

Proof: Given $\langle M\rangle$, we build a new Turing machine M' that accepts the same language as M and moves left three times in a row if and only if it accepts, as follows. For each non-accepting state p of M, the new machine M' has three states p_1, p_2, p_3, with the following transitions:

$$\begin{aligned} \delta'(p_1, a) &= (q_2, b, \Delta), && \text{where } (q, b, \Delta) = \delta(p, a) \text{ and } q \neq \text{accept} \\ \delta'(p_2, a) &= (p_3, a, +1) \\ \delta'(p_3, a) &= (p_1, a, -1) \end{aligned}$$

In other words, after each non-accepting transition, M' moves once to the right and then once to the left. For each transition to accept, M' has a sequence of seven transitions: three steps to the right, then three steps to the left, and then finally accept', all without modifying the tape. (The three steps to the right ensure that M' does not fall off the left end of the tape.)

Finally, M' moves left three times in a row if and only if M accepts w. Thus, if we could decide LEFTTHREE, we could also decide ACCEPT, which is impossible. □

There is no hard and fast rule like Rice's theorem to distinguish decidable behaviors from undecidable behaviors, but I can offer two rules of thumb.

- If it is possible to simulate an arbitrary Turing machine while avoiding the target behavior, then the behavior is not decidable. For example: there is no algorithm to determine whether a given Turing machine reenters its start state, or revisits the left end of the tape, or writes a blank.

- If a Turing machine with the target behavior is limited to a finite number of configurations, or is guaranteed to force an infinite loop after a finite number of transitions, then the behavior is likely to be decidable. For example, there *are* algorithms to determine whether a given Turing machine ever leaves its start state, or reads its entire input string, or writes a non-blank symbol over a blank.

Exercises

1. Let M be an arbitrary Turing machine.
   (a) Describe a Turing machine M^R such that $\textsc{Accept}(M^R) = \textsc{Reject}(M)$ and $\textsc{Reject}(M^R) = \textsc{Accept}(M)$.
   (b) Describe is a Turing machine M^A such that $\textsc{Accept}(M^A) = \textsc{Accept}(M)$ and $\textsc{Reject}(M^A) = \emptyset$.
   (c) Describe is a Turing machine M^H such that $\textsc{Accept}(M^H) = \textsc{Halt}(M)$ and $\textsc{Reject}(M^H) = \emptyset$.

2. (a) Prove that Accept is undecidable.
   (b) Prove that Reject is undecidable.
   (c) Prove that Diverge is undecidable.

3. (a) Prove that NeverReject is undecidable.
   (b) Prove that NeverHalt is undecidable.
   (c) Prove that NeverDiverge is undecidable.

4. Prove that each of the following languages is undecidable.
   (a) $\textsc{AlwaysAccept} := \{\langle M \rangle \mid \textsc{Accept}(M) = \Sigma^*\}$
   (b) $\textsc{AlwaysReject} := \{\langle M \rangle \mid \textsc{Reject}(M) = \Sigma^*\}$
   (c) $\textsc{AlwaysHalt} := \{\langle M \rangle \mid \textsc{Halt}(M) = \Sigma^*\}$
   (d) $\textsc{AlwaysDiverge} := \{\langle M \rangle \mid \textsc{Diverge}(M) = \Sigma^*\}$

5. Let $\mathcal{L}$ be a non-empty proper subset of the set of acceptable languages. Prove that the following languages are undecidable:
   (a) $\textsc{RejectIn}(\mathcal{L}) := \left\{\langle M \rangle \;\middle|\; \textsc{Reject}(M) \in \mathcal{L}\right\}$
   (b) $\textsc{HaltIn}(\mathcal{L}) := \left\{\langle M \rangle \;\middle|\; \textsc{Halt}(M) \in \mathcal{L}\right\}$
   (c) $\textsc{DivergeIn}(\mathcal{L}) := \left\{\langle M \rangle \;\middle|\; \textsc{Diverge}(M) \in \mathcal{L}\right\}$

6. For each of the following decision problems, either *sketch* an algorithm or prove that the problem is undecidable. Recall that w^R denotes the reversal of string w. For each problem, the input is the encoding $\langle M \rangle$ of a Turing machine M.
   (a) Does M accept $\langle M \rangle^R$?
   (b) Does M reject any palindrome?
   (c) Does M accept all palindromes?

(d) Does M diverge only on palindromes?

(e) Is there an input string that forces M to move left?

(f) Is there an input string that forces M to move left three times in a row?

(g) Does M accept the encoding of any Turing machine N such that $\textsc{Accept}(N) = \textsc{SelfDiverge}$?

7. For each of the following decision problems, either *sketch* an algorithm or prove that the problem is undecidable. Recall that w^R denotes the reversal of string w. For each problem, the input is an encoding $\langle M, w\rangle$ of a Turing machine M and its input string w.

(a) Does M accept the string ww^R?

(b) Does M accept either w or w^R?

(c) Does M either accept w or reject w^R?

(d) Does M accept the string w^k for some integer k?

(e) Does M accept w in at most $2^{|w|}$ steps?

(f) If we run M on input w, does M ever change a symbol on its tape?

(g) If we run M on input w, does M ever move to the right?

(h) If we run M on input w, does M ever move to the right twice in a row?

(i) If we run M on input w, does M move its head to the right more than $2^{|w|}$ times (not necessarily consecutively)?

(j) If we run M with input w, does M ever change a $\square$ on the tape to any other symbol?

(k) If we run M with input w, does M ever change a $\square$ on the tape to 1?

(l) If we run M with input w, does M ever write a $\square$?

(m) If we run M with input w, does M ever leave its start state?

(n) If we run M with input w, does M ever reenter its start state?

(o) If we run M with input w, does M ever reenter a state that it previously left? That is, are there states $p \neq q$ such that M moves from state p to state q and then later moves back to state p?

8. Let M be a Turing machine, let w be an arbitrary input string, and let s and t be positive integers integer. We say that M accepts w ***in space s*** if M accepts w after accessing at most the first s cells on the tape, and M accepts w ***in time t*** if M accepts w after at most t transitions.

(a) Prove that the following languages are decidable:

i. $\left\{\langle M, w\rangle \;\middle|\; M \text{ accepts } w \text{ in time } |w|^2\right\}$

ii. $\left\{\langle M, w\rangle \;\middle|\; M \text{ accepts } w \text{ in space } |w|^2\right\}$

(b) Prove that the following languages are undecidable:

i. $\left\{\langle M\rangle \;\middle|\; M \text{ accepts at least one string } w \text{ in time } |w|^2\right\}$

ii. $\left\{\langle M\rangle \;\middle|\; M \text{ accepts at least one string } w \text{ in space } |w|^2\right\}$

9. Let L_0 be an arbitrary language. For any integer $i > 0$, define the language

$$L_i := \left\{ \langle M \rangle \;\middle|\; M \text{ decides } L_{i-1} \right\}.$$

For which integers $i > 0$ is L_i decidable? Obviously the answer depends on the initial language L_0; give a complete characterization of all possible cases. Prove your answer is correct. *[Hint: This question is a lot easier than it looks!]*

10. Argue that each of the following decision problems about programs in your favorite programming language are undecidable.

    (a) Does this program correctly compute Fibonacci numbers?

    (b) Can this program fall into an infinite loop?

    (c) Will the value of this variable ever change?

    (d) Will this program every attempt to deference a null pointer?

    (e) Does this program free every block of memory that it dynamically allocates?

    (f) Is any statement in this program unreachable?

    (g) Do these two programs compute the same function?

*11. Call a Turing machine ***conservative*** if it never writes over its input string. More formally, a Turing machine is conservative if for every transition $\delta(p, a) = (q, b, \Delta)$ where $a \in \Sigma$, we have $b = a$; and for every transition $\delta(p, a) = (q, b, \Delta)$ where $a \notin \Sigma$, we have $b \neq \Sigma$.

    (a) Prove that if M is a conservative Turing machine, then $\textsc{Accept}(M)$ is a regular language.

    (b) Prove that the language $\{\langle M \rangle \mid M \text{ is conservative and } M \text{ accepts } \varepsilon\}$ is undecidable.

Together, these two results imply that every conservative Turing machine accepts the same language as some DFA, but it is impossible to determine *which* DFA.

★12. (a) Prove that it is undecidable whether a given C++ program is syntactically correct. *[Hint: Use templates!]*

    (b) Prove that it is undecidable whether a given ANSI C program is syntactically correct. *[Hint: Use the preprocessor!]*

    (c) Prove that it is undecidable whether a given Perl program is syntactically correct. *[Hint: Does that slash character / delimit a regular expression or represent division?]*

Caveat lector: This is the zeroth (draft) edition of this lecture note. In particular, some topics still need to be written. Please send bug reports and suggestions to jeffe@illinois.edu.

If first you don't succeed, then try and try again.
And if you don't succeed again, just try and try and try.

— Marc Blitzstein, "Useless Song", *The Three Penny Opera* (1954)
Adaptation of Bertold Brecht, "Das Lied von der Unzulänglichkeit menschlichen Strebens" *Die Dreigroschenoper* (1928)

Children need encouragement.
If a kid gets an answer right, tell him it was a lucky guess.
That way he develops a good, lucky feeling.

— Jack Handey, "Deep Thoughts", *Saturday Night Live* (March 21, 1992)

38 Nondeterministic Turing Machines

38.1 Definitions

In his seminal 1936 paper, Turing also defined an extension of his "automatic machines" that he called ***choice machines***, which are now more commonly known as ***nondeterministic Turing machines***. The execution of a nondeterministic Turing machine is *not determined* entirely by its input and its transition function; rather, at each step of its execution, the machine can *choose* from a set of possible transitions. The distinction between deterministic and nondeterministic Turing machines exactly parallels the distinction between deterministic and nondeterministic finite-state automata.

Formally, a nondeterministic Turing machine has all the components of a standard deterministic Turing machine—a finite tape alphabet Γ that contains the input alphabet Σ and a blank symbol $\square$; a finite set Q of internal states with special start, accept, and reject states; and a transition function δ. However, the transition function now has the signature

$$\delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{-1,+1\}}.$$

That is, for each state p and tape symbol a, the output $\delta(p, a)$ of the transition function is a *set* of triples of the form $(q, b, \Delta) \in Q \times \Gamma \times \{-1, +1\}$. Whenever the machine finds itself in state p reading symbol a, the machine *chooses* an arbitrary triple $(q, b, \Delta) \in \delta(p, a)$, and then changes its state to q, writes b to the tape, and moves the head by Δ. If the set $\delta(p, a)$ is empty, the machine moves to the reject state and halts.

The set of all possible transition sequences of a nondeterministic Turing machine N on a given input string w define a rooted tree, called a ***computation tree***. The initial configuration $(\text{start}, w, 0)$ is the root of the computation tree, and the children of any configuration (q, x, i) are the configurations that can be reached from (q, x, i) in one transition. In particular, any configuration whose state is accept or reject is a leaf. For deterministic Turing machines, this computation tree is just a single path, since there is at most one valid transition from every configuration.

38.2 Acceptance and Rejection

Unlike deterministic Turing machines, there is a fundamental asymmetry between the acceptance and rejection criteria for nondeterministic Turing machines. Let N be any nondeterministic Turing machine, and let w be any string.

- ***N accepts w*** if and only if there is *at least one* sequence of valid transitions from the initial configuration (start, w, 0) that leads to the accept state. Equivalently, N accepts w if the computation tree contains at least one accept leaf.
- ***N rejects w*** if and only if *every* sequence of valid transitions from the initial configuration (start, w, 0) leads to the reject state. Equivalently, N rejects w if every path through the computation tree ends with a reject leaf.

In particular, N can accept w even when there are choices that allow the machine to run forever, but rejection requires N to halt after only a finite number of transitions, no matter what choices it makes along the way. Just as for deterministic Turing machines, it is possible that N neither accepts nor rejects w.

Acceptance and rejection of *languages* are defined exactly as they are for deterministic machines. A non-deterministic Turing machine N ***accepts*** a language $L \subseteq \Sigma^*$ if M accepts all strings in L and nothing else; N ***rejects*** L if M rejects every string in L and nothing else; and finally, N ***decides*** L if M accepts L and rejects $\Sigma^* \setminus L$.

38.3 Time and Space Complexity

★★★

- Define "time" and "space".
- TIME($f(n)$) is the class of languages that can be decided by a deterministic multi-tape Turing machine in $O(f(n))$ time.
- NTIME($f(n)$) is the class of languages that can be decided by a nondeterministic multi-tape Turing machine in $O(f(n))$ time.
- SPACE($f(n)$) is the class of languages that can be decided by deterministic multi-tape Turing machine in $O(f(n))$ space.
- NSPACE($f(n)$) is the class of languages that can be decided by a nondeterministic multi-tape Turing machine in $O(f(n))$ space.
- Why multi-tape TMs? Because t steps on any k-tape Turing machine can be simulated in $O(t \log t)$ steps on a two-tape machine [Hennie and Stearns 1966, essentially using lazy counters and amortization], and in $O(t^2)$ steps on a single-tape machine [Hartmanis and Stearns 1965; realign multiple tracks at every simulation step]. Moreover, the latter quadratic bound is tight [Hennie 1965 (palindromes, via communication complexity)].

38.4 Deterministic Simulation

Theorem 1. *For any nondeterministic Turing machine N, there is a deterministic Turing machine M that accepts exactly the same strings and N and rejects exactly the same strings as N. Moreover, if every computation path of N on input x halts after at most t steps, then M halts on input x after at most $O(t^2 r^{2t})$ steps, where r is the maximum size of any transition set in N.*

Proof: I'll describe a deterministic machine M that performs a breadth-first search of the computation tree of N. (The depth-first search performed by a standard recursive backtracking algorithm won't work here. If N's computation tree contains an infinite path, a depth-first search would get stuck in that path without exploring the rest of the tree.)

At the beginning of each simulation round, M's tape contains a string of the form

$$\square\square\cdots\square\bullet\bullet y_1 q_1 z_1 \bullet y_2 q_2 z_2 \bullet \cdots \bullet y_k q_k z_k \bullet\bullet$$

where each substring $y_i q_i z_i$ encodes a configuration $(q_i, y_i z_i, |y_i|)$ of some computation path of N, and $\bullet$ is a new symbol not in the tape alphabet of N. The machine M interprets this sequence of encoded configurations as a queue, with new configurations inserted on the right and old configurations removed

from the left. The double-separators $\bullet\bullet$ uniquely identify the start and end of this queue; outside this queue, the tape is entirely blank.

Specifically, in each round, first M appends the encodings of all configurations than N can reach in one transition from the first encoded configuration $(q_1, y_1 z_1, |y_1|)$; then M erases the first encoded configuration.

$$\cdots\square\square\bullet\bullet\underline{y_1 q_1 z_1}\bullet y_2 q_2 z_2 \bullet \cdots \bullet y_r q_r z_r \bullet\bullet\underline{\square\square\cdots}$$
$$\Downarrow \qquad\qquad\qquad \Downarrow$$
$$\cdots\square\square\underline{\square\square\cdots\square}\bullet\bullet y_2 q_2 z_2 \bullet \cdots \bullet y_k q_k z_k \bullet \underline{\tilde{y}_1\tilde{q}_1\tilde{z}_1 \bullet \tilde{y}_2\tilde{q}_2\tilde{z}_2 \bullet \cdots \bullet \tilde{y}_r\tilde{q}_r\tilde{z}_r \bullet\bullet\square\square\cdots}$$

Suppose each transition set $\delta_N(q,a)$ has size at most r. Then after simulating t steps of N, the tape string of M encoding $O(r^t)$ different configurations of N and therefore has length $L = O(tr^t)$ (not counting the initial blanks). If M begins each simulation phase by moving the initial configuration from the beginning to the end of the tape string, which takes $O(t^2 r^t)$ time, the time for the rest of the the simulation phase is negligible. Altogether, simulating all r^t possibilities for the the tth step of N requires $O(t^2 r^{2t})$ time. We conclude that M can simulate the first t steps of every computation path of N in $\boldsymbol{O(t^2 r^{2t})}$ ***time***, as claimed. □

The running time of this simulation is dominated by the time spent reading from one end of the tape string and writing to the other. It is fairly easy to reduce the running time to $O(tr^t)$ by using either two tapes (a "read tape" containing N-configurations at time t and a "write tape" containing N-configurations at time $t+1$) or two independent heads on the same tape (one at each end of the queue).

38.5 Nondeterminism as Advice

Any nondeterministic Turing machine N can also be simulated by a *deterministic* machine M with *two* inputs: the user input string $w \in \Sigma^*$, and a so-called ***advice*** string $x \in \Omega^*$, where Ω is another finite alphabet. Only the first input string w is actually given by the user. At least for now, we assume that the advice string x is given on a separate read-only tape.

The deterministic machine M simulates N step-by-step, but whenever N has a choice of how to transition, M reads a new symbol from the advice string, and that symbol determines the choice. In fact, without loss of generality, we can assume that M reads a new symbol from the advice string and moves the advice-tape's head to the right on *every* transition. Thus, M's transition function has the form $\delta_M : Q \times \Gamma \times \Omega \to Q \times \Gamma \times \{-1, +1\}$, and we require that

$$\delta_N(q,a) = \{\delta_M(q,a,\omega) \mid \omega \in \Omega\}$$

For example, if N has a binary choice

$$\delta_N(\mathsf{branch}, \mathsf{?}) = \big\{(\mathsf{left}, \mathsf{L}, -1), (\mathsf{right}, \mathsf{R}, +1)\big\},$$

then M might determine this choice by defining

$$\delta_M(\mathsf{branch}, \mathsf{?}, \mathtt{0}) = (\mathsf{left}, \mathsf{L}, -1) \qquad \text{and} \qquad \delta_M(\mathsf{branch}, \mathsf{?}, \mathtt{1}) = (\mathsf{right}, \mathsf{R}, +1)$$

More generally, if every set $\delta_N(p,a)$ has size r, then we let $\Omega = \{1, 2, \ldots, r\}$ and define $\delta_M(q,a,i)$ to be the ith element of $\delta_N(q,a)$ in some canonical order.

Now observe that N accepts a string w if and only if M accepts the pair (w,x) for *some* string $x \in \Omega^*$, and N rejects w if and only if M rejects the pair (w,x) for *all* strings $x \in \Omega^*$.

The "advice" formulation of nondeterminism allows a different strategy for simulation by a standard deterministic Turing machine, which is often called ***dovetailing***. Consider all possible advice strings x, in increasing order of length; listing these advice strings is equivalent to repeatedly incrementing a base-r counter. For each advice string x, simulate M on input (w, x) for exactly $|x|$ transitions.

```
DOVETAIL_M(w):
  for t ← 1 to ∞
      done ← TRUE
      for all strings x ∈ Ω^t
          if M accepts (w, x) in at most t steps
              accept
          if M(w, x) does not halt in at most t steps
              done ← FALSE
      if done
          reject
```

The most straightforward Turing-machine implementation of this algorithm requires three tapes: A read-only input tape containing w, an advice tape containing x (which is also used as a timer for the simulation), and the work tape. This simulation requires $O(tr^t)$ time to simulate all possibilities for t steps of the original non-deterministic machine N.

If we insist on using a standard Turing machine with a single tape and a single head, the simulation becomes slightly more complex, but (unlike our earlier queue-based strategy) not significantly slower. This standard machine S maintains a string of the form $\bullet w \bullet x \bullet z$, where z is the current work-tape string of M (or equivalently, of N), with marks (on a second track) indicating the current positions of the heads on M's work tape and M's advice tape. Simulating a single transition of M now requires $O(|x|)$ steps, because S needs to shuttle its single head between these two marks. Thus, S requires $\boldsymbol{O(t^2 r^t)}$ ***time*** to simulate all possibilities for t steps of the original non-deterministic machine N. This is significantly faster than the queue-based simulation, because we don't record (and therefore don't have to repeatedly scan over) intermediate configurations; recomputing everything from scratch is actually cheaper!

38.6 The Cook-Levin Theorem

★★★

Define SAT and CIRCUITSAT. Non-determinism is fundamentally different from other Turing machine extensions, in that it seems to provide an exponential speedup for some problems, just like NFAs can use exponentially fewer states than DFAs for the same language.

The Cook-Levin Theorem. *If SAT ∈ P, then P=NP.*

Proof: Let $L \subseteq \Sigma^*$ be an arbitrary language in NP, over some fixed alphabet Σ. There must be an integer k and Turing machine M that satisfies the following conditions:

- For all strings $w \in L$, there is at least one string $x \in \Sigma^*$ such that M accepts the string $w\square x$.
- For all strings $w \notin L$ and $x \in \Sigma^*$, M rejects the string $w\square x$.
- For all strings $w, x \in \Sigma^*$, M halts on input $w\square x$ after at most $\max\{1, |w|^k\}$ steps.

Now suppose we are given a string $w \in \Sigma^*$. Let $n = |w|$ and let $N = \max\{1, |w|^k\}$. We construct a boolean formula Φ_w that is satisfiable if and only if $w \in L$, by following the execution of M on input $w\square x$ for some unknown advice string x. Without loss of generality, we can assume that $|x| = N - n - 1$ (since we can extend any shorter string x with blanks.) Our formula Φ_w uses the following boolean variables for all symbols $a \in \Gamma$, all states $q \in Q$, and all integers $0 \le t \le N$ and $0 \le i \le N + 1$.

- $Q_{t,i,q}$ — M is in state q with its head at position i after t transitions.
- $T_{t,i,a}$ — The kth cell of M's work tape contains a after t transitions.

The formula Φ_w is the conjunction of the following constraints:

- **Boundaries:** To simplify later constraints, we include artificial boundary variables just past both ends of the tape:

$$\begin{aligned} Q_{t,i,q} = Q_{t,N+1,q} &= \textsc{False} && \text{for all } 0 \le t \le N \text{ and } q \in Q \\ T_{t,0,a} = T_{t,N+1,a} &= \textsc{False} && \text{for all } 0 \le t \le N \text{ and } a \in \Gamma \end{aligned}$$

- **Initialization:** We have the following values for variables with $t = 0$:

$$\begin{aligned} Q_{0,1,\text{start}} &= \textsc{True} \\ Q_{0,1,q} &= \textsc{False} && \text{for all } q \neq \text{start} \\ H_{0,i,q} &= \textsc{False} && \text{for all } i \neq 1 \text{ and } q \in Q \\ T_{0,i,w_i} &= \textsc{True} && \text{for all } 1 \le i \le n \\ T_{0,i,a} &= \textsc{False} && \text{for all } 1 \le i \le n \text{ and } a \neq w_i \\ T_{0,n+1,\square} &= \textsc{True} \\ T_{0,i,a} &= \textsc{False} && \text{for all } a \neq \square \end{aligned}$$

- **Uniqueness:** The variables $T_{0,i,a}$ with $n+2 \le i \le N$ represent the unknown advice string x; these are the "inputs" to Φ_w. We need some additional constraints ensure that for each i, *exactly one* of these variables is $\textsc{True}$:

$$\left(\bigvee_{a \in \Gamma} T_{0,j,a}\right) \wedge \bigwedge_{a \neq b} \left(\overline{T_{0,j,a}} \vee \overline{T_{0,j,b}}\right)$$

- **Transitions:** For all $1 \le t \le N$ and $1 \le i \le N$, the following constraints simulate the transition from time $t-1$ to time t.

$$Q_{t,i,q} = \bigvee_{\delta(p,a)=(q,\cdot,+1)} \left(Q_{t-1,i-1,p} \wedge T_{t,i-1,a}\right) \vee \bigvee_{\delta(p,a)=(q,\cdot,-1)} \left(Q_{t-i,i+1,p} \wedge T_{t,i+1,a}\right)$$

$$T_{t,i,b} = \bigvee_{\delta(p,a)=(\cdot,b,\cdot)} \left(Q_{t-1,i,p} \wedge T_{t-1,i,a}\right) \vee \left(\bigwedge_{q \in Q} \overline{Q_{t-1,i,q}} \wedge T_{t-1,i,b}\right)$$

- **Output:** We have one final constraint that indicates acceptance.

$$z = \bigvee_{t=0}^{N} \bigvee_{i=1}^{N} Q_{t,i,\text{accept}}$$

A straightforward induction argument implies that *without the acceptance constraint*, any assignment of values to the unknown variables $T_{0,i,a}$ that satisfies the uniqueness constraints determines *unique* values for the other variables in Φ_w, which consistently describe the execution of M. Thus, Φ_w is satisfiable if and only if for some input values $T_{0,i,a}$, the resulting , including acceptance. In other words, Φ_w is

satisfiable if and only if there is a string $x \in \Gamma^*$ such that M accepts the input $w\square x$. We conclude that Φ_w is satisfiable if and only if $w \in L$.

For any any string w of length n, the formula Φ_w has $O(N^2)$ variables and $O(N^2)$ constraints (where the hidden constants depend on the machine M). Every constraint except acceptance has constant length, so altogether Φ_w has length $O(N^2)$. Moreover, we can construct Φ_w in $O(N^2) = O(n^{2k})$ time.

In conclusion, if we could decide SAT for formulas of size n in (say) $O(n^c)$ time, then we could decide membership in L in $O(n^{2kc})$ time, which implies that $L \in \mathrm{P}$. □

Exercises

1. Prove that the following problem is NP-hard, *without* using the Cook-Levin Theorem. Given a string $\langle M, w\rangle$ that encodes a non-deterministic Turing machine M and a string w, does M accept w in at most $|w|$ transitions?

★★★ More excerises!

[I]n his short and broken treatise he provides an eternal example—not of laws, or even of method, for ***there is no method except to be very intelligent****, but of intelligence itself swiftly operating the analysis of sensation to the point of principle and definition.*

— T. S. Eliot on Aristotle, "The Perfect Critic", *The Sacred Wood* (1921)

The nice thing about standards is that you have so many to choose from; furthermore, if you do not like any of them, you can just wait for next year's model.

— Andrew S. Tannenbaum, *Computer Networks* (1981)
Also attributed to Grace Murray Hopper and others

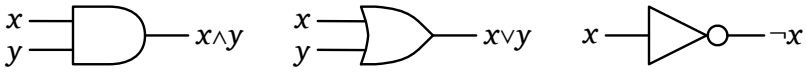
If a problem has no solution, it may not be a problem, but a fact — not to be solved, but to be coped with over time.

— Shimon Peres, as quoted by David Rumsfeld, *Rumsfeld's Rules* (2001)

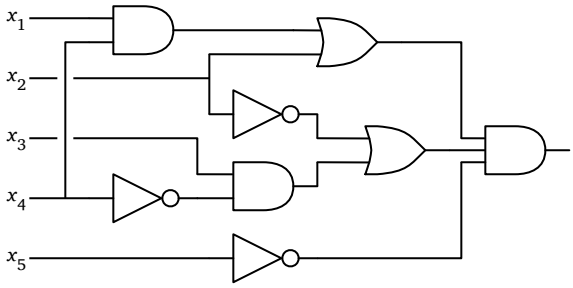
40 NP-Hard Problems

40.1 A Game You Can't Win

A salesman in a red suit who looks suspiciously like Tom Waits presents you with a black steel box with n binary switches on the front and a light bulb on the top. The salesman tells you that the state of the light bulb is controlled by a complex *boolean circuit*—a collection of AND, OR, and NOT gates connected by wires, with one input wire for each switch and a single output wire for the light bulb. He then asks you the following question: Is there a way to set the switches so that the light bulb turns on? If you can answer this question correctly, he will give you the box and a ~~million billion~~ trillion dollars; if you answer incorrectly, or if you die without answering at all, he will take your soul.



An AND gate, an OR gate, and a NOT gate.



A boolean circuit. inputs enter from the left, and the output leaves to the right.

As far as you can tell, the Adversary hasn't connected the switches to the light bulb at all, so no matter how you set the switches, the light bulb will stay off. If you declare that it *is* possible to turn on the light, the Adversary will open the box and reveal that there is no circuit at all. But if you declare that it is *not* possible to turn on the light, before testing all 2^n settings, the Adversary will magically create a circuit inside the box that turns on the light *if and only if* the switches are in one of the settings you haven't tested, and then flip the switches to that setting, turning on the light. (You can't detect the Adversary's cheating, because you can't see inside the box until the end.) The only way to *provably*

answer the Adversary's question correctly is to try all 2^n possible settings. You quickly realize that this will take *far* longer than you expect to live, so you gracefully decline the Adversary's offer.

The Adversary smiles and says, "Ah, yes, of course, you have no reason to trust me. But perhaps I can set your mind at ease." He hands you a large roll of parchment—which you hope was made from sheep skin—with a circuit diagram drawn (or perhaps tattooed) on it. "Here are the complete plans for the circuit inside the box. Feel free to poke around inside the box to make sure the plans are correct. Or build your own circuit from these plans. Or write a computer program to simulate the circuit. Whatever you like. If you discover that the plans don't match the actual circuit in the box, you win the trillion bucks." A few spot checks convince you that the plans have no obvious flaws; subtle cheating appears to be impossible.

But you should still decline the Adversary's generous offer. The problem that the Adversary is posing is called ***circuit satisfiability*** or ***CircuitSat***: Given a boolean circuit, is there is a set of inputs that makes the circuit output True, or conversely, whether the circuit *always* outputs False. For any particular input setting, we can calculate the output of the circuit in polynomial (actually, *linear*) time using depth-first-search. But nobody knows how to solve CircuitSat faster than just trying all 2^n possible inputs to the circuit, but this requires exponential time. On the other hand, nobody has actually *proved* that this is the best we can do; maybe there's a clever algorithm that just hasn't been discovered yet!

40.2 P versus NP

A minimal requirement for an algorithm to be considered "efficient" is that its running time is polynomial: $O(n^c)$ for some constant c, where n is the size of the input.[1] Researchers recognized early on that not all problems can be solved this quickly, but had a hard time figuring out exactly which ones could and which ones couldn't. There are several so-called ***NP-hard*** problems, which most people believe *cannot* be solved in polynomial time, even though nobody can prove a super-polynomial lower bound.

A *decision problem* is a problem whose output is a single boolean value: Yes or No. Let me define three classes of decision problems:

- ***P*** is the set of decision problems that can be solved in polynomial time. Intuitively, P is the set of problems that can be solved quickly.
- ***NP*** is the set of decision problems with the following property: If the answer is Yes, then there is a *proof* of this fact that can be checked in polynomial time. Intuitively, NP is the set of decision problems where we can verify a Yes answer quickly if we have the solution in front of us.
- ***co-NP*** is essentially the opposite of NP. If the answer to a problem in co-NP is No, then there is a proof of this fact that can be checked in polynomial time.

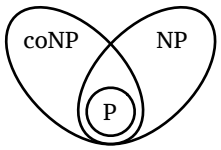
For example, the circuit satisfiability problem is in NP. If the answer is Yes, then any set of m input values that produces True output is a proof of this fact; we can check the proof by evaluating the circuit in polynomial time. It is widely believed that circuit satisfiability is *not* in P or in co-NP, but nobody actually knows.

Every decision problem in P is also in NP. If a problem is in P, we can verify Yes answers in polynomial time recomputing the answer from scratch! Similarly, every problem in P is also in co-NP.

[1] This notion of efficiency was independently formalized by Alan Cobham (The intrinsic computational difficulty of functions. *Logic, Methodology, and Philosophy of Science (Proc. Int. Congress)*, 24–30, 1965), Jack Edmonds (Paths, trees, and flowers. *Canadian Journal of Mathematics* 17:449–467, 1965), and Michael Rabin (Mathematical theory of automata. *Proceedings of the 19th ACM Symposium in Applied Mathematics*, 153–175, 1966), although similar notions were considered more than a decade earlier by Kurt Gödel and John von Neumann.

Perhaps the single most important unanswered question in theoretical computer science—if not all of computer science—if not all of ***science***—is whether the complexity classes P and NP are actually different. Intuitively, it seems obvious to most people that P $\neq$ NP; the homeworks and exams in this class and others have (I hope) convinced you that problems can be incredibly hard to solve, even when the solutions are obvious in retrospect. It's completely obvious; *of course* solving problems from scratch is harder than just checking that a solution is correct. But nobody knows how to prove it! The Clay Mathematics Institute lists P versus NP as the first of its seven Millennium Prize Problems, offering a $1,000,000 reward for its solution. And yes, in fact, several people *have* lost their souls attempting to solve this problem.

A more subtle but still open question is whether the complexity classes NP and co-NP are different. Even if we can verify every YES answer quickly, there's no reason to believe we can also verify NO answers quickly. For example, as far as we know, there is no short proof that a boolean circuit is *not* satisfiable. It is generally believed that NP $\neq$ co-NP, but nobody knows how to prove it.



What we *think* the world looks like.

40.3 NP-hard, NP-easy, and NP-complete

A problem Π is ***NP-hard*** if a polynomial-time algorithm for Π would imply a polynomial-time algorithm for *every problem in NP*. In other words:

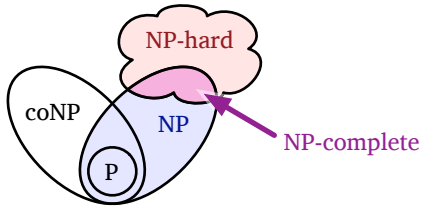
Π is NP-hard $\iff$ If Π can be solved in polynomial time, then P=NP

Intuitively, if we could solve one particular NP-hard problem quickly, then we could quickly solve *any* problem whose solution is easy to understand, using the solution to that one special problem as a subroutine. NP-hard problems are at least as hard as any problem in NP.

Calling a problem NP-hard is like saying 'If I own a dog, then it can speak fluent English.' You probably don't know whether or not I own a dog, but I bet you'repretty sure that I don't own a *talking* dog. Nobody has a mathematical *proof* that dogs can't speak English—the fact that no one has ever heard a dog speak English is evidence, as are the hundreds of examinations of dogs that lacked the proper mouth shape and brainpower, but mere evidence is not a mathematical proof. Nevertheless, no sane person would believe me if I said I owned a dog that spoke fluent English. So the statement 'If I own a dog, then it can speak fluent English' has a natural corollary: No one in their right mind should believe that I own a dog! Likewise, if a problem is NP-hard, no one in their right mind should believe it can be solved in polynomial time.

Finally, a problem is ***NP-complete*** if it is both NP-hard and an element of NP (or 'NP-easy'). NP-complete problems are the hardest problems in NP. If anyone finds a polynomial-time algorithm for even one NP-complete problem, then that would imply a polynomial-time algorithm for *every* NP-complete problem. Literally *thousands* of problems have been shown to be NP-complete, so a polynomial-time algorithm for one (and therefore all) of them seems incredibly unlikely.

It is not immediately clear that *any* decision problems are NP-hard or NP-complete. NP-hardness is already a lot to demand of a problem; insisting that the problem also have a nondeterministic polynomial-time algorithm seems almost completely unreasonable. The following remarkable theorem



More of what we *think* the world looks like.

was first published by Steve Cook in 1971 and independently by Leonid Levin in 1973.[2] I won't even sketch the proof, since I've been (deliberately) vague about the definitions.

The Cook-Levin Theorem. *Circuit satisfiability is NP-complete.*

*40.4 Formal Definitions (*HC SVNT DRACONES*)

Formally, the complexity classes P, NP, and co-NP are defined in terms of *languages* and *Turing machines*. A language is just a set of strings over some finite alphabet Σ; without loss of generality, we can assume that $\Sigma = \{0, 1\}$. P is the set of languages that can be decided in ***P***olynomial time by a deterministic single-tape Turing machine. Similarly, NP is the set of all languages that can be decided in polynomial time by a nondeterministic Turing machine; NP is an abbreviation for ***N***ondeterministic ***P***olynomial-time.

Polynomial time is a sufficient crude requirement that the precise form of Turing machine (number of heads, number of tracks, and so one) is unimportant. In fact, careful analysis of the arguments in Lecture 36 imply that any algorithm that runs on a random-access machine[3] in $T(n)$ time can be simulated by a single-tape, single-track, single-head Turing machine that runs in $O(T(n)^3)$ time. This simulation result allows us to argue formally about computational complexity in terms of standard high-level programming constructs like for-loops and recursion, instead of describing everything directly in terms of Turing machines.

A problem Π is formally NP-hard if and only if, for every language $\Pi' \in$ NP, there is a polynomial-time ***Turing reduction*** from Π' to Π. A Turing reduction just means a reduction that can be executed on a Turing machine; that is, a Turing machine M that can solve Π' using another Turing machine M' for Π as a black-box subroutine. Turing reductions are also called *oracle reductions*; polynomial-time Turing reductions are also called *Cook reductions*.

Researchers in complexity theory prefer to define NP-hardness in terms of polynomial-time ***many-one reductions***, which are also called *Karp reductions*. A *many-one* reduction from one language $L' \subseteq \Sigma^*$ to another language $L \subseteq \Sigma^*$ is an function $f : \Sigma^* \to \Sigma^*$ such that $x \in L'$ if and only if $f(x) \in L$. Then we can define a *language* L to be NP-hard if and only if, for any language $L' \in$ NP, there is a many-one reduction from L' to L that can be computed in polynomial time.

Every Karp reduction "is" a Cook reduction, but not vice versa. Specifically, any Karp reduction from one decision problem Π to another decision Π' is equivalent to transforming the input to Π into the

[2] Levin first reported his results at seminars in Moscow in 1971, while still a PhD student. News of Cook's result did not reach the Soviet Union until at least 1973, after Levin's announcement of his results had been published; in accordance with Stigler's Law, this result is often called 'Cook's Theorem'. Levin was denied his PhD at Moscow University for political reasons; he emigrated to the US in 1978 and earned a PhD at MIT a year later. Cook was denied tenure at Berkeley in 1970, just one year before publishing his seminal paper; he (but not Levin) later won the Turing award for his proof.

[3] Random-access machines are a model of computation that more faithfully models physical computers. A random-access machine has unbounded random-access memory, modeled as an array $M[0..\infty]$ where each address $M[i]$ holds a single w-bit integer, for some fixed integer w, and can read to or write from any memory addresses in constant time. RAM algorithms are formally written in assembly-like language, using instructions like ***ADD i, j, k*** (meaning "$M[i] \leftarrow M[j] + M[k]$"), ***INDIR i, j*** (meaning "$M[i] \leftarrow M[M[j]]$"), and ***IFZGOTO i, ℓ*** (meaning "if $M[i] = 0$, go to line ℓ"). In practice, RAM algorithms can be faithfully described using higher-level pseudocode, as long as we're careful about arithmetic precision.

input for Π', invoking an oracle (that is, a subroutine) for Π', and then returning the answer verbatim. However, as far as we know, not every Cook reduction can be simulated by a Karp reduction.

Complexity theorists prefer Karp reductions primarily because NP is closed under Karp reductions, but is *not* closed under Cook reductions (unless NP=co-NP, which is considered unlikely). There are natural problems that are (1) NP-hard with respect to Cook reductions, but (2) NP-hard with respect to Karp reductions only if P=NP. One trivial example is of such a problem is UNSAT: Given a boolean formula, is it *always false*? On the other hand, many-one reductions apply *only* to decision problems (or more formally, to languages); formally, no optimization or construction problem is Karp-NP-hard.

To make things even more confusing, both Cook and Karp originally defined NP-hardness in terms of ***logarithmic-space*** reductions. Every logarithmic-space reduction is a polynomial-time reduction, but (as far as we know) not vice versa. It is an open question whether relaxing the set of allowed (Cook or Karp) reductions from logarithmic-space to polynomial-time changes the set of NP-hard problems.

Fortunately, none of these subtleties raise their ugly heads in practice—in particular, every algorithmic reduction described in these notes can be formalized as a logarithmic-space many-one reduction—so you can wake up now.

40.5 Reductions and SAT

To prove that any problem other than Circuit satisfiability is NP-hard, we use a *reduction argument*. Reducing problem A to another problem B means describing an algorithm to solve problem A under the assumption that an algorithm for problem B already exists. You're already used to doing reductions, only you probably call it something else, like writing subroutines or utility functions, or modular programming. To prove something is NP-hard, we describe a similar transformation between problems, but not in the direction that most people expect.

You should tattoo the following rule of onto the back of your hand, right next to your Mom's birthday and the *actual* rules of Monopoly.[4]

> **To prove that problem *A* is NP-hard, reduce a known NP-hard problem to *A*.**

In other words, to prove that your problem is hard, you need to describe an algorithm to solve a *different* problem, which you already know is hard, using a mythical algorithm for *your* problem as a subroutine. The essential logic is a proof by contradiction. Your reduction shows implies that if your problem were easy, then the other problem would be easy, too. Equivalently, since you know the other problem is hard, your problem must also be hard.

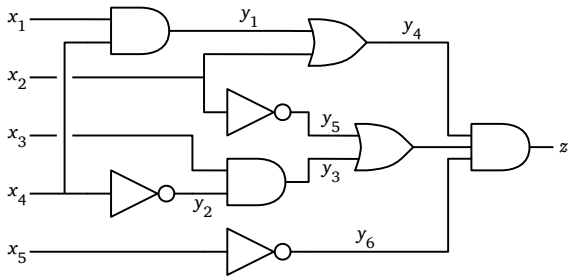
For example, consider the *formula satisfiability* problem, usually just called ***SAT***. The input to SAT is a boolean *formula* like

$$(a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee \overline{(\bar{a} \Rightarrow d)} \vee (c \neq a \wedge b)),$$

and the question is whether it is possible to assign boolean values to the variables $a, b, c, \ldots$ so that the formula evaluates to TRUE.

[4] If a player lands on an available property and declines (or is unable) to buy it, that property is immediately auctioned off to the highest bidder; the player who originally declined the property may bid, and bids may be arbitrarily higher or lower than the list price. Players in Jail can still buy and sell property, buy and sell houses and hotels, and collect rent. The game has 32 houses and 12 hotels; once they're gone, they're gone. In particular, if all houses are already on the board, you cannot downgrade a hotel to four houses; you must sell all three hotels in the group. Players can sell/exchange undeveloped properties, but not buildings or cash. A player landing on Free Parking does not win anything. A player landing on Go gets $200, no more. Railroads are not magic transporters. Finally, Jeff *always* gets the car.

To show that SAT is NP-hard, we need to give a reduction from a known NP-hard problem. The only problem we know is NP-hard so far is circuit satisfiability, so let's start there. Given a boolean circuit, we can transform it into a boolean formula by creating new output variables for each gate, and then just writing down the list of gates separated by ANDs. For example, we can transform the example circuit into a formula as follows:



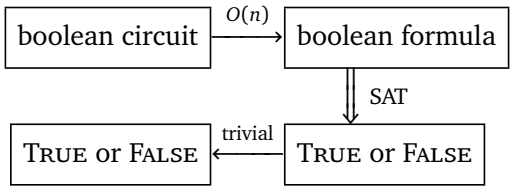
$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge$$
$$(y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (z = y_4 \wedge y_7 \wedge y_6) \wedge z$$

A boolean circuit with gate variables added, and an equivalent boolean formula.

Now the original circuit is satisfiable if and only if the resulting formula is satisfiable. Given a satisfying input to the circuit, we can get a satisfying assignment for the formula by computing the output of every gate. Given a satisfying assignment for the formula, we can get a satisfying input the the circuit by just ignoring the internal gate variables y_i and the output variable z.

We can transform any boolean circuit into a formula in linear time using depth-first search, and the size of the resulting formula is only a constant factor larger than the size of the circuit. Thus, we have a polynomial-time reduction from circuit satisfiability to SAT:

★★★ Redraw reduction cartoons so that the *boxes* represent algorithms, not the arrows.



$$T_{\text{CSAT}}(n) \le O(n) + T_{\text{SAT}}(O(n)) \quad \Longrightarrow \quad T_{\text{SAT}}(n) \ge T_{\text{CSAT}}(\Omega(n)) - O(n)$$

The reduction implies that if we had a polynomial-time algorithm for SAT, then we'd have a polynomial-time algorithm for circuit satisfiability, which would imply that P=NP. So SAT is NP-hard.

To prove that a boolean formula is satisfiable, we only have to specify an assignment to the variables that makes the formula TRUE. We can check the proof in linear time just by reading the formula from left to right, evaluating as we go. So SAT is also in NP, and thus is actually NP-complete.

40.6 3SAT (from SAT)

A special case of SAT that is particularly useful in proving NP-hardness results is called *3SAT*.

A boolean formula is in *conjunctive normal form* (CNF) if it is a conjunction (AND) of several *clauses*, each of which is the disjunction (OR) of several *literals*, each of which is either a variable or its negation.

For example:

$$\overbrace{(a \vee b \vee c \vee d)}^{\text{clause}} \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b})$$

A *3CNF* formula is a CNF formula with exactly three literals per clause; the previous example is not a 3CNF formula, since its first clause has four literals and its last clause has only two. 3SAT is just SAT restricted to 3CNF formulas: Given a 3CNF formula, is there an assignment to the variables that makes the formula evaluate to TRUE?

We could prove that 3SAT is NP-hard by a reduction from the more general SAT problem, but it's easier just to start over from scratch, with a boolean circuit. We perform the reduction in several stages.

1. *Make sure every AND and OR gate has only two inputs.* If any gate has $k > 2$ inputs, replace it with a binary tree of $k-1$ two-input gates.

2. *Write down the circuit as a formula, with one clause per gate.* This is just the previous reduction.

3. *Change every gate clause into a CNF formula.* There are only three types of clauses, one for each type of gate:

$$\begin{aligned} a = b \wedge c &\longmapsto (a \vee \bar{b} \vee \bar{c}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee c) \\ a = b \vee c &\longmapsto (\bar{a} \vee b \vee c) \wedge (a \vee \bar{b}) \wedge (a \vee \bar{c}) \\ a = \bar{b} &\longmapsto (a \vee b) \wedge (\bar{a} \vee \bar{b}) \end{aligned}$$

4. *Make sure every clause has exactly three literals.* Introduce new variables into each one- and two-literal clause, and expand it into two clauses as follows:

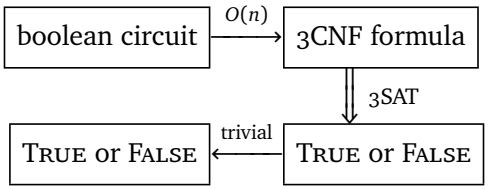
$$\begin{aligned} a &\longmapsto (a \vee x \vee y) \wedge (a \vee \bar{x} \vee y) \wedge (a \vee x \vee \bar{y}) \wedge (a \vee \bar{x} \vee \bar{y}) \\ a \vee b &\longmapsto (a \vee b \vee x) \wedge (a \vee b \vee \bar{x}) \end{aligned}$$

For example, if we start with the same example circuit we used earlier, we obtain the following 3CNF formula. Although this may look a lot more ugly and complicated than the original circuit at first glance, it's actually only a constant factor larger—every binary gate in the original circuit has been transformed into at most five clauses. Even if the formula size were a large *polynomial* function (like n^{573}) of the circuit size, we would still have a valid reduction.

$$\begin{aligned} &(y_1 \vee \overline{x_1} \vee \overline{x_4}) \wedge (\overline{y_1} \vee x_1 \vee z_1) \wedge (\overline{y_1} \vee x_1 \vee \overline{z_1}) \wedge (\overline{y_1} \vee x_4 \vee z_2) \wedge (\overline{y_1} \vee x_4 \vee \overline{z_2}) \\ &\wedge (y_2 \vee x_4 \vee z_3) \wedge (y_2 \vee x_4 \vee \overline{z_3}) \wedge (\overline{y_2} \vee \overline{x_4} \vee z_4) \wedge (\overline{y_2} \vee \overline{x_4} \vee \overline{z_4}) \\ &\wedge (y_3 \vee \overline{x_3} \vee \overline{y_2}) \wedge (\overline{y_3} \vee x_3 \vee z_5) \wedge (\overline{y_3} \vee x_3 \vee \overline{z_5}) \wedge (\overline{y_3} \vee y_2 \vee z_6) \wedge (\overline{y_3} \vee y_2 \vee \overline{z_6}) \\ &\wedge (\overline{y_4} \vee y_1 \vee x_2) \wedge (y_4 \vee \overline{x_2} \vee z_7) \wedge (y_4 \vee \overline{x_2} \vee \overline{z_7}) \wedge (y_4 \vee \overline{y_1} \vee z_8) \wedge (y_4 \vee \overline{y_1} \vee \overline{z_8}) \\ &\wedge (y_5 \vee x_2 \vee z_9) \wedge (y_5 \vee x_2 \vee \overline{z_9}) \wedge (\overline{y_5} \vee \overline{x_2} \vee z_{10}) \wedge (\overline{y_5} \vee \overline{x_2} \vee \overline{z_{10}}) \\ &\wedge (y_6 \vee x_5 \vee z_{11}) \wedge (y_6 \vee x_5 \vee \overline{z_{11}}) \wedge (\overline{y_6} \vee \overline{x_5} \vee z_{12}) \wedge (\overline{y_6} \vee \overline{x_5} \vee \overline{z_{12}}) \\ &\wedge (\overline{y_7} \vee y_3 \vee y_5) \wedge (y_7 \vee \overline{y_3} \vee z_{13}) \wedge (y_7 \vee \overline{y_3} \vee \overline{z_{13}}) \wedge (y_7 \vee \overline{y_5} \vee z_{14}) \wedge (y_7 \vee \overline{y_5} \vee \overline{z_{14}}) \\ &\wedge (y_8 \vee \overline{y_4} \vee \overline{y_7}) \wedge (\overline{y_8} \vee y_4 \vee z_{15}) \wedge (\overline{y_8} \vee y_4 \vee \overline{z_{15}}) \wedge (\overline{y_8} \vee y_7 \vee z_{16}) \wedge (\overline{y_8} \vee y_7 \vee \overline{z_{16}}) \\ &\wedge (y_9 \vee \overline{y_8} \vee \overline{y_6}) \wedge (\overline{y_9} \vee y_8 \vee z_{17}) \wedge (\overline{y_9} \vee y_8 \vee \overline{z_{17}}) \wedge (\overline{y_9} \vee y_6 \vee z_{18}) \wedge (\overline{y_9} \vee y_6 \vee \overline{z_{18}}) \\ &\wedge (y_9 \vee z_{19} \vee z_{20}) \wedge (y_9 \vee \overline{z_{19}} \vee z_{20}) \wedge (y_9 \vee z_{19} \vee \overline{z_{20}}) \wedge (y_9 \vee \overline{z_{19}} \vee \overline{z_{20}}) \end{aligned}$$

This process transforms the circuit into an equivalent 3CNF formula; the output formula is satisfiable if and only if the input circuit is satisfiable. As with the more general SAT problem, the formula is only a

constant factor larger than any reasonable description of the original circuit, and the reduction can be carried out in polynomial time. Thus, we have a polynomial-time reduction from circuit satisfiability to 3SAT:



$$T_{\text{CSAT}}(n) \le O(n) + T_{\text{3SAT}}(O(n)) \quad \Longrightarrow \quad T_{\text{3SAT}}(n) \ge T_{\text{CSAT}}(\Omega(n)) - O(n)$$

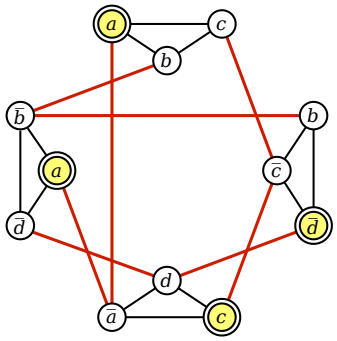
We conclude 3SAT is NP-hard. And because 3SAT is a special case of SAT, it is also in NP. Therefore, 3SAT is NP-complete.

40.7 Maximum Independent Set (from 3SAT)

For the next few problems we consider, the input is a simple, unweighted graph, and the problem asks for the size of the largest or smallest subgraph satisfying some structural property.

Let G be an arbitrary graph. An ***independent set*** in G is a subset of the vertices of G with no edges between them. The *maximum independent set* problem, or simply ***MAXINDSET***, asks for the size of the largest independent set in a given graph.

I'll prove that MAXINDSET is NP-hard (but not NP-complete, since it isn't a decision problem) using a reduction from 3SAT. I'll describe a reduction from a 3CNF formula into a graph that has an independent set of a certain size if and only if the formula is satisfiable. The graph has one node for each instance of each literal in the formula. Two nodes are connected by an edge if (1) they correspond to literals in the same clause, or (2) they correspond to a variable and its inverse. For example, the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ is transformed into the following graph.



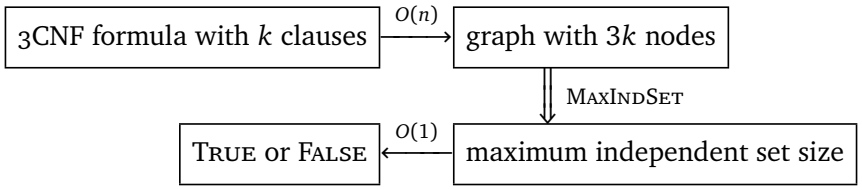
A graph derived from a 3CNF formula, and an independent set of size 4.
Black edges join literals from the same clause; red (heavier) edges join contradictory literals.

Now suppose the original formula had k clauses. Then I claim that the formula is satisfiable if and only if the graph has an independent set of size k.

1. **independent set $\Longrightarrow$ satisfying assignment:** If the graph has an independent set of k vertices, then each vertex must come from a different clause. To obtain a satisfying assignment, we assign the value TRUE to each literal in the independent set. Since contradictory literals are connected by edges, this assignment is consistent. There may be variables that have no literal in the independent set; we can set these to any value we like. The resulting assignment satisfies the original 3CNF formula.

2. **satisfying assignment $\Longrightarrow$ independent set:** If we have a satisfying assignment, then we can choose one literal in each clause that is TRUE. Those literals form an independent set in the graph.

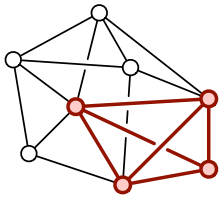
Thus, the reduction is correct. Since the reduction from 3CNF formula to graph takes polynomial time, we conclude that MAXINDSET is NP-hard. Here's a diagram of the reduction:



$$T_{3SAT}(n) \le O(n) + T_{MAXINDSET}(O(n)) \quad \Longrightarrow \quad T_{MAXINDSET}(n) \ge T_{3SAT}(\Omega(n)) - O(n)$$

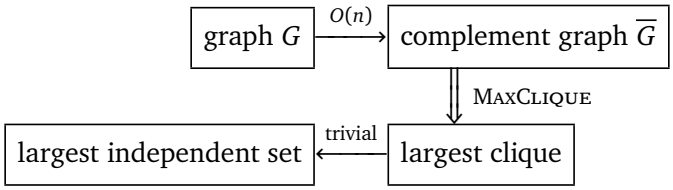
40.8 Clique (from Independent Set)

A *clique* is another name for a complete graph, that is, a graph where every pair of vertices is connected by an edge. The *maximum clique size* problem, or simply MAXCLIQUE, is to compute, given a graph, the number of nodes in its largest complete subgraph.



A graph with maximum clique size 4.

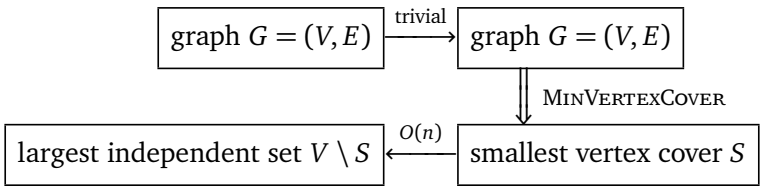
There is an easy proof that MAXCLIQUE is NP-hard, using a reduction from MAXINDSET. Any graph G has an *edge-complement* $\overline{G}$ with the same vertices, but with exactly the opposite set of edges—(u, v) is an edge in $\overline{G}$ if and only if it is *not* an edge in G. A set of vertices is independent in G if and only if the same vertices define a clique in $\overline{G}$. Thus, we can compute the largest independent in a graph simply by computing the largest clique in the complement of the graph.



40.9 Vertex Cover (from Independent Set)

A *vertex cover* of a graph is a set of vertices that touches every edge in the graph. The MINVERTEXCOVER problem is to find the smallest vertex cover in a given graph.

Again, the proof of NP-hardness is simple, and relies on just one fact: If I is an independent set in a graph $G = (V, E)$, then $V \setminus I$ is a vertex cover. Thus, to find the *largest* independent set, we just need to find the vertices that aren't in the *smallest* vertex cover of the same graph.



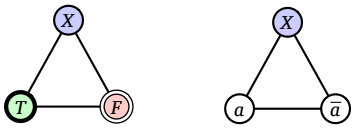
40.10 Graph Coloring (from 3SAT)

A *k-coloring* of a graph is a map $C : V \to \{1, 2, \ldots, k\}$ that assigns one of k 'colors' to each vertex, so that every edge has two different colors at its endpoints. The graph coloring problem is to find the smallest possible number of colors in a legal coloring. To show that this problem is NP-hard, it's enough to consider the special case 3COLORABLE: Given a graph, does it have a 3-coloring?

To prove that 3COLORABLE is NP-hard, we use a reduction from 3SAT. Given a 3CNF formula Φ, we produce a graph G_Φ as follows. The graph consists of a *truth* gadget, one *variable* gadget for each variable in the formula, and one *clause* gadget for each clause in the formula.

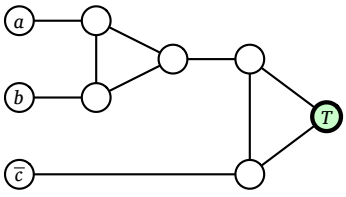
- The truth gadget is just a triangle with three vertices T, F, and X, which intuitively stand for TRUE, FALSE, and OTHER. Since these vertices are all connected, they must have different colors in any 3-coloring. For the sake of convenience, we will *name* those colors TRUE, FALSE, and OTHER. Thus, when we say that a node is colored TRUE, all we mean is that it must be colored the same as the node T.

- The variable gadget for a variable a is also a triangle joining two new nodes labeled a and $\bar{a}$ to node X in the truth gadget. Node a must be colored either TRUE or FALSE, and so node $\bar{a}$ must be colored either FALSE or TRUE, respectively.



The truth gadget and a variable gadget for a.

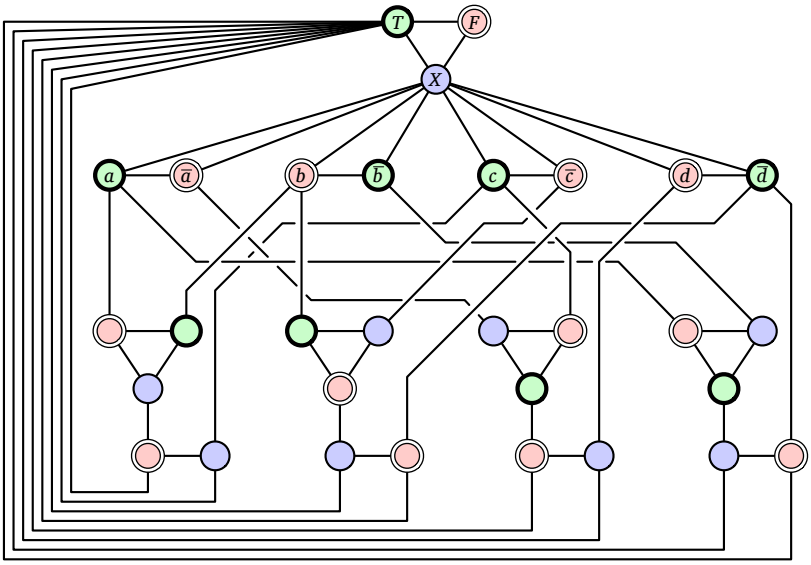
- Finally, each clause gadget joins three literal nodes to node T in the truth gadget using five new unlabeled nodes and ten edges; see the figure below. A straightforward case analysis implies that if all three literal nodes in the clause gadget are colored FALSE, then some edge in the gadget must be monochromatic. Since the variable gadgets force each literal node to be colored either TRUE or FALSE, in any valid 3-coloring, at least one of the three literal nodes is colored TRUE. On the other hand, for any coloring of the literal nodes where at least one literal node is colored TRUE, there is a valid 3-coloring of the clause gadget.



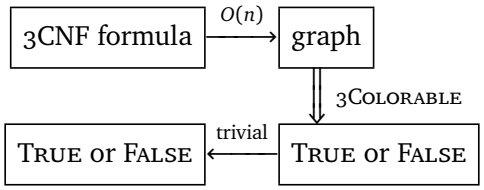
A clause gadget for $(a \vee b \vee \bar{c})$.

The final graph G_Φ contains exactly *one* node T, exactly *one* node F, and exactly *two* nodes a and $\bar{a}$ for each variable. For example, the formula $(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$ that I used to illustrate the MAXCLIQUE reduction would be transformed into the graph shown on the next page. The 3-coloring is one of several that correspond to the satisfying assignment $a = c =$ TRUE, $b = d =$ FALSE.

Now the proof of correctness is just brute force case analysis. If the graph is 3-colorable, then we can extract a satisfying assignment from any 3-coloring—at least one of the three literal nodes in every clause gadget is colored TRUE. Conversely, if the formula is satisfiable, then we can color the graph according to any satisfying assignment.



A 3-colorable graph derived from the satisfiable 3CNF formula
$(a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})$



We can easily verify that a graph has been correctly 3-colored in linear time: just compare the endpoints of every edge. Thus, 3COLORING is in NP, and therefore NP-complete. Moreover, since 3COLORING is a special case of the more general graph coloring problem—What is the minimum number of colors?—the more problem is also NP-hard, but *not* NP-complete, because it's not a decision problem.

40.11 Hamiltonian Cycle (from Vertex Cover)

A ***Hamiltonian cycle*** in a graph is a cycle that visits every vertex exactly once. This is very different from an *Eulerian cycle*, which is actually a closed *walk* that traverses every *edge* exactly once. Eulerian cycles are easy to find and construct in linear time using a variant of depth-first search.

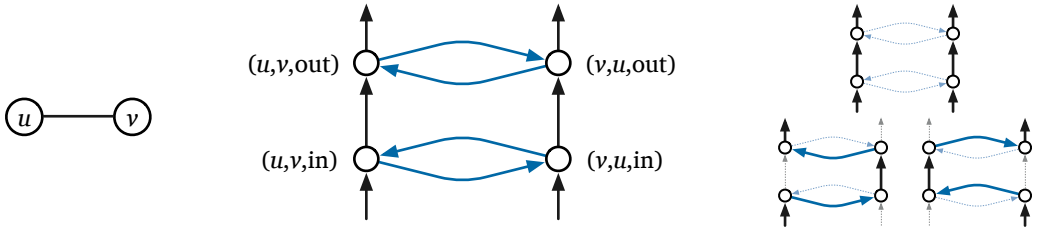
To prove that finding a Hamiltonian cycle in a directed graph is NP-hard, we describe a reduction from the vertex cover problem. Given an undirected graph G and an integer k, we need to transform it into another graph H, such that H has a Hamiltonian cycle if and only if G has a vertex cover of size k. As usual, our transformation uses several gadgets.

- For each undirected edge uv in G, the directed graph H contains an *edge gadget* consisting of four vertices $(u, v, \text{in}), (u, v, \text{out}), (v, u, \text{in}), (v, u, \text{out})$ and six directed edges

$$\begin{array}{lll} (u,v,\text{in}) \to (u,v,\text{out}) & (u,v,\text{in}) \to (v,u,\text{in}) & (v,u,\text{in}) \to (u,v,\text{in}) \\ (v,u,\text{in}) \to (v,u,\text{out}) & (u,v,\text{out}) \to (v,u,\text{out}) & (v,u,\text{out}) \to (u,v,\text{out}) \end{array}$$

  as shown on the next page. Each "in" vertex has an additional incoming edge, and each "out" vertex has an additional outgoing edge. A Hamiltonian cycle must pass through an edge gadget in one of three ways—either straight through on both sides, or with a detour from one side to

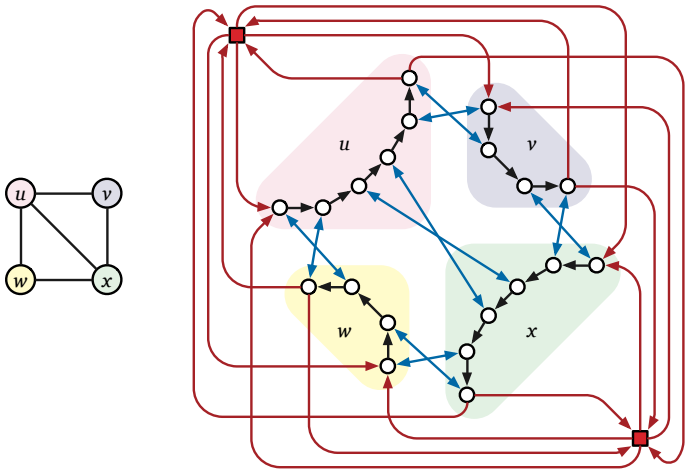
the other and back. Eventually, these options will correspond to both u and v, only u, or only v belonging to some vertex cover.



An edge gadget for uv and its only possible intersections with a Hamiltonian cycle.

- For each vertex u in G, all the edge gadgets for incident edges uv are connected in H into a single directed path, which we call a *vertex chain*. Specifically, suppose vertex u has d neighbors $v_1, v_2, \ldots, v_d$. Then H has $d-1$ additional edges $(u, v_i, \text{out}) \to (u, v_{i+1}, \text{in})$ for each i.
- Finally, H also contains k *cover vertices*, simply numbered 1 through k. Each cover vertex has a directed edge to the first vertex in each vertex chain, and a directed edge from the last vertex in each vertex chain.

An example of our complete transformation is shown below.

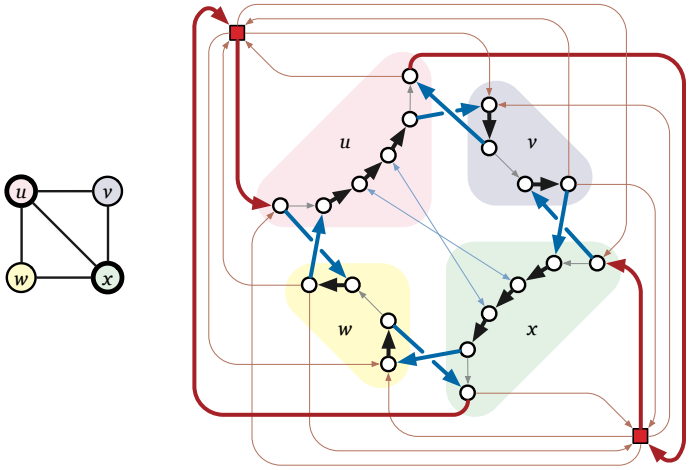


The original graph G and the transformed graph H, where $k = 2$.

Now suppose $C = \{u_1, u_2, \ldots, u_k\}$ is a vertex cover of G. Then H contains a Hamiltonian cycle, constructed as follows. Start at cover vertex 1, through traverse the vertex chain for vu_1, then visit cover vertex 2, then traverse the vertex chain for vu_2, and so forth, eventually returning to cover vertex 1. As we traverse the vertex chain for any vertex u_i, we have a choice for how to proceed when we reach any node (u_i, v, in).

- If $v \in C$, follow the edge $(u_i, v, \text{in}) \to (u_i, v, \text{out})$.
- If $v \notin C$, detour through the path $(u_i, v, \text{in}) \to (v, u_i, \text{in}) \to (v, u_i, \text{out}) \to (u_i, v, \text{out})$.

Thus, for each edge uv of G, the Hamiltonian cycle visits (u, v, in) and (u, v, out) as part of u's vertex chain if $u \in C$ and as part of v's vertex chain otherwise.

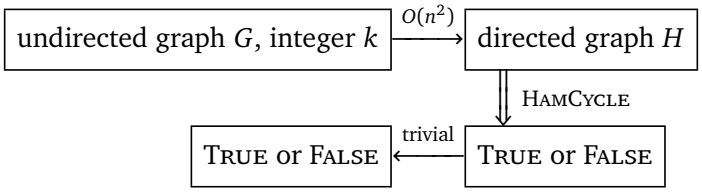


A vertex cover $\{u, x\}$ in G and the corresponding Hamiltonian cycle in H.

Now suppose H contains a Hamiltonian cycle C. This cycle must contain an edge from each cover vertex to the start of some vertex chain. Our case analysis of edge gadgets inductively implies that after C enters the vertex chain for some vertex u, it must traverse the entire vertex chain. Specifically, at each vertex (u, v, in), the cycle must contain either the single edge $(u, v, \text{in}) \to (u, v, \text{out})$ or the detour path $(u, v, \text{in}) \to (v, u, \text{in}) \to (v, u, \text{out}) \to (u, v, \text{out})$, followed by an edge to the next edge gadget in u's vertex chain, or to a cover vertex if this is the last such edge gadget. In particular, if C contains the detour edge $(u, v, \text{in}) \to (v, u, \text{in})$, it does not contain edges between any cover vertex and v's vertex chain. It follows that C traverses exactly k vertex chains. Moreover, these vertex chains describe a vertex cover of the original graph G, because C visits the vertex (u, v, in) for every edge uv in G.

We conclude that G contains a vertex cover of size k if and only if H contains a Hamiltonian cycle.

The transformation from G to H takes at most $O(n^2)$ time; we conclude that the Hamiltonian cycle problem is NP-hard. Moreover, since we can easily verify a Hamiltonian cycle in linear time, the Hamiltonian cycle problem is in NP, and therefore is NP-complete.



A closely related problem to Hamiltonian cycles is the famous *traveling salesman problem*—Given a *weighted* graph G, find the shortest cycle that visits every vertex. Finding the shortest cycle is obviously harder than determining if a cycle exists at all, so the traveling salesman problem is also NP-hard.

Finally, we can prove prove that finding Hamiltonian cycles in *undirected* graphs is NP-hard using a simple reduction from the same problem in *directed* graphs. I'll leave the details of this reduction as an entertaining exercise.

40.12 Subset Sum (from Vertex Cover)

The next problem that we prove NP-hard is the SubsetSum problem considered in the very first lecture on recursion: Given a set X of positive integers and an integer t, determine whether X has a subset whose elements sum to t.

To prove this problem is NP-hard, we once again reduce from VERTEXCOVER. Given a graph G and an integer k, we compute a set X of integer and an integer t, such that X has a subset that sums to t if and only if G has an vertex cover of size k. Our transformation uses just two 'gadgets', which are *integers* representing vertices and edges in G.

Number the *edges* of G arbitrarily from 0 to $m-1$. Our set X contains the integer $b_i := 4^i$ for each edge i, and the integer

$$a_v := 4^m + \sum_{i \in \Delta(v)} 4^i$$

for each vertex v, where $\Delta(v)$ is the set of edges that have v has an endpoint. Alternately, we can think of each integer in X as an $(m+1)$-digit number written in base 4. The mth digit is 1 if the integer represents a vertex, and 0 otherwise; and for each $i < m$, the ith digit is 1 if the integer represents edge i or one of its endpoints, and 0 otherwise. Finally, we set the target sum

$$t := k \cdot 4^m + \sum_{i=0}^{m-1} 2 \cdot 4^i.$$

Now let's prove that the reduction is correct. First, suppose there is a vertex cover of size k in the original graph G. Consider the subset $X_C \subseteq X$ that includes a_v for every vertex v in the vertex cover, and b_i for every edge i that has *exactly one* vertex in the cover. The sum of these integers, written in base 4, has a 2 in each of the first m digits; in the most significant digit, we are summing exactly k 1's. Thus, the sum of the elements of X_C is exactly t.

On the other hand, suppose there is a subset $X' \subseteq X$ that sums to t. Specifically, we must have

$$\sum_{v \in V'} a_v + \sum_{i \in E'} b_i = t$$

for some subsets $V' \subseteq V$ and $E' \subseteq E$. Again, if we sum these base-4 numbers, there are no carries in the first m digits, because for each i there are only three numbers in X whose ith digit is 1. Each edge number b_i contributes only one 1 to the ith digit of the sum, but the ith digit of t is 2. Thus, for each edge in G, at least one of its endpoints must be in V'. In other words, V is a vertex cover. On the other hand, only vertex numbers are larger than 4^m, and $\lfloor t/4^m \rfloor = k$, so V' has at most k elements. (In fact, it's not hard to see that V' has *exactly* k elements.)

For example, given the four-vertex graph used on the previous page to illustrate the reduction to Hamiltonian cycle, our set X might contain the following base-4 integers:

$$\begin{array}{ll} a_u := 111000_4 = 1344 & b_{uv} := 010000_4 = 256 \\ a_v := 110110_4 = 1300 & b_{uw} := 001000_4 = 64 \\ a_w := 101101_4 = 1105 & b_{vw} := 000100_4 = 16 \\ a_x := 100011_4 = 1029 & b_{vx} := 000010_4 = 4 \\ & b_{wx} := 000001_4 = 1 \end{array}$$

If we are looking for a vertex cover of size 2, our target sum would be $t := 222222_4 = 2730$. Indeed, the vertex cover $\{v, w\}$ corresponds to the subset $\{a_v, a_w, b_{uv}, b_{uw}, b_{vx}, b_{wx}\}$, whose sum is $1300 + 1105 + 256 + 64 + 4 + 1 = 2730$.

The reduction can clearly be performed in polynomial time. Since VERTEXCOVER is NP-hard, it follows that SUBSETSUM is NP-hard.

There is one subtle point that needs to be emphasized here. Way back at the beginning of the semester, we developed a dynamic programming algorithm to solve SUBSETSUM in time $O(nt)$. Isn't this a polynomial-time algorithm? idn't we just prove that P=NP? Hey, where's our million dollars? Alas,

life is not so simple. True, the running time is polynomial in n and t, but in order to qualify as a true polynomial-time algorithm, the running time must be a polynomial function *of the size of the input*. The *values* of the elements of X and the target sum t could be exponentially larger than the number of input bits. Indeed, the reduction we just described produces a value of t that is exponentially larger than the size of our original input graph, which would force our dynamic programming algorithm to run in exponential time.

Algorithms like this are said to run in ***pseudo-polynomial time***, and any NP-hard problem with such an algorithm is called ***weakly NP-hard***. Equivalently, a weakly NP-hard problem is one that can be solved in polynomial time when all input numbers are represented in *unary* (as a sum of 1s), but becomes NP-hard when all input numbers are represented in *binary*. If a problem is NP-hard even when all the input numbers are represented in unary, we say that the problem is ***strongly NP-hard***.

40.13 A Frivolous Example

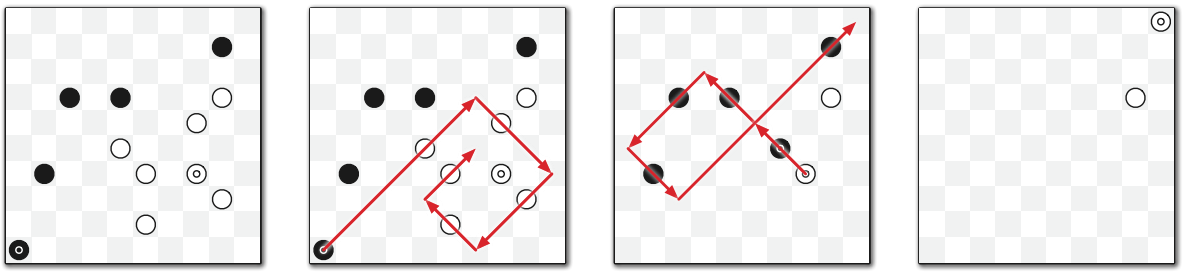
Draughts is a family of board games that have been played for thousands of years. Most Americans are familiar with the version called *checkers* or *English draughts*, but the most common variant worldwide, known as ***international draughts*** or ***Polish draughts***, originated in the Netherlands in the 16th century. For a complete set of rules, the reader should consult Wikipedia; here a few important differences from the Anglo-American game:

- **Flying kings:** As in checkers, a piece that ends a move in the row closest to the opponent becomes a *king* and gains the ability to move backward. Unlike in checkers, however, a king in international draughts can move any distance along a diagonal line in a single turn, as long as the intermediate squares are empty or contain exactly one opposing piece (which is captured).

- **Forced maximum capture:** In each turn, the moving player must capture as many opposing pieces as possible. This is distinct from the forced-capture rule in checkers, which requires only that each player must capture if possible, and that a capturing move ends only when the moving piece cannot capture further. In other words, checkers requires capturing a *maximal* set of opposing pieces on each turn; whereas, international draughts requires a *maximum* capture.

- **Capture subtleties:** As in checkers, captured pieces are removed from the board only at the end of the turn. Any piece can be captured at most once. Thus, when an opposing piece is jumped, that piece remains on the board *but cannot be jumped again* until the end of the turn.

For example, in the first position shown below, each circle represents a piece, and doubled circles represent kings. Black must make the indicated move, capturing five white pieces, because it is not possible to capture more than five pieces, and there is no other move that captures five. Black cannot extend his capture further northeast, because the captured White pieces are still on the board.



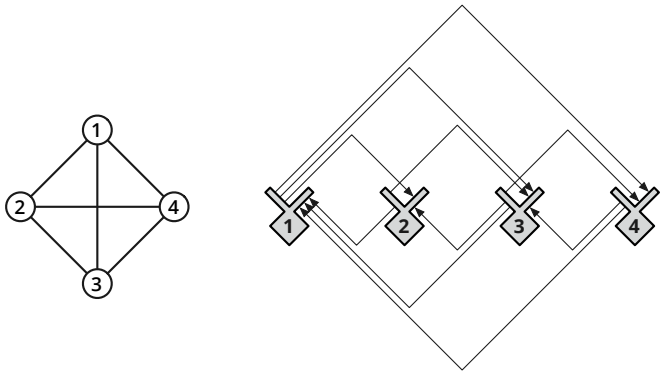
Two forced(!) moves in international draughts.

The actual game, which is played on a 10×10 board with 20 pieces of each color, is computationally trivial; we can precompute the optimal move for both players in every possible board configuration and hard-code the results into a lookup table of constant size. Sure, it's a *big* constant, but it's still just a constant!

But consider the natural generalization of international draughts to an $n \times n$ board. In this setting, ***finding a legal move is actually NP-hard!*** The following reduction from the Hamiltonian cycle problem in directed graphs was discovered by Bob Hearn in 2010.[5] In most two-player games, finding the *best* move is NP-hard (or worse); this is the only example I know of a game where just *following the rules* is an intractable problem!

Given a graph G with n vertices, we construct a board configuration for international draughts, such that White can capture a certain number of black pieces in a single move if and only if G has a Hamiltonian cycle. We treat G as a directed graph, with two arcs $u \to v$ and $v \to u$ in place of each undirected edge uv. Number the vertices arbitrarily from 1 to n. The final draughts configuration has several gadgets.

- The vertices of G are represented by rabbit-shaped *vertex gadgets*, which are evenly spaced along a horizontal line. Each arc $i \to j$ is represented by a path of two diagonal line segments from the "right ear" of vertex gadget i to the "left ear" of vertex gadget j. The path for arc $i \to j$ is located above the vertex gadgets if $i < j$, and below the vertex gadgets if $i > j$.



A high level view of the reduction from Hamiltonian cycle to international draughts.

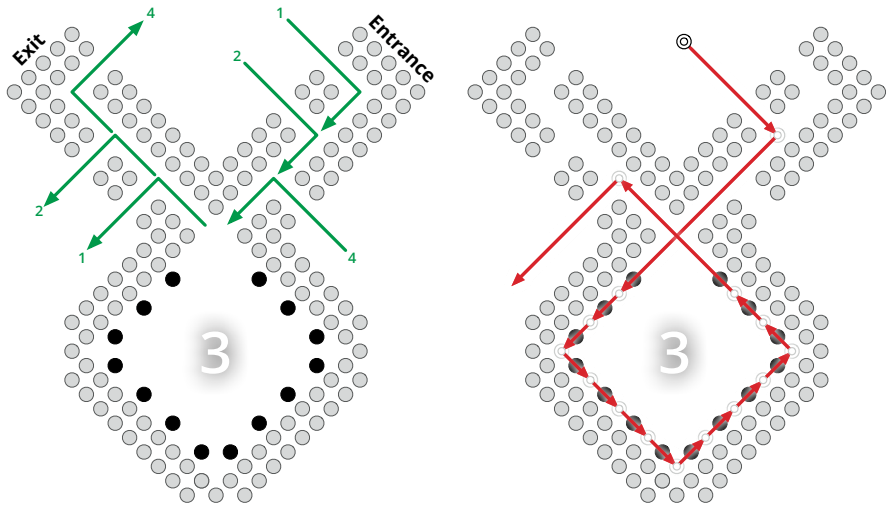
- The bulk of each vertex gadget is a diamond-shaped region called a *vault*. The walls of the vault are composed of two solid layers of black pieces, which cannot be captured; these pieces are drawn as gray circles in the figures. There are N capturable black pieces inside each vault, for some large integer N to be determined later. A white king can enter the vault through the "right ear", capture every internal piece, and then exit through the "left ear". Both ears are hallways, again with walls two pieces thick, with gaps where the arc paths end to allow the white king to enter and leave. The lengths of the "ears" can be adjusted easily to align with the other gadgets.

- For each arc $i \to j$, we have a *corner gadget*, which allows a white king leaving vertex gadget i to be redirected to vertex gadget j.

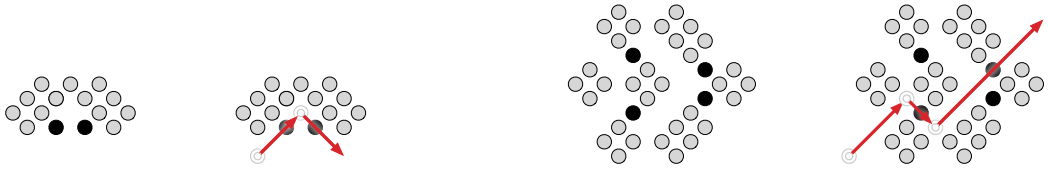
- Finally, wherever two arc paths cross, we have a *crossing gadget*; these gadgets allow the white king to traverse either arc path, but forbid switching from one arc path to the other.

A single white king starts at the bottom corner of one of the vaults. In any legal move, this king must alternate between traversing entire arc paths and clearing vaults. The king can traverse the various

[5] Posted on Theoretical Computer Science Stack Exchange: http://cstheory.stackexchange.com/a/1999/111.



Left: A vertex gadget. Right: A white king emptying the vault.
Gray circles are black pieces that cannot be captured.



A corner gadget and a crossing gadget.

gadgets backward, entering each vault through the exit and vice versa. But the reversal of a Hamiltonian cycle in G is another Hamiltonian cycle in G, so walking backward is fine.

If there is a Hamiltonian cycle in G, the white king can capture at least nN black pieces by visiting each of the other vaults and returning to the starting vault. On the other hand, if there is no Hamiltonian cycle in G, the white king can can capture at most half of the pieces in the starting vault, and thus can capture at most $(n-1/2)N + O(n^3)$ enemy pieces altogether. The $O(n^3)$ term accounts for the corner and crossing gadgets; each edge passes through one corner gadget and at most $n^2/2$ crossing gadgets.

To complete the reduction, we set $N = n^4$. Summing up, we obtain an $O(n^5) \times O(n^5)$ board configuration, with $O(n^5)$ black pieces and one white king. We can clearly construct this board configuration in polynomial time. A complete example of the construction appears on the next page.

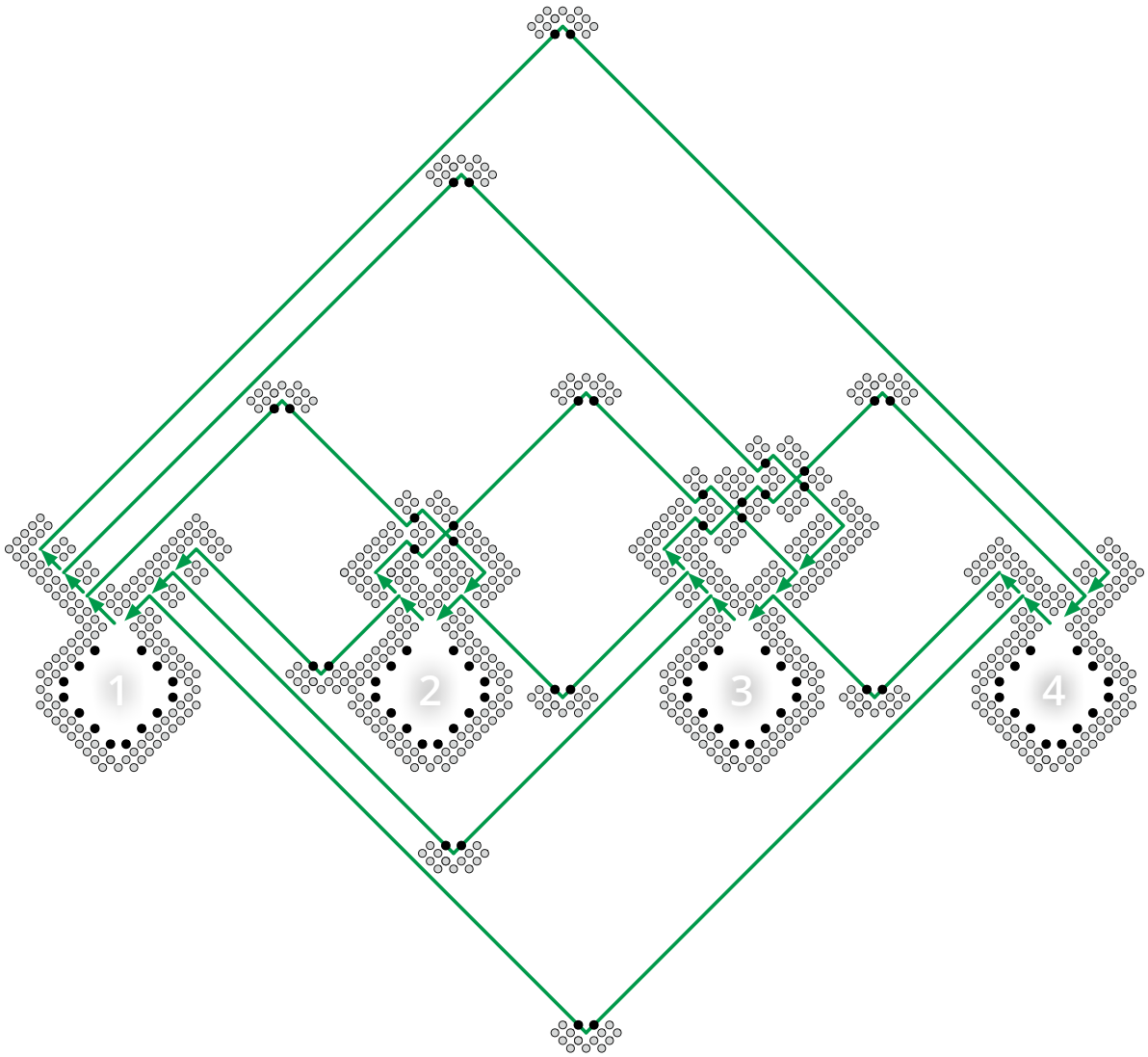
It is still open whether the following related question is NP-hard: Given an $n \times n$ board configuration for international draughts, can (and therefore *must*) White capture *all* the black pieces in a single turn?

40.14 Other Useful NP-hard Problems

Literally thousands of different problems have been proved to be NP-hard. I want to close this note by listing a few NP-hard problems that are useful in deriving reductions. I won't describe the NP-hardness proofs for these problems in detail, but you can find most of them in Garey and Johnson's classic *Scary Black Book of NP-Completeness*.[6]

- PLANARCIRCUITSAT: Given a boolean circuit that can be embedded in the plane so that no two wires cross, is there an input that makes the circuit output TRUE? This problem can be proved

[6]Michael Garey and David Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Co., 1979.



The final draughts configuration for the example graph. (The green arrows are not

NP-hard by reduction from the general circuit satisfiability problem, by replacing each crossing with a small series of gates.

- NotAllEqual3SAT: Given a 3CNF formula, is there an assignment of values to the variables so that every clause contains at least one True literal *and* at least one False literal? This problem can be proved NP-hard by reduction from the usual 3SAT.

- Planar3SAT: Given a 3CNF boolean formula, consider a bipartite graph whose vertices are the clauses and variables, where an edge indicates that a variable (or its negation) appears in a clause. If this graph is planar, the 3CNF formula is also called planar. The Planar3SAT problem asks, given a planar 3CNF formula, whether it has a satisfying assignment. This problem can be proved NP-hard by reduction from PlanarCircuitSAT.[7]

[7] Surprisingly, PlanarNotAllEqual3SAT is solvable in polynomial time!

- EXACT3DIMENSIONALMATCHING or X3M: Given a set S and a collection of three-element subsets of S, called *triples*, is there a sub-collection of disjoint triples that exactly cover S? This problem can be proved NP-hard by a reduction from 3SAT.

- PARTITION: Given a set S of n integers, are there subsets A and B such that $A \cup B = S$, $A \cap B = \emptyset$, and
$$\sum_{a \in A} a = \sum_{b \in B} b?$$
This problem can be proved NP-hard by a simple reduction from SUBSETSUM. Like SUBSETSUM, the PARTITION problem is only weakly NP-hard.

- 3PARTITION: Given a set S of $3n$ integers, can it be partitioned into n disjoint three-element subsets, such that every subset has exactly the same sum? Despite the similar names, this problem is *very* different from PARTITION; sorry, I didn't make up the names. This problem can be proved NP-hard by reduction from X3M. Unlike PARTITION, the 3PARTITION problem is *strongly* NP-hard, that is, it remains NP-hard even if the input numbers are less than some polynomial in n.

- SETCOVER: Given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, find the smallest sub-collection of S_i's that contains all the elements of $\bigcup_i S_i$. This problem is a generalization of both VERTEXCOVER and X3M.

- HITTINGSET: Given a collection of sets $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, find the minimum number of elements of $\bigcup_i S_i$ that hit every set in $\mathcal{S}$. This problem is also a generalization of VERTEXCOVER.

- HAMILTONIANPATH: Given an graph G, is there a path in G that visits every vertex exactly once? This problem can be proved NP-hard either by modifying the reductions from 3SAT or VERTEXCOVER to HAMILTONIANCYCLE, or by a direct reduction from HAMILTONIANCYCLE.

- LONGESTPATH: Given a non-negatively weighted graph G and two vertices u and v, what is the longest simple path from u to v in the graph? A path is *simple* if it visits each vertex at most once. This problem is a generalization of the HAMILTONIANPATH problem. Of course, the corresponding *shortest* path problem is in P.

- STEINERTREE: Given a weighted, undirected graph G with some of the vertices marked, what is the minimum-weight subtree of G that contains every marked vertex? If *every* vertex is marked, the minimum Steiner tree is just the minimum spanning tree; if exactly two vertices are marked, the minimum Steiner tree is just the shortest path between them. This problem can be proved NP-hard by reduction from VERTEXCOVER.

In addition to these dry but useful problems, most interesting puzzles and solitaire games have been shown to be NP-hard, or to have NP-hard generalizations. (Arguably, if a game or puzzle isn't at least NP-hard, it isn't interesting!) Some familiar examples include:

- Minesweeper (by reduction from CIRCUITSAT)[8]
- Tetris (by reduction from 3PARTITION)[9]

[8] Richard Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer* 22(2):9–15, 2000. http://www.mat.bham.ac.uk/R.W.Kaye/minesw/minesw.pdf

[9] Ron Breukelaar*, Erik D. Demaine, Susan Hohenberger*, Hendrik J. Hoogeboom, Walter A. Kosters, and David Liben-Nowell*. Tetris is hard, even to approximate. *International Journal of Computational Geometry and Applications* 14:41–68, 2004.

- Sudoku (by a complex reduction from 3SAT)[10]
- Klondike, aka "Solitaire" (by reduction from 3SAT)[11]
- Flood-It (by reduction from shortest common supersequence)[12]
- Pac-Man (by reduction from HAMILTONIANCYCLE)[13]
- Super Mario Brothers (by reduction from 3SAT)[14]
- Candy Crush Saga (by reduction from a variant of 3SAT)[15]

As of November 2014, nobody has published a proof that a generalization of Threes/2048 or Cookie Clicker is NP-hard, but I'm sure it's only a matter of time.[16]

*40.15 On Beyond Zebra

P and NP are only the first two steps in an enormous hierarchy of complexity classes. To close these notes, let me describe a few more classes of interest.

Polynomial Space. ***PSPACE*** is the set of decision problems that can be solved using polynomial *space*. Every problem in NP (and therefore in P) is also in PSPACE. It is generally believed that NP $\neq$ PSPACE, but nobody can even prove that P $\neq$ PSPACE. A problem Π is ***PSPACE-hard*** if, for any problem Π' that can be solved using polynomial *space*, there is a polynomial-*time* many-one reduction from Π' to Π. A problem is ***PSPACE-complete*** if it is both PSPACE-hard and in PSPACE. If any PSPACE-hard problem is in NP, then PSPACE=NP; similarly, if any PSPACE-hard problem is in P, then PSPACE=P.

The canonical PSPACE-complete problem is the *quantified boolean formula* problem, or ***QBF***: Given a boolean formula Φ that may include any number of universal or existential quantifiers, but no free variables, is Φ equivalent to TRUE? For example, the following expression is a valid input to QBF:

$$\exists a : \forall b : \exists c : (\forall d : a \vee b \vee c \vee \bar{d}) \Leftrightarrow ((b \wedge \bar{c}) \vee (\exists e : \overline{(\bar{a} \Rightarrow e)}) \vee (c \neq a \wedge e))).$$

SAT is provably equivalent the special case of QBF where the input formula contains only existential quantifiers. QBF remains PSPACE-hard even when the input formula must have all its quantifiers at the beginning, the quantifiers strictly alternate between $\exists$ and $\forall$, and the quantified proposition is in conjunctive normal form, with exactly three literals in each clause, for example:

$$\exists a : \forall b : \exists c : \forall d : \big((a \vee b \vee c) \wedge (b \vee \bar{c} \vee \bar{d}) \wedge (\bar{a} \vee c \vee d) \wedge (a \vee \bar{b} \vee \bar{d})\big)$$

[10] Takayuki Yato and Takahiro Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* E86-A(5):1052–1060, 2003. http://www-imai.is.s.u-tokyo.ac.jp/~yato/data2/MasterThesis.pdf.

[11] Luc Longpré and Pierre McKenzie. The complexity of Solitaire. *Proceedings of the 32nd International Mathematical Foundations of Computer Science*, 182–193, 2007.

[12] Raphaël Clifford, Markus Jalsenius, Ashley Montanaro, and Benjamin Sach. The complexity of flood filling games. *Proceedings of the Fifth International Conference on Fun with Algorithms* (FUN'10), 307–318, 2010. http://arxiv.org/abs/1001.4420.

[13] Giovanni Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014. http://giovannivigliettta.com/papers/gaming2.pdf

[14] Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games Are (computationally) hard. *Proceedings of the Seventh International Conference on Fun with Algorithms* (FUN'14), 2014. http://arxiv.org/abs/1203.1895.

[15] Luciano Gualà, Stefano Leucci, Emanuele Natale. Bejeweled, Candy Crush and other match-three games are (NP-)hard. Preprint, March 2014. http://arxiv.org/abs/1403.5830.

[16] Princeton freshman Rahul Mehta actually claimed a proof that a certain generalization of 2048 is PSPACE-hard, but his proof appears to be flawed. [Rahul Mehta. 2048 is (PSPACE) hard, but sometimes easy. *Electronic Colloquium on Computational Complexity*, Report No. 116, 2014. http://eccc.hpi-web.de/report/2014/116/.] On the other hand, Christopher Chen proved that a different(!) generalization of 2048 is in NP, but left the hardness question open. [Christopher Chen. 2048 is in NP. *Open Endings*, March 27, 2014. http://blog.openendings.net/2014/03/2048-is-in-np.html.]

This restricted version of QBF can also be phrased as a two-player strategy question. Suppose two players, Alice and Bob, are given a 3CNF predicate with free variables $x_1, x_2, \ldots, x_n$. The players alternately assign values to the variables in order by index—Alice assigns a value to x_1, Bob assigns a value to x_2, and so on. Alice eventually assigns values to every variable with an odd index, and Bob eventually assigns values to every variable with an even index. Alice wants to make the expression TRUE, and Bob wants to make it FALSE. Assuming Alice and Bob play perfectly, who wins this game? Not surprisingly, most two-player games[17] like tic-tac-toe, reversi, checkers, go, chess, and mancala—or more accurately, appropriate generalizations of these constant-size games to arbitrary board sizes—are PSPACE-hard.

Another canonical PSPACE-hard problem is *NFA totality*: Given a non-deterministic finite-state automaton M over some alphabet Σ, does M accept every string in Σ^*? The closely related problems *NFA equivalence* (Do two given NFAs accept the same language?) and *NFA minimization* (Find the smallest NFA that accepts the same language as a given NFA) are also PSPACE-hard, as are the corresponding questions about regular expressions. (The corresponding questions about *deterministic* finite-state automata are all solvable in polynomial time.)

Exponential time. The next significantly larger complexity class, ***EXP*** (also called EXPTIME), is the set of decision problems that can be solved in exponential time, that is, using at most 2^{n^c} steps for some constant $c > 0$. Every problem in PSPACE (and therefore in NP (and therefore in P)) is also in EXP. It is generally believed that PSPACE $\subsetneq$ EXP, but nobody can even prove that NP $\neq$ EXP. A problem Π is ***EXP-hard*** if, for any problem Π' that can be solved in *exponential* time, there is a *polynomial*-time many-one reduction from Π' to Π. A problem is ***EXP-complete*** if it is both EXP-hard and in EXP. If any EXP-hard problem is in PSPACE, then EXP=PSPACE; similarly, if any EXP-hard probelm is in NP, then EXP=NP. We *do* know that P $\neq$ EXP; in particular, no EXP-hard problem is in P.

Natural generalizations of many interesting 2-player games—like checkers, chess, mancala, and go—are actually EXP-hard. The boundary between PSPACE-complete games and EXP-hard games is rather subtle. For example, there are three ways to draw in chess (the standard 8×8 game): stalemate (the player to move is not in check but has no legal moves), repeating the same board position three times, or moving fifty times without capturing a piece. The $n \times n$ generalization of chess is either in PSPACE or EXP-hard depending on how we generalize these rules. If we declare a draw after (say) n^3 capture-free moves, then every game must end after a polynomial number of moves, so we can simulate all possible games from any given position using only polynomial space. On the other hand, if we ignore the capture-free move rule entirely, the resulting game can last an exponential number of moves, so there no obvious way to detect a repeating position using only polynomial space; indeed, this version of $n \times n$ chess is EXP-hard.

Excelsior! Naturally, even exponential time is not the end of the story. ***NEXP*** is the class of decision problems that can be solve in *nondeterministic* exponential time; equivalently, a decision problem is in NEXP if and only if, for every YES instance, there is a *proof* of this fact that can be checked in exponential time. ***EXPSPACE*** is the set of decision problems that can be solved using exponential *space*. Even these larger complexity classes have hard and complete problems; for example, if we add the intersection operator $\cap$ to the syntax of regular expressions, deciding whether two such expressions describe the same language is EXPSPACE-hard. Beyond EXPSPACE are complexity classes with *doubly*-exponential

[17] For a good (but now slightly dated) overview of known results on the computational complexity of games and puzzles, see Erik D. Demaine and Robert Hearn's survey "Playing Games with Algorithms: Algorithmic Combinatorial Game Theory" at http://arxiv.org/abs/cs.CC/0106019.

resource bounds (EEXP, NEEXP, and EEXPSPACE), then *triply* exponential resource bounds (EEEXP, NEEEXP, and EEEXPSPACE), and so on ad infinitum.

All these complexity classes can be ordered by inclusion as follows:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXP} \subseteq \mathrm{NEXP} \subseteq \mathrm{EXPSPACE} \subseteq \mathrm{EEXP} \subseteq \mathrm{NEEXP} \subseteq \mathrm{EEXPSPACE} \subseteq \mathrm{EEEXP} \subseteq \cdots,$$

Most complexity theorists strongly believe that every inclusion in this sequence is strict; that is, no two of these complexity classes are equal. However, the strongest result that has been proved is that every class in this sequence is strictly contained in the class *three* steps later in the sequence. For example, we have proofs that $\mathrm{P} \neq \mathrm{EXP}$ and $\mathrm{PSPACE} \neq \mathrm{EXPSPACE}$, but not whether $\mathrm{P} \neq \mathrm{PSPACE}$ or $\mathrm{NP} \neq \mathrm{EXP}$.

The limit of this series of increasingly exponential complexity classes is the class ***ELEMENTARY*** of decision problems that can be solved using time or space bounded by a function the form $2 \uparrow^k n$ for some integer k, where

$$2 \uparrow^k n := \begin{cases} n & \text{if } k = 0, \\ 2^{2 \uparrow^{k-1} n} & \text{otherwise.} \end{cases}$$

For example, $2 \uparrow^1 n = 2^n$ and $2 \uparrow^2 n = 2^{2^n}$. You might be tempted to conjecture that every natural decidable problem can be solved in elementary time, but then you would be wrong. Consider the ***extended regular expressions*** defined by recursively combining (possibly empty) strings over some finite alphabet by concatenation (xy), union ($x + y$), Kleene closure (x^*), ***and negation ($\overline{x}$)***. For example, the extended regular expression $\overline{(0+1)^*00(0+1)^*}$ represents the set of strings in $\{0,1\}^*$ that do *not* contain two 0s in a row. It is possible to determine algorithmically whether two extended regular expressions describe identical languages, by recursively converting each expression into an equivalent NFA, converting each NFA into a DFA, and then minimizing the DFA. Unfortunately, however, this problem cannot be solved in only elementary time, intuitively because each layer of recursive negation exponentially increases the number of states in the final DFA.

Exercises

1. (a) Describe and analyze and algorithm to solve PARTITION in time $O(nM)$, where n is the size of the input set and M is the sum of the absolute values of its elements.
   (b) Why doesn't this algorithm imply that P=NP?

2. Consider the following problem, called BOXDEPTH: Given a set of n axis-aligned rectangles in the plane, how big is the largest subset of these rectangles that contain a common point?
   (a) Describe a polynomial-time reduction from BOXDEPTH to MAXCLIQUE.
   (b) Describe and analyze a polynomial-time algorithm for BOXDEPTH. *[Hint: $O(n^3)$ time should be easy, but $O(n \log n)$ time is possible.]*
   (c) Why don't these two results imply that P=NP?

3. A boolean formula is in *disjunctive normal form* (or *DNF*) if it consists of a *disjunction* (OR) or several *terms*, each of which is the conjunction (AND) of one or more literals. For example, the formula
   $$(\overline{x} \wedge y \wedge \overline{z}) \vee (y \wedge z) \vee (x \wedge \overline{y} \wedge \overline{z})$$
   is in disjunctive normal form. DNF-SAT asks, given a boolean formula in disjunctive normal form, whether that formula is satisfiable.

(a) Describe a polynomial-time algorithm to solve DNF-SAT.

(b) What is the error in the following argument that P=NP?

> *Suppose we are given a boolean formula in conjunctive normal form with at most three literals per clause, and we want to know if it is satisfiable. We can use the distributive law to construct an equivalent formula in disjunctive normal form. For example,*
>
> $$(x \vee y \vee \overline{z}) \wedge (\overline{x} \vee \overline{y}) \iff (x \wedge \overline{y}) \vee (y \wedge \overline{x}) \vee (\overline{z} \wedge \overline{x}) \vee (\overline{z} \wedge \overline{y})$$
>
> *Now we can use the algorithm from part (a) to determine, in polynomial time, whether the resulting DNF formula is satisfiable. We have just solved 3SAT in polynomial time. Since 3SAT is NP-hard, we must conclude that P=NP!*

4. (a) Describe a polynomial-time reduction from PARTITION to SUBSETSUM.

   (b) Describe a polynomial-time reduction from SUBSETSUM to PARTITION.

5. (a) Describe a polynomial-time reduction from UNDIRECTEDHAMILTONIANCYCLE to DIRECTEDHAMILTONIANCYCLE.

   (b) Describe a polynomial-time reduction from DIRECTEDHAMILTONIANCYCLE to UNDIRECTEDHAMILTONIANCYCLE.

6. (a) Describe a polynomial-time reduction from HAMILTONIANPATH to HAMILTONIANCYCLE.

   (b) Describe a polynomial-time reduction from HAMILTONIANCYCLE to HAMILTONIANPATH. *[Hint: A polynomial-time reduction may call the black-box subroutine more than once.]*

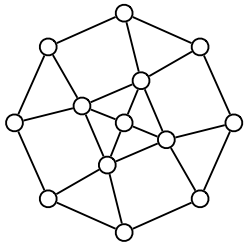
7. (a) Prove that PLANARCIRCUITSAT is NP-hard. *[Hint: Construct a gadget for crossing wires.]*

   (b) Prove that NOTALLEQUAL3SAT is NP-hard.

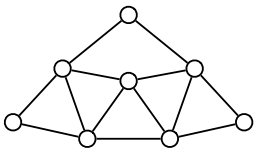
   (c) Prove that the following variant of 3SAT is NP-hard: Given a boolean formula Φ in conjunctive normal form where each clause contains at most 3 literals and each variable appears in at most 3 clauses, does Φ have a satisfying assignment?

8. (a) Using the gadget on the right below, prove that deciding whether a given *planar* graph is 3-colorable is NP-hard. *[Hint: Show that the gadget can be 3-colored, and then replace any crossings in a planar embedding with the gadget appropriately.]*

   (b) Using part (a) and the middle gadget below, prove that deciding whether a planar graph *with maximum degree 4* is 3-colorable is NP-hard. *[Hint: Replace any vertex with degree greater than 4 with a collection of gadgets connected so that no degree is greater than four.]*



(a) Gadget for planar 3-colorability.



(b) Gadget for degree-4 planar 3-colorability.

9. Prove that the following problems are NP-hard.

   (a) Given two undirected graphs G and H, is G isomorphic to a subgraph of H?

   (b) Given an undirected graph G, does G have a spanning tree in which every node has degree at most 17?

   (c) Given an undirected graph G, does G have a spanning tree with at most 42 leaves?

10. **There's something special about the number 3.**

    (a) Describe and analyze a polynomial-time algorithm for 2PARTITION. Given a set S of $2n$ positive integers, your algorithm will determine in polynomial time whether the elements of S can be split into n disjoint pairs whose sums are all equal.

    (b) Describe and analyze a polynomial-time algorithm for 2COLOR. Given an undirected graph G, your algorithm will determine in polynomial time whether G has a proper coloring that uses only two colors.

    (c) Describe and analyze a polynomial-time algorithm for 2SAT. Given a boolean formula Φ in conjunctive normal form, with exactly *two* literals per clause, your algorithm will determine in polynomial time whether Φ has a satisfying assignment.

11. **There's nothing special about the number 3.**

    (a) The problem 12PARTITION is defined as follows: Given a set S of $12n$ positive integers, determine whether the elements of S can be split into n subsets of 12 elements each whose sums are all equal. Prove that 12PARTITION is NP-hard. *[Hint: Reduce from 3PARTITION. It may be easier to consider multisets first.]*

    (b) The problem 12COLOR is defined as follows: Given an undirected graph G, determine whether we can color each vertex with one of twelve colors, so that every edge touches two different colors. Prove that 12COLOR is NP-hard. *[Hint: Reduce from 3COLOR.]*

    (c) The problem 12SAT is defined as follows: Given a boolean formula Φ in conjunctive normal form, with exactly twelve literals per clause, determine whether Φ has a satisfying assignment. Prove that 12SAT is NP-hard. *[Hint: Reduce from 3SAT.]*

*12. Describe a direct polynomial-time reduction from 4COLOR to 3COLOR. (This is a lot harder than the opposite direction.)

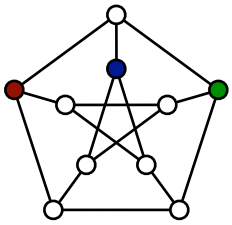
13. This exercise asks you to prove that a certain reduction from VERTEXCOVER to STEINERTREE is correct. Suppose we want to find the smallest vertex cover in a given undirected graph $G = (V, E)$. We construct a new graph $H = (V', E')$ as follows:

    - $V' = V \cup E \cup \{z\}$
    - $E' = \{ve \mid v \in V \text{ is an endpoint of } e \in W\} \cup \{vz \mid v \in V\}$.

    Equivalently, we construct H by subdividing each edge in G with a new vertex, and then connecting all the original vertices of G to a new *apex* vertex z.

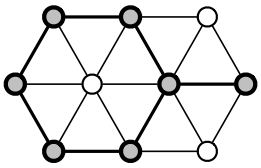
    Prove that G has a vertex cover of size k if and only if there is a subtree of H with $k + |E| + 1$ vertices that contains every vertex in $E \cup \{z\}$.

14. Let $G = (V, E)$ be a graph. A ***dominating set*** in G is a subset S of the vertices such that every vertex in G is either in S or adjacent to a vertex in S. The DominatingSet problem asks, given a graph G and an integer k as input, whether G contains a dominating set of size k. Prove that this problem is NP-hard.



A dominating set of size 3 in the Peterson graph.

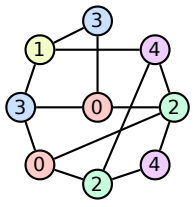
15. A subset S of vertices in an undirected graph G is called ***triangle-free*** if, for every triple of vertices $u, v, w \in S$, at least one of the three edges uv, uw, vw is *absent* from G. Prove that finding the size of the largest triangle-free subset of vertices in a given undirected graph is NP-hard.



A triangle-free subset of 7 vertices.
This is ***not*** the largest triangle-free subset in this graph.

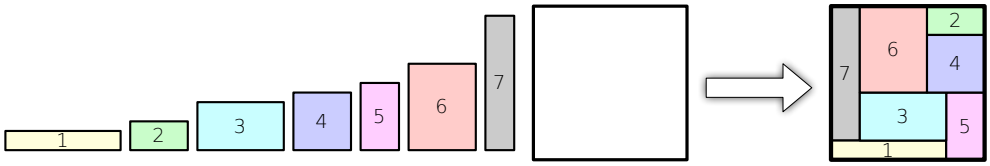
16. *Pebbling* is a solitaire game played on an undirected graph G, where each vertex has zero or more *pebbles*. A single *pebbling move* consists of removing two pebbles from a vertex v and adding one pebble to an arbitrary neighbor of v. (Obviously, the vertex v must have at least two pebbles before the move.) The PebbleDestruction problem asks, given a graph $G = (V, E)$ and a pebble count $p(v)$ for each vertex v, whether is there a sequence of pebbling moves that removes all but one pebble. Prove that PebbleDestruction is NP-hard.

17. Recall that a 5-coloring of a graph G is a function that assigns each vertex of G an 'color' from the set $\{0, 1, 2, 3, 4\}$, such that for any edge uv, vertices u and v are assigned different 'colors'. A 5-coloring is *careful* if the colors assigned to adjacent vertices are not only distinct, but differ by more than 1 (mod 5). Prove that deciding whether a given graph has a careful 5-coloring is NP-hard. *[Hint: Reduce from the standard 5Color problem.]*



A careful 5-coloring.

18. The RectangleTiling problem is defined as follows: Given one large rectangle and several smaller rectangles, determine whether the smaller rectangles can be placed inside the large rectangle with no gaps or overlaps. Prove that RectangleTiling is NP-hard.



A positive instance of the RectangleTiling problem.

19. For each problem below, either describe a polynomial-time algorithm or prove that the problem is NP-hard.

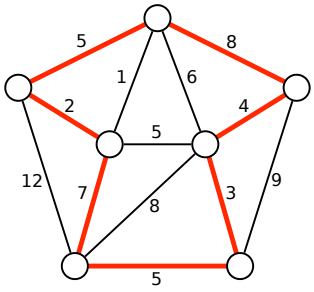
    (a) A *double-Eulerian circuit* in an undirected graph G is a closed walk that traverses every edge in G exactly twice. Given a graph G, does G have a double-Eulerian circuit?

    (b) A *double-Hamiltonian circuit* in an undirected graph G is a closed walk that visits every vertex in G exactly twice. Given a graph G, does G have a double-Hamiltonian circuit?

20. (a) A *tonian path* in a graph G is a path that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian path is NP-hard.

    (b) A *tonian cycle* in a graph G is a cycle that goes through at least half of the vertices of G. Show that determining whether a graph has a tonian cycle is NP-hard. *[Hint: Use part (a).]*

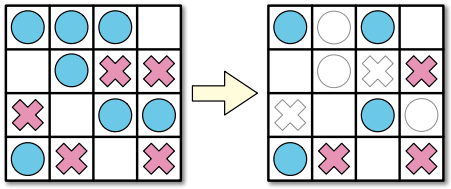
21. Let G be an undirected graph with weighted edges. A *heavy Hamiltonian cycle* is a cycle C that passes through each vertex of G exactly once, such that the total weight of the edges in C is at least half of the total weight of all edges in G. Prove that deciding whether a graph has a heavy Hamiltonian cycle is NP-hard.



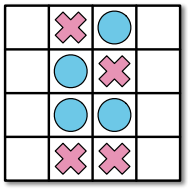
A heavy Hamiltonian cycle. The cycle has total weight 34; the graph has total weight 67.

22. A boolean formula in *exclusive-or conjunctive normal form* (XCNF) is a conjunction (And) of several *clauses*, each of which is the *exclusive*-or of several literals; that is, a clause is true if and only if it contains an odd number of true literals. The XCNF-SAT problem asks whether a given XCNF formula is satisfiable. Either describe a polynomial-time algorithm for XCNF-SAT or prove that it is NP-hard.

23. Consider the following solitaire game. The puzzle consists of an $n \times m$ grid of squares, where each square may be empty, occupied by a red stone, or occupied by a blue stone. The goal of the puzzle is to remove some of the given stones so that the remaining stones satisfy two conditions: (1) every row contains at least one stone, and (2) no column contains stones of both colors. For some initial configurations of stones, reaching this goal is impossible.



A solvable puzzle and one of its many solutions.



An unsolvable puzzle.

Prove that it is NP-hard to determine, given an initial configuration of red and blue stones, whether the puzzle can be solved.

24. You're in charge of choreographing a musical for your local community theater, and it's time to figure out the final pose of the big show-stopping number at the end. ("Streetcar!"") You've decided that each of the n cast members in the show will be positioned in a big line when the song finishes, all with their arms extended and showing off their best spirit fingers.

    The director has declared that during the final flourish, each cast member must either point both their arms up or point both their arms down; it's your job to figure out who points up and who points down. Moreover, in a fit of unchecked power, the director has also given you a list of arrangements that will upset his delicate artistic temperament. Each forbidden arrangement is a subset of the cast members paired with arm positions; for example: "Marge may not point her arms up while Ned, Apu, and Smithers point their arms down."

    Prove that finding an acceptable arrangement of arm positions is NP-hard.

25. The next time you are at a party, one of the guests will suggest everyone play a round of Three-Way Mumbledypeg, a game of skill and dexterity that requires three teams and a knife. The official Rules of Three-Way Mumbledypeg (fixed during the Holy Roman Three-Way Mumbledypeg Council in 1625) require that (1) each team *must* have at least one person, (2) any two people on the same team *must* know each other, and (3) everyone watching the game *must* be on one of the three teams. Of course, it will be a really *fun* party; nobody will want to leave. There will be several pairs of people at the party who don't know each other. The host of the party, having heard thrilling tales of your prowess in all things algorithmic, will hand you a list of which pairs of party-goers know each other and ask you to choose the teams, while he sharpens the knife.

    Either describe and analyze a polynomial time algorithm to determine whether the party-goers can be split into three legal Three-Way Mumbledypeg teams, or prove that the problem is NP-hard.

26. Jeff tries to make his students happy. At the beginning of class, he passes out a questionnaire that lists a number of possible course policies in areas where he is flexible. Every student is asked to respond to each possible course policy with one of "strongly favor", "mostly neutral", or "strongly oppose". Each student may respond with "strongly favor" or "strongly oppose" to at most five questions. Because Jeff's students are very understanding, each student is happy if (but only if) he or she prevails in just one of his or her strong policy preferences. Either describe a polynomial-time

algorithm for setting course policy to maximize the number of happy students, or show that the problem is NP-hard.

27. The party you are attending is going great, but now it's time to line up for ***The Algorithm March (***アルゴリズムこうしん***)***! This dance was originally developed by the Japanese comedy duo Itsumo Kokokara (いつもここから) for the children's television show PythagoraSwitch (ピタゴラスイッチ). The Algorithm March is performed by a line of people; each person in line starts a specific sequence of movements one measure later than the person directly in front of them. Thus, the march is the dance equivalent of a musical round or canon, like "Row Row Row Your Boat".

    Proper etiquette dictates that each marcher must know the person directly in front of them in line, lest a minor mistake during lead to horrible embarrassment between strangers. Suppose you are given a complete list of which people at your party know each other. ***Prove*** that it is NP-hard to determine the largest number of party-goers that can participate in the Algorithm March. You may assume without loss of generality that there are no ninjas at your party.

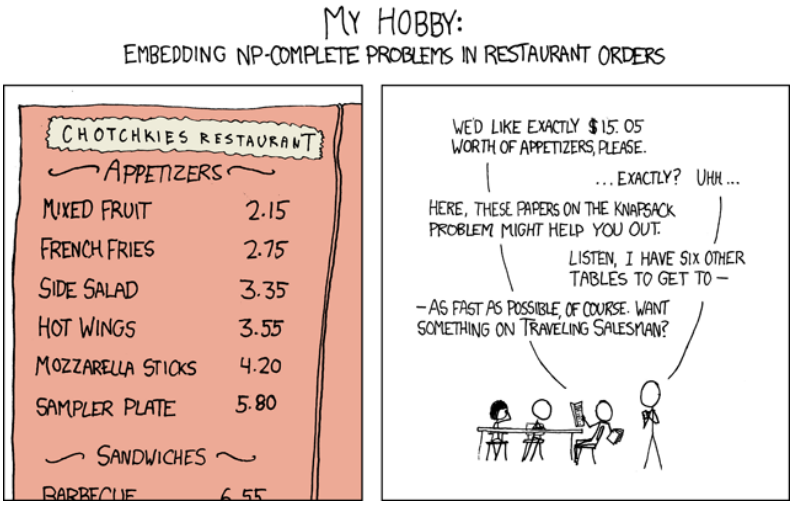
28. (a) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary weighted graph G, the length of the shortest Hamiltonian cycle in G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary weighted graph G, the shortest Hamiltonian cycle in G, using this magic black box as a subroutine.

    (b) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, the number of vertices in the largest complete subgraph of G. Describe and analyze a **polynomial-time** algorithm that computes, given an arbitrary graph G, a complete subgraph of G of maximum size, using this magic black box as a subroutine.

    (c) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary graph G, whether G is 3-colorable. Describe and analyze a **polynomial-time** algorithm that either computes a proper 3-coloring of a given graph or correctly reports that no such coloring exists, using the magic black box as a subroutine. *[Hint: The input to the magic black box is a graph. Just a graph. Vertices and edges. Nothing else.]*

    (d) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary boolean formula Φ, whether Φ is satisfiable. Describe and analyze a **polynomial-time** algorithm that either computes a satisfying assignment for a given boolean formula or correctly reports that no such assignment exists, using the magic black box as a subroutine.

    (e) Suppose you are given a magic black box that can determine **in polynomial time**, given an arbitrary set X of positive integers, whether X can be partitioned into two sets A and B such that $\sum A = \sum B$. Describe and analyze a **polynomial-time** algorithm that either computes an equal partition of a given set of positive integers or correctly reports that no such partition exists, using the magic black box as a subroutine.



[General solutions give you a 50% tip.]

— Randall Munroe, *xkcd* (http://xkcd.com/287/)

Change is certain. Peace is followed by disturbances; departure of evil men by their return. Such recurrences should not constitute occasions for sadness but realities for awareness, so that one may be happy in the interim.

— *I Ching [The Book of Changes]* (c. 1100 BC)

To endure *the idea of the recurrence one needs: freedom from morality; new means against the fact of pain (pain conceived as a tool, as the father of pleasure; there is no cumulative consciousness of displeasure); the enjoyment of all kinds of uncertainty, experimentalism, as a counterweight to this extreme fatalism; abolition of the concept of necessity; abolition of the "will"; abolition of "knowledge-in-itself."*

— Friedrich Nietzsche *The Will to Power* (1884)
[translated by Walter Kaufmann]

Wil Wheaton: *Embrace the dark side!*
Sheldon: *That's not even from your franchise!*

— "The Wheaton Recurrence", *Bing Bang Theory*, April 12, 2010

Solving Recurrences

1 Introduction

A ***recurrence*** is a recursive description of a function, or in other words, a description of a function in terms of itself. Like all recursive structures, a recurrence consists of one or more *base cases* and one or more *recursive cases*. Each of these cases is an equation or inequality, with some function value $f(n)$ on the left side. The base cases give explicit values for a (typically finite, typically small) subset of the possible values of n. The recursive cases relate the function value $f(n)$ to function value $f(k)$ for one or more integers $k < n$; typically, each recursive case applies to an infinite number of possible values of n.

For example, the following recurrence (written in two different but standard ways) describes the identity function $f(n) = n$:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n-1)+1 & \text{otherwise} \end{cases} \qquad \begin{array}{l} f(0) = 0 \\ f(n) = f(n-1)+1 \text{ for all } n > 0 \end{array}$$

In both presentations, the first line is the only base case, and the second line is the only recursive case. The same function can satisfy *many* different recurrences; for example, both of the following recurrences also describe the identity function:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) & \text{otherwise} \end{cases} \qquad f(n) = \begin{cases} 0 & \text{if } n = 0 \\ 2 \cdot f(n/2) & \text{if } n \text{ is even and } n > 0 \\ f(n-1)+1 & \text{if } n \text{ is odd} \end{cases}$$

We say that a particular function ***satisfies*** a recurrence, or is the ***solution*** to a recurrence, if each of the statements in the recurrence is true. Most recurrences—at least, those that we will encounter in this class—have a solution; moreover, if every case of the recurrence is an equation, that solution is unique. Specifically, if we transform the recursive formula into a recursive *algorithm*, the solution to the recurrence is the function computed by that algorithm!

Recurrences arise naturally in the analysis of algorithms, especially recursive algorithms. In many cases, we can express the running time of an algorithm as a recurrence, where the recursive cases of the

recurrence correspond exactly to the recursive cases of the algorithm. Recurrences are also useful tools for solving counting problems—How many objects of a particular kind exist?

By itself, a recurrence is not a satisfying description of the running time of an algorithm or a bound on the number of widgets. Instead, we need a ***closed-form*** solution to the recurrence; this is a *non-recursive* description of a function that satisfies the recurrence. For recurrence *equations*, we sometimes prefer an *exact* closed-form solution, but such a solution may not exist, or may be too complex to be useful. Thus, for most recurrences, especially those arising in algorithm analysis, we are satisfied with an *asymptotic* solution of the form $\Theta(g(n))$, for some explicit (non-recursive) function $g(n)$.

For recursive *inequalities*, we prefer a ***tight*** solution; this is a function that would still satisfy the recurrence if all the inequalities were replaced with the corresponding equations. Again, exactly tight solutions may not exist, or may be too complex to be useful, in which case we seek either a looser bound or an asymptotic solution of the form $O(g(n))$ or $\Omega(g(n))$.

2 The Ultimate Method: Guess and Confirm

Ultimately, there is only one fail-safe method to solve *any* recurrence:

Guess the answer, and then prove it correct by induction.

Later sections of these notes describe techniques to generate guesses that are guaranteed to be correct, provided you use them correctly. But if you're faced with a recurrence that doesn't seem to fit any of these methods, or if you've forgotten how those techniques work, don't despair! If you guess a closed-form solution and then try to verify your guess inductively, usually either the proof will succeed, in which case you're done, or the proof will fail, in which case *your failure will help you refine your guess*. Where you get your initial guess is utterly irrelevant[1]—from a classmate, from a textbook, on the web, from the answer to a different problem, scrawled on a bathroom wall in Siebel, included in a care package from your mom, dictated by the machine elves, whatever. If you can prove that the answer is correct, then it's correct!

2.1 Tower of Hanoi

The classical Tower of Hanoi problem gives us the recurrence $\boldsymbol{T(n) = 2T(n-1) + 1}$ with base case $\boldsymbol{T(0) = 0}$. Just looking at the recurrence we can guess that $T(n)$ is something like 2^n. If we write out the first few values of $T(n)$, we discover that they are each one less than a power of two.

$$T(0) = 0, \quad T(1) = 1, \quad T(2) = 3, \quad T(3) = 7, \quad T(4) = 15, \quad T(5) = 31, \quad T(6) = 63, \quad \ldots,$$

It looks like $\boldsymbol{T(n) = 2^n - 1}$ might be the right answer. Let's check.

$$\begin{aligned} T(0) &= 0 = 2^0 - 1 \quad \checkmark \\ T(n) &= 2T(n-1) + 1 \\ &= 2(2^{n-1} - 1) + 1 && \text{[induction hypothesis]} \\ &= 2^n - 1 \quad \checkmark && \text{[algebra]} \end{aligned}$$

We were right! Hooray, we're done!

[1] . . . except of course during exams, where you aren't supposed to use *any* outside sources

Another way we can guess the solution is by ***unrolling*** the recurrence, by substituting it into itself:

$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ &= 2(2T(n-2)+1)+1 \\ &= 4T(n-2)+3 \\ &= 4(2T(n-3)+1)+3 \\ &= 8T(n-3)+7 \\ &= \cdots \end{aligned}$$

It looks like unrolling the initial Hanoi recurrence k times, for any non-negative integer k, will give us the new recurrence $T(n) = 2^k T(n-k) + (2^k - 1)$. Let's prove this by induction:

$$T(n) = 2T(n-1)+1 \quad \checkmark \qquad [k = 0, \text{ by definition}]$$

$$\begin{aligned} T(n) &= 2^{k-1}T(n-(k-1)) + (2^{k-1}-1) && [\text{inductive hypothesis}] \\ &= 2^{k-1}\big(2T(n-k)+1\big) + (2^{k-1}-1) && [\text{initial recurrence for } T(n-(k-1))] \\ &= 2^k T(n-k) + (2^k - 1) \quad \checkmark && [\text{algebra}] \end{aligned}$$

Our guess was correct! In particular, unrolling the recurrence n times give us the recurrence $T(n) = 2^n T(0) + (2^n - 1)$. Plugging in the base case $T(0) = 0$ give us the closed-form solution $T(n) = 2^n - 1$.

2.2 Fibonacci numbers

Let's try a less trivial example: the Fibonacci numbers $\boldsymbol{F_n = F_{n-1} + F_{n-2}}$ with base cases $\boldsymbol{F_0 = 0}$ and $\boldsymbol{F_1 = 1}$. There is no obvious pattern in the first several values (aside from the recurrence itself), but we can reasonably guess that F_n is exponential in n. Let's try to prove inductively that $F_n \le \alpha \cdot c^n$ for some constants $a > 0$ and $c > 1$ and see how far we get.

$$\begin{aligned} F_n &= F_{n-1} + F_{n-2} \\ &\le \alpha \cdot c^{n-1} + \alpha \cdot c^{n-2} && [\text{“induction hypothesis”}] \\ &\le \alpha \cdot c^n \ \textbf{???} \end{aligned}$$

The last inequality is satisfied if $c^n \ge c^{n-1} + c^{n-2}$, or more simply, if $c^2 - c - 1 \ge 0$. The smallest value of c that works is $\phi = (1+\sqrt{5})/2 \approx 1.618034$; the other root of the quadratic equation has smaller absolute value, so we can ignore it.

So we have *most* of an inductive proof that $\boldsymbol{F_n \le \alpha \cdot \phi^n}$ for *some* constant α. All that we're missing are the base cases, which (we can easily guess) must determine the value of the coefficient a. We quickly compute

$$\frac{F_0}{\phi^0} = \frac{0}{1} = 0 \quad \text{and} \quad \frac{F_1}{\phi^1} = \frac{1}{\phi} \approx 0.618034 > 0,$$

so the base cases of our induction proof are correct as long as $\alpha \ge 1/\phi$. It follows that $\boldsymbol{F_n \le \phi^{n-1}}$ for all $n \ge 0$.

What about a matching lower bound? Essentially the same inductive proof implies that $F_n \ge \beta \cdot \phi^n$ for some constant β, but the only value of β that works for *all* n is the trivial $\beta = 0$! We could try to find some lower-order term that makes the base case non-trivial, but an easier approach is to recall that asymptotic $\Omega(\,)$ bounds only have to work for *sufficiently large* n. So let's ignore the trivial base case $F_0 = 0$ and assume that $F_2 = 1$ is a base case instead. Some more easy calculation gives us

$$\frac{F_2}{\phi^2} = \frac{1}{\phi^2} \approx 0.381966 < \frac{1}{\phi}.$$

Thus, the new base cases of our induction proof are correct as long as $\beta \le 1/\phi^2$, which implies that $\boldsymbol{F_n \ge \phi^{n-2}}$ for all $n \ge 1$.

Putting the upper and lower bounds together, we obtain the tight asymptotic bound $\boldsymbol{F_n = \Theta(\phi^n)}$. It *is* possible to get a more exact solution by speculatively refining and conforming our current bounds, but it's not easy. Fortunately, if we really need it, we can get an exact solution using the *annihilator* method, which we'll see later in these notes.

2.3 Mergesort

Mergesort is a classical recursive divide-and-conquer algorithm for sorting an array. The algorithm splits the array in half, recursively sorts the two halves, and then merges the two sorted subarrays into the final sorted array.

```
MERGESORT(A[1..n]):
  if (n > 1)
      m ← ⌊n/2⌋
      MERGESORT(A[1..m])
      MERGESORT(A[m+1..n])
      MERGE(A[1..n], m)
```

```
MERGE(A[1..n], m):
  i ← 1; j ← m+1
  for k ← 1 to n
      if j > n
          B[k] ← A[i]; i ← i+1
      else if i > m
          B[k] ← A[j]; j ← j+1
      else if A[i] < A[j]
          B[k] ← A[i]; i ← i+1
      else
          B[k] ← A[j]; j ← j+1
  for k ← 1 to n
      A[k] ← B[k]
```

Let $T(n)$ denote the worst-case running time of MERGESORT when the input array has size n. The MERGE subroutine clearly runs in $\Theta(n)$ time, so the function $T(n)$ satisfies the following recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + \Theta(n) & \text{otherwise.} \end{cases}$$

For now, let's consider the special case where n is a power of 2; this assumption allows us to take the floors and ceilings out of the recurrence. (We'll see how to deal with the floors and ceilings later; the short version is that they don't matter.)

Because the recurrence itself is given only asymptotically—in terms of $\Theta(\,)$ expressions—we can't hope for anything but an asymptotic solution. So we can safely simplify the recurrence further by removing the Θ's; any asymptotic solution to the simplified recurrence will also satisfy the original recurrence. (This simplification is actually important for another reason; if we kept the asymptotic expressions, we might be tempted to simplify them inappropriately.)

Our simplified recurrence now looks like this:

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ 2T(n/2) + n & \text{otherwise.} \end{cases}$$

To guess at a solution, let's try unrolling the recurrence.

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &= 2\big(2T(n/4) + n/2\big) + n \\ &= 4T(n/4) + 2n \\ &= 8T(n/8) + 3n = \cdots \end{aligned}$$

It looks like $T(n)$ satisfies the recurrence $T(n) = 2^k T(n/2^k) + kn$ for *any* positive integer k. Let's verify this by induction.

$$\begin{aligned}
T(n) &= 2T(n/2) + n = 2^1 T(n/2^1) + 1 \cdot n \quad \checkmark && [k = 1, \text{ given recurrence}] \\
T(n) &= 2^{k-1} T(n/2^{k-1}) + (k-1)n && [\text{inductive hypothesis}] \\
&= 2^{k-1}\big(2T(n/2^k) + n/2^{k-1}\big) + (k-1)n && [\text{substitution}] \\
&= 2^k T(n/2^k) + kn \quad \checkmark && [\text{algebra}]
\end{aligned}$$

Our guess was right! The recurrence becomes trivial when $n/2^k = 1$, or equivalently, when $k = \log_2 n$:

$$T(n) = nT(1) + n\log_2 n = n\log_2 n + n.$$

Finally, we have to put back the Θ's we stripped off; our final closed-form solution is $\boldsymbol{T(n) = \Theta(n \log n)}$.

2.4 An uglier divide-and-conquer example

Consider the divide-and-conquer recurrence $\boldsymbol{T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n}$. This doesn't fit into the form required by the Master Theorem (which we'll see below), but it still sort of resembles the Mergesort recurrence—the total size of the subproblems at the first level of recursion is n—so let's *guess* that $T(n) = O(n \log n)$, and then try to prove that our guess is correct. (We could also attack this recurrence by unrolling, but let's see how far just guessing will take us.)

Let's start by trying to prove an upper bound $T(n) \le a\, n \lg n$ for all sufficiently large n and some constant a to be determined later:

$$\begin{aligned}
T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + n \\
&\le \sqrt{n} \cdot a\sqrt{n} \lg \sqrt{n} + n && [\text{induction hypothesis}] \\
&= (a/2) n \lg n + n && [\text{algebra}] \\
&\le an \lg n \checkmark && [\text{algebra}]
\end{aligned}$$

The last inequality assumes only that $1 \le (a/2)\log n$,or equivalently, that $n \ge 2^{2/a}$. In other words, the induction proof is correct if n is sufficiently large. So we were right!

But before you break out the champagne, what about the multiplicative constant a? The proof worked for *any* constant a, no matter how small. This strongly suggests that our upper bound $T(n) = O(n \log n)$ is not tight. Indeed, if we try to prove a matching lower bound $T(n) \ge b\, n \log n$ for sufficiently large n, we run into trouble.

$$\begin{aligned}
T(n) &= \sqrt{n} \cdot T(\sqrt{n}) + n \\
&\ge \sqrt{n} \cdot b\sqrt{n} \log \sqrt{n} + n && [\text{induction hypothesis}] \\
&= (b/2) n \log n + n \\
&\not\ge bn \log n
\end{aligned}$$

The last inequality would be correct only if $1 > (b/2)\log n$, but that inequality is false for large values of n, no matter which constant b we choose.

Okay, so $\Theta(n \log n)$ is too big. How about $\Theta(n)$? The lower bound is easy to prove directly:

$$T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n \ge n \checkmark$$

But an inductive proof of the upper bound fails.

$$\begin{aligned} T(n) &= \sqrt{n}\cdot T(\sqrt{n}) + n \\ &\le \sqrt{n}\cdot a\sqrt{n} + n && \text{[induction hypothesis]} \\ &= (a+1)n && \text{[algebra]} \\ &\not\le an \end{aligned}$$

Hmmm. So what's bigger than n and smaller than $n \lg n$? How about $n\sqrt{\lg n}$?

$$\begin{aligned} T(n) = \sqrt{n}\cdot T(\sqrt{n}) + n &\le \sqrt{n}\cdot a\sqrt{n}\sqrt{\lg\sqrt{n}} + n && \text{[induction hypothesis]} \\ &= (a/\sqrt{2})\, n\sqrt{\lg n} + n && \text{[algebra]} \\ &\le a\, n\sqrt{\lg n} \quad \text{for large enough } n \checkmark \end{aligned}$$

Okay, the upper bound checks out; how about the lower bound?

$$\begin{aligned} T(n) = \sqrt{n}\cdot T(\sqrt{n}) + n &\ge \sqrt{n}\cdot b\sqrt{n}\sqrt{\lg\sqrt{n}} + n && \text{[induction hypothesis]} \\ &= (b/\sqrt{2})\, n\sqrt{\lg n} + n && \text{[algebra]} \\ &\not\ge b\, n\sqrt{\lg n} \end{aligned}$$

No, the last step doesn't work. So $\Theta(n\sqrt{\lg n})$ doesn't work.

Okay... what else is between n and $n \lg n$? How about $n \lg\lg n$?

$$\begin{aligned} T(n) = \sqrt{n}\cdot T(\sqrt{n}) + n &\le \sqrt{n}\cdot a\sqrt{n}\lg\lg\sqrt{n} + n && \text{[induction hypothesis]} \\ &= a\, n\lg\lg n - a\, n + n && \text{[algebra]} \\ &\le a\, n\lg\lg n \qquad \textbf{if } \boldsymbol{a \ge 1} \checkmark \end{aligned}$$

Hey look at that! For once, our upper bound proof requires a constraint on the hidden constant a. This is an good indication that we've found the right answer. Let's try the lower bound:

$$\begin{aligned} T(n) = \sqrt{n}\cdot T(\sqrt{n}) + n &\ge \sqrt{n}\cdot b\sqrt{n}\lg\lg\sqrt{n} + n && \text{[induction hypothesis]} \\ &= b\, n\lg\lg n - b\, n + n && \text{[algebra]} \\ &\ge b\, n\lg\lg n \qquad \textbf{if } \boldsymbol{b \le 1} \checkmark \end{aligned}$$

Hey, it worked! We have most of an inductive proof that $T(n) \le an \lg\lg n$ for any $a \ge 1$ and most of an inductive proof that $T(n) \ge bn \lg\lg n$ for any $b \le 1$. Technically, we're still missing the base cases in both proofs, but we can be fairly confident at this point that $\boldsymbol{T(n) = \Theta(n \log\log n)}$.

3 Divide and Conquer Recurrences (Recursion Trees)

Many divide and conquer algorithms give us running-time recurrences of the form

$$\boxed{T(n) = a\, T(n/b) + f(n)} \tag{1}$$

where a and b are constants and $f(n)$ is some other function. There is a simple and general technique for solving many recurrences in this and similar forms, using a ***recursion tree***. The root of the recursion

tree is a box containing the value $f(n)$; the root has a children, each of which is the root of a (recursively defined) recursion tree for the function $T(n/b)$.

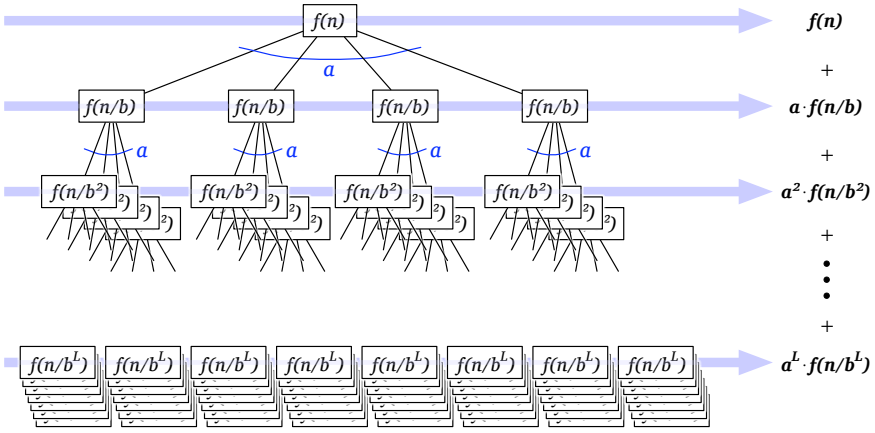
Equivalently, a recursion tree is a complete a-ary tree where each node at depth i contains the value $f(n/b^i)$. The recursion stops when we get to the base case(s) of the recurrence. Because we're only looking for asymptotic bounds, the exact base case doesn't matter; we can safely assume that $T(1) = \Theta(1)$, or even that $T(n) = \Theta(1)$ for all $n \le 10^{100}$. I'll also assume for simplicity that n is an integral power of b; we'll see how to avoid this assumption later (but to summarize: it doesn't matter).

Now $T(n)$ is just the sum of all values stored in the recursion tree. For each i, the ith level of the tree contains a^i nodes, each with value $f(n/b^i)$. Thus,

$$\boxed{T(n) = \sum_{i=0}^{L} a^i f(n/b^i)} \tag{Σ}$$

where L is the depth of the recursion tree. We easily see that $\boldsymbol{L = \log_b n}$, because $n/b^L = 1$. The base case $f(1) = \Theta(1)$ implies that the last non-zero term in the summation is $\Theta(a^L) = \Theta(a^{\log_b n}) = \Theta(n^{\log_b a})$.

For *most* divide-and-conquer recurrences, the level-by-level sum (Σ) is a *geometric* series—each term is a constant factor larger or smaller than the previous term. In this case, only the largest term in the geometric series matters; all of the other terms are swallowed up by the $\Theta(\cdot)$ notation.



A recursion tree for the recurrence $T(n) = a\,T(n/b) + f(n)$

Here are several examples of the recursion-tree technique in action:

- **Mergesort (simplified): $\boldsymbol{T(n) = 2T(n/2) + n}$**

  There are 2^i nodes at level i, each with value $n/2^i$, so every term in the level-by-level sum (Σ) is the same:

  $$T(n) = \sum_{i=0}^{L} n.$$

  The recursion tree has $L = \log_2 n$ levels, so $\boldsymbol{T(n) = \Theta(n \log n)}$.

- **Randomized selection: $\boldsymbol{T(n) = T(3n/4) + n}$**

  The recursion tree is a single path. The node at depth i has value $(3/4)^i n$, so the level-by-level sum (Σ) is a decreasing geometric series:

  $$T(n) = \sum_{i=0}^{L} (3/4)^i n.$$

This geometric series is dominated by its initial term n, so $\boldsymbol{T(n) = \Theta(n)}$. The recursion tree has $L = \log_{4/3} n$ levels, but so what?

- **Karatsuba's multiplication algorithm: $\boldsymbol{T(n) = 3T(n/2) + n}$**

  There are 3^i nodes at depth i, each with value $n/2^i$, so the level-by-level sum (Σ) is an increasing geometric series:

$$T(n) = \sum_{i=0}^{L} (3/2)^i n.$$

  This geometric series is dominated by its final term $(3/2)^L n$. Each leaf contributes 1 to this term; thus, the final term is equal to the number of leaves in the tree! The recursion tree has $L = \log_2 n$ levels, and therefore $3^{\log_2 n} = n^{\log_2 3}$ leaves, so $\boldsymbol{T(n) = \Theta(n^{\log_2 3})}$.

- **$\boldsymbol{T(n) = 2T(n/2) + n/\lg n}$**

  The sum of all the nodes in the ith level is $n/(\lg n - i)$. This implies that the depth of the tree is at most $\lg n - 1$. The level sums are neither constant nor a geometric series, so we just have to evaluate the overall sum directly.

  Recall (or if you're seeing this for the first time: Behold!) that the nth *harmonic number* H_n is the sum of the reciprocals of the first n positive integers:

$$H_n := \sum_{i=1}^{n} \frac{1}{i}$$

  It's not hard to show that $H_n = \Theta(\log n)$; in fact, we have the stronger inequalities $\ln(n+1) \le H_n \le \ln n + 1$.

$$T(n) = \sum_{i=0}^{\lg n - 1} \frac{n}{\lg n - i} = \sum_{j=1}^{\lg n} \frac{n}{j} = n H_{\lg n} = \boldsymbol{\Theta(n \lg \lg n)}$$

- **$\boldsymbol{T(n) = 4T(n/2) + n \lg n}$**

  There are 4^i nodes at each level i, each with value $(n/2^i)\lg(n/2^i) = (n/2^i)(\lg n - i)$; again, the depth of the tree is at most $\lg n - 1$. We have the following summation:

$$T(n) = \sum_{i=0}^{\lg n - 1} n 2^i (\lg n - i)$$

  We can simplify this sum by substituting $j = \lg n - i$:

$$T(n) = \sum_{j=i}^{\lg n} n 2^{\lg n - j} j = \sum_{j=i}^{\lg n} \frac{n^2 j}{2^j} = n^2 \sum_{j=i}^{\lg n} \frac{j}{2^j} = \boldsymbol{\Theta(n^2)}$$

  The last step uses the fact that $\sum_{i=1}^{\infty} j/2^j = 2$. Although this is not quite a geometric series, it is still dominated by its largest term.

- **Ugly divide and conquer:** $T(n) = \sqrt{n} \cdot T(\sqrt{n}) + n$

  We solved this recurrence earlier by guessing the right answer and verifying, but we can use recursion trees to get the correct answer directly. The *degree* of the nodes in the recursion tree is no longer constant, so we have to be a bit more careful, but the same basic technique still applies. It's not hard to see that the nodes in any level sum to n. The depth L satisfies the identity $n^{2^{-L}} = 2$ (we can't get all the way down to 1 by taking square roots), so $L = \lg\lg n$ and $\boldsymbol{T(n) = \Theta(n \lg\lg n)}$.

- **Randomized quicksort:** $T(n) = T(3n/4) + T(n/4) + n$

  This recurrence isn't in the standard form described earlier, but we can still solve it using recursion trees. Now modes in the same level of the recursion tree have different values, and different leaves are at different levels. However, the nodes in any *complete* level (that is, above any of the leaves) sum to n. Moreover, every leaf in the recursion tree has depth between $\log_4 n$ and $\log_{4/3} n$. To derive an upper bound, we overestimate $T(n)$ by ignoring the base cases and extending the tree downward to the level of the *deepest* leaf. Similarly, to derive a lower bound, we overestimate $T(n)$ by counting only nodes in the tree up to the level of the *shallowest* leaf. These observations give us the upper and lower bounds $n \log_4 n \le T(n) \le n \log_{4/3} n$. Since these bounds differ by only a constant factor, we have $\boldsymbol{T(n) = \Theta(n \log n)}$.

- **Deterministic selection:** $T(n) = T(n/5) + T(7n/10) + n$

  Again, we have a lopsided recursion tree. If we look only at complete levels of the tree, we find that the level sums form a descending geometric series $T(n) = n + 9n/10 + 81n/100 + \cdots$. We can get an upper bound by ignoring the base cases entirely and growing the tree out to infinity, and we can get a lower bound by only counting nodes in complete levels. Either way, the geometric series is dominated by its largest term, so $\boldsymbol{T(n) = \Theta(n)}$.

- **Randomized search trees:** $T(n) = \frac{1}{4}T(n/4) + \frac{3}{4}T(3n/4) + 1$

  This looks like a divide-and-conquer recurrence, but what does it mean to have a quarter of a child? The right approach is to imagine that each node in the recursion tree has a *weight* in addition to its value. Alternately, we get a standard recursion tree again if we add a second real parameter to the recurrence, defining $T(n) = T(n, 1)$, where

  $$T(n, \alpha) = T(n/4, \alpha/4) + T(3n/4, 3\alpha/4) + \alpha.$$

  In each complete level of the tree, the (weighted) node values sum to exactly 1. The leaves of the recursion tree are at different levels, but all between $\log_4 n$ and $\log_{4/3} n$. So we have upper and lower bounds $\log_4 n \le T(n) \le \log_{4/3} n$, which differ by only a constant factor, so $\boldsymbol{T(n) = \Theta(\log n)}$.

- **Ham-sandwich trees:** $T(n) = T(n/2) + T(n/4) + 1$

  Again, we have a lopsided recursion tree. If we only look at complete levels, we find that the level sums form an *ascending* geometric series $T(n) = 1 + 2 + 4 + \cdots$, so the solution is dominated by the number of leaves. The recursion tree has $\log_4 n$ complete levels, so there are more than $2^{\log_4 n} = n^{\log_4 2} = \sqrt{n}$; on the other hand, every leaf has depth at most $\log_2 n$, so the total number of leaves is at most $2^{\log_2 n} = n$. Unfortunately, the crude bounds $\boldsymbol{\sqrt{n} \ll T(n) \ll n}$ are the best we can derive using the techniques we know so far!

The following theorem completely describes the solution for any divide-and-conquer recurrence in the 'standard form' $T(n) = aT(n/b) + f(n)$, where a and b are constants and $f(n)$ is a polynomial. This theorem allows us to bypass recursion trees for 'standard' recurrences, but many people (including Jeff) find it harder to remember than the more general recursion-tree technique. Your mileage may vary.

The Master Theorem. *The recurrence $T(n) = aT(n/b) + f(n)$ can be solved as follows.*

- *If $a\,f(n/b) = \kappa\, f(n)$ for some constant $\kappa < 1$, then $T(n) = \Theta(f(n))$.*
- *If $a\,f(n/b) = K\, f(n)$ for some constant $K > 1$, then $T(n) = \Theta(n^{\log_b a})$.*
- *If $a\,f(n/b) = f(n)$, then $T(n) = \Theta(f(n)\log_b n)$.*
- *If none of these three cases apply, you're on your own.*

Proof: If $f(n)$ is a *constant factor larger* than $a\,f(b/n)$, then by induction, the sum is a descending geometric series. The sum of any geometric series is a constant times its largest term. In this case, the largest term is the first term $f(n)$.

If $f(n)$ is a *constant factor smaller* than $a\,f(b/n)$, then by induction, the sum is an ascending geometric series. The sum of any geometric series is a constant times its largest term. In this case, this is the last term, which by our earlier argument is $\Theta(n^{\log_b a})$.

Finally, if $a\,f(b/n) = f(n)$, then by induction, each of the $L+1$ terms in the sum is equal to $f(n)$. □

*4 The Nuclear Bomb

Finally, let me describe *without proof* a powerful generalization of the recursion tree method, first published by Lebanese researchers Mohamad Akra and Louay Bazzi in 1998. Consider a general divide-and-conquer recurrence of the form

$$T(n) = \sum_{i=1}^{k} a_i T(n/b_i) + f(n),$$

where k is a constant, $a_i > 0$ and $b_i > 1$ are constants for all i, and $f(n) = \Omega(n^c)$ and $f(n) = O(n^d)$ for some constants $0 < c \le d$. (As usual, we assume the standard base case $T(\Theta(1)) = \Theta(1)$).) Akra and Bazzi prove that this recurrence has the closed-form asymptotic solution

$$T(n) = \Theta\left(n^\rho \left(1 + \int_1^n \frac{f(u)}{u^{\rho+1}}\,du\right)\right),$$

where ρ is the unique real solution to the equation

$$\sum_{i=1}^{k} a_i / b_i^\rho = 1.$$

In particular, the Akra-Bazzi theorem immediately implies the following form of the Master Theorem:

$$T(n) = aT(n/b) + n^c \implies T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } c < \log_b a - \varepsilon \\ \Theta(n^c \log n) & \text{if } c = \log_b a \\ \Theta(n^c) & \text{if } c > \log_b a + \varepsilon \end{cases}$$

The Akra-Bazzi theorem does not require that the parameters a_i and b_i are integers, or even rationals; on the other hand, even when all parameters are integers, the characteristic equation $\sum_i a_i/b_i^\rho = 1$ may have no analytical solution.

Here are a few examples of recurrences that are difficult (or impossible) for recursion trees, but have easy solutions using the Akra-Bazzi theorem.

- **Randomized quicksort: $T(n) = T(3n/4) + T(n/4) + n$**

  The equation $(3/4)^\rho + (1/4)^\rho = 1$ has the unique solution $\rho = 1$, and therefore

  $$T(n) = \Theta\left(n\left(1 + \int_1^n \frac{1}{u}\,du\right)\right) = O(n \log n).$$

- **Deterministic selection: $T(n) = T(n/5) + T(7n/10) + n$**

  The equation $(1/5)^\rho + (7/10)^\rho = 1$ has no analytical solution. However, we easily observe that $(1/5)^x + (7/10)^x$ is a decreasing function of x, and therefore $0 < \rho < 1$. Thus, we have

  $$\int_1^n \frac{f(u)}{u^{\rho+1}}\,du = \int_1^n u^{-\rho}\,du = \frac{u^{1-\rho}}{1-\rho}\Bigg|_{u=1}^{n} = \frac{n^{1-\rho}-1}{1-\rho} = \Theta(n^{1-\rho}),$$

  and therefore

  $$T(n) = \Theta(n^\rho \cdot (1 + \Theta(n^{1-\rho})) = \Theta(n).$$

  (A bit of numerical computation gives the approximate value $\rho \approx 0.83978$, but why bother?)

- **Randomized search trees: $T(n) = \frac{1}{4}T(n/4) + \frac{3}{4}T(3n/4) + 1$**

  The equation $\frac{1}{4}(\frac{1}{4})^\rho + \frac{3}{4}(\frac{3}{4})^\rho = 1$ has the unique solution $\rho = 0$, and therefore

  $$T(n) = \Theta\left(1 + \int_1^n \frac{1}{u}\,du\right) = \Theta(\log n).$$

- **Ham-sandwich trees: $T(n) = T(n/2) + T(n/4) + 1$.** Recall that we could only prove the very weak bounds $\sqrt{n} \ll T(n) \ll n$ using recursion trees. The equation $(1/2)^\rho + (1/4)^\rho = 1$ has the unique solution $\rho = \log_2((1+\sqrt{5})/2) \approx 0.69424$, which can be obtained by setting $x = 2^\rho$ and solving for x. Thus, we have

  $$\int_1^n \frac{1}{u^{\rho+1}}\,du = \frac{u^{-\rho}}{-\rho}\Bigg|_{u=1}^{n} = \frac{1-n^{-\rho}}{\rho} = \Theta(1)$$

  and therefore

  $$T(n) = \Theta\left(n^\rho(1 + \Theta(1))\right) = \Theta(n^{\lg\phi}).$$

The Akra-Bazzi method is that it can solve *almost* any divide-and-conquer recurrence with just a few lines of calculation. (There are a few nasty exceptions like $T(n) = \sqrt{n}\,T(\sqrt{n}) + n$ where we have to fall back on recursion trees.) On the other hand, the steps appear to be magic, which makes the method hard to remember, and for most divide-and-conquer recurrences that arise in practice, there are much simpler solution techniques.

*5 Linear Recurrences (Annihilators)

Another common class of recurrences, called ***linear*** recurrences, arises in the context of recursive backtracking algorithms and counting problems. These recurrences express each function value $f(n)$ as a *linear* combination of a small number of nearby values $f(n-1), f(n-2), f(n-3), \ldots$. The Fibonacci recurrence is a typical example:

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{cases}$$

It turns out that the solution to *any* linear recurrence is a simple combination of polynomial and exponential functions in n. For example, we can verify by induction that the linear recurrence

$$T(n) = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{if } n = 1 \text{ or } n = 2 \\ 3T(n-1) - 8T(n-2) + 4T(n-3) & \text{otherwise} \end{cases}$$

has the closed-form solution $\boldsymbol{T(n) = (n-3)2^n + 4}$. First we check the base cases:

$$\begin{aligned} T(0) &= (0-3)2^0 + 4 = 1 \quad \checkmark \\ T(1) &= (1-3)2^1 + 4 = 0 \quad \checkmark \\ T(2) &= (2-3)2^2 + 4 = 0 \quad \checkmark \end{aligned}$$

And now the recursive case:

$$\begin{aligned} T(n) &= 3T(n-1) - 8T(n-2) + 4T(n-3) \\ &= 3((n-4)2^{n-1} + 4) - 8((n-5)2^{n-2} + 4) + 4((n-6)2^{n-3} + 4) \\ &= \left(\frac{3}{2} - \frac{8}{4} + \frac{4}{8}\right) n \cdot 2^n - \left(\frac{12}{2} - \frac{40}{4} + \frac{24}{8}\right) 2^n + (2 - 8 + 4) \cdot 4 \\ &= (n-3) \cdot 2^n + 4 \quad \checkmark \end{aligned}$$

But how could we have possibly come up with that solution? In this section, I'll describe a general method for solving linear recurrences that's arguably easier than the induction proof!

5.1 Operators

Our technique for solving linear recurrences relies on the theory of ***operators***. Operators are higher-order functions, which take one or more functions as input and produce different functions as output. For example, your first two semesters of calculus focus almost exclusively on the *differential* and *integral* operators $\frac{d}{dx}$ and $\int dx$. All the operators we will need are combinations of three elementary building blocks:

- ***Sum:*** $(f + g)(n) := f(n) + g(n)$
- ***Scale:*** $(\alpha \cdot f)(n) := \alpha \cdot (f(n))$
- ***Shift:*** $(\boldsymbol{E}f)(n) := f(n+1)$

The shift and scale operators are ***linear***, which means they can be distributed over sums; for example, for any functions f, g, and h, we have $\boldsymbol{E}(f - 3(g - h)) = \boldsymbol{E}f + (-3)\boldsymbol{E}g + 3\boldsymbol{E}h$.

We can combine these building blocks to obtain more complex *compound* operators. For example, the compound operator $\boldsymbol{E} - 2$ is defined by setting $(\boldsymbol{E} - 2)f := \boldsymbol{E}f + (-2)f$ for any function f. We can also apply the shift operator twice: $(\boldsymbol{E}(\boldsymbol{E}f))(n) = f(n+2)$; we write usually $\boldsymbol{E}^2 f$ as a synonym for $\boldsymbol{E}(\boldsymbol{E}f)$. More generally, for any positive integer k, the operator $\boldsymbol{E}^k$ shifts its argument k times: $\boldsymbol{E}^k f(n) = f(n+k)$. Similarly, $(\boldsymbol{E} - 2)^2$ is shorthand for the operator $(\boldsymbol{E} - 2)(\boldsymbol{E} - 2)$, which applies $(\boldsymbol{E} - 2)$ twice.

For example, here are the results of applying different operators to the function $f(n) = 2^n$:

$$\begin{aligned}
2f(n) &= 2 \cdot 2^n = 2^{n+1} \\
3f(n) &= 3 \cdot 2^n \\
\boldsymbol{E}f(n) &= 2^{n+1} \\
\boldsymbol{E}^2 f(n) &= 2^{n+2} \\
(\boldsymbol{E} - 2)f(n) &= \boldsymbol{E}f(n) - 2f(n) = 2^{n+1} - 2^{n+1} = 0 \\
(\boldsymbol{E}^2 - 1)f(n) &= \boldsymbol{E}^2 f(n) - f(n) = 2^{n+2} - 2^n = 3 \cdot 2^n
\end{aligned}$$

These compound operators can be manipulated exactly as though they were polynomials over the 'variable' $\boldsymbol{E}$. In particular, we can 'factor' compound operators into 'products' of simpler operators, and the order of the factors is unimportant. For example, the compound operators $\boldsymbol{E}^2 - 3\boldsymbol{E} + 2$ and $(\boldsymbol{E} - 1)(\boldsymbol{E} - 2)$ are equivalent:

$$\text{Let} \quad g(n) := (\boldsymbol{E} - 2)f(n) = f(n+1) - 2f(n).$$

$$\begin{aligned}
\text{Then} \quad (\boldsymbol{E} - 1)(\boldsymbol{E} - 2)f(n) &= (\boldsymbol{E} - 1)g(n) \\
&= g(n+1) - g(n) \\
&= (f(n+2) - 2f(n-1)) - (f(n+1) - 2f(n)) \\
&= f(n+2) - 3f(n+1) + 2f(n) \\
&= (\boldsymbol{E}^2 - 3\boldsymbol{E} + 2)f(n). \quad \checkmark
\end{aligned}$$

It is an easy exercise to confirm that $\boldsymbol{E}^2 - 3\boldsymbol{E} + 2$ is also equivalent to the operator $(\boldsymbol{E} - 2)(\boldsymbol{E} - 1)$.

The following table summarizes everything we need to remember about operators.

Operator	Definition
addition	$(f + g)(n) := f(n) + g(n)$
subtraction	$(f - g)(n) := f(n) - g(n)$
multiplication	$(\alpha \cdot f)(n) := \alpha \cdot (f(n))$
shift	$\boldsymbol{E}f(n) := f(n+1)$
k-fold shift	$\boldsymbol{E}^k f(n) := f(n+k)$
composition	$(\boldsymbol{X} + \boldsymbol{Y})f := \boldsymbol{X}f + \boldsymbol{Y}f$
	$(\boldsymbol{X} - \boldsymbol{Y})f := \boldsymbol{X}f - \boldsymbol{Y}f$
	$\boldsymbol{X}\boldsymbol{Y}f := \boldsymbol{X}(\boldsymbol{Y}f) = \boldsymbol{Y}(\boldsymbol{X}f)$
distribution	$\boldsymbol{X}(f + g) = \boldsymbol{X}f + \boldsymbol{X}g$

5.2 Annihilators

An ***annihilator*** of a function f is any nontrivial operator that transforms f into the zero function. (We can trivially annihilate any function by multiplying it by zero, so as a technical matter, we do not consider

the zero operator to be an annihilator.) Every compound operator we consider annihilates a specific class of functions; conversely, every function composed of polynomial and exponential functions has a unique (minimal) annihilator.

We have already seen that the operator $(\boldsymbol{E}-2)$ annihilates the function 2^n. It's not hard to see that the operator $(\boldsymbol{E}-c)$ annihilates the function $\alpha \cdot c^n$, for any constants c and α. More generally, the operator $(\boldsymbol{E}-c)$ annihilates the function a^n if and only if $c = a$:

$$(\boldsymbol{E}-c)a^n = \boldsymbol{E}a^n - c \cdot a^n = a^{n+1} - c \cdot a^n = (a-c)a^n.$$

Thus, $(\boldsymbol{E}-2)$ is essentially the *only* annihilator of the function 2^n.

What about the function $2^n + 3^n$? The operator $(\boldsymbol{E}-2)$ annihilates the function 2^n, but leaves the function 3^n unchanged. Similarly, $(\boldsymbol{E}-3)$ annihilates 3^n while *negating* the function 2^n. But if we apply *both* operators, we annihilate both terms:

$$\begin{aligned}(\boldsymbol{E}-2)(2^n+3^n) &= \boldsymbol{E}(2^n+3^n) - 2(2^n+3^n) \\ &= (2^{n+1}+3^{n+1}) - (2^{n+1}+2\cdot 3^n) = 3^n \\ \Longrightarrow (\boldsymbol{E}-3)(\boldsymbol{E}-2)(2^n+3^n) &= (\boldsymbol{E}-3)3^n = 0\end{aligned}$$

In general, for any integers $a \neq b$, the operator $(\boldsymbol{E}-a)(\boldsymbol{E}-b) = (\boldsymbol{E}-b)(\boldsymbol{E}-a) = (\boldsymbol{E}^2 - (a+b)\boldsymbol{E} + ab)$ annihilates any function of the form $\alpha a^n + \beta b^n$, but nothing else.

What about the operator $(\boldsymbol{E}-a)(\boldsymbol{E}-a) = (\boldsymbol{E}-a)^2$? It turns out that this operator annihilates all functions of the form $(\alpha n + \beta)a^n$:

$$\begin{aligned}(\boldsymbol{E}-a)((\alpha n+\beta)a^n) &= (\alpha(n+1)+\beta)a^{n+1} - a(\alpha n+\beta)a^n \\ &= \alpha a^{n+1} \\ \Longrightarrow (\boldsymbol{E}-a)^2((\alpha n+\beta)a^n) &= (\boldsymbol{E}-a)(\alpha a^{n+1}) = 0\end{aligned}$$

More generally, the operator $(\boldsymbol{E}-a)^d$ annihilates all functions of the form $p(n)\cdot a^n$, where $p(n)$ is a polynomial of degree at most $d-1$. For example, $(\boldsymbol{E}-1)^3$ annihilates any polynomial of degree at most 2.

The following table summarizes everything we need to remember about annihilators.

Operator	**Functions annihilated**	
$\boldsymbol{E}-1$	α	
$\boldsymbol{E}-a$	αa^n	
$(\boldsymbol{E}-a)(\boldsymbol{E}-b)$	$\alpha a^n + \beta b^n$	[if $a \neq b$]
$(\boldsymbol{E}-a_0)(\boldsymbol{E}-a_1)\cdots(\boldsymbol{E}-a_k)$	$\sum_{i=0}^{k} \alpha_i a_i^n$	[if a_i distinct]
$(\boldsymbol{E}-1)^2$	$\alpha n + \beta$	
$(\boldsymbol{E}-a)^2$	$(\alpha n+\beta)a^n$	
$(\boldsymbol{E}-a)^2(\boldsymbol{E}-b)$	$(\alpha n+\beta)a^b + \gamma b^n$	[if $a \neq b$]
$(\boldsymbol{E}-a)^d$	$\left(\sum_{i=0}^{d-1} \alpha_i n^i\right)a^n$	

If $\boldsymbol{X}$ annihilates f, then $\boldsymbol{X}$ also annihilates $\boldsymbol{E}f$.
If $\boldsymbol{X}$ annihilates both f and g, then $\boldsymbol{X}$ also annihilates $f \pm g$.
If $\boldsymbol{X}$ annihilates f, then $\boldsymbol{X}$ also annihilates αf, for any constant α.

If $\boldsymbol{X}$ annihilates f and $\boldsymbol{Y}$ annihilates g, then $\boldsymbol{XY}$ annihilates $f \pm g$.

5.3 Annihilating Recurrences

Given a linear recurrence for a function, it's easy to extract an annihilator for that function. For many recurrences, we only need to rewrite the recurrence in operator notation. Once we have an annihilator, we can factor it into operators of the form $(\boldsymbol{E} - c)$; the table on the previous page then gives us a generic solution with some unknown coefficients. If we are given explicit base cases, we can determine the coefficients by examining a few small cases; in general, this involves solving a small system of linear equations. If the base cases are not specified, the generic solution almost always gives us an asymptotic solution. Here is the technique step by step:

1. Write the recurrence in operator form
2. Extract an annihilator for the recurrence
3. Factor the annihilator (if necessary)
4. Extract the *generic solution* from the annihilator
5. Solve for coefficients using base cases (if known)

Here are several examples of the technique in action:

- $\boldsymbol{r(n) = 5r(n-1)}$**, where** $\boldsymbol{r(0) = 3}$**.**

  1. We can write the recurrence in operator form as follows:

     $$r(n) = 5r(n-1) \implies r(n+1) - 5r(n) = 0 \implies (\boldsymbol{E} - 5)r(n) = 0.$$

  2. We immediately see that $(\boldsymbol{E} - 5)$ annihilates the function $r(n)$.
  3. The annihilator $(\boldsymbol{E} - 5)$ is already factored.
  4. Consulting the annihilator table on the previous page, we find the generic solution $r(n) = \alpha 5^n$ for some constant α.
  5. The base case $r(0) = 3$ implies that $\alpha = 3$.

  We conclude that $\boldsymbol{r(n) = 3 \cdot 5^n}$. We can easily verify this closed-form solution by induction:

  $$\begin{aligned} r(0) &= 3 \cdot 5^0 = 3 \quad \checkmark && [\text{definition}] \\ r(n) &= 5r(n-1) && [\text{definition}] \\ &= 5 \cdot (3 \cdot 5^{n-1}) && [\text{induction hypothesis}] \\ &= 5^n \cdot 3 \quad \checkmark && [\text{algebra}] \end{aligned}$$

- ***Fibonacci numbers:*** $\boldsymbol{F(n) = F(n-1) + F(n-2)}$**, where** $\boldsymbol{F(0) = 0}$ ***and*** $\boldsymbol{F(1) = 1}$**.**

  1. We can rewrite the recurrence as $(\boldsymbol{E}^2 - \boldsymbol{E} - 1)F(n) = 0$.
  2. The operator $\boldsymbol{E}^2 - \boldsymbol{E} - 1$ clearly annihilates $F(n)$.
  3. The quadratic formula implies that the annihilator $\boldsymbol{E}^2 - \boldsymbol{E} - 1$ factors into $(\boldsymbol{E} - \phi)(\boldsymbol{E} - \hat{\phi})$, where $\phi = (1 + \sqrt{5})/2 \approx 1.618034$ is the golden ratio and $\hat{\phi} = (1 - \sqrt{5})/2 = 1 - \phi = -1/\phi \approx -0.618034$.
  4. The annihilator implies that $\boldsymbol{F(n) = \alpha\phi^n + \hat{\alpha}\hat{\phi}^n}$ for some unknown constants α and $\hat{\alpha}$.

5. The base cases give us two equations in two unknowns:

$$F(0) = 0 = \alpha + \hat{\alpha}$$
$$F(1) = 1 = \alpha\phi + \hat{\alpha}\hat{\phi}$$

Solving this system of equations gives us $\alpha = 1/(2\phi - 1) = 1/\sqrt{5}$ and $\hat{\alpha} = -1/\sqrt{5}$.

We conclude with the following exact closed form for the nth Fibonacci number:

$$\boldsymbol{F(n) = \frac{\phi^n - \hat{\phi}^n}{\sqrt{5}} = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n}$$

With all the square roots in this formula, it's quite amazing that Fibonacci numbers are integers. However, if we do all the math correctly, all the square roots cancel out when i is an integer. (In fact, this is pretty easy to prove using the binomial theorem.)

- ***Towers of Hanoi:* $\boldsymbol{T(n) = 2T(n-1) + 1}$, *where* $\boldsymbol{T(0) = 0}$.** This is our first example of a *non-homogeneous* recurrence, which means the recurrence has one or more non-recursive terms.

  1. We can rewrite the recurrence as $(\boldsymbol{E} - 2)T(n) = 1$.
  2. The operator $(\boldsymbol{E} - 2)$ doesn't quite annihilate the function; it leaves a *residue* of 1. But we can annihilate the residue by applying the operator $(\boldsymbol{E} - 1)$. Thus, the compound operator $(\boldsymbol{E} - 1)(\boldsymbol{E} - 2)$ annihilates the function.
  3. The annihilator is already factored.
  4. The annihilator table gives us the generic solution $T(n) = \alpha 2^n + \beta$ for some unknown constants α and β.
  5. The base cases give us $T(0) = 0 = \alpha 2^0 + \beta$ and $T(1) = 1 = \alpha 2^1 + \beta$. Solving this system of equations, we find that $\alpha = 1$ and $\beta = -1$.

  We conclude that $\boldsymbol{T(n) = 2^n - 1}$.

For the remaining examples, I won't explicitly enumerate the steps in the solution.

- ***Height-balanced trees:* $\boldsymbol{H(n) = H(n-1) + H(n-2) + 1}$, *where* $\boldsymbol{H(-1) = 0}$ *and* $\boldsymbol{H(0) = 1}$.** (Yes, we're starting at -1 instead of 0. So what?)

  We can rewrite the recurrence as $(\boldsymbol{E}^2 - \boldsymbol{E} - 1)H = 1$. The residue 1 is annihilated by $(\boldsymbol{E} - 1)$, so the compound operator $(\boldsymbol{E} - 1)(\boldsymbol{E}^2 - \boldsymbol{E} - 1)$ annihilates the recurrence. This operator factors into $(\boldsymbol{E} - 1)(\boldsymbol{E} - \phi)(\boldsymbol{E} - \hat{\phi})$, where $\phi = (1 + \sqrt{5})/2$ and $\hat{\phi} = (1 - \sqrt{5})/2$. Thus, we get the generic solution $H(n) = \alpha \cdot \phi^n + \beta + \gamma \cdot \hat{\phi}^n$, for some unknown constants α, β, γ that satisfy the following system of equations:

$$\begin{aligned} H(-1) = 0 = \alpha\phi^{-1} + \beta + \gamma\hat{\phi}^{-1} &= \alpha/\phi + \beta - \gamma/\hat{\phi} \\ H(0) = 1 = \alpha\phi^0 + \beta + \gamma\hat{\phi}^0 &= \alpha + \beta + \gamma \\ H(1) = 2 = \alpha\phi^1 + \beta + \gamma\hat{\phi}^1 &= \alpha\phi + \beta + \gamma\hat{\phi} \end{aligned}$$

Solving this system (using Cramer's rule or Gaussian elimination), we find that $\alpha = (\sqrt{5} + 2)/\sqrt{5}$, $\beta = -1$, and $\gamma = (\sqrt{5} - 2)/\sqrt{5}$. We conclude that

$$\boldsymbol{H(n) = \frac{\sqrt{5}+2}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - 1 + \frac{\sqrt{5}-2}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n.}$$

- $\boldsymbol{T(n) = 3T(n-1) - 8T(n-2) + 4T(n-3)}$**, where** $\boldsymbol{T(0) = 1}$**,** $\boldsymbol{T(1) = 0}$**, and** $\boldsymbol{T(2) = 0}$**.** This was our original example of a linear recurrence.

  We can rewrite the recurrence as $(E^3 - 3E^2 + 8E - 4)T = 0$, so we immediately have an annihilator $E^3 - 3E^2 + 8E - 4$. Using high-school algebra, we can factor the annihilator into $(E-2)^2(E-1)$, which implies the generic solution $T(n) = \alpha n 2^n + \beta 2^n + \gamma$. The constants α, β, and γ are determined by the base cases:

$$\begin{aligned} T(0) &= 1 = \alpha \cdot 0 \cdot 2^0 + \beta 2^0 + \gamma = \beta + \gamma \\ T(1) &= 0 = \alpha \cdot 1 \cdot 2^1 + \beta 2^1 + \gamma = 2\alpha + 2\beta + \gamma \\ T(2) &= 0 = \alpha \cdot 2 \cdot 2^2 + \beta 2^2 + \gamma = 8\alpha + 4\beta + \gamma \end{aligned}$$

  Solving this system of equations, we find that $\alpha = 1$, $\beta = -3$, and $\gamma = 4$, so $\boldsymbol{T(n) = (n-3)2^n + 4}$.

- $\boldsymbol{T(n) = T(n-1) + 2T(n-2) + 2^n - n^2}$

  We can rewrite the recurrence as $(E^2 - E - 2)T(n) = E^2(2^n - n^2)$. Notice that we had to shift up the non-recursive parts of the recurrence when we expressed it in this form. The operator $(E-2)(E-1)^3$ annihilates the residue $2^n - n^2$, and therefore also annihilates the shifted residue $E^2(2^n - n^2)$. Thus, the operator $(E-2)(E-1)^3(E^2-E-2)$ annihilates the entire recurrence. We can factor the quadratic factor into $(E-2)(E+1)$, so the annihilator factors into $(E-2)^2(E-1)^3(E+1)$. So the generic solution is $\boldsymbol{T(n) = \alpha n 2^n + \beta 2^n + \gamma n^2 + \delta n + \varepsilon + \eta(-1)^n}$. The coefficients α, β, γ, δ, ε, η satisfy a system of six equations determined by the first six function values $T(0)$ through $T(5)$. For almost[2] every set of base cases, we have $\alpha \neq 0$, which implies that $\boldsymbol{T(n) = \Theta(n2^n)}$.

For a more detailed explanation of the annihilator method, see George Lueker, Some techniques for solving recurrences, *ACM Computing Surveys* 12(4):419-436, 1980.

6 Transformations

Sometimes we encounter recurrences that don't fit the structures required for recursion trees or annihilators. In many of those cases, we can transform the recurrence into a more familiar form, by defining a new function in terms of the one we want to solve. There are many different kinds of transformations, but these three are probably the most useful:

- **Domain transformation:** Define a new function $S(n) = T(f(n))$ with a simpler recurrence, for some simple function f.
- **Range transformation:** Define a new function $S(n) = f(T(n))$ with a simpler recurrence, for some simple function f.
- **Difference transformation:** Simplify the recurrence for $T(n)$ by considering the difference function $\Delta T(n) = T(n) - T(n-1)$.

Here are some examples of these transformations in action.

- **Unsimplified Mergesort:** $\boldsymbol{T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n)}$

  When n is a power of 2, we can simplify the mergesort recurrence to $T(n) = 2T(n/2) + \Theta(n)$, which has the solution $T(n) = \Theta(n \log n)$. Unfortunately, for other values values of n, this simplified

[2] In fact, the only possible solutions with $\alpha = 0$ have the form $-2^{n-1} - n^2/2 - 5n/2 + \eta(-1)^n$ for some constant η.

recurrence is incorrect. When n is odd, then the recurrence calls for us to sort a fractional number of elements! Worse yet, if n is not a power of 2, we will *never* reach the base case $T(1) = 1$.

So we really need to solve the original recurrence. We have no hope of getting an *exact* solution, even if we ignore the $\Theta(\,)$ in the recurrence; the floors and ceilings will eventually kill us. But we can derive a tight asymptotic solution using a domain transformation—we can rewrite the function $T(n)$ as a nested function $S(f(n))$, where $f(n)$ is a simple function and the function $S(\,)$ has an simpler recurrence.

First let's overestimate the time bound, once by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:

$$T(n) \le 2T\left(\lceil n/2 \rceil\right) + n \le 2T(n/2+1) + n.$$

Now we define a new function $S(n) = T(n+\alpha)$, where α is a unknown constant, chosen so that $S(n)$ satisfies the Master-Theorem-ready recurrence $S(n) \le 2S(n/2) + O(n)$. To figure out the correct value of α, we compare two versions of the recurrence for the function $T(n+\alpha)$:

$$\begin{aligned} S(n) \le 2S(n/2) + O(n) \quad &\Longrightarrow \quad T(n+\alpha) \le 2T(n/2+\alpha) + O(n) \\ T(n) \le 2T(n/2+1) + n \quad &\Longrightarrow \quad T(n+\alpha) \le 2T((n+\alpha)/2+1) + n + \alpha \end{aligned}$$

For these two recurrences to be equal, we need $n/2 + \alpha = (n+\alpha)/2 + 1$, which implies that $\alpha = 2$. The Master Theorem now tells us that $S(n) = O(n \log n)$, so

$$T(n) = S(n-2) = O((n-2)\log(n-2)) = O(n \log n).$$

A similar argument implies the matching lower bound $T(n) = \Omega(n \log n)$. So $\boldsymbol{T(n) = \Theta(n \log n)}$ after all, just as though we had ignored the floors and ceilings from the beginning!

Domain transformations are useful for removing floors, ceilings, and lower order terms from the arguments of any recurrence that otherwise looks like it ought to fit either the Master Theorem or the recursion tree method. But now that we know this, we don't need to bother grinding through the actual gory details!

- **Ham-Sandwich Trees: $\boldsymbol{T(n) = T(n/2) + T(n/4) + 1}$**

  As we saw earlier, the recursion tree method only gives us the uselessly loose bounds $\sqrt{n} \ll T(n) \ll n$ for this recurrence, and the recurrence is in the wrong form for annihilators. The authors who discovered ham-sandwich trees (Yes, this is a real data structure!) solved this recurrence by guessing the solution and giving a complicated induction proof. We got a tight solution using the Akra-Bazzi method, but who can remember that?

  In fact, a simple domain transformation allows us to solve the recurrence in just a few lines. We define a new function $t(k) = T(2^k)$, which satisfies the simpler linear recurrence $t(k) = t(k-1) + t(k-2) + 1$. This recurrence should immediately remind you of Fibonacci numbers. Sure enough, the annihilator method implies the solution $t(k) = \Theta(\phi^k)$, where $\phi = (1+\sqrt{5})/2$ is the golden ratio. We conclude that

  $$T(n) = t(\lg n) = \Theta(\phi^{\lg n}) = \boldsymbol{\Theta(n^{\lg \phi})} \approx \Theta(n^{0.69424}).$$

  This is the same solution we obtained earlier using the Akra-Bazzi theorem.

  Many other divide-and-conquer recurrences can be similarly transformed into linear recurrences and then solved with annihilators. Consider once more the simplified mergesort recurrence

$T(n) = 2T(n/2) + n$. The function $t(k) = T(2^k)$ satisfies the recurrence $t(k) = 2t(k-1) + 2^k$. The annihilator method gives us the generic solution $t(k) = \Theta(k \cdot 2^k)$, which implies that $T(n) = t(\lg n) = \Theta(n \log n)$, just as we expected.

On the other hand, for some recurrences like $T(n) = T(n/3) + T(2n/3) + n$, the recursion tree method gives an easy solution, but there's no way to transform the recurrence into a form where we can apply the annihilator method directly.[3]

- **Random Binary Search Trees: $T(n) = \frac{1}{4}T(n/4) + \frac{3}{4}T(3n/4) + 1$**

  This looks like a divide-and-conquer recurrence, so we might be tempted to apply recursion trees, but what does it mean to have a quarter of a child? If we're not comfortable with weighted recursion trees (or the Akra-Bazzi theorem), we can instead apply the following range transformation. The function $U(n) = n \cdot T(n)$ satisfies the more palatable recurrence $U(n) = U(n/4) + U(3n/4) + n$. As we've already seen, recursion trees imply that $U(n) = \Theta(n \log n)$, which immediately implies that $\boldsymbol{T(n) = \Theta(\log n)}$.

- **Randomized Quicksort: $T(n) = \dfrac{2}{n}\displaystyle\sum_{k=0}^{n-1} T(k) + n$**

  This is our first example of a ***full history*** recurrence; each function value $T(n)$ is defined in terms of *all* previous function values $T(k)$ with $k < n$. Before we can apply any of our existing techniques, we need to convert this recurrence into an equivalent *limited history* form by shifting and subtracting away common terms. To make this step slightly easier, we first multiply both sides of the recurrence by n to get rid of the fractions.

$$n \cdot T(n) = 2\sum_{k=0}^{n-1} T(j) + n^2 \qquad \text{[multiply both sides by } n\text{]}$$

$$(n-1) \cdot T(n-1) = 2\sum_{k=0}^{n-2} T(j) + (n-1)^2 \qquad \text{[shift]}$$

$$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1 \qquad \text{[subtract]}$$

$$T(n) = \frac{n+1}{n}T(n-1) + 2 - \frac{1}{n} \qquad \text{[simplify]}$$

  We can solve this limited-history recurrence using another functional transformation. We define a new function $t(n) = T(n)/(n+1)$, which satisfies the simpler recurrence

$$t(n) = t(n-1) + \frac{2}{n+1} - \frac{1}{n(n+1)},$$

  which we can easily unroll into a summation. If we only want an asymptotic solution, we can simplify the final recurrence to $t(n) = t(n-1) + \Theta(1/n)$, which unrolls into a very familiar summation:

$$t(n) = \sum_{i=1}^{n} \Theta(1/i) = \Theta(H_n) = \Theta(\log n).$$

[3] However, we can still get a solution via functional transformations as follows. The function $t(k) = T((3/2)^k)$ satisfies the recurrence $t(n) = t(n-1) + t(n-\lambda) + (3/2)^k$, where $\lambda = \log_{3/2} 3 = 2.709511\ldots$. The ***characteristic function*** for this recurrence is $(r^\lambda - r^{\lambda-1} - 1)(r - 3/2)$, which has a double root at $r = 3/2$ and nowhere else. Thus, $t(k) = \Theta(k(3/2)^k)$, which implies that $T(n) = t(\log_{3/2} n) = \Theta(n \log n)$. This line of reasoning is the core of the Akra-Bazzi method.

Finally, substituting $T(n) = (n+1)t(n)$ gives us a solution to the original recurrence: $\boldsymbol{T(n) = \Theta(n \log n)}$.

Exercises

1. For each of the following recurrences, first ***guess*** an exact closed-form solution, and then prove your guess is correct. You are free to use any method you want to make your guess—unrolling the recurrence, writing out the first several values, induction proof template, recursion trees, annihilators, transformations, 'It looks like that other one', whatever—but please describe your method. All functions are from the non-negative integers to the reals. If it simplifies your solutions, express them in terms of Fibonacci numbers F_n, harmonic numbers H_n, binomial coefficients $\binom{n}{k}$, factorials $n!$, and/or the floor and ceiling functions $\lfloor x \rfloor$ and $\lceil x \rceil$.

   (a) $A(n) = A(n-1) + 1$, where $A(0) = 0$.

   (b) $B(n) = \begin{cases} 0 & \text{if } n < 5 \\ B(n-5) + 2 & \text{otherwise} \end{cases}$

   (c) $C(n) = C(n-1) + 2n - 1$, where $C(0) = 0$.

   (d) $D(n) = D(n-1) + \binom{n}{2}$, where $D(0) = 0$.

   (e) $E(n) = E(n-1) + 2^n$, where $E(0) = 0$.

   (f) $F(n) = 3 \cdot F(n-1)$, where $F(0) = 1$.

   (g) $G(n) = \frac{G(n-1)}{G(n-2)}$, where $G(0) = 1$ and $G(1) = 2$. *[Hint: This is easier than it looks.]*

   (h) $H(n) = H(n-1) + 1/n$, where $H(0) = 0$.

   (i) $I(n) = I(n-2) + 3/n$, where $I(0) = I(1) = 0$. *[Hint: Consider even and odd n separately.]*

   (j) $J(n) = J(n-1)^2$, where $J(0) = 2$.

   (k) $K(n) = K(\lfloor n/2 \rfloor) + 1$, where $K(0) = 0$.

   (l) $L(n) = L(n-1) + L(n-2)$, where $L(0) = 2$ and $L(1) = 1$.
   [Hint: Write the solution in terms of Fibonacci numbers.]

   (m) $M(n) = M(n-1) \cdot M(n-2)$, where $M(0) = 2$ and $M(1) = 1$.
   [Hint: Write the solution in terms of Fibonacci numbers.]

   (n) $N(n) = 1 + \sum_{k=1}^{n} (N(k-1) + N(n-k))$, where $N(0) = 1$.

   (p) $P(n) = \sum_{k=0}^{n-1} (k \cdot P(k-1))$, where $P(0) = 1$.

   (q) $Q(n) = \frac{1}{2 - Q(n-1)}$, where $Q(0) = 0$.

   (r) $R(n) = \max_{1 \le k \le n} \{R(k-1) + R(n-k) + n\}$

   (s) $S(n) = \max_{1 \le k \le n} \{S(k-1) + S(n-k) + 1\}$

   (t) $T(n) = \min_{1 \le k \le n} \{T(k-1) + T(n-k) + n\}$

(u) $U(n) = \min_{1 \le k \le n} \{U(k-1) + U(n-k) + 1\}$

(v) $V(n) = \max_{n/3 \le k \le 2n/3} \{V(k-1) + V(n-k) + n\}$

2. Use recursion trees or the Akra-Bazzi theorem to solve each of the following recurrences.

(a) $A(n) = 2A(n/4) + \sqrt{n}$

(b) $B(n) = 2B(n/4) + n$

(c) $C(n) = 2C(n/4) + n^2$

(d) $D(n) = 3D(n/3) + \sqrt{n}$

(e) $E(n) = 3E(n/3) + n$

(f) $F(n) = 3F(n/3) + n^2$

(g) $G(n) = 4G(n/2) + \sqrt{n}$

(h) $H(n) = 4H(n/2) + n$

(i) $I(n) = 4I(n/2) + n^2$

(j) $J(n) = J(n/2) + J(n/3) + J(n/6) + n$

(k) $K(n) = K(n/2) + K(n/3) + K(n/6) + n^2$

(l) $L(n) = L(n/15) + L(n/10) + 2L(n/6) + \sqrt{n}$

*(m) $M(n) = 2M(n/3) + 2M(2n/3) + n$

(n) $N(n) = \sqrt{2n}\,N(\sqrt{2n}) + \sqrt{n}$

(p) $P(n) = \sqrt{2n}\,P(\sqrt{2n}) + n$

(q) $Q(n) = \sqrt{2n}\,Q(\sqrt{2n}) + n^2$

(r) $R(n) = R(n-3) + 8^n$ — Don't use annihilators!

(s) $S(n) = 2S(n-2) + 4^n$ — Don't use annihilators!

(t) $T(n) = 4T(n-1) + 2^n$ — Don't use annihilators!

3. Make up a bunch of linear recurrences and then solve them using annihilators.

4. Solve the following recurrences, using any tricks at your disposal.

(a) $T(n) = \sum_{i=1}^{\lg n} T(n/2^i) + n$ *[Hint: Assume n is a power of 2.]*

(b) More to come...