CCES – College of Computer Engineering and Sciences

Prince Sattam Bin Abdulaziz University

CS3401 – Algorithm Design and Analysis

Chapter 1

Introduction to Algorithm Design

Academic Year 2026 – 2027

Dr. Khalil Chebil

Dr. Esraa Eldesouky

Chapter Overview

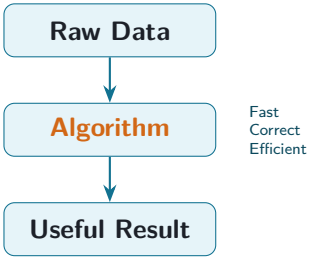
Section 1

Why Algorithms Matter

The Invisible Engine Behind Modern Technology

Every second:

- **Google** processes >100 000 search queries
- **Spotify** streams to >600 million users
- **GPS** recalculates routes for millions of drivers
- **Banks** clear thousands of transactions



The Question

What tells a computer *how* to solve these problems correctly and at speed?

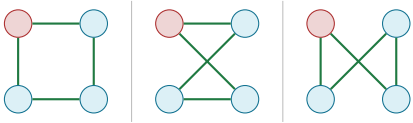
A Motivating Example: Route Planning

Problem: Given n locations, find the shortest round trip that visits every location and returns to the start.

Brute-force idea: enumerate *every* possible route, compute its length, and return the shortest.

- $n = 4$: 3 possible routes (shown right)
- $n = 5$: 12 possible routes
- $n = 6$: 60 possible routes
- $n = 10$: $\approx 181\,000$ routes
- $n = 20$: $\approx 6 \times 10^{16}$ routes

The 3 routes for $n = 4$



General Formula

Number of routes $= \dfrac{(n-1)!}{2}$

— checking them all is **impossible** for large n

Section 2

What is an Algorithm?

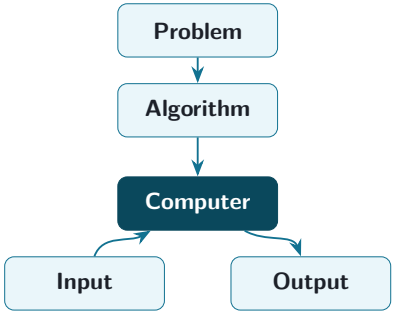
Definition

Algorithm

A **finite**, **unambiguous** sequence of instructions that, given some *input*, produces a corresponding *output* and terminates in a finite number of steps.

Key properties (DIOFEG)

- **D**efiniteness – each step is precisely defined
- **I**nput – zero or more inputs
- **O**utput – at least one output
- **F**initeness – must terminate
- **E**ffectiveness – steps are basic and doable
- **G**enerality – works for a class of inputs



Special Features of Algorithms

- **Non-ambiguity** — every instruction has exactly one interpretation
- **Range of inputs** — must handle all valid inputs, not just examples
- **Multiple representations** — the same algorithm can be expressed as pseudocode, flowchart, or code
- **Multiple algorithms** — one problem can have many solutions
- **Different efficiencies** — different algorithms for the same problem can differ enormously in speed

Same Problem, Three Algorithms

Computing $\gcd(60, 24)$:

1. **Euclid's algorithm** — elegant, fast
2. **Consecutive integer checking** — slow, limited
3. **Middle-school factoring** — complex, very slow

All three give the answer 12 — but not equally fast.

Section 3

Case Study — Greatest Common Divisor

Problem Statement

Greatest Common Divisor

Given two non-negative integers m and n (not both zero), find $\gcd(m, n)$ — the *largest integer that divides both*.

Examples

- $\gcd(60,\ 24) = 12$
- $\gcd(60,\ \ 0) = 60$
- $\gcd(\ 7,\ \ 5) = 1$ (coprime)
- $\gcd(\ 0,\ \ 0) =$? undefined

Three classical approaches

Algorithm	Efficiency	Notes
Euclid's	$\mathcal{O}(\log \min(m, n))$	Best choice
Consecutive check	$\mathcal{O}(n)$	Fails for $n = 0$
Middle-school	Very slow	Ambiguous steps

Algorithm 1: Euclid's Algorithm

Key insight:

$$\gcd(m,\ n) = \gcd(n,\ m \bmod n)$$

Repeat until the second argument becomes 0.

Pseudocode

Input: $m, n \geq 0$, not both zero
Output: $\gcd(m, n)$

```
1: while n ≠ 0
2:     r ← m mod n
3:     m ← n;  n ← r
4: end while
5: return m
```

Trace: $\gcd(60, 24)$

m	n	$r = m \bmod n$
60	**24**	**12**
24	**12**	**0**
12	**0**	—

Result: $\gcd(60, 24) =$ 12

Efficiency

The number of iterations is $\mathcal{O}(\log \min(m, n))$

Algorithm 1: Implementation

Recursive version

```
int gcd(int m, int n) {
    if (n == 0) return m;
    return gcd(n, m % n);
}
```

Iterative version

```
int gcd(int m, int n) {
    while (n != 0) {
        int r = m % n;
        m = n;
        n = r;
    }
    return m;
}
```

Why the iterative version is preferred

- No function-call overhead
- Constant stack space
- Same time complexity $\mathcal{O}(\log \min(m, n))$

Convergence guarantee

After two consecutive iterations, the second argument decreases by *at least half* — hence $\mathcal{O}(\log \min(m, n))$.

Algorithm 2: Consecutive Integer Checking

Idea: Start from $t = \min(m, n)$ and test each candidate downward until we find one that divides both.

Pseudocode

1: $t \leftarrow \min(m, n)$
2: **while** $t > 0$
3: **if** $m \bmod t = 0$ **and** $n \bmod t = 0$ **then**
4: **return** t
5: **end if**
6: $t \leftarrow t - 1$
7: **end while**

Limitations

- **Fails** when either input is **zero** ($\min(m, 0) = 0$, loop never runs)
- Worst case: $\mathcal{O}(\min(m, n))$ iterations
- Much slower than Euclid's for large inputs

Example: $\gcd(60, 24)$

- $t = 24 \ldots 13$: 12 wasted steps
- $t = 12$: $12 \mid 60$ and $12 \mid 24$ ✓

Algorithm 3: Middle-School Procedure

Steps:

1. Find the prime factorization of m
2. Find the prime factorization of n
3. Identify all common prime factors (with multiplicity)
4. Multiply them to get $\gcd(m, n)$

Example: $\gcd(60, 24)$

$$60 = 2 \cdot 2 \cdot 3 \cdot 5 \qquad 24 = 2 \cdot 2 \cdot 3 \cdot 2$$

$$\gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12$$

Why this algorithm is problematic

- Prime factorization is **not defined unambiguously** in the problem — it is itself a hard sub-problem
- Step 3 ("find common factors") is also not clearly specified
- Fails when an input equals 1
- Overall complexity: much worse than $\mathcal{O}(\log n)$

Lesson

Even a familiar school method can be a *bad* algorithm.

Sieve of Eratosthenes

Goal: Produce all primes $\leq n$ (needed by the middle-school method).

Pseudocode

```
1: for p ← 2 to n   A[p] ← p
2: end for
3: for p ← 2 to ⌊√n⌋
4:     if A[p] ≠ 0 then
5:         j ← p^2
6:         while j ≤ n
7:             A[j] ← 0;   j ← j + p
8:         end while
9:     end if
10: end for
11: Output all A[p] ≠ 0
```

Trace for $n = 25$ (successive passes, $p = 2, 3, 5$):

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

Primes ≤ 25: **2, 3, 5, 7, 11, 13, 17, 19, 23**

Complexity: $\mathcal{O}(n \log \log n)$

Section 4

Algorithm Correctness

What Does "Correct" Mean?

Definition: Correctness

An algorithm is **correct** if, for *every* valid input, it terminates and produces the *right output*.

Why it is non-trivial:

- The set of valid inputs is usually *infinite*
- Testing a few examples is **not a proof**
- Intuition can be deceived by edge cases
- Formal proof (mathematical induction, loop invariants) is required

Common pitfall

"It works on the examples I tried" $\neq$ correct algorithm.

A single *counter-example* is enough to *disprove* correctness.

Good practice

After designing an algorithm, actively try to find inputs where it *fails*.

Case Study: The Shortest Tour Problem

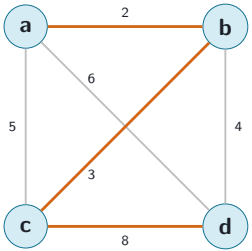
Problem: Given n cities with known pairwise distances, find the *shortest* round tour that visits each city exactly once (Travelling Salesman Problem — TSP).

Applications:

- Delivery route optimization
- Circuit board drilling
- DNA sequencing

Proposed heuristic: Nearest Neighbor (NN)

1. Start at any city
2. Repeatedly visit the *nearest unvisited* city
3. Return to the start



NN tour: a→b→c→d→a = 19

Optimal: a→c→b→d→a = 18

Nearest-Neighbor Counter-Example

Points on a number line:

NN from point 4 (value 0):

- Visit 5 (dist 1), then 3 (dist 2), then 2 (dist 4), then 1 (dist 16), back to 4 (dist 21)
- Total: **44 units**

Optimal tour: $1 \to 2 \to 3 \to 4 \to 5 \to 1 =$ 22 units

Conclusion

The Nearest-Neighbor heuristic **does not always** produce an optimal tour.

Key lesson

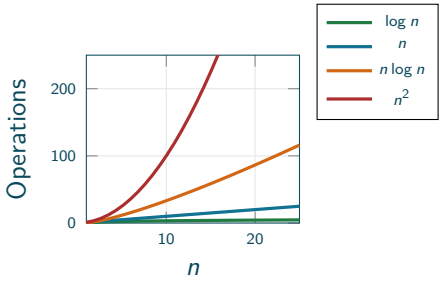
- Seeking *counter-examples* is a vital part of algorithm design
- A plausible algorithm is not necessarily a correct one
- TSP has no known efficient exact algorithm (it is NP-hard)

Section 5

Algorithm Efficiency

Why Efficiency Matters

- Hardware improvements are *incremental*; algorithmic improvements can be *exponential*
- A $10\times$ faster CPU helps — switching from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$ helps *much more*
- Real systems must scale: from 10 000 to 10 000 000 users



Software always outstrips hardware

Even the fastest computer cannot save a poorly designed exponential-time algorithm for large inputs.

How to Measure Efficiency

Machine-independent approach:

- Analyse *pseudocode*, not actual code
- Assume an *idealized model*: each basic operation takes one unit of time
- Count the *number of basic operations* as a function of the *input size n*

$$T(n) \approx c_{\text{op}} \cdot C(n)$$

c_{op}: time per operation $C(n)$: operation count

Typical Input Size Measures

Problem	Size
Sorting	$n = \#$ elements
Matrix multiply	$n \times n$ dimension
Graph problems	$\#$ vertices $+$ edges
Primality test	$\#$ digits of n

Typical Basic Operations

key comparison, addition, multiplication, array access

Faster Algorithm vs. Faster CPU

Algorithm S1 (slow, on a fast machine):

- $T(n) = 2n^2$ operations, CPU @ 10^{10} ops/sec
- For $n = 10^6$: 2×10^{12} ops $\rightarrow$ 200 seconds

Algorithm S2 (fast, on a slow machine):

- $T(n) = 50n \log_2 n$ ops, CPU @ 10^8 ops/sec
- For $n = 10^6$: $\approx 10^9$ ops $\rightarrow$ **10 seconds**

Takeaway

A *better algorithm* on a *slower machine* beats a *worse algorithm* on a faster machine for large inputs.

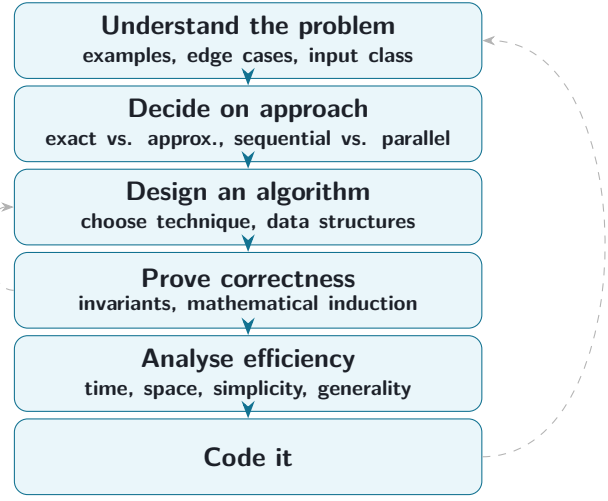
Algorithm	$n = 10^4$	$n = 10^6$
$\mathcal{O}(n^2)$	1.0 s	2.8 hr
$\mathcal{O}(n \log n)$	0.001 s	0.10 s
$\mathcal{O}(n)$	0.0001s	0.01 s

(Assuming 10^8 ops/sec)

Section 6

Algorithm Design

The Algorithm Design and Analysis Process

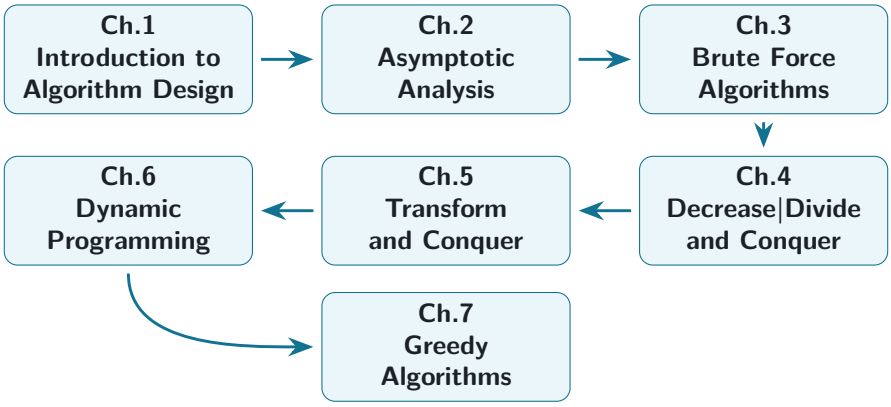


Algorithm Design Techniques

Techniques covered in this course	
Brute Force	try all possibilities
Decrease & Conquer	reduce to smaller instance
Divide & Conquer	split into equal halves
Transform & Conquer	change representation
Dynamic Programming	overlapping sub-problems
Greedy Algorithms	locally optimal choices

Important Problem Types	
Sorting	reorder a list
Searching	find an element
String processing	text, patterns
Graph problems	paths, flows, trees
Combinatorial	permutations, subsets
Geometric	points, lines, hulls
Numerical	equations, integrals

Course Roadmap



Chapter 1 Summary

Key concepts:

- An **algorithm** is a finite, unambiguous, effective sequence of steps producing a correct output
- The same problem can have *many* algorithms with very different properties
- **Correctness** requires proof — counter-examples are your best diagnostic tool
- **Efficiency** is measured in a machine-independent way using asymptotic notation

Three take-aways

1. Choose algorithms — don't just code the first idea
2. A faster algorithm on a slower machine beats a slower algorithm on a faster machine

Next chapter

Chapter 2: Asymptotic analysis, Big-O, Big-Θ, Big-Ω, recurrences, and running-time calculation