https://github.com/zhedahht/CodingInterviewChinese2/tree/master/06_PrintListInReversedOrder

测试用例：

- 功能测试（输入的链表有多个节点；输入的链表只有一个节点）。
- 特殊输入测试（输入的链表头节点指针为 nullptr）。

本题考点：

- 考查应聘者对单向链表的理解和编程能力。
- 考查应聘者对循环、递归和栈 3 个相互关联的概念的理解。

2.3.4 树

树是一种在实际编程中经常遇到的数据结构。它的逻辑很简单：除根节点之外每个节点只有一个父节点，根节点没有父节点；除叶节点之外所有节点都有一个或多个子节点，叶节点没有子节点。父节点和子节点之间用指针链接。由于树的操作会涉及大量的指针，因此与树有关的面试题都不太容易。当面试官想考查应聘者在有复杂指针操作的情况下写代码的能力时，他往往会想到用与树有关的面试题。

面试的时候提到的树，大部分是二叉树。所谓二叉树是树的一种特殊结构，在二叉树中每个节点最多只能有两个子节点。在二叉树中最重要的操作莫过于遍历，即按照某一顺序访问树中的所有节点。通常树有如下几种遍历方式。

- 前序遍历：先访问根节点，再访问左子节点，最后访问右子节点。图 2.5 中的二叉树的前序遍历的顺序是 10、6、4、8、14、12、16。
- 中序遍历：先访问左子节点，再访问根节点，最后访问右子节点。图 2.5 中的二叉树的中序遍历的顺序是 4、6、8、10、12、14、16。
- 后序遍历：先访问左子节点，再访问右子节点，最后访问根节点。图 2.5 中的二叉树的后序遍历的顺序是 4、8、6、12、16、14、10。

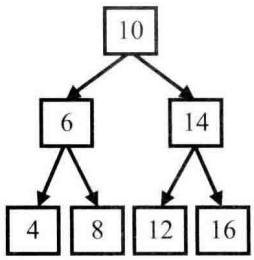

图 2.5　一个二叉树的例子

这 3 种遍历都有递归和循环两种不同的实现方法，每种遍历的递归实现都比循环实现要简洁很多。很多面试官喜欢直接或间接考查遍历（详见面试题 26“树的子结构”、面试题 34“二叉树中和为某一值的路径”、面试题 55“二叉树的深度”）的具体代码实现，面试题 7“重建二叉树”、面试题 33“二叉搜索树的后序遍历序列”也是考查对遍历特点的理解，因此应聘者应该对这 3 种遍历的 6 种实现方法都了如指掌。

- 宽度优先遍历：先访问树的第一层节点，再访问树的第二层节点……一直到访问到最下面一层节点。在同一层节点中，以从左到右的顺序依次访问。我们可以对包括二叉树在内的所有树进行宽度优先遍历。图 2.5 中的二叉树的宽度优先遍历的顺序是 10、6、14、4、8、12、16。

面试题 32“从上到下打印二叉树”就是考查宽度优先遍历算法的题目。

二叉树有很多特例，二叉搜索树就是其中之一。在二叉搜索树中，左子节点总是小于或等于根节点，而右子节点总是大于或等于根节点。图 2.5 中的二叉树就是一棵二叉搜索树。我们可以平均在 $O(\log n)$的时间内根据数值在二叉搜索树中找到一个节点。二叉搜索树的面试题有很多，如面试题 36“二叉搜索树与双向链表”、面试题 68“树中两个节点的最低公共祖先”。

二叉树的另外两个特例是堆和红黑树。堆分为最大堆和最小堆。在最大堆中根节点的值最大，在最小堆中根节点的值最小。有很多需要快速找到最大值或者最小值的问题都可以用堆来解决。红黑树是把树中的节点定义为红、黑两种颜色，并通过规则确保从根节点到叶节点的最长路径的长度不超过最短路径的两倍。在 C++的 STL 中，set、multiset、map、multimap 等数据结构都是基于红黑树实现的。与堆和红黑树相关的面试题，请参考面试题 40“最小的 k 个数”。

面试题 7：重建二叉树

题目：输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如，输入前序遍历序列{1, 2, 4, 7, 3, 5, 6, 8}和中序遍历序列{4, 7, 2, 1, 5, 3, 8, 6}，则重建如图 2.6 所示的二叉树并输出它的头节点。二叉树节点的定义如下：

```
struct BinaryTreeNode
{
    int                    m_nValue;
    BinaryTreeNode*        m_pLeft;
    BinaryTreeNode*        m_pRight;
};
```

在二叉树的前序遍历序列中，第一个数字总是树的根节点的值。但在中序遍历序列中，根节点的值在序列的中间，左子树的节点的值位于根节点的值的左边，而右子树的节点的值位于根节点的值的右边。因此我们需要扫描中序遍历序列，才能找到根节点的值。

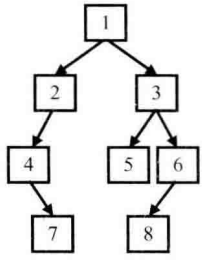

图 2.6　根据前序遍历序列{1, 2, 4, 7, 3, 5, 6, 8}和中序遍历序列{4, 7, 2, 1, 5, 3, 8, 6}重建的二叉树

如图 2.7 所示，前序遍历序列的第一个数字 1 就是根节点的值。扫描中序遍历序列，就能确定根节点的值的位置。根据中序遍历的特点，在根节点的值 1 前面的 3 个数字都是左子树节点的值，位于 1 后面的数字都是右子树节点的值。

由于在中序遍历序列中，有 3 个数字是左子树节点的值，因此左子树共有 3 个左子节点。同样，在前序遍历序列中，根节点后面的 3 个数字就是 3 个左子树节点的值，再后面的所有数字都是右子树节点的值。这样我们就在前序遍历和中序遍历两个序列中分别找到了左、右子树对应的子序列。

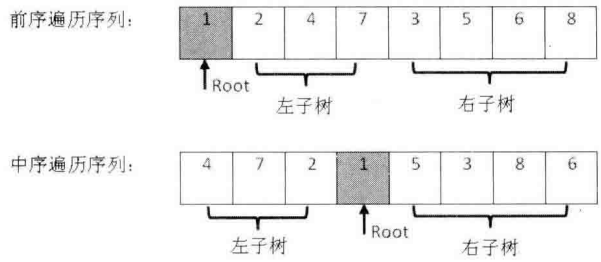

图 2.7　在二叉树的前序遍历和中序遍历序列中确定根节点的值、左子树节点的值和右子树节点的值

既然我们已经分别找到了左、右子树的前序遍历序列和中序遍历序列，我们可以用同样的方法分别构建左、右子树。也就是说，接下来的事情可以用递归的方法去完成。

在想清楚如何在前序遍历和中序遍历序列中确定左、右子树的子序列之后，我们可以写出如下的递归代码：

```
BinaryTreeNode* Construct(int* preorder, int* inorder, int length)
{
    if(preorder == nullptr || inorder == nullptr || length <= 0)
        return nullptr;

    return ConstructCore(preorder, preorder + length - 1,
        inorder, inorder + length - 1);
}

BinaryTreeNode* ConstructCore
(
    int* startPreorder, int* endPreorder,
    int* startInorder, int* endInorder
)
{
    // 前序遍历序列的第一个数字是根节点的值
    int rootValue = startPreorder[0];
    BinaryTreeNode* root = new BinaryTreeNode();
    root->m_nValue = rootValue;
    root->m_pLeft = root->m_pRight = nullptr;

    if(startPreorder == endPreorder)
    {
        if(startInorder == endInorder
            && *startPreorder == *startInorder)
            return root;
        else
            throw std::exception("Invalid input.");
    }
```

```
    // 在中序遍历序列中找到根节点的值
    int* rootInorder = startInorder;
    while(rootInorder <= endInorder && *rootInorder != rootValue)
        ++ rootInorder;

    if(rootInorder == endInorder && *rootInorder != rootValue)
        throw std::exception("Invalid input.");

    int leftLength = rootInorder - startInorder;
    int* leftPreorderEnd = startPreorder + leftLength;
    if(leftLength > 0)
    {
        // 构建左子树
        root->m_pLeft = ConstructCore(startPreorder + 1,
            leftPreorderEnd, startInorder, rootInorder - 1);
    }
    if(leftLength < endPreorder - startPreorder)
    {
        // 构建右子树
        root->m_pRight = ConstructCore(leftPreorderEnd + 1,
            endPreorder, rootInorder + 1, endInorder);
    }

    return root;
}
```

在函数 ConstructCore 中，我们先根据前序遍历序列的第一个数字创建根节点，接下来在中序遍历序列中找到根节点的位置，这样就能确定左、右子树节点的数量。在前序遍历和中序遍历序列中划分了左、右子树节点的值之后，我们就可以递归地调用函数 ConstructCore 去分别构建它的左、右子树。

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/07_ConstructBinaryTree

测试用例：

- 普通二叉树（完全二叉树；不完全二叉树）。
- 特殊二叉树（所有节点都没有右子节点的二叉树；所有节点都没有左子节点的二叉树；只有一个节点的二叉树）。

- 特殊输入测试（二叉树的根节点指针为 nullptr；输入的前序遍历序列和中序遍历序列不匹配）。

本题考点：

- 考查应聘者对二叉树的前序遍历和中序遍历的理解程度。只有对二叉树的不同遍历算法有了深刻的理解，应聘者才有可能在遍历序列中划分出左、右子树对应的子序列。
- 考查应聘者分析复杂问题的能力。我们把构建二叉树的大问题分解成构建左、右子树的两个小问题。我们发现小问题和大问题在本质上是一致的，因此可以用递归的方式解决。更多关于分解复杂问题的讨论，请参考本书的 4.4 节。

面试题 8：二叉树的下一个节点

题目：给定一棵二叉树和其中的一个节点，如何找出中序遍历序列的下一个节点？树中的节点除了有两个分别指向左、右子节点的指针，还有一个指向父节点的指针。

在图 2.8 中的二叉树的中序遍历序列是{d, b, h, e, i, a, f, c, g}。我们将以这棵树为例来分析如何找出二叉树的下一个节点。

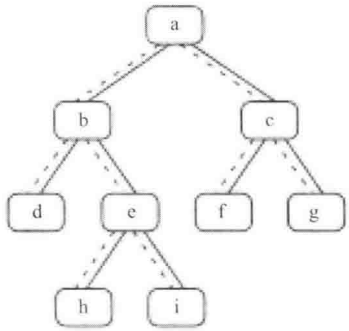

图 2.8　一棵有 9 个节点的二叉树。树中从父节点指向子节点的指针用实线表示，从子节点指向父节点的指针用虚线表示

如果一个节点有右子树，那么它的下一个节点就是它的右子树中的最左子节点。也就是说，从右子节点出发一直沿着指向左子节点的指针，我

们就能找到它的下一个节点。例如，图 2.8 中节点 b 的下一个节点是 h，节点 a 的下一个节点是 f。

接着我们分析一个节点没有右子树的情形。如果节点是它父节点的左子节点，那么它的下一个节点就是它的父节点。例如，图 2.8 中节点 d 的下一个节点是 b，节点 f 的下一个节点是 c。

如果一个节点既没有右子树，并且它还是它父节点的右子节点，那么这种情形就比较复杂。我们可以沿着指向父节点的指针一直向上遍历，直到找到一个是它父节点的左子节点的节点。如果这样的节点存在，那么这个节点的父节点就是我们要找的下一个节点。

为了找到图 2.8 中节点 i 的下一个节点，我们沿着指向父节点的指针向上遍历，先到达节点 e。由于节点 e 是它父节点 b 的右节点，我们继续向上遍历到达节点 b。节点 b 是它父节点 a 的左子节点，因此节点 b 的父节点 a 就是节点 i 的下一个节点。

找出节点 g 的下一个节点的步骤类似。我们先沿着指向父节点的指针到达节点 c。由于节点 c 是它父节点 a 的右子节点，我们继续向上遍历到达节点 a。由于节点 a 是树的根节点，它没有父节点，因此节点 g 没有下一个节点。

我们用如下的 C++代码从二叉树中找出一个节点的下一个节点：

```
BinaryTreeNode* GetNext(BinaryTreeNode* pNode)
{
    if(pNode == nullptr)
        return nullptr;

    BinaryTreeNode* pNext = nullptr;
    if(pNode->m_pRight != nullptr)
    {
        BinaryTreeNode* pRight = pNode->m_pRight;
        while(pRight->m_pLeft != nullptr)
            pRight = pRight->m_pLeft;

        pNext = pRight;
    }
    else if(pNode->m_pParent != nullptr)
    {
        BinaryTreeNode* pCurrent = pNode;
        BinaryTreeNode* pParent = pNode->m_pParent;
        while(pParent != nullptr && pCurrent == pParent->m_pRight)
        {
            pCurrent = pParent;
```

```
            pParent = pParent->m_pParent;
        }

        pNext = pParent;
    }

    return pNext;
}
```

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/08_NextNodeInBinaryTrees

测试用例：

- 普通二叉树（完全二叉树；不完全二叉树）。
- 特殊二叉树（所有节点都没有右子节点的二叉树；所有节点都没有左子节点的二叉树；只有一个节点的二叉树；二叉树的根节点指针为 nullptr）。
- 不同位置的节点的下一个节点（下一个节点为当前节点的右子节点、右子树的最左子节点、父节点、跨层的父节点等；当前节点没有下一个节点）。

本题考点：

- 考查应聘者对二叉树中序遍历的理解程度。只有对二叉树的遍历算法有了深刻的理解，应聘者才有可能准确找出每个节点的中序遍历的下一个节点。
- 考查应聘者分析复杂问题的能力。应聘者只有画出二叉树的结构图、通过具体的例子找出中序遍历下一个节点的规律，才有可能设计出可行的算法。关于画图和举例解决复杂问题的讨论，请参考本书的 4.2 和 4.3 节。

2.3.5 栈和队列

栈是一个非常常见的数据结构，它在计算机领域被广泛应用，比如操作系统会给每个线程创建一个栈用来存储函数调用时各个函数的参数、返回地址及临时变量等。栈的特点是后进先出，即最后被压入（push）栈的元素会第一个被弹出（pop）。在面试题 31“栈的压入、弹出序列”中，我们再详细分析进栈和出栈序列的特点。

通常栈是一个不考虑排序的数据结构，我们需要 $O(n)$时间才能找到栈中最大或者最小的元素。如果想要在 $O(1)$时间内得到栈的最大值或者最小值，则需要对栈做特殊的设计，详见面试题 30“包含 min 函数的栈”。

队列是另外一种很重要的数据结构。和栈不同的是，队列的特点是先进先出，即第一个进入队列的元素将会第一个出来。在 2.3.4 节介绍的树的宽度优先遍历算法中，我们在遍历某一层树的节点时，把节点的子节点放到一个队列里，以备下一层节点的遍历。详细的代码参见面试题 32“从上到下打印二叉树”。

栈和队列虽然是特点针锋相对的两个数据结构，但有意思的是它们却相互联系。请看面试题 9“用两个栈实现队列”，同时读者也可以考虑如何用两个队列实现栈。

面试题 9：用两个栈实现队列

题目：用两个栈实现一个队列。队列的声明如下，请实现它的两个函数 appendTail 和 deleteHead，分别完成在队列尾部插入节点和在队列头部删除节点的功能。

```
template <typename T> class CQueue
{
public:
    CQueue(void);
    ~CQueue(void);

    void appendTail(const T& node);
    T deleteHead();

private:
    stack<T> stack1;
    stack<T> stack2;
};
```

从上述队列的声明中可以看出，一个队列包含了两个栈 stack1 和 stack2，因此这道题的意图是要求我们操作这两个“先进后出”的栈实现一个“先进先出”的队列 CQueue。

我们通过一个具体的例子来分析往该队列插入和删除元素的过程。首先插入一个元素 a，不妨先把它插入 stack1，此时 stack1 中的元素有{a}，stack2 为空。再压入两个元素 b 和 c，还是插入 stack1，此时 stack1 中的元素有{a, b, c}，其中 c 位于栈顶，而 stack2 仍然是空的，如图 2.9（a）所示。

这时候我们试着从队列中删除一个元素。按照队列先入先出的规则，由于 a 比 b、c 先插入队列中，最先被删除的元素应该是 a。元素 a 存储在 stack1 中，但并不在栈顶上，因此不能直接进行删除。注意到 stack2 一直没有被使用过，现在是让 stack2 发挥作用的时候了。如果我们把 stack1 中的元素逐个弹出并压入 stack2，则元素在 stack2 中的顺序正好和原来在 stack1 中的顺序相反。因此经过 3 次弹出 stack1 和压入 stack2 的操作之后，stack1 为空，而 stack2 中的元素是{c,b,a}，这时候就可以弹出 stack2 的栈顶 a 了。此时的 stack1 为空，而 stack2 的元素为{c,b}，其中 b 在栈顶，如图 2.9（b）所示。

如果我们还想继续删除队列的头部应该怎么办呢？剩下的两个元素是 b 和 c，b 比 c 早进入队列，因此 b 应该先删除。而此时 b 恰好又在栈顶上，因此直接弹出 stack2 的栈顶即可。在这次弹出操作之后，stack1 仍然为空，而 stack2 中的元素为{c}，如图 2.9（c）所示。

从上面的分析中我们可以总结出删除一个元素的步骤：当 stack2 不为空时，在 stack2 中的栈顶元素是最先进入队列的元素，可以弹出。当 stack2 为空时，我们把 stack1 中的元素逐个弹出并压入 stack2。由于先进入队列的元素被压到 stack1 的底端，经过弹出和压入操作之后就处于 stack2 的顶端，又可以直接弹出。

接下来再插入一个元素 d。我们还是把它压入 stack1，如图 2.9（d）所示，这样会不会有问题呢？我们考虑下一次删除队列的头部 stack2 不为空，直接弹出它的栈顶元素 c，如图 2.9（e）所示。而 c 的确比 d 先进入队列，应该在 d 之前从队列中删除，因此不会出现任何矛盾。

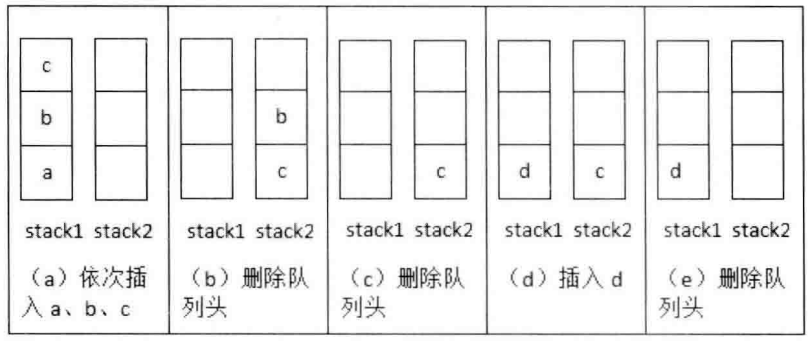

图 2.9　用两个栈模拟一个队列的操作

总结完每一次在队列中插入和删除操作的过程之后，我们就可以开始动手写代码了。参考代码如下：

```
template<typename T> void CQueue<T>::appendTail(const T& element)
{
    stack1.push(element);
}

template<typename T> T CQueue<T>::deleteHead()
{
    if(stack2.size()<= 0)
    {
        while(stack1.size()>0)
        {
            T& data = stack1.top();
            stack1.pop();
            stack2.push(data);
        }
    }

    if(stack2.size() == 0)
        throw new exception("queue is empty");

    T head = stack2.top();
    stack2.pop();

    return head;
}
```

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/09_QueueWithTwoStacks

测试用例：

- 往空的队列里添加、删除元素。
- 往非空的队列里添加、删除元素。
- 连续删除元素直至队列为空。

本题考点：

- 考查应聘者对栈和队列的理解。
- 考查应聘者写与模板相关的代码的能力。
- 考查应聘者分析复杂问题的能力。本题解法的代码虽然只有二十几行，但形成正确的思路却不容易。应聘者能否通过具体的例子分析问题，通过画图的手段把抽象的问题形象化，从而解决这个相对复杂的问题，是能否顺利通过面试的关键。

相关题目：

用两个队列实现一个栈。

我们通过一系列栈的压入和弹出操作来分析用两个队列模拟一个栈的过程。如图 2.10（a）所示，我们先往栈内压入一个元素 a。由于两个队列现在都是空的，我们可以选择把 a 插入两个队列的任意一个。我们不妨把 a 插入 queue1。接下来继续往栈内压入 b、c 两个元素，我们把它们都插入 queue1。这个时候 queue1 包含 3 个元素 a、b 和 c，其中 a 位于队列的头部，c 位于队列的尾部。

现在我们考虑从栈内弹出一个元素。根据栈的后入先出原则，最后被压入栈的 c 应该最先被弹出。由于 c 位于 queue1 的尾部，而我们每次只能从队列的头部删除元素，因此我们可以先从 queue1 中依次删除元素 a、b 并插入 queue2，再从 queue1 中删除元素 c。这就相当于从栈中弹出元素 c 了，如图 2.10（b）所示。我们可以用同样的方法从栈内弹出元素 b，如图 2.10（c）所示。

接下来我们考虑往栈内压入一个元素 d。此时 queue1 已经有一个元素，我们就把 d 插入 queue1 的尾部，如图 2.10（d）所示。如果我们再从栈内弹出一个元素，那么此时被弹出的应该是最后被压入的 d。由于 d 位于 queue1 的尾部，我们只能先从头删除 queue1 的元素并插入 queue2，直到在 queue1 中遇到 d 再直接把它删除，如图 2.10（e）所示。

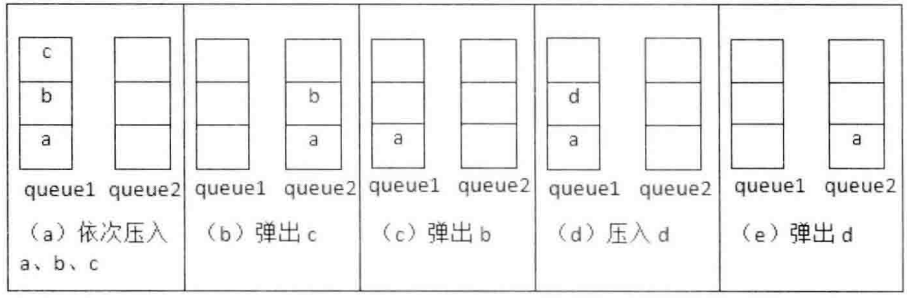

图 2.10　用两个队列模拟一个栈的操作

2.4 算法和数据操作

和数据结构一样，考查算法的面试题也备受面试官的青睐。有很多算法都可以用递归和循环两种不同的方式实现。通常基于递归的实现方法代码会比较简洁，但性能不如基于循环的实现方法。在面试的时候，我们可以根据题目的特点，甚至可以和面试官讨论选择合适的方法编程。

通常排序和查找是面试时考查算法的重点。在准备面试的时候，我们应该重点掌握二分查找、归并排序和快速排序，做到能随时正确、完整地写出它们的代码。

如果面试题要求在二维数组（可能具体表现为迷宫或者棋盘等）上搜索路径，那么我们可以尝试用回溯法。通常回溯法很适合用递归的代码实现。只有当面试官限定不可以用递归实现的时候，我们再考虑用栈来模拟递归的过程。

如果面试题是求某个问题的最优解，并且该问题可以分为多个子问题，那么我们可以尝试用动态规划。在用自上而下的递归思路去分析动态规划问题的时候，我们会发现子问题之间存在重叠的更小的子问题。为了避免

不必要的重复计算，我们用自下而上的循环代码来实现，也就是把子问题的最优解先算出来并用数组（一般是一维或者二维数组）保存下来，接下来基于子问题的解计算大问题的解。

如果我们告诉面试官动态规划的思路之后，面试官还在提醒说在分解子问题的时候是不是存在某个特殊的选择，如果采用这个特殊的选择将一定能得到最优解，那么，通常面试官这样的提示意味着该面试题可能适用于贪婪算法。当然，面试官也会要求应聘者证明贪婪选择的确最终能够得到最优解。

位运算可以看成一类特殊的算法，它是把数字表示成二进制之后对 0 和 1 的操作。由于位运算的对象为二进制数字，所以不是很直观，但掌握它也不难，因为总共只有与、或、异或、左移和右移 5 种位运算。

2.4.1　递归和循环

如果我们需要重复地多次计算相同的问题，则通常可以选择用递归或者循环两种不同的方法。递归是在一个函数的内部调用这个函数自身。而循环则是通过设置计算的初始值及终止条件，在一个范围内重复运算。比如求 1+2+…+n，我们可以用递归或者循环两种方式求出结果。对应的代码如下：

```
int AddFrom1ToN_Recursive(int n)
{
    return n <= 0 ? 0 : n + AddFrom1ToN_Recursive(n - 1);
}

int AddFrom1ToN_Iterative(int n)
{
    int result = 0;
    for(int i = 1; i <= n; ++ i)
        result += i;

    return result;
}
```

通常递归的代码会比较简洁。在上面的例子中，递归的代码只有一条语句，而循环则需要 4 条语句。在树的前序、中序、后序遍历算法的代码中，递归的实现明显要比循环简单得多。在面试的时候，如果面试官没有特别的要求，则应聘者可以尽量多采用递归的方法编程。

面试小提示：

通常基于递归实现的代码比基于循环实现的代码要简洁很多，更加容易实现。如果面试官没有特殊要求，则应聘者可以优先采用递归的方法编程。

递归虽然有简洁的优点，但它同时也有显著的缺点。递归由于是函数调用自身，而函数调用是有时间和空间的消耗的：每一次函数调用，都需要在内存栈中分配空间以保存参数、返回地址及临时变量，而且往栈里压入数据和弹出数据都需要时间。这就不难理解上述的例子中递归实现的效率不如循环。

另外，递归中有可能很多计算都是重复的，从而对性能带来很大的负面影响。递归的本质是把一个问题分解成两个或者多个小问题。如果多个小问题存在相互重叠的部分，就存在重复的计算。在面试题 10“斐波那契数列”及面试题 60“n 个骰子的点数”中，我们将详细地分析递归和循环的性能区别。

通常应用动态规划解决问题时我们都是用递归的思路分析问题，但由于递归分解的子问题中存在大量的重复，因此我们总是用自下而上的循环来实现代码。我们将在面试题 14“剪绳子”、面试题 47“礼物的最大价值”及面试题 48“最长不含重复字符的子字符串”中详细讨论如何用递归分析问题并基于循环写代码。

除效率之外，递归还有可能引起更严重的问题：调用栈溢出。在前面的分析中提到需要为每一次函数调用在内存栈中分配空间，而每个进程的栈的容量是有限的。当递归调用的层级太多时，就会超出栈的容量，从而导致调用栈溢出。在上述例子中，如果输入的参数比较小，如 10，则它们都能返回结果 55。但如果输入的参数很大，如 5000，那么递归代码在运行的时候就会出错，但运行循环的代码能得到正确的结果 12502500。

面试题 10：斐波那契数列

题目一：求斐波那契数列的第 n 项。

写一个函数，输入 n，求斐波那契（Fibonacci）数列的第 n 项。斐波那契数列的定义如下：

$$f(n)=\begin{cases}0 & n=0\\ 1 & n=1\\ f(n-1)+f(n-2) & n>1\end{cases}$$

❖ 效率很低的解法，挑剔的面试官不会喜欢

很多 C 语言教科书在讲述递归函数的时候，都会用斐波那契数列作为例子，因此很多应聘者对这道题的递归解法都很熟悉。他们看到这道题的时候心中会忍不住一阵窃喜，因为他们能很快写出如下代码：

```
long long Fibonacci(unsigned int n)
{
    if(n <= 0)
        return 0;

    if(n == 1)
        return 1;

    return Fibonacci(n - 1) + Fibonacci(n - 2);
}
```

我们的教科书上反复用这个问题来讲解递归函数，并不能说明递归的解法最适合这道题目。面试官会提示我们上述递归的解法有很严重的效率问题并要求我们分析原因。

我们以求解 $f(10)$为例来分析递归的求解过程。想求得 $f(10)$，需要先求得 $f(9)$和 $f(8)$。同样，想求得 $f(9)$，需要先求得 $f(8)$和 $f(7)$……我们可以用树形结构来表示这种依赖关系，如图 2.11 所示。

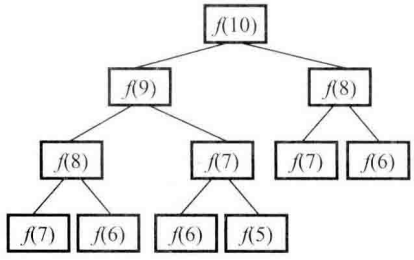

图 2.11　基于递归求斐波那契数列的第 10 项的调用过程

我们不难发现，在这棵树中有很多节点是重复的，而且重复的节点数会随着 n 的增大而急剧增加，这意味着计算量会随着 n 的增大而急剧增大。事实上，用递归方法计算的时间复杂度是以 n 的指数的方式递增的。读者不妨求斐波那契数列的第 100 项试试，感受一下这样递归会慢到什么程度。

❖ 面试官期待的实用解法

其实改进的方法并不复杂。上述递归代码之所以慢，是因为重复的计算太多，我们只要想办法避免重复计算就行了。比如我们可以把已经得到的数列中间项保存起来，在下次需要计算的时候我们先查找一下，如果前面已经计算过就不用再重复计算了。

更简单的办法是从下往上计算，首先根据 $f(0)$和 $f(1)$算出 $f(2)$，再根据 $f(1)$和 $f(2)$算出 $f(3)$……以此类推就可以算出第 n 项了。很容易理解，这种思路的时间复杂度是 $O(n)$。实现代码如下：

```
long long Fibonacci(unsigned n)
{
    int result[2] = {0, 1};
    if(n < 2)
        return result[n];

    long long   fibNMinusOne = 1;
    long long   fibNMinusTwo = 0;
    long long   fibN = 0;
    for(unsigned int i = 2; i <= n; ++ i)
    {
        fibN = fibNMinusOne + fibNMinusTwo;

        fibNMinusTwo = fibNMinusOne;
        fibNMinusOne = fibN;
    }

    return fibN;
}
```

❖ 时间复杂度 $O(\log n)$但不够实用的解法

通常面试到这里也就差不多了，尽管我们还有比这更快的 $O(\log n)$算法。由于这种算法需要用到一个很生僻的数学公式，因此很少有面试官会要求我们掌握。不过以防不时之需，我们还是简要介绍一下这种算法。

在介绍这种方法之前，我们先介绍一个数学公式：

$$\begin{bmatrix} f(n) & f(n-1) \\ f(n-1) & f(n-2) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$$

这个公式用数学归纳法不难证明，感兴趣的读者不妨自己证明一下。有了这个公式，我们只需要求得矩阵 $\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1}$ 即可得到 $f(n)$。现在的问题转

换为如何求矩阵$\begin{bmatrix}1 & 1\\ 1 & 0\end{bmatrix}$的乘方。如果只是简单地从 0 开始循环，$n$ 次方需要 n 次运算，则其时间复杂度仍然是 $O(n)$，并不比前面的方法快。但我们可以考虑乘方的如下性质：

$$a^n=\begin{cases}a^{n/2}\cdot a^{n/2} & n\text{ 为偶数}\\ a^{(n-1)/2}\cdot a^{(n-1)/2}\cdot a & n\text{ 为奇数}\end{cases}$$

从上面的公式中我们可以看出，我们想求得 n 次方，就要先求得 $n/2$ 次方，再把 $n/2$ 次方的结果平方一下即可。这可以用递归的思路实现。

由于很少有面试官要求编程实现这种思路，本书中不再列出完整的代码，感兴趣的读者请参考附带的源代码。不过这种基于递归用 $O(\log n)$的时间求得 n 次方的算法却值得我们重视。我们在面试题 16“数值的整数次方”中再详细讨论这种算法。

❖ 解法比较

用不同的方法求解斐波那契数列的时间效率大不相同。第一种基于递归的解法虽然直观但时间效率很低，在实际软件开发中不会用这种方法，也不可能得到面试官的青睐。第二种方法把递归的算法用循环实现，极大地提高了时间效率。第三种方法把求斐波那契数列转换成求矩阵的乘方，是一种很有创意的算法。虽然我们可以用 $O(\log n)$求得矩阵的 n 次方，但由于隐含的时间常数较大，很少会有软件采用这种算法。另外，实现这种解法的代码也很复杂，不太适合面试。因此第三种方法不是一种实用的算法，不过应聘者可以用它来展示自己的知识面。

除了面试官直接要求编程实现斐波那契数列，还有不少面试题可以看成斐波那契数列的应用。

题目二：青蛙跳台阶问题。

一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级台阶。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

首先我们考虑最简单的情况。如果只有 1 级台阶，那显然只有一种跳法。如果有 2 级台阶，那就有两种跳法：一种是分两次跳，每次跳 1 级；另一种就是一次跳 2 级。

接着我们再来讨论一般情况。我们把 n 级台阶时的跳法看成 n 的函数，记为 $f(n)$。当 $n>2$ 时，第一次跳的时候就有两种不同的选择：一是第一次只跳 1 级，此时跳法数目等于后面剩下的 $n-1$ 级台阶的跳法数目，即为 $f(n-1)$；二是第一次跳 2 级，此时跳法数目等于后面剩下的 $n-2$ 级台阶的跳法数目，即为 $f(n-2)$。因此，n 级台阶的不同跳法的总数 $f(n)=f(n-1)+f(n-2)$。分析到这里，我们不难看出这实际上就是斐波那契数列了。

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/10_Fibonacci

测试用例：

- 功能测试（如输入 3、5、10 等）。
- 边界值测试（如输入 0、1、2）。
- 性能测试（输入较大的数字，如 40、50、100 等）。

本题考点：

- 考查应聘者对递归、循环的理解及编码能力。
- 考查应聘者对时间复杂度的分析能力。
- 如果面试官采用的是青蛙跳台阶的问题，那么同时还在考查应聘者的数学建模能力。

本题扩展：

在青蛙跳台阶的问题中，如果把条件改成：一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级……它也可以跳上 n 级，此时该青蛙跳上一个 n 级的台阶总共有多少种跳法？我们用数学归纳法可以证明 $f(n)=2^{n-1}$。

相关题目：

我们可以用 2×1（图 2.12 的左边）的小矩形横着或者竖着去覆盖更大的矩形。请问用 8 个 2×1 的小矩形无重叠地覆盖一个 2×8 的大矩形（图 2.12 的右边），总共有多少种方法？

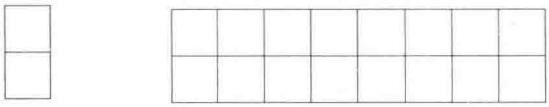

图 2.12　一个 2×1 的矩形和 2×8 的矩形

我们先把 2×8 的覆盖方法记为 $f(8)$。用第一个 2×1 的小矩形去覆盖大矩形的最左边时有两种选择：竖着放或者横着放。当竖着放的时候，右边还剩下 2×7 的区域，这种情形下的覆盖方法记为 $f(7)$。接下来考虑横着放的情况。当 2×1 的小矩形横着放在左上角的时候，左下角必须和横着放一个 2×1 的小矩形，而在右边还剩下 2×6 的区域，这种情形下的覆盖方法记为 $f(6)$，因此 $f(8)=f(7)+f(6)$。此时我们可以看出，这仍然是斐波那契数列。

2.4.2　查找和排序

查找和排序都是在程序设计中经常用到的算法。查找相对而言较为简单，不外乎顺序查找、二分查找、哈希表查找和二叉排序树查找。在面试的时候，不管是用循环还是用递归，面试官都期待应聘者能够信手拈来写出完整正确的二分查找代码，否则可能连继续面试的兴趣都没有。面试题 11“旋转数组的最小数字”和面试题 53“在排序数组中查找数字”都可以用二分查找算法解决。

面试小提示：

如果面试题是要求在排序的数组（或者部分排序的数组）中查找一个数字或者统计某个数字出现的次数，那么我们都可以尝试用二分查找算法。

哈希表和二叉排序树查找的重点在于考查对应的数据结构而不是算法。哈希表最主要的优点是我们利用它能够在 $O(1)$时间内查找某一元素，是效率最高的查找方式；但其缺点是需要额外的空间来实现哈希表。面试题 50“第一个只出现一次的字符”就是用哈希表的特性来实现高效查找的。

与二叉排序树查找算法对应的数据结构是二叉搜索树，我们将在面试题 33“二叉搜索树的后序遍历序列”和面试题 36“二叉搜索树与双向链表”中详细介绍二叉搜索树的特点。

排序比查找要复杂一些。面试官会经常要求应聘者比较插入排序、冒泡排序、归并排序、快速排序等不同算法的优劣。强烈建议应聘者在准备面试的时候，一定要对各种排序算法的特点烂熟于胸，能够从额外空间消耗、平均时间复杂度和最差时间复杂度等方面去比较它们的优缺点。需要特别强调的是，很多公司的面试官喜欢在面试环节要求应聘者写出快速排序的代码。应聘者不妨自己写一个快速排序的函数并用各种数据进行测试。当测试都通过之后，再和经典的实现进行比较，看看有什么区别。

实现快速排序算法的关键在于先在数组中选择一个数字，接下来把数组中的数字分为两部分，比选择的数字小的数字移到数组的左边，比选择的数字大的数字移到数组的右边。这个函数可以如下实现：

```
int Partition(int data[], int length, int start, int end)
{
    if(data == nullptr || length <= 0 || start < 0 || end >= length)
        throw new std::exception("Invalid Parameters");

    int index = RandomInRange(start, end);
    Swap(&data[index], &data[end]);

    int small = start - 1;
    for(index = start; index < end; ++ index)
    {
        if(data[index] < data[end])
        {
            ++ small;
            if(small != index)
                Swap(&data[index], &data[small]);
        }
    }

    ++ small;
    Swap(&data[small], &data[end]);

    return small;
}
```

函数 RandomInRange 用来生成一个在 start 和 end 之间的随机数，函数 Swap 的作用是用来交换两个数字。接下来我们可以用递归的思路分别对每次选中的数字的左右两边排序。下面就是递归实现快速排序的参考代码：

```
void QuickSort(int data[], int length, int start, int end)
{
```

```
    if(start == end)
        return;

    int index = Partition(data, length, start, end);
    if(index > start)
        QuickSort(data, length, start, index - 1);
    if(index < end)
        QuickSort(data, length, index + 1, end);
}
```

对一个长度为 n 的数组排序，只需把 start 设为 0、把 end 设为 n-1，调用函数 QuickSort 即可。

在前面的代码中，函数 Partition 除了可以用在快速排序算法中，还可以用来实现在长度为 n 的数组中查找第 k 大的数字。面试题 39“数组中出现次数超过一半的数字”和面试题 40“最小的 k 个数”都可以用这个函数来解决。

不同的排序算法适用的场合也不尽相同。快速排序虽然总体的平均效率是最好的，但也不是任何时候都是最优的算法。比如数组本身已经排好序了，而每一轮排序的时候都以最后一个数字作为比较的标准，此时快速排序的效率只有 $O(n^2)$。因此，在这种场合快速排序就不是最优的算法。在面试的时候，如果面试官要求实现一个排序算法，那么应聘者一定要问清楚这个排序应用的环境是什么、有哪些约束条件，在得到足够多的信息之后再选择最合适的排序算法。下面来看一个面试的片段。

面试官：请实现一个排序算法，要求时间效率为 $O(n)$。

应聘者：对什么数字进行排序，有多少个数字？

面试官：我们想对公司所有员工的年龄排序。我们公司总共有几万名员工。

应聘者：也就是说数字的大小是在一个较小的范围之内的，对吧？

面试官：嗯，是的。

应聘者：可以使用辅助空间吗？

面试官：看你用多少辅助内存。只允许使用常量大小辅助空间，不得超过 $O(n)$。

在面试的时候应聘者不要怕问面试官问题，只有多提问，应聘者才有可能明了面试官的意图。在上面的例子中，该应聘者通过几个问题就弄清楚了需排序的数字在一个较小的范围内，并且可以用辅助内存。知道了这

些限制条件，就不难写出如下代码了：

```
void SortAges(int ages[], int length)
{
    if(ages == nullptr || length <= 0)
        return;

    const int oldestAge = 99;
    int timesOfAge[oldestAge + 1];

    for(int i = 0; i <= oldestAge; ++ i)
        timesOfAge[i] = 0;

    for(int i = 0; i < length; ++ i)
    {
        int age = ages[i];
        if(age < 0 || age > oldestAge)
            throw new std::exception("age out of range.");

        ++ timesOfAge[age];
    }

    int index = 0;
    for(int i = 0; i <= oldestAge; ++ i)
    {
        for(int j = 0; j < timesOfAge[i]; ++ j)
        {
            ages[index] = i;
            ++ index;
        }
    }
}
```

公司员工的年龄有一个范围。在上面的代码中，允许的范围是 0～99 岁。数组 timesOfAge 用来统计每个年龄出现的次数。某个年龄出现了多少次，就在数组 ages 里设置几次该年龄，这就相当于给数组 ages 排序了。该方法用长度 100 的整数数组作为辅助空间换来了 $O(n)$的时间效率。

面试题 11：旋转数组的最小数字

题目：把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个递增排序的数组的一个旋转，输出旋转数组的最小元素。例如，数组{3, 4, 5, 1, 2}为{1, 2, 3, 4, 5}的一个旋转，该数组的最小值为 1。

这道题最直观的解法并不难，从头到尾遍历数组一次，我们就能找出最小的元素。这种思路的时间复杂度显然是 $O(n)$。但是这种思路没有利用输入的旋转数组的特性，肯定达不到面试官的要求。

我们注意到旋转之后的数组实际上可以划分为两个排序的子数组，而且前面子数组的元素都大于或者等于后面子数组的元素。我们还注意到最小的元素刚好是这两个子数组的分界线。在排序的数组中我们可以用二分查找法实现 $O(\log n)$的查找。本题给出的数组在一定程度上是排序的，因此我们可以试着用二分查找法的思路来寻找这个最小的元素。

和二分查找法一样，我们用两个指针分别指向数组的第一个元素和最后一个元素。按照题目中旋转的规则，第一个元素应该是大于或者等于最后一个元素的（这其实不完全对，还有特例，后面再加以讨论）。

接着我们可以找到数组中间的元素。如果该中间元素位于前面的递增子数组，那么它应该大于或者等于第一个指针指向的元素。此时数组中最小的元素应该位于该中间元素的后面。我们可以把第一个指针指向该中间元素，这样可以缩小寻找的范围。移动之后的第一个指针仍然位于前面的递增子数组。

同样，如果中间元素位于后面的递增子数组，那么它应该小于或者等于第二个指针指向的元素。此时该数组中最小的元素应该位于该中间元素的前面。我们可以把第二个指针指向该中间元素，这样也可以缩小寻找的范围。移动之后的第二个指针仍然位于后面的递增子数组。

不管是移动第一个指针还是第二个指针，查找范围都会缩小到原来的一半。接下来我们再用更新之后的两个指针重复做新一轮的查找。

按照上述思路，第一个指针总是指向前面递增数组的元素，而第二个指针总是指向后面递增数组的元素。最终第一个指针将指向前面子数组的最后一个元素，而第二个指针会指向后面子数组的第一个元素。也就是它们最终会指向两个相邻的元素，而第二个指针指向的刚好是最小的元素。这就是循环结束的条件。

以前面的数组{3, 4, 5, 1, 2}为例，我们先把第一个指针指向第 0 个元素，把第二个指针指向第 4 个元素，如图 2.13（a）所示。位于两个指针中间（在数组中的下标是 2）的数字是 5，它大于第一个指针指向的数字。因此中间数字 5 一定位于第一个递增子数组，并且最小的数字一定位于它的后面。因此我们可以移动第一个指针，让它指向数组的中间，如图 2.13（b）所示。

此时位于这两个指针中间（在数组中的下标是 3）的数字是 1，它小于第二个指针指向的数字。因此这个中间数字 1 一定位于第二个递增字数组，

并且最小的数字一定位于它的前面或者它自己就是最小的数字。因此我们可以移动第二个指针，让它指向两个指针中间的元素，即下标为 3 的元素，如图 2.13（c）所示。

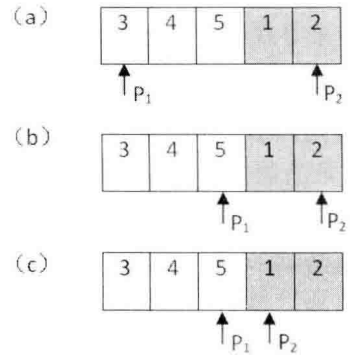

图 2.13　在数组{3, 4, 5, 1, 2}中查找最小值的过程

注：旋转数组中包含两个递增排序的子数组，有阴影的是第二个子数组。（a）把 P_1 指向数组的第一个数字，P_2 指向数组的最后一个数字。由于 P_1 和 P_2 中间的数字 5 大于 P_1 指向的数字，中间的数字在第一个子数组中。下一步把 P_1 指向中间的数字。（b）P_1 和 P_2 中间的数字 1 小于 P_2 指向的数字，中间的数字在第二个子数组中。下一步把 P_2 指向中间的数字。（c）P_1 和 P_2 指向两个相邻的数字，则 P_2 指向的是数组中的最小数字。

此时两个指针的距离是 1，表明第一个指针已经指向第一个递增子数组的末尾，而第二个指针指向第二个递增子数组的开头。第二个子数组的第一个数字就是最小的数字，因此第二个指针指向的数字就是我们查找的结果。

基于这个思路，我们可以写出如下代码：

```
int Min(int* numbers, int length)
{
    if(numbers == nullptr || length <= 0)
        throw new std::exception("Invalid parameters");

    int index1 = 0;
    int index2 = length - 1;
    int indexMid = index1;
    while(numbers[index1] >= numbers[index2])
    {
        if(index2 - index1 == 1)
        {
            indexMid = index2;
```

```
            break;
        }

        indexMid = (index1 + index2) / 2;
        if(numbers[indexMid] >= numbers[index1])
            index1 = indexMid;
        else if(numbers[indexMid] <= numbers[index2])
            index2 = indexMid;
    }

    return numbers[indexMid];
}
```

前面我们提到，在旋转数组中，由于是把递增排序数组前面的若干个数字搬到数组的后面，因此第一个数字总是大于或者等于最后一个数字。但按照定义还有一个特例：如果把排序数组的前面的 0 个元素搬到最后面，即排序数组本身，这仍然是数组的一个旋转，我们的代码需要支持这种情况。此时，数组中的第一个数字就是最小的数字，可以直接返回。这就是在上面的代码中，把 indexMid 初始化为 index1 的原因。一旦发现数组中第一个数字小于最后一个数字，表明该数组是排序的，就可以直接返回第一个数字。

上述代码是否就完美了呢？面试官会告诉我们其实不然。他将提示我们再仔细分析下标为 index1 和 index2（index1 和 index2 分别与图 2.13 中 P_1 和 P_2 相对应）的两个数相同的情况。在前面的代码中，当这两个数相同，并且它们中间的数字（indexMid 指向的数字）也相同时，我们把 indexMid 赋值给 index1，也就是认为此时最小的数字位于中间数字的后面。是不是一定这样？

我们再来看一个例子。数组{1, 0, 1, 1, 1}和数组{1, 1, 1, 0, 1}都可以看成递增排序数组{0, 1, 1, 1, 1}的旋转，图 2.14 分别画出它们由最小数字分隔开的两个子数组。

图 2.14　数组{0, 1, 1, 1, 1}的两个旋转{1, 0, 1, 1, 1}和{1, 1, 1, 0, 1}

注：在这两个数组中，第一个数字、最后一个数字和中间数字都是 1，我们无法确定中间的数字 1 是属于第一个递增子数组还是属于第二个递增子数组。第二个递增子数组用灰色背景表示。

在这两种情况中，第一个指针和第二个指针指向的数字都是 1，并且两个指针中间的数字也是 1，这 3 个数字相同。在第一种情况中，中间数字（下标为 2）位于后面的子数组；在第二种情况中，中间数字（下标为 2）位于前面的子数组。因此，当两个指针指向的数字及它们中间的数字三者相同的时候，我们无法判断中间的数字是位于前面的子数组还是后面的子数组，也就无法移动两个指针来缩小查找的范围。此时，我们不得不采用顺序查找的方法。

在把问题分析清楚形成清晰的思路之后，我们就可以把前面的代码修改为：

```
int Min(int* numbers, int length)
{
    if(numbers == nullptr || length <= 0)
        throw new std::exception("Invalid parameters");

    int index1 = 0;
    int index2 = length - 1;
    int indexMid = index1;
    while(numbers[index1] >= numbers[index2])
    {
        if(index2 - index1 == 1)
        {
            indexMid = index2;
            break;
        }

        indexMid = (index1 + index2) / 2;

        // 如果下标为 index1、index2 和 indexMid 指向的三个数字相等，
        // 则只能顺序查找
        if(numbers[index1] == numbers[index2]
            && numbers[indexMid] == numbers[index1])
            return MinInOrder(numbers, index1, index2);

        if(numbers[indexMid] >= numbers[index1])
            index1 = indexMid;
        else if(numbers[indexMid] <= numbers[index2])
            index2 = indexMid;
    }

    return numbers[indexMid];
}

int MinInOrder(int* numbers, int index1, int index2)
{
    int result = numbers[index1];
    for(int i = index1 + 1; i <= index2; ++i)
    {
        if(result > numbers[i])
```

```
            result = numbers[i];
        }

        return result;
    }
```

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/11_MinNumberInRotatedArray

测试用例：

- 功能测试（输入的数组是升序排序数组的一个旋转，数组中有重复数字或者没有重复数字）。
- 边界值测试（输入的数组是一个升序排序的数组，只包含一个数字的数组）。
- 特殊输入测试（输入 nullptr 指针）。

本题考点：

- 考查应聘者对二分查找的理解。本题变换了二分查找的条件，输入的数组不是排序的，而是排序数组的一个旋转。这要求我们对二分查找的过程有深刻的理解。
- 考查应聘者的沟通能力和学习能力。本题面试官提出了一个新的概念：数组的旋转。我们要在很短的时间内学习、理解这个新概念。在面试过程中，如果面试官提出新的概念，那么我们可以主动和面试官沟通，多问几个问题，把概念弄清楚。
- 考查应聘者思维的全面性。排序数组本身是数组旋转的一个特例。另外，我们要考虑到数组中有相同数字的特例。如果不能很好地处理这些特例，就很难写出让面试官满意的完美代码。

2.4.3 回溯法

回溯法可以看成蛮力法的升级版，它从解决问题每一步的所有可能选项里系统地选择出一个可行的解决方案。回溯法非常适合由多个步骤组成的问题，并且每个步骤都有多个选项。当我们在某一步选择了其中一个选项时，就进入下一步，然后又面临新的选项。我们就这样重复选择，直至到达最终的状态。

用回溯法解决的问题的所有选项可以形象地用树状结构表示。在某一步有 n 个可能的选项，那么该步骤可以看成是树状结构中的一个节点，每个选项看成树中节点连接线，经过这些连接线到达该节点的 n 个子节点。树的叶节点对应着终结状态。如果在叶节点的状态满足题目的约束条件，那么我们找到了一个可行的解决方案。

如果在叶节点的状态不满足约束条件，那么只好回溯到它的上一个节点再尝试其他的选项。如果上一个节点所有可能的选项都已经试过，并且不能到达满足约束条件的终结状态，则再次回溯到上一个节点。如果所有节点的所有选项都已经尝试过仍然不能到达满足约束条件的终结状态，则该问题无解。

接下来以面试题 12 为例，分析用回溯法解决问题的过程。由于矩阵的第一行第二个字母'b'和路径"bfce"的第一个字符相等，我们就从这里开始分析。根据题目的要求，我们此时有 3 个选项，分别是向左到达字母'a'、向右到达字母't'、向下到达字母'f'。我们先尝试选项'a'，由于此时不可能得到路径"bfce"，因此不得不回到节点'b'尝试下一个选项't'。同样，经过节点't'也不可能得到路径"bfce"，因此再次回到节点'b'尝试下一个选项'f'。

在节点'f'我们也有 3 个选项，向左、向右都能到达字母'c'、向下到达字母'd'。我们先选择向左到达字母'c'，此时只有一个选择，即向下到达字母'j'。由于此时的路径为"bfcj"，不满足题目的约束条件，我们只好回到上一个节点左边的节点'c'。注意到左边的节点'c'的所有选项都已经尝试过，因此只好再回溯到上一个节点'f'并尝试它的下一个选项，即向右到达节点'c'。

在右边的节点'c'我们有两个选择，即向右到达节点's'，或者向下到达节点'e'。由于经过节点's'不可能找到满足条件的路径，我们再选择节点'e'，此时路径上的字母刚好组成字符串"bfce"，满足题目的约束条件，因此我们找到了符合要求的解决方案，如图 2.15 所示。

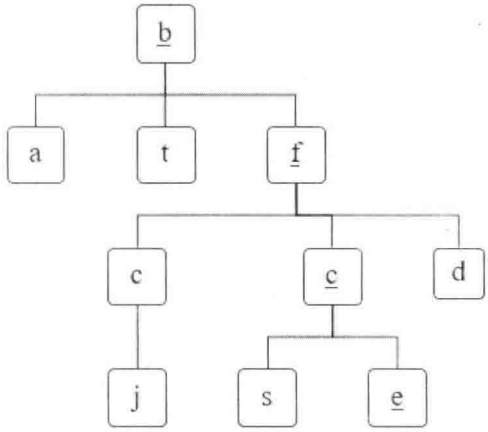

图 2.15　面试题 12 从字母 b 开始的选项组成的树状结构

通常回溯法算法适合用递归实现代码。当我们到达某一个节点时，尝试所有可能的选项并在满足条件的前提下递归地抵达下一个节点。

面试题 12：矩阵中的路径

题目：请设计一个函数，用来判断在一个矩阵中是否存在一条包含某字符串所有字符的路径。路径可以从矩阵中的任意一格开始，每一步可以在矩阵中向左、右、上、下移动一格。如果一条路径经过了矩阵的某一格，那么该路径不能再次进入该格子。例如，在下面的 3×4 的矩阵中包含一条字符串“bfce”的路径（路径中的字母用下画线标出）。但矩阵中不包含字符串“abfb”的路径，因为字符串的第一个字符 b 占据了矩阵中的第一行第二个格子之后，路径不能再次进入这个格子。

a　b　t　g

c　f　c　s

j　d　e　h

这是一个可以用回溯法解决的典型题。首先，在矩阵中任选一个格子作为路径的起点。假设矩阵中某个格子的字符为 ch，并且这个格子将对应于路径上的第 i 个字符。如果路径上的第 i 个字符不是 ch，那么这个格子不可能处在路径上的第 i 个位置。如果路径上的第 i 个字符正好是 ch，那么到相邻的格子寻找路径上的第 i+1 个字符。除矩阵边界上的格子之外，其他格

子都有 4 个相邻的格子。重复这个过程，直到路径上的所有字符都在矩阵中找到相应的位置。

由于回溯法的递归特性，路径可以被看成一个栈。当在矩阵中定位了路径中前 n 个字符的位置之后，在与第 n 个字符对应的格子的周围都没有找到第 $n+1$ 个字符，这时候只好在路径上回到第 $n-1$ 个字符，重新定位第 n 个字符。

由于路径不能重复进入矩阵的格子，所以还需要定义和字符矩阵大小一样的布尔值矩阵，用来标识路径是否已经进入了每个格子。

下面的代码实现了这个回溯算法：

```
bool hasPath(char* matrix, int rows, int cols, char* str)
{
    if(matrix == nullptr || rows < 1 || cols < 1 || str == nullptr)
        return false;

    bool *visited = new bool[rows * cols];
    memset(visited, 0, rows * cols);

    int pathLength = 0;
    for(int row = 0; row < rows; ++row)
    {
        for(int col = 0; col < cols; ++col)
        {
            if(hasPathCore(matrix, rows, cols, row, col, str,
                pathLength, visited))
            {
                return true;
            }
        }
    }

    delete[] visited;

    return false;
}

bool hasPathCore(const char* matrix, int rows, int cols, int row,
    int col, const char* str, int& pathLength, bool* visited)
{
    if(str[pathLength] == '\0')
        return true;

    bool hasPath = false;
    if(row >= 0 && row < rows && col >= 0 && col < cols
            && matrix[row * cols + col] == str[pathLength]
            && !visited[row * cols + col])
    {
        ++pathLength;
```

```
        visited[row * cols + col] = true;

        hasPath = hasPathCore(matrix, rows, cols, row, col - 1,
                    str, pathLength, visited)
                || hasPathCore(matrix, rows, cols, row - 1, col,
                    str, pathLength, visited)
                || hasPathCore(matrix, rows, cols, row, col + 1,
                    str, pathLength, visited)
                || hasPathCore(matrix, rows, cols, row + 1, col,
                    str, pathLength, visited);

        if(!hasPath)
        {
            --pathLength;
            visited[row * cols + col] = false;
        }
    }

    return hasPath;
}
```

当矩阵中坐标为 (row, col)的格子和路径字符串中下标为 pathLength 的字符一样时，从 4 个相邻的格子(row, col−1)、(row−1, col)、(row, col + 1) 和 (row + 1, col) 中去定位路径字符串中下标为 pathLength+1 的字符。

如果 4 个相邻的格子都没有匹配字符串中下标为 pathLength+1 的字符，则表明当前路径字符串中下标为 pathLength 的字符在矩阵中的定位不正确，我们需要回到前一个字符（pathLength−1），然后重新定位。

一直重复这个过程，直到路径字符串上的所有字符都在矩阵中找到合适的位置（此时 str[pathLength] == '\0'）。

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/12_StringPathInMatrix

测试用例：

- 功能测试（在多行多列的矩阵中存在或者不存在路径）。
- 边界值测试（矩阵只有一行或者只有一列；矩阵和路径中的所有字母都是相同的）。
- 特殊输入测试（输入 nullptr 指针）。

本题考点：

- 考查应聘者对回溯法的理解。通常在二维矩阵上找路径这类问题都可以应用回溯法解决。
- 考查应聘者对数组的编程能力。我们一般都把矩阵看成一个二维的数组。只有对数组的特性充分了解，才有可能快速、正确地实现回溯法的代码。

面试题 13：机器人的运动范围

题目：地上有一个 m 行 n 列的方格。一个机器人从坐标(0, 0)的格子开始移动，它每次可以向左、右、上、下移动一格，但不能进入行坐标和列坐标的数位之和大于 k 的格子。例如，当 k 为 18 时，机器人能够进入方格(35, 37)，因为 3+5+3+7=18。但它不能进入方格(35, 38)，因为 3+5+3+8=19。请问该机器人能够到达多少个格子？

和前面的题目类似，这个方格也可以看作一个 $m \times n$ 的矩阵。同样，在这个矩阵中，除边界上的格子之外，其他格子都有 4 个相邻的格子。

机器人从坐标(0, 0)开始移动。当它准备进入坐标为(i, j)的格子时，通过检查坐标的数位和来判断机器人是否能够进入。如果机器人能够进入坐标为(i, j)的格子，则再判断它能否进入 4 个相邻的格子$(i, j-1)$、$(i-1, j)$、$(i, j+1)$和$(i+1, j)$。因此，我们可以用如下的代码来实现回溯算法：

```
int movingCount(int threshold, int rows, int cols)
{
    if(threshold < 0 || rows <= 0 || cols <= 0)
        return 0;

    bool *visited = new bool[rows * cols];
    for(int i = 0; i < rows * cols; ++i)
        visited[i] = false;

    int count = movingCountCore(threshold, rows, cols,
                        0, 0, visited);

    delete[] visited;

    return count;
}

int movingCountCore(int threshold, int rows, int cols, int row,
    int col, bool* visited){
```

```
    int count = 0;
    if(check(threshold, rows, cols, row, col, visited))
    {
        visited[row * cols + col] = true;

        count = 1 + movingCountCore(threshold, rows, cols,
                    row - 1, col, visited)
                + movingCountCore(threshold, rows, cols,
                    row, col - 1, visited)
                + movingCountCore(threshold, rows, cols,
                    row + 1, col, visited)
                + movingCountCore(threshold, rows, cols,
                    row, col + 1, visited);
    }

    return count;
}
```

下面的函数 check 用来判断机器人能否进入坐标为(row, col)的方格，而函数 getDigitSum 用来得到一个数字的数位之和。

```
bool check(int threshold, int rows, int cols, int row, int col,
    bool* visited)
{
    if(row >=0 && row < rows && col >= 0 && col < cols
        && getDigitSum(row) + getDigitSum(col) <= threshold
        && !visited[row* cols + col])
        return true;

    return false;
}

int getDigitSum(int number)
{
    int sum = 0;
    while(number > 0)
    {
        sum += number % 10;
        number /= 10;
    }

    return sum;
}
```

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/13_RobotMove

测试用例：

- 功能测试（方格为多行多列；k 为正数）。
- 边界值测试（方格只有一行或者只有一列；k 等于 0）。
- 特殊输入测试（k 为负数）。

本题考点：

- 考查应聘者对回溯法的理解。通常物体或者人在二维方格运动这类问题都可以用回溯法解决。
- 考查应聘者对数组编程的能力。我们一般都把矩阵看成一个二维的数组。只有对数组的特性充分了解，才有可能快速、正确地实现回溯法的代码编写。

2.4.4 动态规划与贪婪算法

动态规划现在是编程面试中的热门话题。如果面试题是求一个问题的最优解（通常是求最大值或者最小值），而且该问题能够分解成若干个子问题，并且子问题之间还有重叠的更小的子问题，就可以考虑用动态规划来解决这个问题。

我们在应用动态规划之前要分析能否把大问题分解成小问题，分解后的每个小问题也存在最优解。如果把小问题的最优解组合起来能够得到整个问题的最优解，那么我们可以应用动态规划解决这个问题。

例如在面试题 14 中，我们如何把长度为 n 的绳子剪成若干段，使得得到的各段长度的乘积最大。这个问题的目标是求剪出的各段绳子长度的乘积最大值，也就是求一个问题的最优解——这是可以应用动态规划求解的问题的第一个特点。

我们把长度为 n 的绳子剪成若干段后得到的乘积最大值定义为函数 $f(n)$。假设我们把第一刀剪在长度为 i（$0<i<n$）的位置，于是把绳子剪成了长度分别为 i 和 $n-i$ 的两段。我们要想得到整个问题的最优解 $f(n)$，那么要同样用最优化的方法把长度为 i 和 $n-i$ 的两段分别剪成若干段，使得它们各自剪出的每段绳子的长度乘积最大。也就是说整体问题的最优解是依赖各

个子问题的最优解——这是可以应用动态规划求解的问题的第二个特点。

我们把大问题分解成若干个小问题，这些小问题之间还有相互重叠的更小的子问题——这是可以应用动态规划求解的问题的第三个特点。还是以面试题 14 为例，假设绳子最初的长度为 10，我们可以把绳子剪成长度分别为 4 和 6 的两段，也就是 $f(4)$和 $f(6)$都是 $f(10)$的子问题。接下来分别求解这两个子问题。我们可以把长度为 4 的绳子剪成均为 2 的两段，即 $f(2)$是 $f(4)$的子问题。同样，我们也可以把长度为 6 的绳子剪成长度分别为 2 和 4 的两段，即 $f(2)$和 $f(4)$都是 $f(6)$的子问题。我们注意到 $f(2)$是 $f(4)$和 $f(6)$公共的更小的子问题。

由于子问题在分解大问题的过程中重复出现，为了避免重复求解子问题，我们可以用从下往上的顺序先计算小问题的最优解并存储下来，再以此为基础求取大问题的最优解。从上往下分析问题，从下往上求解问题，这是可以应用动态规划求解的问题的第四个特点。在应用动态规划解决问题的时候，我们总是从解决最小问题开始，并把已经解决的子问题的最优解存储下来（大部分面试题都是存储在一维或者二维数组里），并把子问题的最优解组合起来逐步解决大的问题。

在应用动态规划的时候，我们每一步都可能面临若干个选择。在求解面试题 14 时，我们在剪第一刀的时候就有 $n-1$ 个选择。我们可以剪在长度为 $1,2,\cdots,n-1$ 的任意位置。由于我们事先不知道剪在哪个位置是最优的解法，只好把所有的可能都尝试一遍，然后比较得出最优的剪法。如果用数学的语言来表示，这就是 $f(n)=\max(f(i)\times f(n-i))$，其中 $0<i<n$。

贪婪算法和动态规划不一样。当我们应用贪婪算法解决问题的时候，每一步都可以做出一个贪婪的选择，基于这个选择，我们确定能够得到最优解。还是以面试题 14“剪绳子”为例，如果绳子的长度大于 5，则每次都剪出一段长度为 3 的绳子。如果剩下的绳子的长度仍然大于 5，则接着剪出一段长度为 3 的绳子。接下来重复这个步骤，直到剩下的绳子的长度小于 5。剪出一段长度为 3 的绳子，就是我们在每一步做出的贪婪选择。为什么这样的贪婪选择能得到最优解？这是我们应用贪婪算法时都需要问的问题，需要用数学方式来证明贪婪选择是正确的。

面试题 14：剪绳子

题目：给你一根长度为 n 的绳子，请把绳子剪成 m 段（m、n 都是整数，$n>1$ 并且 $m>1$），每段绳子的长度记为 $k[0],k[1],\cdots,k[m]$。请问 $k[0]\times k[1]\times\cdots\times k[m]$可能的最大乘积是多少？例如，当绳子的长度是 8 时，我们把它剪成长度分别为 2、3、3 的三段，此时得到的最大乘积是 18。

我们有两种不同的方法解决这个问题。先用常规的需要 $O(n^2)$时间和 $O(n)$空间的动态规划的思路，接着用只需要 $O(1)$时间和空间的贪婪算法来分析解决这个问题。

❖ 动态规划

首先定义函数 $f(n)$为把长度为 n 的绳子剪成若干段后各段长度乘积的最大值。在剪第一刀的时候，我们有 $n-1$ 种可能的选择，也就是剪出来的第一段绳子的可能长度分别为 $1,2,\cdots,n-1$。因此 $f(n)=\max(f(i)\times f(n-i))$，其中 $0<i<n$。

这是一个从上至下的递归公式。由于递归会有很多重复的子问题，从而有大量不必要的重复计算。一个更好的办法是按照从下而上的顺序计算，也就是说我们先得到 $f(2)$、$f(3)$，再得到 $f(4)$、$f(5)$，直到得到 $f(n)$。

当绳子的长度为 2 时，只可能剪成长度都为 1 的两段，因此 $f(2)$等于 1。当绳子的长度为 3 时，可能把绳子剪成长度分别为 1 和 2 的两段或者长度都为 1 的三段，由于 $1\times2>1\times1\times1$，因此 $f(3)=2$。

下面是这一思路对应的参考代码：

```
int maxProductAfterCutting_solution1(int length)
{
    if(length < 2)
        return 0;
    if(length == 2)
        return 1;
    if(length == 3)
        return 2;

    int* products = new int[length + 1];
    products[0] = 0;
    products[1] = 1;
    products[2] = 2;
    products[3] = 3;

    int max = 0;
```

```
    for(int i = 4; i <= length; ++i)
    {
        max = 0;
        for(int j = 1; j <= i / 2; ++j)
        {
            int product = products[j] * products[i - j];
            if(max < product)
                max = product;

            products[i] = max;
        }
    }

    max = products[length];
    delete[] products;

    return max;
}
```

在上述代码中，子问题的最优解存储在数组 products 里。数组中第 i 个元素表示把长度为 i 的绳子剪成若干段之后各段长度乘积的最大值，即 $f(i)$。我们注意到代码中第一个 for 循环变量 i 是顺序递增的，这意味着计算顺序是自下而上的。因此在求 $f(i)$之前，对于每一个 j（$0<i<j$）而言，$f(j)$都已经求解出来了，并且结果保存在 projects[j]里。为了求解 $f(i)$，我们需要求出所有可能的 $f(j)\times f(i-j)$并比较得出它们的最大值。这就是代码中第二个 for 循环的功能。

❖ 贪婪算法

如果我们按照如下的策略来剪绳子，则得到的各段绳子的长度的乘积将最大：当 n≥5 时，我们尽可能多地剪长度为 3 的绳子；当剩下的绳子长度为 4 时，把绳子剪成两段长度为 2 的绳子。这种思路对应的参考代码如下：

```
int maxProductAfterCutting_solution2(int length)
{
    if(length < 2)
        return 0;
    if(length == 2)
        return 1;
    if(length == 3)
        return 2;

    // 尽可能多地剪去长度为 3 的绳子段
    int timesOf3 = length / 3;

    // 当绳子最后剩下的长度为 4 的时候，不能再剪去长度为 3 的绳子段。
```

```
    // 此时更好的方法是把绳子剪成长度为 2 的两段，因为 2×2 > 3×1
    if(length - timesOf3 * 3 == 1)
        timesOf3 -= 1;

    int timesOf2 = (length - timesOf3 * 3) / 2;

    return (int) (pow(3, timesOf3)) * (int) (pow(2, timesOf2));
}
```

接下来我们证明这种思路的正确性。首先，当 $n \geqslant 5$ 的时候，我们可以证明 $2(n-2)>n$ 并且 $3(n-3)>n$。也就是说，当绳子剩下的长度大于或者等于 5 的时候，我们就把它剪成长度为 3 或者 2 的绳子段。另外，当 $n \geqslant 5$ 时，$3(n-3) \geqslant 2(n-2)$，因此我们应该尽可能地多剪长度为 3 的绳子段。

前面证明的前提是 $n \geqslant 5$。那么当绳子的长度为 4 呢？在长度为 4 的绳子上剪一刀，有两种可能的结果：剪成长度分别为 1 和 3 的两根绳子，或者两根长度都为 2 的绳子。注意到 $2\times2>1\times3$，同时 $2\times2=4$，也就是说，当绳子长度为 4 时其实没有必要剪，只是题目的要求是至少要剪一刀。

源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/14_CuttingRope

测试用例：

- 功能测试（绳子的初始长度大于 5）。
- 边界值测试（绳子的初始长度分别为 0、1、2、3、4）。

本题考点：

- 考查应聘者的抽象建模能力。应聘者需要把一个具体的场景抽象成一个能够用动态规划或者贪婪算法解决的模型。更多关于抽象建模能力的讨论请参考本书 6.4 节。
- 考查应聘者对动态规划和贪婪算法的理解。能够灵活运用动态规划解决问题的关键是具备从上到下分析问题、从下到上解决问题的能力，而灵活运用贪婪算法则需要扎实的数学基本功。