LeetCode 101：和你一起你轻松刷题（C++）

LeetCode 101: A LeetCode Grinding Guide (C++ Version)

作者：高畅 Chang Gao

版本：正式版 1.12

一个面向有 C++ 编程基础，但缺乏刷题经验的读者的教科书和工具书。

序

在 2018 年这个奇妙的秋天，我前往美国卡内基梅隆大学攻读硕士项目。为了准备实习秋招，我从夏天就开始整理 LeetCode 的题目；经过几个月的刷题，我也整理了几百道题，但是缺少系统性的归纳和总结。时隔一年，我于 2019 年秋季在 GitHub 上用 Markdown 做了一个初步的总结，按照算法和数据结构进行分类，整理了差不多 200 道题，用于自己在面试前查漏补缺。然而，在这个简单的总结里，每道题只有简单的题目描述和题解代码，并没有详细的解释说明。除了我之外的其他人很难读懂代码的思路。

有了刷题的积累和不错的运气，我很快就在毕业前找到了工作。当时我的一位朋友对我开玩笑说，你刷了这么多题，却在找到工作后停止了面试，是不是有点亏啊。我笑了笑，心想我并不是会这么做的人；但是的确，刷了这么多题却没有派上太多用场。2019 年冬季毕业后，我宅在家里做着入职前的准备，同时刷着魔兽世界的坐骑成就。不知怎的，我突然萌生了一个念想，既然我刷了这么多题，也有了初步的总结，不如把它们好好地归纳总结一下，做一个便于他人阅读和学习的电子书。Bang! Here comes the book.

本书分为算法和数据结构两大部分，又细分了十五个章节，详细讲解了刷 LeetCode 时常用的技巧。我把题目精简到了 101 道，一是呼应了本书的标题，二是不想让读者阅读和练习时间过长。这么做不太好的一点是，如果只练习这 101 道题，读者可能对算法和数据结构的掌握不够扎实。因此在每一章节的末尾，我都加上了一些推荐的练习题，并给出了一些解法提示，希望读者在理解每一章节后把练习题也完成。如果本书反响热烈，我也会后续加上他们的题解。

本书以 C++ 作为编程语言。对于 Java 用户，绝大部分的算法和数据结构都可以找到对应的写法，语法上也只需要小修改。对于 Python 等其它用户，由于语法差别略大，这本书可能并不会特别适合你。由于本书的目的不是学习 C++ 语言，因此行文时我不会过多解释语法细节，而且会适当使用一些 C++11 或更新的语法。截止于 2019 年年末，所有的书内代码在 LeetCode 上都是可以正常运行的，并且在保持易读的基础上，几乎都是最快或最省空间的解法。

请注意，刷题只是提高面试乃至工作能力的一小部分。在计算机科学的海洋里，值得探索的东西太多，并不建议您花过多时间刷题。并且要成为一个优秀的计算机科学家，刷题只是入职的敲门砖，提高各种专业技能、打好专业基础、以及了解最新的专业方向或许更加重要。

由于本书的目的是分享和教学，因此本书永久免费，也禁止任何营利性利用。欢迎学术目的的分享和传阅。由于我不对 LeetCode 的任何题目拥有版权，一切题目版权以 LeetCode 官方为准。

感谢 GitHub 用户 CyC2018 的 LeetCode 题解，它对于我早期的整理起到了很大的帮助作用。感谢 ElegantBook 提供的精美 LaTeX 模版，使得我可以轻松地把 Markdown 笔记变成看起来更专业的电子书。另外，书的封面图片是我于 2019 年元月，在尼亚加拉大瀑布的加拿大侧拍摄的风景；在此感谢海澄兄同我一起旅行拍照。

重要声明

本书永久免费地址：github.com/changgyhub/leetcode_101

由于本书是作者本人于闲余时间完成的，可能会有不少纰漏，部分题目的解释可能也存在不详细或者不清楚的情况。欢迎在 GitHub 提 issue 指正，我会将您加入鸣谢名单。

鸣谢名单

GitHub 用户：quweikoala、szlghl1、mag1cianag、woodpenker、yangCoder96、cluckl、shaoshuai-luo、KivenGood、alicegong、hitYunhongXu、shengchen1998、jihchi、hapoyige、hitYunhongXu

目　录

1 题目分类 1

2 最易懂的贪心算法 3
2.1 算法解释 . . . 3
2.2 分配问题 . . . 3
2.3 区间问题 . . . 5
2.4 练习 . . . 6

3 玩转双指针 8
3.1 算法解释 . . . 8
3.2 Two Sum . . . 9
3.3 归并两个有序数组 . . . 10
3.4 快慢指针 . . . 10
3.5 滑动窗口 . . . 11
3.6 练习 . . . 13

4 居合斩！二分查找 14
4.1 算法解释 . . . 14
4.2 求开方 . . . 14
4.3 查找区间 . . . 15
4.4 旋转数组查找数字 . . . 17
4.5 练习 . . . 18

5 千奇百怪的排序算法 19
5.1 常用排序算法 . . . 19
5.2 快速选择 . . . 21
5.3 桶排序 . . . 22
5.4 练习 . . . 23

6 一切皆可搜索 24
6.1 算法解释 . . . 24
6.2 深度优先搜索 . . . 24
6.3 回溯法 . . . 29
6.4 广度优先搜索 . . . 34
6.5 练习 . . . 38

7 深入浅出动态规划 40
7.1 算法解释 . . . 40
7.2 基本动态规划：一维 . . . 40
7.3 基本动态规划：二维 . . . 43
7.4 分割类型题 . . . 47
7.5 子序列问题 . . . 49
7.6 背包问题 . . . 51
7.7 字符串编辑 . . . 57
7.8 股票交易 . . . 59
7.9 练习 . . . 62

8 化繁为简的分治法 64
8.1 算法解释 . . . 64
8.2 表达式问题 . . . 64
8.3 练习 . . . 66

9 巧解数学问题 67
9.1 引言 . . . 67
9.2 公倍数与公因数 . . . 67
9.3 质数 . . . 67
9.4 数字处理 . . . 69
9.5 随机与取样 . . . 71
9.6 练习 . . . 74

10 神奇的位运算 76
10.1 常用技巧 . . . 76
10.2 位运算基础问题 . . . 76
10.3 二进制特性 . . . 78
10.4 练习 . . . 80

11 妙用数据结构 81
11.1 C++ STL . . . 81
11.2 数组 . . . 82
11.3 栈和队列 . . . 85
11.4 单调栈 . . . 88
11.5 优先队列 . . . 89
11.6 双端队列 . . . 93
11.7 哈希表 . . . 94
11.8 多重集合和映射 . . . 98
11.9 前缀和与积分图 . . . 99
11.10练习 . . . 102

12 令人头大的字符串 104
12.1 引言 104
12.2 字符串比较 104
12.3 字符串理解 107
12.4 字符串匹配 108
12.5 练习 109

13 指针三剑客之一：链表 110
13.1 数据结构介绍 110
13.2 链表的基本操作 110
13.3 其它链表技巧 113
13.4 练习 114

14 指针三剑客之二：树 116
14.1 数据结构介绍 116
14.2 树的递归 116
14.3 层次遍历 122
14.4 前中后序遍历 123
14.5 二叉查找树 125
14.6 字典树 129
14.7 练习 131

15 指针三剑客之三：图 134
15.1 数据结构介绍 134
15.2 二分图 134
15.3 拓扑排序 136
15.4 练习 137

16 更加复杂的数据结构 138
16.1 引言 138
16.2 并查集 138
16.3 复合数据结构 140
16.4 练习 142

17 后记 143

第 1 章 题目分类

打开 LeetCode 网站，如果我们按照题目类型数量分类，最多的几个题型有数组、动态规划、数学、字符串、树、哈希表、深度优先搜索、二分查找、贪心算法、广度优先搜索、双指针等等。本书将包括上述题型以及网站上绝大多数流行的题型，并且按照难易程度和类型进行分类。

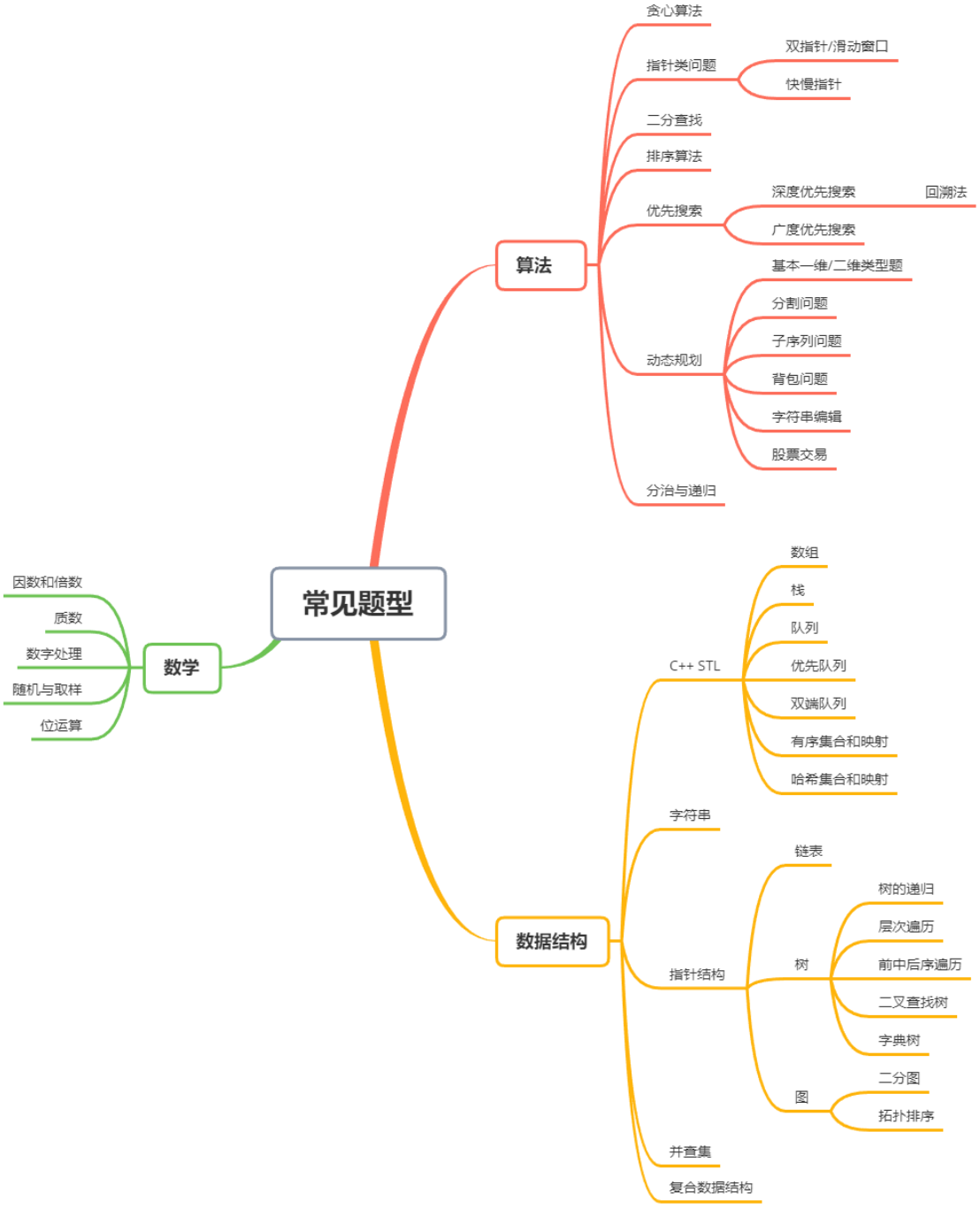

图 1.1: 题型思维导图

第一个大分类是算法。本书先从最简单的贪心算法讲起，然后逐渐进阶到二分查找、排序算法和搜索算法，最后是难度比较高的动态规划和分治算法。

第二个大分类是数学，包括偏向纯数学的数学问题，和偏向计算机知识的位运算问题。这类问题通常用来测试你是否聪敏，在实际工作中并不常用，笔者建议可以优先把精力放在其它大类上。

第三个大分类是数据结构，包括 C++ STL 内包含的常见数据结构、字符串处理、链表、树和图。其中，链表、树、和图都是用指针表示的数据结构，且前者是后者的子集。最后我们也将介绍一些更加复杂的数据结构，比如经典的并查集和 LRU。

第 2 章 最易懂的贪心算法

内容提要

- 算法解释
- 区间问题
- 分配问题

2.1 算法解释

顾名思义，贪心算法或贪心思想采用贪心的策略，保证每次操作都是局部最优的，从而使最后得到的结果是全局最优的。

举一个最简单的例子：小明和小王喜欢吃苹果，小明可以吃五个，小王可以吃三个。已知苹果园里有吃不完的苹果，求小明和小王一共最多吃多少个苹果。在这个例子中，我们可以选用的贪心策略为，每个人吃自己能吃的最多数量的苹果，这在每个人身上都是局部最优的。又因为全局结果是局部结果的简单求和，且局部结果互不相干，因此局部最优的策略也同样是全局最优的策略。

2.2 分配问题

455. Assign Cookies (Easy)

题目描述

有一群孩子和一堆饼干，每个孩子有一个饥饿度，每个饼干都有一个大小。每个孩子只能吃一个饼干，且只有饼干的大小不小于孩子的饥饿度时，这个孩子才能吃饱。求解最多有多少孩子可以吃饱。

输入输出样例

输入两个数组，分别代表孩子的饥饿度和饼干的大小。输出最多有多少孩子可以吃饱的数量。

```
Input: [1,2], [1,2,3]
Output: 2
```

在这个样例中，我们可以给两个孩子喂 [1,2]、[1,3]、[2,3] 这三种组合的任意一种。

题解

因为饥饿度最小的孩子最容易吃饱，所以我们先考虑这个孩子。为了尽量使得剩下的饼干可以满足饥饿度更大的孩子，所以我们应该把大于等于这个孩子饥饿度的、且大小最小的饼干给这个孩子。满足了这个孩子之后，我们采取同样的策略，考虑剩下孩子里饥饿度最小的孩子，直到没有满足条件的饼干存在。

简而言之，这里的贪心策略是，给剩余孩子里最小饥饿度的孩子分配最小的能饱腹的饼干。

至于具体实现，因为我们需要获得大小关系，一个便捷的方法就是把孩子和饼干分别排序。这样我们就可以从饥饿度最小的孩子和大小最小的饼干出发，计算有多少个对子可以满足条件。

注意 对数组或字符串排序是常见的操作，方便之后的大小比较。

注意 在之后的讲解中，若我们谈论的是对连续空间的变量进行操作，我们并不会明确区分数组和字符串，因为他们本质上都是在连续空间上的有序变量集合。一个字符串 “abc” 可以被看作一个数组 [‘a’,‘b’,‘c’]。

```
int findContentChildren(vector<int>& children, vector<int>& cookies) {
    sort(children.begin(), children.end());
    sort(cookies.begin(), cookies.end());
    int child = 0, cookie = 0;
    while (child < children.size() && cookie < cookies.size()) {
        if (children[child] <= cookies[cookie]) ++child;
        ++cookie;
    }
    return child;
}
```

135. Candy (Hard)

题目描述

一群孩子站成一排，每一个孩子有自己的评分。现在需要给这些孩子发糖果，规则是如果一个孩子的评分比自己身旁的一个孩子要高，那么这个孩子就必须得到比身旁孩子更多的糖果；所有孩子至少要有一个糖果。求解最少需要多少个糖果。

输入输出样例

输入是一个数组，表示孩子的评分。输出是最少糖果的数量。

```
Input: [1,0,2]
Output: 5
```

在这个样例中，最少的糖果分法是 [2,1,2]。

题解

做完了题目 455，你会不会认为存在比较关系的贪心策略一定需要排序或是选择？虽然这一道题也是运用贪心策略，但我们只需要简单的两次遍历即可：把所有孩子的糖果数初始化为 1；先从左往右遍历一遍，如果右边孩子的评分比左边的高，则右边孩子的糖果数更新为左边孩子的糖果数加 1；再从右往左遍历一遍，如果左边孩子的评分比右边的高，且左边孩子当前的糖果数不大于右边孩子的糖果数，则左边孩子的糖果数更新为右边孩子的糖果数加 1。通过这两次遍历，分配的糖果就可以满足题目要求了。这里的贪心策略即为，在每次遍历中，只考虑并更新相邻一侧的大小关系。

在样例中，我们初始化糖果分配为 [1,1,1]，第一次遍历更新后的结果为 [1,1,2]，第二次遍历更新后的结果为 [2,1,2]。

```cpp
int candy(vector<int>& ratings) {
    int size = ratings.size();
    if (size < 2) {
        return size;
    }
    vector<int> num(size, 1);
    for (int i = 1; i < size; ++i) {
        if (ratings[i] > ratings[i-1]) {
            num[i] = num[i-1] + 1;
        }
    }
    for (int i = size - 1; i > 0; --i) {
        if (ratings[i] < ratings[i-1]) {
            num[i-1] = max(num[i-1], num[i] + 1);
        }
    }
    return accumulate(num.begin(), num.end(), 0); // std::accumulate 可以很方便
        地求和
}
```

2.3 区间问题

435. Non-overlapping Intervals (Medium)

题目描述

给定多个区间，计算让这些区间互不重叠所需要移除区间的最少个数。起止相连不算重叠。

输入输出样例

输入是一个数组，数组由多个长度固定为 2 的数组组成，表示区间的开始和结尾。输出一个整数，表示需要移除的区间数量。

```
Input: [[1,2], [2,4], [1,3]]
Output: 1
```

在这个样例中，我们可以移除区间 [1,3]，使得剩余的区间 [[1,2], [2,4]] 互不重叠。

题解

求最少的移除区间个数，等价于尽量多保留不重叠的区间。在选择要保留区间时，区间的结尾十分重要：选择的区间结尾越小，余留给其它区间的空间就越大，就越能保留更多的区间。因此，我们采取的贪心策略为，优先保留结尾小且不相交的区间。

具体实现方法为，先把区间按照结尾的大小进行增序排序，每次选择结尾最小且和前一个选择的区间不重叠的区间。我们这里使用 C++ 的 Lambda，结合 `std::sort()` 函数进行自定义排序。

在样例中，排序后的数组为 [[1,2], [1,3], [2,4]]。按照我们的贪心策略，首先初始化为区间 [1,2]；由于 [1,3] 与 [1,2] 相交，我们跳过该区间；由于 [2,4] 与 [1,2] 不相交，我们将其保留。因此最终保留的区间为 [[1,2], [2,4]]。

注意 需要根据实际情况判断按区间开头排序还是按区间结尾排序。

```
int eraseOverlapIntervals(vector<vector<int>>& intervals) {
    if (intervals.empty()) {
        return 0;
    }
    int n = intervals.size();
    sort(intervals.begin(), intervals.end(), [](vector<int>& a, vector<int>& b)
        {
        return a[1] < b[1];
    });
    int removed = 0, prev = intervals[0][1];
    for (int i = 1; i < n; ++i) {
        if (intervals[i][0] < prev) {
            ++removed;
        } else {
            prev = intervals[i][1];
        }
    }
    return removed;
}
```

2.4 练习

基础难度

605. Can Place Flowers (Easy)

采取什么样的贪心策略，可以种植最多的花朵呢?

452. Minimum Number of Arrows to Burst Balloons (Medium)

这道题和题目 435 十分类似，但是稍有不同，具体是哪里不同呢?

763. Partition Labels (Medium)

为了满足你的贪心策略，是否需要一些预处理?

注意 在处理数组前，统计一遍信息（如频率、个数、第一次出现位置、最后一次出现位置等）可以使题目难度大幅降低。

122. Best Time to Buy and Sell Stock II (Easy)

股票交易题型里比较简单的题目，在不限制交易次数的情况下，怎样可以获得最大利润呢?

进阶难度

406. Queue Reconstruction by Height (Medium)

温馨提示，这道题可能同时需要排序和插入操作。

665. Non-decreasing Array (Easy)

需要仔细思考你的贪心策略在各种情况下，是否仍然是最优解。

第 3 章 玩转双指针

内容提要

- 算法解释
- Two Sum
- 归并两个有序数组
- 快慢指针
- 滑动窗口

3.1 算法解释

双指针主要用于遍历数组，两个指针指向不同的元素，从而协同完成任务。也可以延伸到多个数组的多个指针。

若两个指针指向同一数组，遍历方向相同且不会相交，则也称为滑动窗口（两个指针包围的区域即为当前的窗口），经常用于区间搜索。

若两个指针指向同一数组，但是遍历方向相反，则可以用来进行搜索，待搜索的数组往往是排好序的。

对于 C++ 语言，指针还可以玩出很多新的花样。一些常见的关于指针的操作如下。

指针与常量

```
int x;
int * p1 = &x; // 指针可以被修改，值也可以被修改
const int * p2 = &x; // 指针可以被修改，值不可以被修改（const int）
int * const p3 = &x; // 指针不可以被修改（* const），值可以被修改
const int * const p4 = &x; // 指针不可以被修改，值也不可以被修改
```

指针函数与函数指针

```
// addition是指针函数，一个返回类型是指针的函数
int* addition(int a, int b) {
    int* sum = new int(a + b);
    return sum;
}

int subtraction(int a, int b) {
    return a - b;
}

int operation(int x, int y, int (*func)(int, int)) {
    return (*func)(x,y);
}

// minus是函数指针，指向函数的指针
int (*minus)(int, int) = subtraction;
```

```
int* m = addition(1, 2);
int n = operation(3, *m, minus);
```

3.2 Two Sum

167. Two Sum II - Input array is sorted (Easy)

题目描述

在一个增序的整数数组里找到两个数，使它们的和为给定值。已知有且只有一对解。

输入输出样例

输入是一个数组（numbers）和一个给定值（target）。输出是两个数的位置，从 1 开始计数。

```
Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
```

在这个样例中，第一个数字（2）和第二个数字（7）的和等于给定值（9）。

题解

因为数组已经排好序，我们可以采用方向相反的双指针来寻找这两个数字，一个初始指向最小的元素，即数组最左边，向右遍历；一个初始指向最大的元素，即数组最右边，向左遍历。

如果两个指针指向元素的和等于给定值，那么它们就是我们要的结果。如果两个指针指向元素的和小于给定值，我们把左边的指针右移一位，使得当前的和增加一点。如果两个指针指向元素的和大于给定值，我们把右边的指针左移一位，使得当前的和减少一点。

可以证明，对于排好序且有解的数组，双指针一定能遍历到最优解。证明方法如下：假设最优解的两个数的位置分别是 l 和 r。我们假设在左指针在 l 左边的时候，右指针已经移动到了 r；此时两个指针指向值的和小于给定值，因此左指针会一直右移直到到达 l。同理，如果我们假设在右指针在 r 右边的时候，左指针已经移动到了 l；此时两个指针指向值的和大于给定值，因此右指针会一直左移直到到达 r。所以双指针在任何时候都不可能处于 (l,r) 之间，又因为不满足条件时指针必须移动一个，所以最终一定会收敛在 l 和 r。

```
vector<int> twoSum(vector<int>& numbers, int target) {
    int l = 0, r = numbers.size() - 1, sum;
    while (l < r) {
        sum = numbers[l] + numbers[r];
        if (sum == target) break;
        if (sum < target) ++l;
        else --r;
    }
    return vector<int>{l + 1, r + 1};
}
```

3.3 归并两个有序数组

88. Merge Sorted Array (Easy)

题目描述

给定两个有序数组，把两个数组合并为一个。

输入输出样例

输入是两个数组和它们分别的长度 m 和 n。其中第一个数组的长度被延长至 $m+n$，多出的 n 位被 0 填补。题目要求把第二个数组归并到第一个数组上，不需要开辟额外空间。

```
Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
Output: nums1 = [1,2,2,3,5,6]
```

题解

因为这两个数组已经排好序，我们可以把两个指针分别放在两个数组的末尾，即 nums1 的 $m-1$ 位和 nums2 的 $n-1$ 位。每次将较大的那个数字复制到 nums1 的后边，然后向前移动一位。因为我们也要定位 nums1 的末尾，所以我们还需要第三个指针，以便复制。

在以下的代码里，我们直接利用 m 和 n 当作两个数组的指针，再额外创立一个 pos 指针，起始位置为 $m+n-1$。每次向前移动 m 或 n 的时候，也要向前移动 pos。这里需要注意，如果 nums1 的数字已经复制完，不要忘记把 nums2 的数字继续复制；如果 nums2 的数字已经复制完，剩余 nums1 的数字不需要改变，因为它们已经被排好序。

注意 这里我们使用了 ++ 和--的小技巧：a++ 和 ++a 都是将 a 加 1，但是 a++ 返回值为 a，而 ++a 返回值为 a+1。如果只是希望增加 a 的值，而不需要返回值，则推荐使用 ++a，其运行速度会略快一些。

```
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
    int pos = m-- + n-- - 1;
    while (m >= 0 && n >= 0) {
        nums1[pos--] = nums1[m] > nums2[n]? nums1[m--]: nums2[n--];
    }
    while (n >= 0) {
        nums1[pos--] = nums2[n--];
    }
}
```

3.4 快慢指针

142. Linked List Cycle II (Medium)

题目描述

给定一个链表，如果有环路，找出环路的开始点。

输入输出样例

输入是一个链表，输出是链表的一个节点。如果没有环路，返回一个空指针。

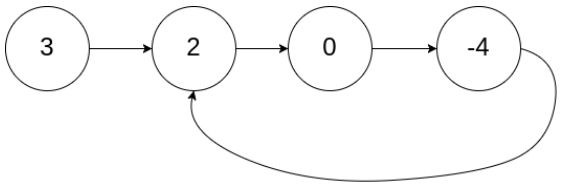

图 3.1: 题目 142 - 输入样例

在这个样例中，值为 2 的节点即为环路的开始点。

如果没有特殊说明，LeetCode 采用如下的数据结构表示链表。

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};
```

题解

对于链表找环路的问题，有一个通用的解法——快慢指针（Floyd 判圈法）。给定两个指针，分别命名为 slow 和 fast，起始位置在链表的开头。每次 fast 前进两步，slow 前进一步。如果 fast 可以走到尽头，那么说明没有环路；如果 fast 可以无限走下去，那么说明一定有环路，且一定存在一个时刻 slow 和 fast 相遇。当 slow 和 fast 第一次相遇时，我们将 fast 重新移动到链表开头，并让 slow 和 fast 每次都前进一步。当 slow 和 fast 第二次相遇时，相遇的节点即为环路的开始点。

```
ListNode *detectCycle(ListNode *head) {
    ListNode *slow = head, *fast = head;
    // 判断是否存在环路
    do {
        if (!fast || !fast->next) return nullptr;
        fast = fast->next->next;
        slow = slow->next;
    } while (fast != slow);
    // 如果存在，查找环路节点
    fast = head;
    while (fast != slow){
        slow = slow->next;
        fast = fast->next;
    }
    return fast;
}
```

3.5 滑动窗口

76. Minimum Window Substring (Hard)

题目描述

给定两个字符串 S 和 T，求 S 中包含 T 所有字符的最短连续子字符串的长度，同时要求时间复杂度不得超过 $O(n)$。

输入输出样例

输入是两个字符串 S 和 T，输出是一个 S 字符串的子串。

```
Input: S = "ADOBECODEBANC", T = "ABC"
Output: "BANC"
```

在这个样例中，S 中同时包含一个 A、一个 B、一个 C 的最短子字符串是 “BANC”。

题解

本题使用滑动窗口求解，即两个指针 l 和 r 都是从最左端向最右端移动，且 l 的位置一定在 r 的左边或重合。注意本题虽然在 for 循环里出现了一个 while 循环，但是因为 while 循环负责移动 l 指针，且 l 只会从左到右移动一次，因此总时间复杂度仍然是 $O(n)$。本题使用了长度为 128 的数组来映射字符，也可以用哈希表替代；其中 chars 表示目前每个字符缺少的数量，flag 表示每个字符是否在 T 中存在。

```
string minWindow(string S, string T) {
    vector<int> chars(128, 0);
    vector<bool> flag(128, false);
    // 先统计T中的字符情况
    for (int i = 0; i < T.size(); ++i) {
        flag[T[i]] = true;
        ++chars[T[i]];
    }
    // 移动滑动窗口，不断更改统计数据
    int cnt = 0, l = 0, min_l = 0, min_size = S.size() + 1;
    for (int r = 0; r < S.size(); ++r) {
        if (flag[S[r]]) {
            if (--chars[S[r]] >= 0) {
                ++cnt;
            }
            // 若目前滑动窗口已包含T中全部字符，
            // 则尝试将l右移，在不影响结果的情况下获得最短子字符串
            while (cnt == T.size()) {
                if (r - l + 1 < min_size) {
                    min_l = l;
                    min_size = r - l + 1;
                }
                if (flag[S[l]] && ++chars[S[l]] > 0) {
                    --cnt;
                }
                ++l;
            }
        }
    }
    return min_size > S.size()? "": S.substr(min_l, min_size);
}
```

3.6 练习

基础难度

633. Sum of Square Numbers (Easy)

Two Sum 题目的变形题之一。

680. Valid Palindrome II (Easy)

Two Sum 题目的变形题之二。

524. Longest Word in Dictionary through Deleting (Medium)

归并两个有序数组的变形题。

进阶难度

340. Longest Substring with At Most K Distinct Characters (Hard)

需要利用其它数据结构方便统计当前的字符状态。

第 4 章 居合斩！二分查找

内容提要

- 算法解释
- 查找区间
- 求开方
- 旋转数组查找数字

4.1 算法解释

二分查找也常被称为二分法或者折半查找，每次查找时通过将待查找区间分成两部分并只取一部分继续查找，将查找的复杂度大大减少。对于一个长度为 $O(n)$ 的数组，二分查找的时间复杂度为 $O(\log n)$。

举例来说，给定一个排好序的数组 {3,4,5,6,7}，我们希望查找 4 在不在这个数组内。第一次折半时考虑中位数 5，因为 5 大于 4, 所以如果 4 存在于这个数组，那么其必定存在于 5 左边这一半。于是我们的查找区间变成了 {3,4,5}。（注意，根据具体情况和您的刷题习惯，这里的 5 可以保留也可以不保留，并不影响时间复杂度的级别。）第二次折半时考虑新的中位数 4，正好是我们需要查找的数字。于是我们发现，对于一个长度为 5 的数组，我们只进行了 2 次查找。如果是遍历数组，最坏的情况则需要查找 5 次。

我们也可以用更加数学的方式定义二分查找。给定一个在 $[a,b]$ 区间内的单调函数 $f(x)$，若 $f(a)$ 和 $f(b)$ 正负性相反，那么必定存在一个解 c，使得 $f(c)=0$。在上个例子中，$f(x)$ 是离散函数 $f(x)=x+2$，查找 4 是否存在等价于求 $f(x)-4=0$ 是否有离散解。因为 $f(1)-4=3-4=-1<0$、$f(5)-4=7-4=3>0$，且函数在区间内单调递增，因此我们可以利用二分查找求解。如果最后二分到了不能再分的情况，如只剩一个数字，且剩余区间里不存在满足条件的解，则说明不存在离散解，即 4 不在这个数组内。

具体到代码上，二分查找时区间的左右端取开区间还是闭区间在绝大多数时候都可以，因此有些初学者会容易搞不清楚如何定义区间开闭性。这里我提供两个小诀窍，第一是尝试熟练使用一种写法，比如左闭右开（满足 C++、Python 等语言的习惯）或左闭右闭（便于处理边界条件），尽量只保持这一种写法；第二是在刷题时思考如果最后区间只剩下一个数或者两个数，自己的写法是否会陷入死循环，如果某种写法无法跳出死循环，则考虑尝试另一种写法。

二分查找也可以看作双指针的一种特殊情况，但我们一般会将二者区分。双指针类型的题，指针通常是一步一步移动的，而在二分查找里，指针每次移动半个区间长度。

4.2 求开方

69. Sqrt(x) (Easy)

题目描述

给定一个非负整数，求它的开方，向下取整。

输入输出样例

输入一个整数，输出一个整数。

```
Input: 8
Output: 2
```

8 的开方结果是 2.82842...，向下取整即是 2。

题解

我们可以把这道题想象成，给定一个非负整数 a，求 $f(x) = x^2 - a = 0$ 的解。因为我们只考虑 $x \geq 0$，所以 $f(x)$ 在定义域上是单调递增的。考虑到 $f(0) = -a \leq 0$，$f(a) = a^2 - a \geq 0$，我们可以对 $[0, a]$ 区间使用二分法找到 $f(x) = 0$ 的解。

在以下的代码里，为了防止除以 0，我们把 $a = 0$ 的情况单独考虑，然后对区间 $[1, a]$ 进行二分查找。我们使用了左闭右闭的写法。

注意 $mid = (l + r)/2$ 可能会因为 $l + r$ 溢出而错误，因而采用 $mid = l + (r - l)/2$ 的写法。

```
int mySqrt(int a) {
    if (a == 0) return a;
    int l = 1, r = a, mid, sqrt;
    while (l <= r) {
        mid = l + (r - l) / 2;
        sqrt = a / mid;
        if (sqrt == mid) {
            return mid;
        } else if (mid > sqrt) {
            r = mid - 1;
        } else {
            l = mid + 1;
        }
    }
    return r;
}
```

另外，这道题还有一种更快的算法——牛顿迭代法，其公式为 $x_{n+1} = x_n - f(x_n)/f'(x_n)$。给定 $f(x) = x^2 - a = 0$，这里的迭代公式为 $x_{n+1} = (x_n + a/x_n)/2$，其代码如下。

注意 这里为了防止 int 超上界，我们使用 long 来存储乘法结果。

```
int mySqrt(int a) {
    long x = a;
    while (x * x > a) {
        x = (x + a / x) / 2;
    }
    return x;
}
```

4.3 查找区间

34. Find First and Last Position of Element in Sorted Array (Medium)

题目描述

给定一个增序的整数数组和一个值，查找该值第一次和最后一次出现的位置。

输入输出样例

输入是一个数组和一个值，输出为该值第一次出现的位置和最后一次出现的位置（从 0 开始）；如果不存在该值，则两个返回值都设为-1。

```
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]
```

数字 8 在第 3 位第一次出现，在第 4 位最后一次出现。

题解

这道题可以看作是自己实现 C++ 里的 `lower_bound` 和 `upper_bound` 函数。这里我们尝试使用左闭右开的写法，当然左闭右闭也可以。

```cpp
// 主函数
vector<int> searchRange(vector<int>& nums, int target) {
    if (nums.empty()) return vector<int>{-1, -1};
    int lower = lower_bound(nums, target);
    int upper = upper_bound(nums, target) - 1; // 这里需要减1位
    if (lower == nums.size() || nums[lower] != target) {
        return vector<int>{-1, -1};
    }
    return vector<int>{lower, upper};
}

// 辅函数
int lower_bound(vector<int> &nums, int target) {
    int l = 0, r = nums.size(), mid;
    while (l < r) {
        mid = l + (r - l) / 2;
        if (nums[mid] >= target) {
            r = mid;
        } else {
            l = mid + 1;
        }
    }
    return l;
}

// 辅函数
int upper_bound(vector<int> &nums, int target) {
    int l = 0, r = nums.size(), mid;
    while (l < r) {
        mid = l + (r - l) / 2;
        if (nums[mid] > target) {
            r = mid;
        } else {
```

```
            l = mid + 1;
        }
    }
    return l;
}
```

4.4 旋转数组查找数字

81. Search in Rotated Sorted Array II (Medium)

题目描述

一个原本增序的数组被首尾相连后按某个位置断开（如 [1,2,2,3,4,5] → [2,3,4,5,1,2]，在第一位和第二位断开），我们称其为旋转数组。给定一个值，判断这个值是否存在于这个旋转数组中。

输入输出样例

输入是一个数组和一个值，输出是一个布尔值，表示数组中是否存在该值。

```
Input: nums = [2,5,6,0,0,1,2], target = 0
Output: true
```

题解

即使数组被旋转过，我们仍然可以利用这个数组的递增性，使用二分查找。对于当前的中点，如果它指向的值小于等于右端，那么说明右区间是排好序的；反之，那么说明左区间是排好序的。如果目标值位于排好序的区间内，我们可以对这个区间继续二分查找；反之，我们对于另一半区间继续二分查找。

注意，因为数组存在重复数字，如果中点和左端的数字相同，我们并不能确定是左区间全部相同，还是右区间完全相同。在这种情况下，我们可以简单地将左端点右移一位，然后继续进行二分查找。

```
bool search(vector<int>& nums, int target) {
    int l = 0, r = nums.size() - 1;
    while (l <= r) {
        int mid = l + (r - l) / 2;
        if (nums[mid] == target) {
            return true;
        }
        if (nums[l] == nums[mid]) {
            // 无法判断哪个区间是增序的
            ++l;
        } else if (nums[mid] <= nums[r]) {
            // 右区间是增序的
            if (target > nums[mid] && target <= nums[r]) {
                l = mid + 1;
            } else {
                r = mid - 1;
            }
```

```
        } else {
            // 左区间是增序的
            if (target >= nums[l] && target < nums[mid]) {
                r = mid - 1;
            } else {
                l = mid + 1;
            }
        }
    }
    return false;
}
```

4.5 练习

基础难度

154. Find Minimum in Rotated Sorted Array II (Medium)

旋转数组的变形题之一。

540. Single Element in a Sorted Array (Medium)

在出现独立数之前和之后，奇偶位数的值发生了什么变化？

进阶难度

4. Median of Two Sorted Arrays (Hard)

需要对两个数组同时进行二分搜索。

第 5 章　千奇百怪的排序算法

内容提要

- 常用排序算法
- 快速排序
- 桶排序

5.1　常用排序算法

以下是一些最基本的排序算法。虽然在 C++ 里可以通过 `std::sort()` 快速排序，而且刷题时很少需要自己手写排序算法，但是熟习各种排序算法可以加深自己对算法的基本理解，以及解出由这些排序算法引申出来的题目。

快速排序（Quicksort）

我们采用左闭右开的二分写法。

```
void quick_sort(vector<int> &nums, int l, int r) {
    if (l + 1 >= r) {
        return;
    }
    int first = l, last = r - 1, key = nums[first];
    while (first < last){
        while(first < last && nums[last] >= key) {
            --last;
        }
        nums[first] = nums[last];
        while (first < last && nums[first] <= key) {
            ++first;
        }
        nums[last] = nums[first];
    }
    nums[first] = key;
    quick_sort(nums, l, first);
    quick_sort(nums, first + 1, r);
}
```

归并排序（Merge Sort）

```
void merge_sort(vector<int> &nums, int l, int r, vector<int> &temp) {
    if (l + 1 >= r) {
        return;
    }
    // divide
    int m = l + (r - l) / 2;
    merge_sort(nums, l, m, temp);
```

```
    merge_sort(nums, m, r, temp);
    // conquer
    int p = l, q = m, i = l;
    while (p < m || q < r) {
        if (q >= r || (p < m && nums[p] <= nums[q])) {
            temp[i++] = nums[p++];
        } else {
            temp[i++] = nums[q++];
        }
    }
    for (i = l; i < r; ++i) {
        nums[i] = temp[i];
    }
}
```

插入排序（Insertion Sort）

```
void insertion_sort(vector<int> &nums, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = i; j > 0 && nums[j] < nums[j-1]; --j) {
            swap(nums[j], nums[j-1]);
        }
    }
}
```

冒泡排序（Bubble Sort）

```
void bubble_sort(vector<int> &nums, int n) {
    bool swapped;
    for (int i = 1; i < n; ++i) {
        swapped = false;
        for (int j = 1; j < n - i + 1; ++j) {
            if (nums[j] < nums[j-1]) {
                swap(nums[j], nums[j-1]);
                swapped = true;
            }
        }
        if (!swapped) {
            break;
        }
   }
}
```

选择排序（Selection Sort）

```
void selection_sort(vector<int> &nums, int n) {
    int mid;
    for (int i = 0; i < n - 1; ++i) {
        mid = i;
```

```
        for (int j = i + 1; j < n; ++j) {
            if (nums[j] < nums[mid]) {
                mid = j;
            }
        }
        swap(nums[mid], nums[i]);
    }
}
```

以上排序代码调用方法为

```
void sort() {
    vector<int> nums = {1,3,5,7,2,6,4,8,9,2,8,7,6,0,3,5,9,4,1,0};
    vector<int> temp(nums.size());
    sort(nums.begin(), nums.end());
    quick_sort(nums, 0, nums.size());
    merge_sort(nums, 0, nums.size(), temp);
    insertion_sort(nums, nums.size());
    bubble_sort(nums, nums.size());
    selection_sort(nums, nums.size());
}
```

5.2 快速选择

215. Kth Largest Element in an Array

题目描述

在一个未排序的数组中，找到第 k 大的数字。

输入输出样例

输入一个数组和一个目标值 k，输出第 k 大的数字。题目默认一定有解。

```
Input: [3,2,1,5,6,4] and k = 2
Output: 5
```

题解

快速选择一般用于求解 k-th Element 问题，可以在 $O(n)$ 时间复杂度，$O(1)$ 空间复杂度完成求解工作。快速选择的实现和快速排序相似，不过只需要找到第 k 大的枢（pivot）即可，不需要对其左右再进行排序。与快速排序一样，快速选择一般需要先打乱数组，否则最坏情况下时间复杂度为 $O(n^2)$，我们这里为了方便省略掉了打乱的步骤。

```
// 主函数
int findKthLargest(vector<int>& nums, int k) {
    int l = 0, r = nums.size() - 1, target = nums.size() - k;
    while (l < r) {
        int mid = quickSelection(nums, l, r);
        if (mid == target) {
```

```
            return nums[mid];
        }
        if (mid < target) {
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    return nums[l];
}

// 辅函数 - 快速选择
int quickSelection(vector<int>& nums, int l, int r) {
    int i = l + 1, j = r;
    while (true) {
        while (i < r && nums[i] <= nums[l]) {
            ++i;
        }
        while (l < j && nums[j] >= nums[l]) {
            --j;
        }
        if (i >= j) {
            break;
        }
        swap(nums[i], nums[j]);
    }
    swap(nums[l], nums[j]);
    return j;
}
```

5.3 桶排序

347. Top K Frequent Elements (Medium)

题目描述

给定一个数组，求前 k 个最频繁的数字。

输入输出样例

输入是一个数组和一个目标值 k。输出是一个长度为 k 的数组。

```
Input: nums = [1,1,1,1,2,2,3,4], k = 2
Output: [1,2]
```

在这个样例中，最频繁的两个数是 1 和 2。

题解

顾名思义，桶排序的意思是为每个值设立一个桶，桶内记录这个值出现的次数（或其它属性），然后对桶进行排序。针对样例来说，我们先通过桶排序得到四个桶 [1,2,3,4]，它们的值分别为 [4,2,1,1]，表示每个数字出现的次数。

紧接着，我们对桶的频次进行排序，前 k 大个桶即是前 k 个频繁的数。这里我们可以使用各种排序算法，甚至可以再进行一次桶排序，把每个旧桶根据频次放在不同的新桶内。针对样例来说，因为目前最大的频次是 4，我们建立 [1,2,3,4] 四个新桶，它们分别放入的旧桶为 [[3,4],[2],[],[1]]，表示不同数字出现的频率。最后，我们从后往前遍历，直到找到 k 个旧桶。

```
vector<int> topKFrequent(vector<int>& nums, int k) {
    unordered_map<int, int> counts;
    int max_count = 0;
    for (const int & num : nums) {
        max_count = max(max_count, ++counts[num]);
    }

    vector<vector<int>> buckets(max_count + 1);
    for (const auto & p : counts) {
        buckets[p.second].push_back(p.first);
    }

    vector<int> ans;
    for (int i = max_count; i >= 0 && ans.size() < k; --i) {
        for (const int & num : buckets[i]) {
            ans.push_back(num);
            if (ans.size() == k) {
                break;
            }
        }
    }
    return ans;
}
```

5.4 练习

基础难度

451. Sort Characters By Frequency (Medium)

桶排序的变形题。

进阶难度

75. Sort Colors (Medium)

很经典的荷兰国旗问题，考察如何对三个重复且打乱的值进行排序。

第 6 章 一切皆可搜索

内容提要

- 算法解释
- 深度优先搜索
- 回溯法
- 广度优先搜索

6.1 算法解释

深度优先搜索和广度优先搜索是两种最常见的优先搜索方法，它们被广泛地运用在图和树等结构中进行搜索。

6.2 深度优先搜索

深度优先搜索（depth-first seach，DFS）在搜索到一个新的节点时，立即对该新节点进行遍历；因此遍历需要用先入后出的栈来实现，也可以通过与栈等价的递归来实现。对于树结构而言，由于总是对新节点调用遍历，因此看起来是向着“深”的方向前进。

考虑如下一颗简单的树。我们从 1 号节点开始遍历，假如遍历顺序是从左子节点到右子节点，那么按照优先向着“深”的方向前进的策略，假如我们使用递归实现，我们的遍历过程为 1（起始节点）->2（遍历更深一层的左子节点）->4（遍历更深一层的左子节点）->2（无子节点，返回父结点）->1（子节点均已完成遍历，返回父结点）->3（遍历更深一层的右子节点）->1（无子节点，返回父结点）-> 结束程序（子节点均已完成遍历）。如果我们使用栈实现，我们的栈顶元素的变化过程为 1->2->4->3。

```
    1
   / \
  2   3
 /
4
```

深度优先搜索也可以用来检测环路：记录每个遍历过的节点的父节点，若一个节点被再次遍历且父节点不同，则说明有环。我们也可以用之后会讲到的拓扑排序判断是否有环路，若最后存在入度不为零的点，则说明有环。

有时我们可能会需要对已经搜索过的节点进行标记，以防止在遍历时重复搜索某个节点，这种做法叫做状态记录或记忆化（memoization）。

695. Max Area of Island (Easy)

题目描述

给定一个二维的 0-1 矩阵，其中 0 表示海洋，1 表示陆地。单独的或相邻的陆地可以形成岛屿，每个格子只与其上下左右四个格子相邻。求最大的岛屿面积。

输入输出样例

输入是一个二维数组，输出是一个整数，表示最大的岛屿面积。

```
Input:
[[1,0,1,1,0,1,0,1],
 [1,0,1,1,0,1,1,1],
 [0,0,0,0,0,0,0,1]]
Output: 6
```

最大的岛屿面积为 6，位于最右侧。

题解

此题是十分标准的搜索题，我们可以拿来练手深度优先搜索。一般来说，深度优先搜索类型的题可以分为主函数和辅函数，主函数用于遍历所有的搜索位置，判断是否可以开始搜索，如果可以即在辅函数进行搜索。辅函数则负责深度优先搜索的递归调用。当然，我们也可以使用栈（stack）实现深度优先搜索，但因为栈与递归的调用原理相同，而递归相对便于实现，因此刷题时笔者推荐使用递归式写法，同时也方便进行回溯（见下节）。不过在实际工程上，直接使用栈可能才是最好的选择，一是因为便于理解，二是更不易出现递归栈满的情况。我们先展示使用栈的写法。

```
vector<int> direction{-1, 0, 1, 0, -1};

int maxAreaOfIsland(vector<vector<int>>& grid) {
    int m = grid.size(), n = m? grid[0].size(): 0, local_area, area = 0, x, y;
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (grid[i][j]) {
                local_area = 1;
                grid[i][j] = 0;
                stack<pair<int, int>> island;
                island.push({i, j});
                while (!island.empty()) {
                    auto [r, c] = island.top();
                    island.pop();
                    for (int k = 0; k < 4; ++k) {
                        x = r + direction[k], y = c + direction[k+1];
                        if (x >= 0 && x < m &&
                            y >= 0 && y < n && grid[x][y] == 1) {
                            grid[x][y] = 0;
                            ++local_area;
                            island.push({x, y});
                        }
                    }
                }
                area = max(area, local_area);
            }
        }
    }
    return area;
}
```

这里我们使用了一个小技巧，对于四个方向的遍历，可以创造一个数组 [-1, 0, 1, 0, -1]，每相邻两位即为上下左右四个方向之一。

在辅函数里，一个一定要注意的点是辅函数内递归搜索时，边界条件的判定。边界判定一般有两种写法，一种是先判定是否越界，只有在合法的情况下才进行下一步搜索（即判断放在调用递归函数前）；另一种是不管三七二十一先进行下一步搜索，待下一步搜索开始时再判断是否合法（即判断放在辅函数第一行）。我们这里分别展示这两种写法。

第一种递归写法为：

```
vector<int> direction{-1, 0, 1, 0, -1};

// 主函数
int maxAreaOfIsland(vector<vector<int>>& grid) {
    if (grid.empty() || grid[0].empty()) return 0;
    int max_area = 0;
    for (int i = 0; i < grid.size(); ++i) {
        for (int j = 0; j < grid[0].size(); ++j) {
            if (grid[i][j] == 1) {
                max_area = max(max_area, dfs(grid, i, j));
            }
        }
    }
    return max_area;
}

// 辅函数
int dfs(vector<vector<int>>& grid, int r, int c) {
    if (grid[r][c] == 0) return 0;
    grid[r][c] = 0;
    int x, y, area = 1;
    for (int i = 0; i < 4; ++i) {
        x = r + direction[i], y = c + direction[i+1];
        if (x >= 0 && x < grid.size() && y >= 0 && y < grid[0].size()) {
            area += dfs(grid, x, y);
        }
    }
    return area;
}
```

第二种递归写法为：

```
// 主函数
int maxAreaOfIsland(vector<vector<int>>& grid) {
    if (grid.empty() || grid[0].empty()) return 0;
    int max_area = 0;
    for (int i = 0; i < grid.size(); ++i) {
        for (int j = 0; j < grid[0].size(); ++j) {
            max_area = max(max_area, dfs(grid, i, j));
        }
    }
    return max_area;
}

// 辅函数
int dfs(vector<vector<int>>& grid, int r, int c) {
    if (r < 0 || r >= grid.size() ||
        c < 0 || c >= grid[0].size() || grid[r][c] == 0) {
        return 0;
```

```
    }
    grid[r][c] = 0;
    return 1 + dfs(grid, r + 1, c) + dfs(grid, r - 1, c) +
               dfs(grid, r, c + 1) + dfs(grid, r, c - 1);
}
```

547. Friend Circles (Medium)

题目描述

给定一个二维的 0-1 矩阵，如果第 (i, j) 位置是 1，则表示第 i 个人和第 j 个人是朋友。已知朋友关系是可以传递的，即如果 a 是 b 的朋友，b 是 c 的朋友，那么 a 和 c 也是朋友，换言之这三个人处于同一个朋友圈之内。求一共有多少个朋友圈。

输入输出样例

输入是一个二维数组，输出是一个整数，表示朋友圈数量。因为朋友关系具有对称性，该二维数组为对称矩阵。同时，因为自己是自己的朋友，对角线上的值全部为 1。

```
Input:
[[1,1,0],
 [1,1,0],
 [0,0,1]]
Output: 2
```

在这个样例中，[1,2] 处于一个朋友圈，[3] 处于一个朋友圈。

题解

对于题目 695，图的表示方法是，每个位置代表一个节点，每个节点与上下左右四个节点相邻。而在这一道题里面，每一行（列）表示一个节点，它的每列（行）表示是否存在一个相邻节点。因此题目 695 拥有 $m \times n$ 个节点，每个节点有 4 条边；而本题拥有 n 个节点，每个节点最多有 n 条边，表示和所有人都是朋友，最少可以有 1 条边，表示自己与自己相连。当清楚了图的表示方法后，这道题与题目 695 本质上是同一道题：搜索朋友圈（岛屿）的个数（最大面积）。我们这里采用递归的第一种写法。

```
// 主函数
int findCircleNum(vector<vector<int>>& friends) {
    int n = friends.size(), count = 0;
    vector<bool> visited(n, false);
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(friends, i, visited);
            ++count;
        }
    }
    return count;
}

// 辅函数
void dfs(vector<vector<int>>& friends, int i, vector<bool>& visited) {
```

```
    visited[i] = true;
    for (int k = 0; k < friends.size(); ++k) {
        if (friends[i][k] == 1 && !visited[k]) {
            dfs(friends, k, visited);
        }
    }
}
```

417. Pacific Atlantic Water Flow (Medium)

题目描述

给定一个二维的非负整数矩阵，每个位置的值表示海拔高度。假设左边和上边是太平洋，右边和下边是大西洋，求从哪些位置向下流水，可以流到太平洋和大西洋。水只能从海拔高的位置流到海拔低或相同的位置。

输入输出样例

输入是一个二维的非负整数数组，表示海拔高度。输出是一个二维的数组，其中第二个维度大小固定为 2，表示满足条件的位置坐标。

```
Input:
  太平洋  ~   ~   ~   ~   ~
       ~  1   2   2   3  (5) *
       ~  3   2   3  (4) (4) *
       ~  2   4  (5)  3   1  *
       ~ (6) (7) 1   4   5  *
       ~ (5) 1   1   2   4  *
          *   *   *   *   * 大西洋
Output: [[0, 4], [1, 3], [1, 4], [2, 2], [3, 0], [3, 1], [4, 0]]
```

在这个样例中，有括号的区域为满足条件的位置。

题解

虽然题目要求的是满足向下流能到达两个大洋的位置，如果我们对所有的位置进行搜索，那么在不剪枝的情况下复杂度会很高。因此我们可以反过来想，从两个大洋开始向上流，这样我们只需要对矩形四条边进行搜索。搜索完成后，只需遍历一遍矩阵，满足条件的位置即为两个大洋向上流都能到达的位置。

```
vector<int> direction{-1, 0, 1, 0, -1};

// 主函数
vector<vector<int>> pacificAtlantic(vector<vector<int>>& matrix) {
    if (matrix.empty() || matrix[0].empty()) {
        return {};
    }
    vector<vector<int>> ans;
    int m = matrix.size(), n = matrix[0].size();
    vector<vector<bool>> can_reach_p(m, vector<bool>(n, false));
    vector<vector<bool>> can_reach_a(m, vector<bool>(n, false));
```

```
    for (int i = 0; i < m; ++i) {
        dfs(matrix, can_reach_p, i, 0);
        dfs(matrix, can_reach_a, i, n - 1);
    }
    for (int i = 0; i < n; ++i) {
        dfs(matrix, can_reach_p, 0, i);
        dfs(matrix, can_reach_a, m - 1, i);
    }
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; ++j) {
            if (can_reach_p[i][j] && can_reach_a[i][j]) {
                ans.push_back(vector<int>{i, j});
            }
        }
    }
    return ans;
}

// 辅函数
void dfs(const vector<vector<int>>& matrix, vector<vector<bool>>& can_reach,
    int r, int c) {
    if (can_reach[r][c]) {
        return;
    }
    can_reach[r][c] = true;
    int x, y;
    for (int i = 0; i < 4; ++i) {
        x = r + direction[i], y = c + direction[i+1];
        if (x >= 0 && x < matrix.size()
            && y >= 0 && y < matrix[0].size() &&
            matrix[r][c] <= matrix[x][y]) {
            dfs(matrix, can_reach, x, y);
        }
    }
}
```

6.3 回溯法

回溯法（backtracking）是优先搜索的一种特殊情况，又称为试探法，常用于需要记录节点状态的深度优先搜索。通常来说，排列、组合、选择类问题使用回溯法比较方便。

顾名思义，回溯法的核心是回溯。在搜索到某一节点的时候，如果我们发现目前的节点（及其子节点）并不是需求目标时，我们回退到原来的节点继续搜索，并且把在目前节点修改的状态还原。这样的好处是我们可以始终只对图的总状态进行修改，而非每次遍历时新建一个图来储存状态。在具体的写法上，它与普通的深度优先搜索一样，都有 [修改当前节点状态]→[递归子节点] 的步骤，只是多了回溯的步骤，变成了 [修改当前节点状态]→[递归子节点]→[回改当前节点状态]。

没有接触过回溯法的读者可能会不明白我在讲什么，这也完全正常，希望以下几道题可以让您理解回溯法。如果还是不明白，可以记住两个小诀窍，一是按引用传状态，二是所有的状态修改在递归完成后回改。

回溯法修改一般有两种情况，一种是修改最后一位输出，比如排列组合；一种是修改访问标记，比如矩阵里搜字符串。

46. Permutations (Medium)

题目描述

给定一个无重复数字的整数数组，求其所有的排列方式。

输入输出样例

输入是一个一维整数数组，输出是一个二维数组，表示输入数组的所有排列方式。

```
Input: [1,2,3]
Output: [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,2,1], [3,1,2]]
```

可以以任意顺序输出，只要包含了所有排列方式即可。

题解

怎样输出所有的排列方式呢？对于每一个当前位置 i，我们可以将其于之后的任意位置交换，然后继续处理位置 i+1，直到处理到最后一位。为了防止我们每此遍历时都要新建一个子数组储存位置 i 之前已经交换好的数字，我们可以利用回溯法，只对原数组进行修改，在递归完成后再修改回来。

我们以样例 [1,2,3] 为例，按照这种方法，我们输出的数组顺序为 [[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,2,1], [3,1,2]]，可以看到所有的排列在这个算法中都被考虑到了。

```
// 主函数
vector<vector<int>> permute(vector<int>& nums) {
    vector<vector<int>> ans;
    backtracking(nums, 0, ans);
    return ans;
}

// 辅函数
void backtracking(vector<int> &nums, int level, vector<vector<int>> &ans) {
    if (level == nums.size() - 1) {
        ans.push_back(nums);
        return;
    }
    for (int i = level; i < nums.size(); i++) {
        swap(nums[i], nums[level]); // 修改当前节点状态
        backtracking(nums, level+1, ans); // 递归子节点
        swap(nums[i], nums[level]); // 回改当前节点状态
    }
}
```

77. Combinations (Medium)

题目描述

给定一个整数 n 和一个整数 k，求在 1 到 n 中选取 k 个数字的所有组合方法。

输入输出样例

输入是两个正整数 n 和 k，输出是一个二维数组，表示所有组合方式。

```
Input: n = 4, k = 2
Output: [[2,4], [3,4], [2,3], [1,2], [1,3], [1,4]]
```

这里二维数组的每个维度都可以以任意顺序输出。

题解

类似于排列问题，我们也可以进行回溯。排列回溯的是交换的位置，而组合回溯的是否把当前的数字加入结果中。

```
// 主函数
vector<vector<int>> combine(int n, int k) {
    vector<vector<int>> ans;
    vector<int> comb(k, 0);
    int count = 0;
    backtracking(ans, comb, count, 1, n, k);
    return ans;
}

// 辅函数
void backtracking(vector<vector<int>>& ans, vector<int>& comb, int& count, int
    pos, int n, int k) {
    if (count == k) {
        ans.push_back(comb);
        return;
    }
    for (int i = pos; i <= n; ++i) {
        comb[count++] = i; // 修改当前节点状态
        backtracking(ans, comb, count, i + 1, n, k); // 递归子节点
        --count; // 回改当前节点状态
    }
}
```

79. Word Search (Medium)

题目描述

给定一个字母矩阵，所有的字母都与上下左右四个方向上的字母相连。给定一个字符串，求字符串能不能在字母矩阵中寻找到。

输入输出样例

输入是一个二维字符数组和一个字符串，输出是一个布尔值，表示字符串是否可以被寻找到。

```
Input: word = "ABCCED", board =
[['A','B','C','E'],
```

```
 ['S','F','C','S'],
 ['A','D','E','E']]
Output: true
```

从左上角的'A' 开始，我们可以先向右、再向下、最后向左，找到连续的"ABCCED"。

题解

不同于排列组合问题，本题采用的并不是修改输出方式，而是修改访问标记。在我们对任意位置进行深度优先搜索时，我们先标记当前位置为已访问，以避免重复遍历（如防止向右搜索后又向左返回）；在所有的可能都搜索完成后，再回改当前位置为未访问，防止干扰其它位置搜索到当前位置。使用回溯法，我们可以只对一个二维的访问矩阵进行修改，而不用把每次的搜索状态作为一个新对象传入递归函数中。

```cpp
// 主函数
bool exist(vector<vector<char>>& board, string word) {
    if (board.empty()) return false;
    int m = board.size(), n = board[0].size();
    vector<vector<bool>> visited(m, vector<bool>(n, false));
    bool find = false;
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            backtracking(i, j, board, word, find, visited, 0);
        }
    }
    return find;
}

// 辅函数
void backtracking(int i, int j, vector<vector<char>>& board, string& word, bool
    & find, vector<vector<bool>>& visited, int pos) {
    if (i < 0 || i >= board.size() || j < 0 || j >= board[0].size()) {
        return;
    }
    if (visited[i][j] || find || board[i][j] != word[pos]) {
        return;
    }
    if (pos == word.size() - 1) {
        find = true;
        return;
    }
    visited[i][j] = true; // 修改当前节点状态
    // 递归子节点
    backtracking(i + 1, j, board, word, find, visited, pos + 1);
    backtracking(i - 1, j, board, word, find, visited, pos + 1);
    backtracking(i, j + 1, board, word, find, visited, pos + 1);
    backtracking(i, j - 1, board, word, find, visited, pos + 1);
    visited[i][j] = false; // 回改当前节点状态
}
```

51. N-Queens (Hard)

题目描述

给定一个大小为 n 的正方形国际象棋棋盘，求有多少种方式可以放置 n 个皇后并使得她们互不攻击，即每一行、列、左斜、右斜最多只有一个皇后。

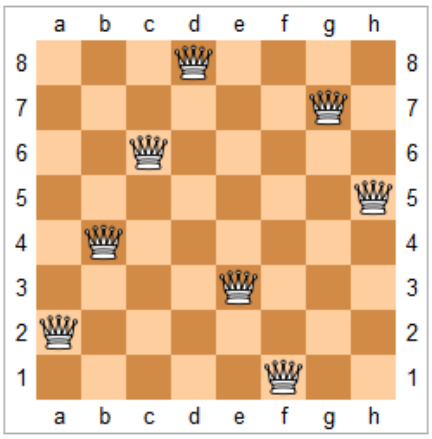

图 6.1: 题目 51 - 八皇后的一种解法

输入输出样例

输入是一个整数 n，输出是一个二维字符串数组，表示所有的棋盘表示方法。

```
Input: 4
Output: [
 [".Q..", // Solution 1
  "...Q",
  "Q...",
  "..Q."],
 ["..Q.", // Solution 2
  "Q...",
  "...Q",
  ".Q.."]
]
```

在这个样例中，点代表空白位置，Q 代表皇后。

题解

类似于在矩阵中寻找字符串，本题也是通过修改状态矩阵来进行回溯。不同的是，我们需要对每一行、列、左斜、右斜建立访问数组，来记录它们是否存在皇后。

本题有一个隐藏的条件，即满足条件的结果中每一行或列有且仅有一个皇后。这是因为我们一共只有 n 行和 n 列。所以如果我们通过对每一行遍历来插入皇后，我们就不需要对行建立访问数组了。

```
// 主函数
vector<vector<string>> solveNQueens(int n) {
    vector<vector<string>> ans;
    if (n == 0) {
        return ans;
```

```
    }
    vector<string> board(n, string(n, '.'));
    vector<bool> column(n, false), ldiag(2*n-1, false), rdiag(2*n-1, false);
    backtracking(ans, board, column, ldiag, rdiag, 0, n);
    return ans;
}

// 辅函数
void backtracking(vector<vector<string>> &ans, vector<string> &board, vector<
    bool> &column, vector<bool> &ldiag, vector<bool> &rdiag, int row, int n) {
    if (row == n) {
        ans.push_back(board);
        return;
    }
    for (int i = 0; i < n; ++i) {
        if (column[i] || ldiag[n-row+i-1] || rdiag[row+i]) {
            continue;
        }
        // 修改当前节点状态
        board[row][i] = 'Q';
        column[i] = ldiag[n-row+i-1] = rdiag[row+i] = true;
        // 递归子节点
        backtracking(ans, board, column, ldiag, rdiag, row+1, n);
        // 回改当前节点状态
        board[row][i] = '.';
        column[i] = ldiag[n-row+i-1] = rdiag[row+i] = false;
    }
}
```

6.4 广度优先搜索

广度优先搜索（breadth-first search，BFS）不同与深度优先搜索，它是一层层进行遍历的，因此需要用先入先出的队列而非先入后出的栈进行遍历。由于是按层次进行遍历，广度优先搜索时按照“广”的方向进行遍历的，也常常用来处理最短路径等问题。

考虑如下一颗简单的树。我们从 1 号节点开始遍历，假如遍历顺序是从左子节点到右子节点，那么按照优先向着“广”的方向前进的策略，队列顶端的元素变化过程为 [1]->[2->3]->[4]，其中方括号代表每一层的元素。

```
   1
  / \
 2   3
/
4
```

这里要注意，深度优先搜索和广度优先搜索都可以处理可达性问题，即从一个节点开始是否能达到另一个节点。因为深度优先搜索可以利用递归快速实现，很多人会习惯使用深度优先搜索刷此类题目。实际软件工程中，笔者很少见到递归的写法，因为一方面难以理解，另一方面可能产生栈溢出的情况；而用栈实现的深度优先搜索和用队列实现的广度优先搜索在写法上并没有太大差异，因此使用哪一种搜索方式需要根据实际的功能需求来判断。

934. Shortest Bridge (Medium)

题目描述

给定一个二维 0-1 矩阵，其中 1 表示陆地，0 表示海洋，每个位置与上下左右相连。已知矩阵中有且只有两个岛屿，求最少要填海造陆多少个位置才可以将两个岛屿相连。

输入输出样例

输入是一个二维整数数组，输出是一个非负整数，表示需要填海造陆的位置数。

```
Input:
[[1,1,1,1,1],
 [1,0,0,0,1],
 [1,0,1,0,1],
 [1,0,0,0,1],
 [1,1,1,1,1]]
Output: 1
```

题解

本题实际上是求两个岛屿间的最短距离，因此我们可以先通过任意搜索方法找到其中一个岛屿，然后利用广度优先搜索，查找其与另一个岛屿的最短距离。

```
vector<int> direction{-1, 0, 1, 0, -1};

// 主函数
int shortestBridge(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    queue<pair<int, int>> points;
    // dfs寻找第一个岛屿，并把1全部赋值为2
    bool flipped = false;
    for (int i = 0; i < m; ++i) {
        if (flipped) break;
        for (int j = 0; j < n; ++j) {
            if (grid[i][j] == 1) {
                dfs(points, grid, m, n, i, j);
                flipped = true;
                break;
            }
        }

    }
    // bfs寻找第二个岛屿，并把过程中经过的0赋值为2
    int x, y;
    int level = 0;
    while (!points.empty()){
        ++level;
        int n_points = points.size();
        while (n_points--) {
            auto [r, c] = points.front();
            points.pop();
            for (int k = 0; k < 4; ++k) {
                x = r + direction[k], y = c + direction[k+1];
                if (x >= 0 && y >= 0 && x < m && y < n) {
```

```
                if (grid[x][y] == 2) {
                    continue;
                }
                if (grid[x][y] == 1) {
                    return level;
                }
                points.push({x, y});
                grid[x][y] = 2;
            }
        }
    }
    return 0;
}

// 辅函数
void dfs(queue<pair<int, int>>& points, vector<vector<int>>& grid, int m, int n
    , int i, int j) {
    if (i < 0 || j < 0 || i == m || j == n || grid[i][j] == 2) {
        return;
    }
    if (grid[i][j] == 0) {
        points.push({i, j});
        return;
    }
    grid[i][j] = 2;
    dfs(points, grid, m, n, i - 1, j);
    dfs(points, grid, m, n, i + 1, j);
    dfs(points, grid, m, n, i, j - 1);
    dfs(points, grid, m, n, i, j + 1);
}
```

126. Word Ladder II (Hard)

题目描述

给定一个起始字符串和一个终止字符串，以及一个单词表，求是否可以将起始字符串每次改一个字符，直到改成终止字符串，且所有中间的修改过程表示的字符串都可以在单词表里找到。若存在，输出需要修改次数最少的所有更改方式。

输入输出样例

输入是两个字符串，输出是一个二维字符串数组，表示每种字符串修改方式。

```
Input: beginWord = "hit", endWord = "cog",
wordList = ["hot","dot","dog","lot","log","cog"]
Output:
[["hit","hot","dot","dog","cog"],
 ["hit","hot","lot","log","cog"]]
```

题解

我们可以把起始字符串、终止字符串、以及单词表里所有的字符串想象成节点。若两个字符串只有一个字符不同，那么它们相连。因为题目需要输出修改次数最少的所有修改方式，因此我们可以使用广度优先搜索，求得起始节点到终止节点的最短距离。

我们同时还使用了一个小技巧：我们并不是直接从起始节点进行广度优先搜索，直到找到终止节点为止；而是从起始节点和终止节点分别进行广度优先搜索，每次只延展当前层节点数最少的那一端，这样我们可以减少搜索的总结点数。举例来说，假设最短距离为 4，如果我们只从一端搜索 4 层，总遍历节点数最多是 $1+2+4+8+16=31$；而如果我们从两端各搜索两层，总遍历节点数最多只有 $2\times(1+2+4)=14$。

在搜索结束后，我们还需要通过回溯法来重建所有可能的路径。

```
// 主函数
vector<vector<string>> findLadders(string beginWord, string endWord, vector<
    string>& wordList) {
    vector<vector<string>> ans;
    unordered_set<string> dict;
    for (const auto &w: wordList){
        dict.insert(w);
    }
    if (!dict.count(endWord)) {
        return ans;
    }
    dict.erase(beginWord);
    dict.erase(endWord);
    unordered_set<string> q1{beginWord}, q2{endWord};
    unordered_map<string, vector<string>> next;
    bool reversed = false, found = false;
    while (!q1.empty()) {
        unordered_set<string> q;
        for (const auto &w: q1) {
            string s = w;
            for (size_t i = 0; i < s.size(); i++) {
                char ch = s[i];
                for (int j = 0; j < 26; j++) {
                    s[i] = j + 'a';
                    if (q2.count(s)) {
                        reversed? next[s].push_back(w): next[w].push_back(s);
                        found = true;
                    }
                    if (dict.count(s)) {
                        reversed? next[s].push_back(w): next[w].push_back(s);
                        q.insert(s);
                    }
                }
                s[i] = ch;
            }
        }
        if (found) {
            break;
        }
        for (const auto &w: q) {
            dict.erase(w);
        }
```

```
        if (q.size() <= q2.size()) {
            q1 = q;
        } else {
            reversed = !reversed;
            q1 = q2;
            q2 = q;
        }
    }
    if (found) {
        vector<string> path = {beginWord};
        backtracking(beginWord, endWord, next, path, ans);
    }
    return ans;
}

// 辅函数
void backtracking(const string &src, const string &dst, unordered_map<string,
    vector<string>> &next, vector<string> &path, vector<vector<string>> &ans) {
    if (src == dst) {
        ans.push_back(path);
        return;
    }
    for (const auto &s: next[src]) {
        path.push_back(s);
        backtracking(s, dst, next, path, ans);
        path.pop_back();
    }
}
```

6.5 练习

基础难度

130. Surrounded Regions (Medium)

先从最外侧填充，然后再考虑里侧。

257. Binary Tree Paths (Easy)

输出二叉树中所有从根到叶子的路径，回溯法使用与否有什么区别?

进阶难度

47. Permutations II (Medium)

排列题的 follow-up，如何处理重复元素?

40. Combination Sum II (Medium)

组合题的 follow-up，如何处理重复元素?

37. Sudoku Solver (Hard)

十分经典的数独题，可以利用回溯法求解。事实上对于数独类型的题，有很多进阶的搜索方法和剪枝策略可以提高速度，如启发式搜索。

310. Minimum Height Trees (Medium)

如何将这道题转为搜索类型题？是使用深度优先还是广度优先呢？

第 7 章 深入浅出动态规划

内容提要

- ❑ 算法解释
- ❑ 基本动态规划：一维
- ❑ 基本动态规划：二维
- ❑ 分割类型题
- ❑ 子序列问题
- ❑ 背包问题
- ❑ 字符串编辑
- ❑ 股票交易

7.1 算法解释

这里我们引用一下维基百科的描述：“动态规划（Dynamic Programming, DP）在查找有很多重叠子问题的情况的最优解时有效。它将问题重新组合成子问题。为了避免多次解决这些子问题，它们的结果都逐渐被计算并被保存，从简单的问题直到整个问题都被解决。因此，动态规划保存递归时的结果，因而不会在解决同样的问题时花费时间 …… 动态规划只能应用于有最优子结构的问题。最优子结构的意思是局部最优解能决定全局最优解（对有些问题这个要求并不能完全满足，故有时需要引入一定的近似）。简单地说，问题能够分解成子问题来解决。”

通俗一点来讲，动态规划和其它遍历算法（如深/广度优先搜索）都是将原问题拆成多个子问题然后求解，他们之间最本质的区别是，动态规划保存子问题的解，避免重复计算。解决动态规划问题的关键是找到状态转移方程，这样我们可以通过计算和储存子问题的解来求解最终问题。

同时，我们也可以对动态规划进行空间压缩，起到节省空间消耗的效果。这一技巧笔者将在之后的题目中介绍。

在一些情况下，动态规划可以看成是带有状态记录（memoization）的优先搜索。状态记录的意思为，如果一个子问题在优先搜索时已经计算过一次，我们可以把它的结果储存下来，之后遍历到该子问题的时候可以直接返回储存的结果。动态规划是自下而上的，即先解决子问题，再解决父问题；而用带有状态记录的优先搜索是自上而下的，即从父问题搜索到子问题，若重复搜索到同一个子问题则进行状态记录，防止重复计算。如果题目需求的是最终状态，那么使用动态搜索比较方便；如果题目需要输出所有的路径，那么使用带有状态记录的优先搜索会比较方便。

7.2 基本动态规划：一维

70. Climbing Stairs (Easy)

题目描述

给定 n 节台阶，每次可以走一步或走两步，求一共有多少种方式可以走完这些台阶。

输入输出样例

输入是一个数字，表示台阶数量；输出是爬台阶的总方式。

```
Input: 3
Output: 3
```

在这个样例中，一共有三种方法走完这三节台阶：每次走一步；先走一步，再走两步；先走两步，再走一步。

题解

这是十分经典的斐波那契数列题。定义一个数组 dp，dp[i] 表示走到第 i 阶的方法数。因为我们每次可以走一步或者两步，所以第 i 阶可以从第 i-1 或 i-2 阶到达。换句话说，走到第 i 阶的方法数即为走到第 i-1 阶的方法数加上走到第 i-2 阶的方法数。这样我们就得到了状态转移方程 dp[i] = dp[i-1] + dp[i-2]。注意边界条件的处理。

```
int climbStairs(int n) {
    if (n <= 2) return n;
    vector<int> dp(n + 1, 1);
    for (int i = 2; i <= n; ++i) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

进一步的，我们可以对动态规划进行空间压缩。因为 dp[i] 只与 dp[i-1] 和 dp[i-2] 有关，因此可以只用两个变量来存储 dp[i-1] 和 dp[i-2]，使得原来的 $O(n)$ 空间复杂度优化为 $O(1)$ 复杂度。

```
int climbStairs(int n) {
    if (n <= 2) return n;
    int pre2 = 1, pre1 = 2, cur;
    for (int i = 2; i < n; ++i) {
        cur = pre1 + pre2;
        pre2 = pre1;
        pre1 = cur;
    }
    return cur;
}
```

198. House Robber (Easy)

题目描述

假如你是一个劫匪，并且决定抢劫一条街上的房子，每个房子内的钱财数量各不相同。如果你抢了两栋相邻的房子，则会触发警报机关。求在不触发机关的情况下最多可以抢劫多少钱。

输入输出样例

输入是一个一维数组，表示每个房子的钱财数量；输出是劫匪可以最多抢劫的钱财数量。

```
Input: [2,7,9,3,1]
Output: 12
```

在这个样例中，最多的抢劫方式为抢劫第 1、3、5 个房子。

题解

定义一个数组 dp，dp[i] 表示抢劫到第 i 个房子时，可以抢劫的最大数量。我们考虑 dp[i]，此时可以抢劫的最大数量有两种可能，一种是我们选择不抢劫这个房子，此时累计的金额即为 dp[i-1]；另一种是我们选择抢劫这个房子，那么此前累计的最大金额只能是 dp[i-2]，因为我们不能够抢劫第 i-1 个房子，否则会触发警报机关。因此本题的状态转移方程为 dp[i] = max(dp[i-1], nums[i-1] + dp[i-2])。

```
int rob(vector<int>& nums) {
    if (nums.empty()) return 0;
    int n = nums.size();
    vector<int> dp(n + 1, 0);
    dp[1] = nums[0];
    for (int i = 2; i <= n; ++i) {
        dp[i] = max(dp[i-1], nums[i-1] + dp[i-2]);
    }
    return dp[n];
}
```

同样的，我们可以像题目 70 那样，对空间进行压缩。

```
int rob(vector<int>& nums) {
    if (nums.empty()) return 0;
    int n = nums.size();
    if (n == 1) return nums[0];
    int pre2 = 0, pre1 = 0, cur;
    for (int i = 0; i < n; ++i) {
        cur = max(pre2 + nums[i], pre1);
        pre2 = pre1;
        pre1 = cur;
    }
    return cur;
}
```

413. Arithmetic Slices (Medium)

题目描述

给定一个数组，求这个数组中连续且等差的子数组一共有多少个。

输入输出样例

输入是一个一维数组，输出是满足等差条件的连续字数组个数。

```
Input: nums = [1,2,3,4]
Output: 3
```

在这个样例中，等差数列有 [1,2,3]、[2,3,4] 和 [1,2,3,4]。

题解

这道题略微特殊，因为要求是等差数列，可以很自然的想到子数组必定满足 num[i] - num[i-1] = num[i-1] - num[i-2]。然而由于我们对于 dp 数组的定义通常为以 i 结尾的，满足某些条件的子数组数量，而等差子数组可以在任意一个位置终结，因此此题在最后需要对 dp 数组求和。

```
int numberOfArithmeticSlices(vector<int>& nums) {
    int n = nums.size();
    if (n < 3) return 0;
    vector<int> dp(n, 0);
    for (int i = 2; i < n; ++i) {
        if (nums[i] - nums[i-1] == nums[i-1] - nums[i-2]) {
            dp[i] = dp[i-1] + 1;
        }
    }
    return accumulate(dp.begin(), dp.end(), 0);
}
```

7.3 基本动态规划：二维

64. Minimum Path Sum (Medium)

题目描述

给定一个 $m \times n$ 大小的非负整数矩阵，求从左上角开始到右下角结束的、经过的数字的和最小的路径。每次只能向右或者向下移动。

输入输出样例

输入是一个二维数组，输出是最优路径的数字和。

```
Input:
[[1,3,1],
 [1,5,1],
 [4,2,1]]
Output: 7
```

在这个样例中，最短路径为 1->3->1->1->1。

题解

我们可以定义一个同样是二维的 dp 数组，其中 dp[i][j] 表示从左上角开始到 (i, j) 位置的最优路径的数字和。因为每次只能向下或者向右移动，我们可以很容易得到状态转移方程 dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j]，其中 grid 表示原数组。

```
int minPathSum(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    vector<vector<int>> dp(m, vector<int>(n, 0));
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
```

```
            if (i == 0 && j == 0) {
                dp[i][j] = grid[i][j];
            } else if (i == 0) {
                dp[i][j] = dp[i][j-1] + grid[i][j];
            } else if (j == 0) {
                dp[i][j] = dp[i-1][j] + grid[i][j];
            } else {
                dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j];
            }
        }
    }
    return dp[m-1][n-1];
}
```

因为 dp 矩阵的每一个值只和左边和上面的值相关，我们可以使用空间压缩将 dp 数组压缩为一维。对于第 i 行，在遍历到第 j 列的时候，因为第 j-1 列已经更新过了，所以 dp[j-1] 代表 dp[i][j-1] 的值；而 dp[j] 待更新，当前存储的值是在第 i-1 行的时候计算的，所以代表 dp[i-1][j] 的值。

注意 如果不是很熟悉空间压缩技巧，笔者推荐您优先尝试写出非空间压缩的解法，如果时间充裕且力所能及再进行空间压缩。

```
int minPathSum(vector<vector<int>>& grid) {
    int m = grid.size(), n = grid[0].size();
    vector<int> dp(n, 0);
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (i == 0 && j == 0) {
                dp[j] = grid[i][j];
            } else if (i == 0) {
                dp[j] = dp[j-1] + grid[i][j];
            } else if (j == 0) {
                dp[j] = dp[j] + grid[i][j];
            } else {
                dp[j] = min(dp[j], dp[j-1]) + grid[i][j];
            }
        }
    }
    return dp[n-1];
}
```

542. 01 Matrix (Medium)

题目描述

给定一个由 0 和 1 组成的二维矩阵，求每个位置到最近的 0 的距离。

输入输出样例

输入是一个二维 0-1 数组，输出是一个同样大小的非负整数数组，表示每个位置到最近的 0 的距离。

```
Input:
[[0,0,0],
```

```
 [0,1,0],
 [1,1,1]]

Output:
[[0,0,0],
 [0,1,0],
 [1,2,1]]
```

题解

一般来说，因为这道题涉及到四个方向上的最近搜索，所以很多人的第一反应可能会是广度优先搜索。但是对于一个大小 $O(mn)$ 的二维数组，对每个位置进行四向搜索，最坏情况的时间复杂度（即全是 1）会达到恐怖的 $O(m^2n^2)$。一种办法是使用一个 dp 数组做 memoization，使得广度优先搜索不会重复遍历相同位置；另一种更简单的方法是，我们从左上到右下进行一次动态搜索，再从右下到左上进行一次动态搜索。两次动态搜索即可完成四个方向上的查找。

```
vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
    if (matrix.empty()) return {};
    int n = matrix.size(), m = matrix[0].size();
    vector<vector<int>> dp(n, vector<int>(m, INT_MAX - 1));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            if (matrix[i][j] == 0) {
                dp[i][j] = 0;
            } else {
                if (j > 0) {
                    dp[i][j] = min(dp[i][j], dp[i][j-1] + 1);
                }
                if (i > 0) {
                    dp[i][j] = min(dp[i][j], dp[i-1][j] + 1);
                }
            }
        }
    }
    for (int i = n - 1; i >= 0; --i) {
        for (int j = m - 1; j >= 0; --j) {
            if (matrix[i][j] != 0) {
                if (j < m - 1) {
                    dp[i][j] = min(dp[i][j], dp[i][j+1] + 1);
                }
                if (i < n - 1) {
                    dp[i][j] = min(dp[i][j], dp[i+1][j] + 1);
                }
            }
        }
    }
    return dp;
}
```

221. Maximal Square (Medium)

题目描述

给定一个二维的 0-1 矩阵，求全由 1 构成的最大正方形面积。

输入输出样例

输入是一个二维 0-1 数组，输出是最大正方形面积。

```
Input:
[["1","0","1","0","0"],
 ["1","0","1","1","1"],
 ["1","1","1","1","1"],
 ["1","0","0","1","0"]]
Output: 4
```

题解

对于在矩阵内搜索正方形或长方形的题型，一种常见的做法是定义一个二维 dp 数组，其中 dp[i][j] 表示满足题目条件的、以 (i, j) 为右下角的正方形或者长方形的属性。对于本题，则表示以 (i, j) 为右下角的全由 1 构成的最大正方形面积。如果当前位置是 0，那么 dp[i][j] 即为 0；如果当前位置是 1，我们假设 dp[i][j] = k^2，其充分条件为 dp[i-1][j-1]、dp[i][j-1] 和 dp[i-1][j] 的值必须都不小于 $(k-1)^2$，否则 (i, j) 位置不可以构成一个边长为 k 的正方形。同理，如果这三个值中的的最小值为 $k-1$，则 (i, j) 位置一定且最大可以构成一个边长为 k 的正方形。

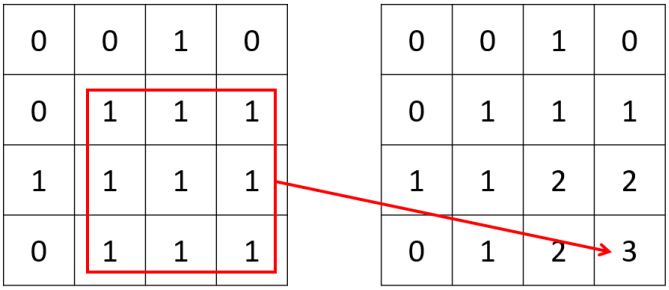

图 7.1: 题目 542 - 左边为一个 0-1 矩阵，右边为其对应的 dp 矩阵，我们可以发现最大的正方形边长为 3

```
int maximalSquare(vector<vector<char>>& matrix) {
    if (matrix.empty() || matrix[0].empty()) {
        return 0;
    }
    int m = matrix.size(), n = matrix[0].size(), max_side = 0;
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (matrix[i-1][j-1] == '1') {
                dp[i][j] = min(dp[i-1][j-1], min(dp[i][j-1], dp[i-1][j])) + 1;
            }
            max_side = max(max_side, dp[i][j]);
        }
    }
    return max_side * max_side;
```

```
}
```

7.4 分割类型题

279. Perfect Squares (Medium)

题目描述

给定一个正整数，求其最少可以由几个完全平方数相加构成。

输入输出样例

输入是给定的正整数，输出也是一个正整数，表示输入的数字最少可以由几个完全平方数相加构成。

```
Input: n = 13
Output: 2
```

在这个样例中，13 的最少构成方法为 4+9。

题解

对于分割类型题，动态规划的状态转移方程通常并不依赖相邻的位置，而是依赖于满足分割条件的位置。我们定义一个一维矩阵 dp，其中 dp[i] 表示数字 i 最少可以由几个完全平方数相加构成。在本题中，位置 i 只依赖 i - k^2 的位置，如 i - 1、i - 4、i - 9 等等，才能满足完全平方分割的条件。因此 dp[i] 可以取的最小值即为 1 + min(dp[i-1], dp[i-4], dp[i-9] $\cdots$)。

```
int numSquares(int n) {
    vector<int> dp(n + 1, INT_MAX);
    dp[0] = 0;
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j * j <= i; ++j) {
            dp[i] = min(dp[i], dp[i-j*j] + 1);
        }
    }
    return dp[n];
}
```

91. Decode Ways (Medium)

题目描述

已知字母 A-Z 可以表示成数字 1-26。给定一个数字串，求有多少种不同的字符串等价于这个数字串。

输入输出样例

输入是一个由数字组成的字符串，输出是满足条件的解码方式总数。

```
Input: "226"
Output: 3
```

在这个样例中，有三种解码方式：BZ(2 26)、VF(22 6) 或 BBF(2 2 6)。

题解

这是一道很经典的动态规划题，难度不大但是十分考验耐心。这是因为只有 1-26 可以表示字母，因此对于一些特殊情况，比如数字 0 或者当相邻两数字大于 26 时，需要有不同的状态转移方程，详见如下代码。

```
int numDecodings(string s) {
    int n = s.length();
    if (n == 0) return 0;
    int prev = s[0] - '0';
    if (!prev) return 0;
    if (n == 1) return 1;
    vector<int> dp(n + 1, 1);
    for (int i = 2; i <= n; ++i) {
        int cur = s[i-1] - '0';
        if ((prev == 0 || prev > 2) && cur == 0) {
            return 0;
        }
        if ((prev < 2 && prev > 0) || prev == 2 && cur < 7) {
            if (cur) {
                dp[i] = dp[i-2] + dp[i-1];
            } else {
                dp[i] = dp[i-2];
            }
        } else {
            dp[i] = dp[i-1];
        }
        prev = cur;
    }
    return dp[n];
}
```

139. Word Break (Medium)

题目描述

给定一个字符串和一个字符串集合，求是否存在一种分割方式，使得原字符串分割后的子字符串都可以在集合内找到。

输入输出样例

```
Input: s = "applepenapple", wordDict = ["apple", "pen"]
Output: true
```

在这个样例中，字符串可以被分割为 [“apple” , “pen” , “apple”]。

题解

类似于完全平方数分割问题，这道题的分割条件由集合内的字符串决定，因此在考虑每个分割位置时，需要遍历字符串集合，以确定当前位置是否可以成功分割。注意对于位置 0，需要初始化值为真。

```
bool wordBreak(string s, vector<string>& wordDict) {
    int n = s.length();
    vector<bool> dp(n + 1, false);
    dp[0] = true;
    for (int i = 1; i <= n; ++i) {
        for (const string & word: wordDict) {
            int len = word.length();
            if (i >= len && s.substr(i - len, len) == word) {
                dp[i] = dp[i] || dp[i - len];
            }
        }
    }
    return dp[n];
}
```

7.5 子序列问题

300. Longest Increasing Subsequence (Medium)

题目描述

给定一个未排序的整数数组，求最长的递增子序列。

注意 按照 LeetCode 的习惯，子序列（subsequence）不必连续，子数组（subarray）或子字符串（substring）必须连续。

输入输出样例

输入是一个一维数组，输出是一个正整数，表示最长递增子序列的长度。

```
Input: [10,9,2,5,3,7,101,18]
Output: 4
```

在这个样例中，最长递增子序列之一是 [2,3,7,18]。

题解

对于子序列问题，第一种动态规划方法是，定义一个 dp 数组，其中 dp[i] 表示以 i 结尾的子序列的性质。在处理好每个位置后，统计一遍各个位置的结果即可得到题目要求的结果。

在本题中，dp[i] 可以表示以 i 结尾的、最长子序列长度。对于每一个位置 i，如果其之前的某个位置 j 所对应的数字小于位置 i 所对应的数字，则我们可以获得一个以 i 结尾的、长度为 dp[j] + 1 的子序列。为了遍历所有情况，我们需要 i 和 j 进行两层循环，其时间复杂度为 $O(n^2)$。

```
int lengthOfLIS(vector<int>& nums) {
    int max_length = 0, n = nums.size();
    if (n <= 1) return n;
    vector<int> dp(n, 1);
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < i; ++j) {
            if (nums[i] > nums[j]) {
                dp[i] = max(dp[i], dp[j] + 1);
            }
        }
        max_length = max(max_length, dp[i]);
    }
    return max_length;
}
```

本题还可以使用二分查找将时间复杂度降低为 $O(n \log n)$。我们定义一个 dp 数组，其中 dp[k] 存储长度为 k+1 的最长递增子序列的最后一个数字。我们遍历每一个位置 i，如果其对应的数字大于 dp 数组中所有数字的值，那么我们把它放在 dp 数组尾部，表示最长递增子序列长度加 1；如果我们发现这个数字在 dp 数组中比数字 a 大、比数字 b 小，则我们将 b 更新为此数字，使得之后构成递增序列的可能性增大。以这种方式维护的 dp 数组永远是递增的，因此可以用二分查找加速搜索。

以样例为例，对于数组 [10,9,2,5,3,7,101,18]，我们每轮的更新查找情况为：

```
num     dp
10      [10]
9       [9]
2       [2]
5       [2,5]
3       [2,3]
7       [2,3,7]
101     [2,3,7,101]
18      [2,3,7,18]
```

最终我们就获得了 [2,3,7,18] 这个最长递增数组之一。该算法的代码实现如下。

```
int lengthOfLIS(vector<int>& nums) {
    int n = nums.size();
    if (n <= 1) return n;
    vector<int> dp;
    dp.push_back(nums[0]);
    for (int i = 1; i < n; ++i) {
        if (dp.back() < nums[i]) {
            dp.push_back(nums[i]);
        } else {
            auto itr = lower_bound(dp.begin(), dp.end(), nums[i]);
            *itr = nums[i];
        }
    }
    return dp.size();
```

```
}
```

1143. Longest Commom Subsequence (Medium)

题目描述

给定两个字符串，求它们最长的公共子序列长度。

输入输出样例

输入是两个字符串，输出是一个整数，表示它们满足题目条件的长度。

```
Input: text1 = "abcde", text2 = "ace"
Output: 3
```

在这个样例中，最长公共子序列是“ace”。

题解

对于子序列问题，第二种动态规划方法是，定义一个 dp 数组，其中 dp[i] 表示到位置 i 为止的子序列的性质，并不必须以 i 结尾。这样 dp 数组的最后一位结果即为题目所求，不需要再对每个位置进行统计。

在本题中，我们可以建立一个二维数组 dp，其中 dp[i][j] 表示到第一个字符串位置 i 为止、到第二个字符串位置 j 为止、最长的公共子序列长度。这样一来我们就可以很方便地分情况讨论这两个位置对应的字母相同与不同的情况了。

```
int longestCommonSubsequence(string text1, string text2) {
    int m = text1.length(), n = text2.length();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (text1[i-1] == text2[j-1]) {
                dp[i][j] = dp[i-1][j-1] + 1;
            } else {
                dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
            }
        }
    }
    return dp[m][n];
}
```

7.6 背包问题

背包问题是一种组合优化的 NP 完全问题：有 N 个物品和容量为 W 的背包，每个物品都有自己的体积 w 和价值 v，求拿哪些物品可以使得背包所装下物品的总价值最大。如果限定每种物品只能选择 0 个或 1 个，则问题称为 0-1 背包问题；如果不限定每种物品的数量，则问题称为无界背包问题或完全背包问题。

我们可以用动态规划来解决背包问题。以 0-1 背包问题为例。我们可以定义一个二维数组 dp 存储最大价值，其中 dp[i][j] 表示前 i 件物品体积不超过 j 的情况下能达到的最大价值。在我们遍历到第 i 件物品时，在当前背包总容量为 j 的情况下，如果我们不将物品 i 放入背包，那么 dp[i][j] = dp[i-1][j]，即前 i 个物品的最大价值等于只取前 i-1 个物品时的最大价值；如果我们将物品 i 放入背包，假设第 i 件物品体积为 w，价值为 v，那么我们得到 dp[i][j] = dp[i-1][j-w] + v。我们只需在遍历过程中对这两种情况取最大值即可，总时间复杂度和空间复杂度都为 $O(NW)$。

```
int knapsack(vector<int> weights, vector<int> values, int N, int W) {
    vector<vector<int>> dp(N + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= N; ++i) {
        int w = weights[i-1], v = values[i-1];
        for (int j = 1; j <= W; ++j) {
            if (j >= w) {
                dp[i][j] = max(dp[i-1][j], dp[i-1][j-w] + v);
            } else {
                dp[i][j] = dp[i-1][j];
            }
        }
    }
    return dp[N][W];
}
```

dp[i][j]	j = 1	j = 2	j = 3	j = 4	j = 5
i = 0					
i = 1					
i = 2					
i = 3					
i = 4					

图 7.2: 0-1 背包问题 - 状态转移矩阵样例

我们可以进一步对 0-1 背包进行空间优化，将空间复杂度降低为 $O(W)$。如图所示，假设我们目前考虑物品 i = 2，且其体积为 w = 2，价值为 v = 3；对于背包容量 j，我们可以得到 dp[2][j] = max(dp[1][j], dp[1][j-2] + 3)。这里可以发现我们永远只依赖于上一排 i = 1 的信息，之前算过的其他物品都不需要再使用。因此我们可以去掉 dp 矩阵的第一个维度，在考虑物品 i 时变成 dp[j] = max(dp[j], dp[j-w] + v)。这里要注意我们在遍历每一行的时候必须逆向遍历，这样才能够调用上一行物品 i-1 时 dp[j-w] 的值；若按照从左往右的顺序进行正向遍历，则 dp[j-w] 的值在遍历到 j 之前就已经被更新成物品 i 的值了。

```
int knapsack(vector<int> weights, vector<int> values, int N, int W) {
    vector<int> dp(W + 1, 0);
    for (int i = 1; i <= N; ++i) {
        int w = weights[i-1], v = values[i-1];
        for (int j = W; j >= w; --j) {
            dp[j] = max(dp[j], dp[j-w] + v);
        }
    }
    return dp[W];
```

```
}
```

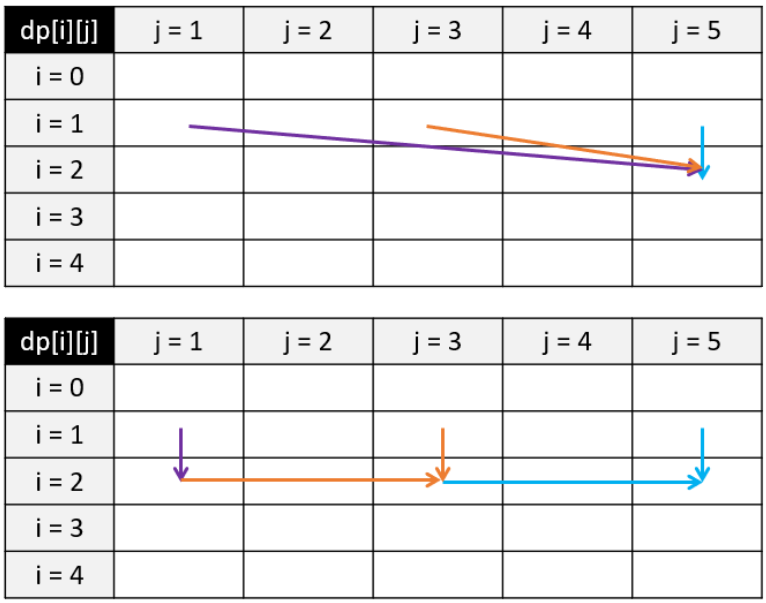

图 7.3: 完全背包问题 - 状态转移矩阵样例

在完全背包问题中，一个物品可以拿多次。如图上半部分所示，假设我们遍历到物品 i = 2，且其体积为 w = 2，价值为 v = 3；对于背包容量 j = 5，最多只能装下 2 个该物品。那么我们的状态转移方程就变成了 dp[2][5] = max(dp[1][5], dp[1][3] + 3, dp[1][1] + 6)。如果采用这种方法，假设背包容量无穷大而物体的体积无穷小，我们这里的比较次数也会趋近于无穷大，远超 $O(NW)$ 的时间复杂度。

怎么解决这个问题呢？我们发现在 dp[2][3] 的时候我们其实已经考虑了 dp[1][3] 和 dp[2][1] 的情况，而在时 dp[2][1] 也已经考虑了 dp[1][1] 的情况。因此，如图下半部分所示，对于拿多个物品的情况，我们只需考虑 dp[2][3] 即可，即 dp[2][5] = max(dp[1][5], dp[2][3] + 3)。这样，我们就得到了完全背包问题的状态转移方程：dp[i][j] = max(dp[i-1][j], dp[i][j-w] + v)，其与 0-1 背包问题的差别仅仅是把状态转移方程中的第二个 i-1 变成了 i。

```
int knapsack(vector<int> weights, vector<int> values, int N, int W) {
    vector<vector<int>> dp(N + 1, vector<int>(W + 1, 0));
    for (int i = 1; i <= N; ++i) {
        int w = weights[i-1], v = values[i-1];
        for (int j = 1; j <= W; ++j) {
            if (j >= w) {
                dp[i][j] = max(dp[i-1][j], dp[i][j-w] + v);
            } else {
                dp[i][j] = dp[i-1][j];
            }
        }
    }
    return dp[N][W];
}
```

同样的，我们也可以利用空间压缩将时间复杂度降低为 $O(W)$。这里要注意我们在遍历每一行的时候必须正向遍历，因为我们需要利用当前物品在第 j-w 列的信息。

```
int knapsack(vector<int> weights, vector<int> values, int N, int W) {
    vector<int> dp(W + 1, 0);
    for (int i = 1; i <= N; ++i) {
        int w = weights[i-1], v = values[i-1];
        for (int j = w; j <= W; ++j) {
            dp[j] = max(dp[j], dp[j-w] + v);
        }
    }
    return dp[W];
}
```

注意 压缩空间时到底需要正向还是逆向遍历呢？物品和体积哪个放在外层，哪个放在内层呢？这取决于状态转移方程的依赖关系。在思考空间压缩前，不妨将状态转移矩阵画出来，方便思考如何进行空间压缩。

如果您实在不想仔细思考，这里有个简单的口诀：0-1 背包对物品的迭代放在外层，里层的体积或价值逆向遍历；完全背包对物品的迭代放在里层，外层的体积或价值正向遍历。

416. Partition Equal Subset Sum (Medium)

题目描述

给定一个正整数数组，求是否可以把这个数组分成和相等的两部分。

输入输出样例

输入是一个一维正整数数组，输出时一个布尔值，表示是否可以满足题目要求。

```
Input: [1,5,11,5]
Output: true
```

在这个样例中，满足条件的分割方法是 [1,5,5] 和 [11]。

题解

本题等价于 0-1 背包问题，设所有数字和为 sum，我们的目标是选取一部分物品，使得它们的总和为 sum/2。这道题不需要考虑价值，因此我们只需要通过一个布尔值矩阵来表示状态转移矩阵。注意边界条件的处理。

```
bool canPartition(vector<int> &nums) {
    int sum = accumulate(nums.begin(), nums.end(), 0);
    if (sum % 2) return false;
    int target = sum / 2, n = nums.size();
    vector<vector<bool>> dp(n + 1, vector<bool>(target + 1, false));
    dp[0][0] = true;
    for (int i = 1; i <= n; ++i) {
        for (int j = 0; j <= target; ++j) {
            if (j < nums[i-1]) {
                dp[i][j] = dp[i-1][j];
            } else {
                dp[i][j] = dp[i-1][j] || dp[i-1][j-nums[i-1]];
            }
```

```
        }
    }
    return dp[n][target];
}
```

同样的，我们也可以对本题进行空间压缩。注意对数字和的遍历需要逆向。

```
bool canPartition(vector<int> &nums) {
    int sum = accumulate(nums.begin(), nums.end(), 0);
    if (sum % 2) return false;
    int target = sum / 2, n = nums.size();
    vector<bool> dp(target + 1, false);
    dp[0] = true;
    for (int i = 1; i <= n; ++i) {
        for (int j = target; j >= nums[i-1]; --j) {
            dp[j] = dp[j] || dp[j-nums[i-1]];
        }
    }
    return dp[target];
}
```

474. Ones and Zeroes (Medium)

题目描述

给定 m 个数字 0 和 n 个数字 1，以及一些由 0-1 构成的字符串，求利用这些数字最多可以构成多少个给定的字符串，字符串只可以构成一次。

输入输出样例

输入两个整数 m 和 n，表示 0 和 1 的数量，以及一个一维字符串数组，表示待构成的字符串；输出是一个整数，表示最多可以生成的字符串个数。

```
Input: Array = {"10", "0001", "111001", "1", "0"}, m = 5, n = 3
Output: 4
```

在这个样例中，我们可以用 5 个 0 和 3 个 1 构成 [“10”，“0001”，“1”，“0”]。

题解

这是一个多维费用的 0-1 背包问题，有两个背包大小，0 的数量和 1 的数量。我们在这里直接展示三维空间压缩到二维后的写法。

```
// 主函数
int findMaxForm(vector<string>& strs, int m, int n) {
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (const string & str: strs) {
        auto [count0, count1] = count(str);
        for (int i = m; i >= count0; --i) {
            for (int j = n; j >= count1; --j) {
                dp[i][j] = max(dp[i][j], 1 + dp[i-count0][j-count1]);
```

```
            }
        }
    }
    return dp[m][n];
}

// 辅函数
pair<int, int> count(const string & s){
    int count0 = s.length(), count1 = 0;
    for (const char & c: s) {
        if (c == '1') {
            ++count1;
            --count0;
        }
    }
    return make_pair(count0, count1);
}
```

322. Coin Change (Medium)

题目描述

给定一些硬币的面额，求最少可以用多少颗硬币组成给定的金额。

输入输出样例

输入一个一维整数数组，表示硬币的面额；以及一个整数，表示给定的金额。输出一个整数，表示满足条件的最少的硬币数量。若不存在解，则返回-1。

```
Input: coins = [1, 2, 5], amount = 11
Output: 3
```

在这个样例中，最少的组合方法是 11 = 5 + 5 + 1。

题解

因为每个硬币可以用无限多次，这道题本质上是完全背包。我们直接展示二维空间压缩为一维的写法。

这里注意，我们把 dp 数组初始化为 amount + 1 而不是-1 的原因是，在动态规划过程中有求最小值的操作，如果初始化成-1 则会导致结果始终为-1。至于为什么取这个值，是因为 i 最大可以取 amount，而最多的组成方式是只用 1 元硬币，因此 amount + 1 一定大于所有可能的组合方式，取最小值时一定不会是它。在动态规划完成后，若结果仍然是此值，则说明不存在满足条件的组合方法，返回-1。

```
int coinChange(vector<int>& coins, int amount) {
    if (coins.empty()) return -1;
    vector<int> dp(amount + 1, amount + 1);
    dp[0] = 0;
    for (int i = 1; i <= amount; ++i) {
        for (const int & coin : coins) {
            if (i >= coin) {
```

```
                dp[i] = min(dp[i], dp[i-coin] + 1);
            }
        }
    }
    return dp[amount] == amount + 1? -1: dp[amount];
}
```

7.7 字符串编辑

72. Edit Distance (Hard)

题目描述

给定两个字符串，已知你可以删除、替换和插入任意字符串的任意字符，求最少编辑几步可以将两个字符串变成相同。

输入输出样例

输入是两个字符串，输出是一个整数，表示最少的步骤。

```
Input: word1 = "horse", word2 = "ros"
Output: 3
```

在这个样例中，一种最优编辑方法是（1）horse -> rorse （2）rorse -> rose（3）rose -> ros。

题解

类似于题目 1143，我们使用一个二维数组 dp[i][j]，表示将第一个字符串到位置 i 为止，和第二个字符串到位置 j 为止，最多需要几步编辑。当第 i 位和第 j 位对应的字符相同时，dp[i][j] 等于 dp[i-1][j-1]；当二者对应的字符不同时，修改的消耗是 dp[i-1][j-1]+1，插入 i 位置/删除 j 位置的消耗是 dp[i][j-1] + 1，插入 j 位置/删除 i 位置的消耗是 dp[i-1][j] + 1。

```
int minDistance(string word1, string word2) {
    int m = word1.length(), n = word2.length();
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (int i = 0; i <= m; ++i) {
        for (int j = 0; j <= n; ++j) {
            if (i == 0) {
                dp[i][j] = j;
            } else if (j == 0) {
                dp[i][j] = i;
            } else {
                dp[i][j] = min(
                    dp[i-1][j-1] + ((word1[i-1] == word2[j-1])? 0: 1),
                    min(dp[i-1][j] + 1, dp[i][j-1] + 1));
            }
        }
    }
    return dp[m][n];
}
```

650. 2 Keys Keyboard (Medium)

题目描述

给定一个字母 A，已知你可以每次选择复制全部字符，或者粘贴之前复制的字符，求最少需要几次操作可以把字符串延展到指定长度。

输入输出样例

输入是一个正整数，代表指定长度；输出是一个整数，表示最少操作次数。

```
Input: 3
Output: 3
```

在这个样例中，一种最优的操作方法是先复制一次，再粘贴两次。

题解

不同于以往通过加减实现的动态规划，这里需要乘除法来计算位置，因为粘贴操作是倍数增加的。我们使用一个一维数组 dp，其中位置 i 表示延展到长度 i 的最少操作次数。对于每个位置 j，如果 j 可以被 i 整除，那么长度 i 就可以由长度 j 操作得到，其操作次数等价于把一个长度为 1 的 A 延展到长度为 i/j。因此我们可以得到递推公式 dp[i] = dp[j] + dp[i/j]。

```
int minSteps(int n) {
    vector<int> dp(n + 1);
    for (int i = 2; i <= n; ++i) {
        dp[i] = i;
        for (int j = 2; j * j <= i; ++j) {
            if (i % j == 0) {
                dp[i] = dp[j] + dp[i/j];
                break;
            }
        }
    }
    return dp[n];
}
```

10. Regular Expression Matching (Hard)

题目描述

给定一个字符串和一个正则表达式（regular expression, regex），求该字符串是否可以被匹配。

输入输出样例

输入是一个待匹配字符串和一个用字符串表示的正则表达式，输出是一个布尔值，表示是否可以匹配成功。

```
Input: s = "aab", p = "c*a*b"
Output: true
```

在个样例中，我们可以重复 c 零次，重复 a 两次。

题解

我们可以使用一个二维数组 dp，其中 dp[i][j] 表示以 i 截止的字符串是否可以被以 j 截止的正则表达式匹配。根据正则表达式的不同情况，即字符、星号，点号，我们可以分情况讨论来更新 dp 数组，其具体代码如下。

```
bool isMatch(string s, string p) {
    int m = s.size(), n = p.size();
    vector<vector<bool>> dp(m + 1, vector<bool>(n + 1, false));
    dp[0][0] = true;
    for (int i = 1; i < n + 1; ++i) {
        if (p[i-1] == '*') {
            dp[0][i] = dp[0][i-2];
        }
    }
    for (int i = 1; i < m + 1; ++i) {
        for (int j = 1; j < n + 1; ++j) {
            if (p[j-1] == '.') {
                dp[i][j] = dp[i-1][j-1];
            } else if (p[j-1] != '*') {
                dp[i][j] = dp[i-1][j-1] && p[j-1] == s[i-1];
            } else if (p[j-2] != s[i-1] && p[j-2] != '.') {
                dp[i][j] = dp[i][j-2];
            } else {
                dp[i][j] = dp[i][j-1] || dp[i-1][j] || dp[i][j-2];
            }
        }
    }
    return dp[m][n];
}
```

7.8 股票交易

股票交易类问题通常可以用动态规划来解决。对于稍微复杂一些的股票交易类问题，比如需要冷却时间或者交易费用，则可以用通过动态规划实现的状态机来解决。

121. Best Time to Buy and Sell Stock (Easy)

题目描述

给定一段时间内每天某只股票的固定价格，已知你只可以买卖各一次，求最大的收益。

输入输出样例

输入一个一维整数数组，表示每天的股票价格；输出一个整数，表示最大的收益。

```
Input: [7,1,5,3,6,4]
Output: 5
```

在这个样例中，最大的利润为在第二天价格为 1 时买入，在第五天价格为 6 时卖出。

题解

我们可以遍历一遍数组，在每一个位置 i 时，记录 i 位置之前所有价格中的最低价格，然后将当前的价格作为售出价格，查看当前收益是不是最大收益即可。

```
int maxProfit(vector<int>& prices) {
    int sell = 0, buy = INT_MIN;
    for (int i = 0; i < prices.size(); ++i) {
        buy = max(buy, -prices[i]);
        sell = max(sell, buy + prices[i]);
    }
    return sell;
}
```

188. Best Time to Buy and Sell Stock IV (Hard)

题目描述

给定一段时间内每天某只股票的固定价格，已知你只可以买卖各 k 次，且每次只能拥有一支股票，求最大的收益。

输入输出样例

输入一个一维整数数组，表示每天的股票价格；以及一个整数，表示可以买卖的次数 k。输出一个整数，表示最大的收益。

```
Input: [3,2,6,5,0,3], k = 2
Output: 7
```

在这个样例中，最大的利润为在第二天价格为 2 时买入，在第三天价格为 6 时卖出；再在第五天价格为 0 时买入，在第六天价格为 3 时卖出。

题解

如果 k 大于总天数的一半，那么我们一旦发现可以赚钱就可以进行买卖；这里一半的原因是因为当天股价是不变的，因此一次买卖需要两天。如果 k 小于总天数，我们可以建立两个动态规划数组 buy 和 sell，对于每天的股票价格，buy[j] 表示在第 j 次买入时的最大收益，sell[j] 表示在第 j 次卖出时的最大收益。

```
// 主函数
int maxProfit(int k, vector<int>& prices) {
    int days = prices.size();
    if (days < 2) {
        return 0;
    }
    if (k * 2 >= days) {
        return maxProfitUnlimited(prices);
    }
    vector<int> buy(k + 1, INT_MIN), sell(k + 1, 0);
    for (int i = 0; i < days; ++i) {
```

```
        for (int j = 1; j <= k; ++j) {
            buy[j] = max(buy[j], sell[j-1] - prices[i]);
            sell[j] = max(sell[j], buy[j] + prices[i]);
        }
    }
    return sell[k];
}

// 辅函数
int maxProfitUnlimited(vector<int> prices) {
    int maxProfit = 0;
    for (int i = 1; i < prices.size(); ++i) {
        if (prices[i] > prices[i-1]) {
            maxProfit += prices[i] - prices[i-1];
        }
    }
    return maxProfit;
}
```

309. Best Time to Buy and Sell Stock with Cooldown (Medium)

题目描述

给定一段时间内每天某只股票的固定价格，已知每次卖出之后必须冷却一天，且每次只能拥有一支股票，求最大的收益。

输入输出样例

输入一个一维整数数组，表示每天的股票价格；输出一个整数，表示最大的收益。

```
Input: [1,2,3,0,2]
Output: 3
```

在这个样例中，最大的利润获取操作是买入、卖出、冷却、买入、卖出。

题解

我们可以使用状态机来解决这类复杂的状态转移问题，通过建立多个状态以及它们的转移方式，我们可以很容易地推导出各个状态的转移方程。如图所示，我们可以建立四个状态来表示带有冷却的股票交易，以及它们的之间的转移方式。

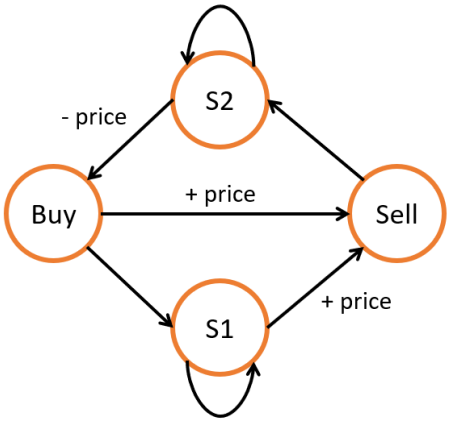

图 7.4: 题目 309 - 状态机状态转移

```
int maxProfit(vector<int>& prices) {
    int n = prices.size();
    if (n == 0) {
        return 0;
    }
    vector<int> buy(n), sell(n), s1(n), s2(n);
    s1[0] = buy[0] = -prices[0];
    sell[0] = s2[0] = 0;
    for (int i = 1; i < n; i++) {
        buy[i] = s2[i-1] - prices[i];
        s1[i] = max(buy[i-1], s1[i-1]);
        sell[i] = max(buy[i-1], s1[i-1]) + prices[i];
        s2[i] = max(s2[i-1], sell[i-1]);
    }
    return max(sell[n-1], s2[n-1]);
}
```

7.9 练习

基础难度

213. House Robber II (Medium)

强盗抢劫题目的 follow-up，如何处理环形数组呢?

53. Maximum Subarray (Easy)

经典的一维动态规划题目，试着把一维空间优化为常量吧。

343. Integer Break (Medium)

分割类型题，先尝试用动态规划求解，再思考是否有更简单的解法。

583. Delete Operation for Two Strings (Medium)

最长公共子序列的变种题。

进阶难度

646. Maximum Length of Pair Chain (Medium)

最长递增子序列的变种题，同样的，尝试用二分进行加速。

376. Wiggle Subsequence (Medium)

最长摆动子序列，通项公式比较特殊，需要仔细思考。

494. Target Sum (Medium)

如果告诉你这道题是 0-1 背包，你是否会有一些思路?

714. Best Time to Buy and Sell Stock with Transaction Fee (Medium)

建立状态机，股票交易类问题就会迎刃而解。

第 8 章 化繁为简的分治法

内容提要

- 算法解释
- 表达式问题

8.1 算法解释

顾名思义，分治问题由“分”（divide）和“治”（conquer）两部分组成，通过把原问题分为子问题，再将子问题进行处理合并，从而实现对原问题的求解。我们在排序章节展示的归并排序就是典型的分治问题，其中“分”即为把大数组平均分成两个小数组，通过递归实现，最终我们会得到多个长度为 1 的子数组；“治”即为把已经排好序的两个小数组合成为一个排好序的大数组，从长度为 1 的子数组开始，最终合成一个大数组。

我们也使用数学表达式来表示这个过程。定义 $T(n)$ 表示处理一个长度为 n 的数组的时间复杂度，则归并排序的时间复杂度递推公式为 $T(n) = 2T(n/2) + O(n)$。其中 $2T(n/2)$ 表示我们分成了两个长度减半的子问题，$O(n)$ 则为合并两个长度为 $n/2$ 数组的时间复杂度。

那么怎么利用这个递推公式得到最终的时间复杂度呢？这里我们可以利用著名的主定理（Master theorem）求解：

定理 8.1. 主定理

考虑 $T(n) = aT(n/b) + f(n)$，定义 $k = \log_b a$

1. 如果 $f(n) = O(n^p)$ 且 $p < k$，那么 $T(n) = O(n^K)$
2. 如果存在 $c \geq 0$ 满足 $f(n) = O(n^k \log^c n)$，那么 $T(n) = O(n^k \log^{c+1} n)$
3. 如果 $f(n) = O(n^p)$ 且 $p > k$，那么 $T(n) = O(f(n))$

♡

通过主定理我们可以知道，归并排序属于第二种情况，且时间复杂度为 $O(n \log n)$。其他的分治问题也可以通过主定理求得时间复杂度。

另外，自上而下的分治可以和 memoization 结合，避免重复遍历相同的子问题。如果方便推导，也可以换用自下而上的动态规划方法求解。

8.2 表达式问题

241. Different Ways to Add Parentheses (Medium)

题目描述

给定一个只包含加、减和乘法的数学表达式，求通过加括号可以得到多少种不同的结果。

输入输出样例

输入是一个字符串，表示数学表达式；输出是一个数组，存储所有不同的加括号结果。

```
Input: "2-1-1"
Output: [0, 2]
```

在这个样例中，有两种加括号结果：((2-1)-1) = 0 和 (2-(1-1)) = 2。

题解

利用分治思想，我们可以把加括号转化为，对于每个运算符号，先执行处理两侧的数学表达式，再处理此运算符号。注意边界情况，即字符串内无运算符号，只有数字。

```
vector<int> diffWaysToCompute(string input) {
    vector<int> ways;
    for (int i = 0; i < input.length(); i++) {
        char c = input[i];
        if (c == '+' || c == '-' || c == '*') {
            vector<int> left = diffWaysToCompute(input.substr(0, i));
            vector<int> right = diffWaysToCompute(input.substr(i + 1));
            for (const int & l: left) {
                for (const int & r: right) {
                    switch (c) {
                        case '+': ways.push_back(l + r); break;
                        case '-': ways.push_back(l - r); break;
                        case '*': ways.push_back(l * r); break;
                    }
                }
            }
        }
    }
    if (ways.empty()) ways.push_back(stoi(input));
    return ways;
}
```

我们发现，某些被 divide 的子字符串可能重复出现多次，因此我们可以用 memoization 来去重。或者与其我们从上到下用分治处理 +memoization，不如直接从下到上用动态规划处理。

```
vector<int> diffWaysToCompute(string input) {
    vector<int> data;
    vector<char> ops;
    int num = 0;
    char op = ' ';
    istringstream ss(input + "+");
    while (ss >> num && ss >> op) {
        data.push_back(num);
        ops.push_back(op);
    }
    int n = data.size();
    vector<vector<vector<int>>> dp(n, vector<vector<int>>(n, vector<int>()));
    for (int i = 0; i < n; ++i) {
        for (int j = i; j >= 0; --j) {
            if (i == j) {
                dp[j][i].push_back(data[i]);
            } else {
                for (int k = j; k < i; k += 1) {
                    for (auto left : dp[j][k]) {
```

```
                    for (auto right : dp[k+1][i]) {
                        int val = 0;
                        switch (ops[k]) {
                           case '+': val = left + right; break;
                           case '-': val = left - right; break;
                           case '*': val = left * right; break;
                        }
                        dp[j][i].push_back(val);
                    }
                }
            }
        }
    }
    return dp[0][n-1];
}
```

8.3 练习

基础难度

932. Beautiful Array (Medium)

试着用从上到下的分治（递归）写法求解，最好加上 memoization；再用从下到上的动态规划写法求解。

进阶难度

312. Burst Balloons (Hard)

试着用从上到下的分治（递归）写法求解，最好加上 memoization；再用从下到上的动态规划写法求解。

第 9 章 巧解数学问题

内容提要

- 引言
- 公倍数与公因数
- 质数
- 数字处理
- 随机与取样

9.1 引言

对于 LeetCode 上数量不少的数学题，我们尽量将其按照类型划分讲解。然而很多数学题的解法并不通用，我们也很难在几道题里把所有的套路讲清楚，因此我们只选择了几道经典或是典型的题目，供大家参考。

9.2 公倍数与公因数

利用辗转相除法，我们可以很方便地求得两个数的最大公因数（greatest common divisor，gcd）；将两个数相乘再除以最大公因数即可得到最小公倍数（least common multiple, lcm）。

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a% b);
}

int lcm(int a, int b) {
    return a * b / gcd(a, b);
}
```

进一步地，我们也可以通过扩展欧几里得算法（extended gcd）在求得 a 和 b 最大公因数的同时，也得到它们的系数 x 和 y，从而使 ax + by = gcd(a, b)。

```
int xGCD(int a, int b, int &x, int &y) {
    if (!b) {
       x = 1, y = 0;
       return a;
    }
    int x1, y1, gcd = xGCD(b, a % b, x1, y1);
    x = y1, y = x1 - (a / b) * y1;
    return gcd;
}
```

9.3 质数

质数又称素数，指的是指在大于 1 的自然数中，除了 1 和它本身以外不再有其他因数的自然数。值得注意的是，每一个数都可以分解成质数的乘积。

204. Count Primes (Easy)

题目描述

给定一个数字 n，求小于 n 的质数的个数。

输入输出样例

输入一个整数，输出也是一个整数，表示小于输入数的质数的个数。

```
Input: 10
Output: 4
```

在这个样例中，小于 10 的质数只有 [2,3,5,7]。

题解

埃拉托斯特尼筛法（Sieve of Eratosthenes，简称埃氏筛法）是非常常用的，判断一个整数是否是质数的方法。并且它可以在判断一个整数 n 时，同时判断所小于 n 的整数，因此非常适合这道题。其原理也十分易懂：从 1 到 n 遍历，假设当前遍历到 m，则把所有小于 n 的、且是 m 的倍数的整数标为和数；遍历完成后，没有被标为和数的数字即为质数。

```
int countPrimes(int n) {
    if (n <= 2) return 0;
    vector<bool> prime(n, true);
    int count = n - 2; // 去掉不是质数的1
    for (int i = 2; i <= n; ++i) {
        if (prime[i]) {
            for (int j = 2 * i; j < n; j += i) {
                if (prime[j]) {
                    prime[j] = false;
                    --count;
                }
            }
        }
    }
    return count;
}
```

利用质数的一些性质，我们可以进一步优化该算法。

```
int countPrimes(int n) {
    if (n <= 2) return 0;
    vector<bool> prime(n, true);
    int i = 3, sqrtn = sqrt(n), count = n / 2; // 偶数一定不是质数
    while (i <= sqrtn) { // 最小质因子一定小于等于开方数
        for (int j = i * i; j < n; j += 2 * i) { // 避免偶数和重复遍历
            if (prime[j]) {
                --count;
                prime[j] = false;
            }
        }
        do {
            i += 2;
```

```
        } while (i <= sqrtn && !prime[i]); // 避免偶数和重复遍历
    }
    return count;
}
```

9.4 数字处理

504. Base 7 (Easy)

题目描述

给定一个十进制整数，求它在七进制下的表示。

输入输出样例

输入一个整数，输出一个字符串，表示其七进制。

```
Input: 100
Output: "202"
```

在这个样例中，100 的七进制表示法来源于 101 = 2 * 49 + 0 * 7 + 2 * 1。

题解

进制转换类型的题，通常是利用除法和取模（mod）来进行计算，同时也要注意一些细节，如负数和零。如果输出是数字类型而非字符串，则也需要考虑是否会超出整数上下界。

```
string convertToBase7(int num) {
    if (num == 0) return "0";
    bool is_negative = num < 0;
    if (is_negative) num = -num;
    string ans;
    while (num) {
        int a = num / 7, b = num % 7;
        ans = to_string(b) + ans;
        num = a;
    }
    return is_negative? "-" + ans: ans;
}
```

172. Factorial Trailing Zeroes

题目描述

给定一个非负整数，判断它的阶乘结果的结尾有几个 0。

输入输出样例

输入一个非负整数，输出一个非负整数，表示输入的阶乘结果的结尾有几个 0。

```
Input: 12
Output: 2
```

在这个样例中，12 != 479001600 的结尾有两个 0。

题解

每个尾部的 0 由 2 × 5 = 10 而来，因此我们可以把阶乘的每一个元素拆成质数相乘，统计有多少个 2 和 5。明显的，质因子 2 的数量远多于质因子 5 的数量，因此我们可以只统计阶乘结果里有多少个质因子 5。

```
int trailingZeroes(int n) {
    return n == 0? 0: n / 5 + trailingZeroes(n / 5);
}
```

415. Add Strings (Easy)

题目描述

给定两个由数字组成的字符串，求它们相加的结果。

输入输出样例

输入是两个字符串，输出是一个整数，表示输入的数字和。

```
Input: num1 = "99", num2 = "1"
Output: 100
```

题解

因为相加运算是从后往前进行的，所以可以先翻转字符串，再逐位计算。这种类型的题考察的是细节，如进位、位数差等等。

```
string addStrings(string num1, string num2) {
    string output("");
    reverse(num1.begin(), num1.end());
    reverse(num2.begin(), num2.end());
    int onelen = num1.length(), twolen = num2.length();
    if (onelen <= twolen){
        swap(num1, num2);
        swap(onelen, twolen);
    }
    int addbit = 0;
    for (int i = 0; i < twolen; ++i){
        int cur_sum = (num1[i]-'0') + (num2[i]-'0') + addbit;
        output += to_string((cur_sum) % 10);
        addbit = cur_sum < 10? 0: 1;
    }
    for (int i = twolen; i < onelen; ++i){
        int cur_sum = (num1[i]-'0') + addbit;
```

```
        output += to_string((cur_sum) % 10);
        addbit = cur_sum < 10? 0: 1;
    }
    if (addbit) {
        output += "1";
    }
    reverse(output.begin(), output.end());
    return output;
}
```

326. Power of Three (Easy)

题目描述

判断一个数字是否是 3 的次方。

输入输出样例

输入一个整数，输出一个布尔值。

```
Input: n = 27
Output: true
```

题解

有两种方法，一种是利用对数。设 $\log_3^n = m$，如果 n 是 3 的整数次方，那么 m 一定是整数。

```
bool isPowerOfThree(int n) {
    return fmod(log10(n) / log10(3), 1) == 0;
}
```

另一种方法是，因为在 int 范围内 3 的最大次方是 $3^{19} = 1162261467$，如果 n 是 3 的整数次方，那么 1162261467 除以 n 的余数一定是零；反之亦然。

```
bool isPowerOfThree(int n) {
    return n > 0 && 1162261467 % n == 0;
}
```

9.5 随机与取样

384. Shuffle an Array (Medium)

题目描述

给定一个数组，要求实现两个指令函数。第一个函数“shuffle”可以随机打乱这个数组，第二个函数“reset”可以恢复原来的顺序。

输入输出样例

输入是一个存有整数数字的数组，和一个包含指令函数名称的数组。输出是一个二维数组，表示每个指令生成的数组。

```
Input: nums = [1,2,3], actions: ["shuffle","shuffle","reset"]
Output: [[2,1,3],[3,2,1],[1,2,3]]
```

在这个样例中，前两次打乱的结果只要是随机生成即可。

题解

我们采用经典的 Fisher-Yates 洗牌算法，原理是通过随机交换位置来实现随机打乱，有正向和反向两种写法，且实现非常方便。注意这里“reset”函数以及类的构造函数的实现细节。

```cpp
class Solution {
    vector<int> origin;
public:
    Solution(vector<int> nums): origin(std::move(nums)) {}

    vector<int> reset() {
        return origin;
    }

    vector<int> shuffle() {
        if (origin.empty()) return {};
        vector<int> shuffled(origin);
        int n = origin.size();
        // 可以使用反向或者正向洗牌，效果相同。
        // 反向洗牌:
        for (int i = n - 1; i >= 0; --i) {
            swap(shuffled[i], shuffled[rand() % (i + 1)]);
        }
        // 正向洗牌:
        // for (int i = 0; i < n; ++i) {
        //     int pos = rand() % (n - i);
        //     swap(shuffled[i], shuffled[i+pos]);
        // }
        return shuffled;
    }
};
```

528. Random Pick with Weight (Medium)

题目描述

给定一个数组，数组每个位置的值表示该位置的权重，要求按照权重的概率去随机采样。

输入输出样例

输入是一维正整数数组，表示权重；和一个包含指令字符串的一维数组，表示运行几次随机采样。输出是一维整数数组，表示随机采样的整数在数组中的位置。

```
Input: weights = [1,3], actions: ["pickIndex","pickIndex","pickIndex"]
Output: [0,1,1]
```

在这个样例中，每次选择的位置都是不确定的，但选择第 0 个位置的期望为 1/4，选择第 1 个位置的期望为 3/4。

题解

我们可以先使用 `partial_sum` 求前缀和（即到每个位置为止之前所有数字的和），这个结果对于正整数数组是单调递增的。每当需要采样时，我们可以先随机产生一个数字，然后使用二分法查找其在前缀和中的位置，以模拟加权采样的过程。这里的二分法可以用 `lower_bound` 实现。

以样例为例，权重数组 [1,3] 的前缀和为 [1,4]。如果我们随机生成的数字为 1，那么 `lower_bound` 返回的位置为 0；如果我们随机生成的数字是 2、3、4，那么 `lower_bound` 返回的位置为 1。

关于前缀和的更多技巧，我们将在接下来的章节中继续深入讲解。

```
class Solution {
vector<int> sums;
public:
    Solution(vector<int> weights): sums(std::move(weights)) {
        partial_sum(sums.begin(), sums.end(), sums.begin());
    }

    int pickIndex() {
        int pos = (rand() % sums.back()) + 1;
        return lower_bound(sums.begin(), sums.end(), pos) - sums.begin();
    }
};
```

382. Linked List Random Node (Medium)

题目描述

给定一个单向链表，要求设计一个算法，可以随机取得其中的一个数字。

输入输出样例

输入是一个单向链表，输出是一个数字，表示链表里其中一个节点的值。

```
Input: 1->2->3->4->5
Output: 3
```

在这个样例中，我们有均等的概率得到任意一个节点，比如 3。

题解

不同于数组，在未遍历完链表前，我们无法知道链表的总长度。这里我们就可以使用水库采样：遍历一次链表，在遍历到第 m 个节点时，有 $\frac{1}{m}$ 的概率选择这个节点覆盖掉之前的节点选择。

我们提供一个简单的，对于水库算法随机性的证明。对于长度为 n 的链表的第 m 个节点，最后被采样的充要条件是它被选择，且之后的节点都没有被选择。这种情况发生的概率为 $\frac{1}{m} \times \frac{m}{m+1} \times \frac{m+1}{m+2} \times \cdots \times \frac{n-1}{n} = \frac{1}{n}$。因此每个点都有均等的概率被选择。

```
class Solution {
    ListNode* head;
public:
    Solution(ListNode* n): head(n) {}

    int getRandom() {
        int ans = head->val;
        ListNode* node = head->next;
        int i = 2;
        while (node) {
            if ((rand() % i) == 0) {
                ans = node->val;
            }
            ++i;
            node = node->next;
        }
        return ans;
    }
};
```

9.6 练习

基础难度

168. Excel Sheet Column Title (Easy)

7 进制转换的变种题，需要注意的一点是从 1 开始而不是 0。

67. Add Binary (Easy)

字符串加法的变种题。

238. Product of Array Except Self (Medium)

你可以不使用除法做这道题吗？我们很早之前讲过的题目 135 或许能给你一些思路。

进阶难度

462. Minimum Moves to Equal Array Elements II (Medium)

这道题是笔者最喜欢的 LeetCode 题目之一，需要先推理出怎么样移动是最优的，再考虑如何进行移动。你或许需要一些前些章节讲过的算法。

169. Majority Element (Easy)

如果想不出简单的解决方法，搜索一下 Boyer-Moore Majority Vote 算法吧。

470. Implement Rand10() Using Rand7() (Medium)

如何用一个随机数生成器生成另一个随机数生成器？你可能需要利用原来的生成器多次。

202. Happy Number (Easy)

你可以简单的用一个 while 循环解决这道题，但是有没有更好的解决办法？如果我们把每个数字想象成一个节点，是否可以转化为环路检测？

第 10 章 神奇的位运算

内容提要

- 常用技巧
- 二进制特性
- 位运算基础问题

10.1 常用技巧

位运算是算法题里比较特殊的一种类型，它们利用二进制位运算的特性进行一些奇妙的优化和计算。常用的位运算符号包括：“^”按位异或、“&”按位与、“|”按位或、“~”取反、“<<”算术左移和“>>”算术右移。以下是一些常见的位运算特性，其中 0s 和 1s 分别表示只由 0 或 1 构成的二进制数字。

```
x ^ 0s = x      x & 0s = 0      x | 0s = x
x ^ 1s = ~x     x & 1s = x      x | 1s = 1s
x ^ x = 0       x & x = x       x | x = x
```

除此之外，n & (n - 1) 可以去除 n 的位级表示中最低的那一位，例如对于二进制表示 11110100，减去 1 得到 11110011，这两个数按位与得到 11110000。n & (-n) 可以得到 n 的位级表示中最低的那一位，例如对于二进制表示 11110100，取负得到 00001100，这两个数按位与得到 00000100。还有更多的并不常用的技巧，若读者感兴趣可以自行研究，这里不再赘述。

10.2 位运算基础问题

461. Hamming Distance (Easy)

题目描述

给定两个十进制数字，求它们二进制表示的汉明距离（Hamming distance，即不同位的个数）。

输入输出样例

输入是两个十进制整数，输出是一个十进制整数，表示两个输入数字的汉明距离。

```
Input: x = 1, y = 4
Output: 2
```

在这个样例中，1 的二进制是 0001，4 的二进制是 0100，一共有两位不同。

题解

对两个数进行按位异或操作，统计有多少个 1 即可。

```
int hammingDistance(int x, int y) {
    int diff = x ^ y, ans = 0;
    while (diff) {
        ans += diff & 1;
        diff >>= 1;
    }
    return ans;
}
```

190. Reverse Bits (Easy)

题目描述

给定一个十进制整数，输出它在二进制下的翻转结果。

输入输出样例

输入和输出都是十进制整数。

```
Input: 43261596 (00000010100101000001111010011100)
Output: 964176192 (00111001011110000010100101000000)
```

题解

使用算术左移和右移，可以很轻易地实现二进制的翻转。

```
uint32_t reverseBits(uint32_t n) {
    uint32_t ans = 0;
    for (int i = 0; i < 32; ++i) {
        ans <<= 1;
        ans += n & 1;
        n >>= 1;
    }
    return ans;
}
```

136. Single Number (Easy)

题目描述

给定一个整数数组，这个数组里只有一个数次出现了一次，其余数字出现了两次，求这个只出现一次的数字。

输入输出样例

输入是一个一维整数数组，输出是该数组内的一个整数。

```
Input: [4,1,2,1,2]
Output: 4
```

题解

我们可以利用 x ∧ x = 0 和 x ∧ 0 = x 的特点，将数组内所有的数字进行按位异或。出现两次的所有数字按位异或的结果是 0，0 与出现一次的数字异或可以得到这个数字本身。

```
int singleNumber(vector<int>& nums) {
    int ans = 0;
    for (const int & num: nums) {
        ans ^= num;
    }
    return ans;
}
```

10.3 二进制特性

利用二进制的一些特性，我们可以把位运算使用到更多问题上。

例如，我们可以利用二进制和位运算输出一个数组的所有子集。假设我们有一个长度为 n 的数组，我们可以生成长度为 n 的所有二进制，1 表示选取该数字，0 表示不选取。这样我们就获得了 2^n 个子集。

342. Power of Four (Easy)

题目描述

给定一个整数，判断它是否是 4 的次方。

输入输出样例

输入是一个整数，输出是一个布尔值，表示判断结果。

```
Input: 16
Output: true
```

在这个样例中，16 是 4 的二次方，因此返回值为真。

题解

首先我们考虑一个数字是不是 2 的（整数）次方：如果一个数字 n 是 2 的整数次方，那么它的二进制一定是 0...010...0 这样的形式；考虑到 $n-1$ 的二进制是 0...001...1，这两个数求按位与的结果一定是 0。因此如果 n & (n - 1) 为 0，那么这个数是 2 的次方。

如果这个数也是 4 的次方，那二进制表示中 1 的位置必须为奇数位。我们可以把 n 和二进制的 10101...101（即十进制下的 1431655765）做按位与，如果结果不为 0，那么说明这个数是 4 的次方。

```
bool isPowerOfFour(int n) {
    return n > 0 && !(n & (n - 1)) && (n & 1431655765);
}
```

318. Maximum Product of Word Lengths (Medium)

题目描述

给定多个字母串，求其中任意两个字母串的长度乘积的最大值，且这两个字母串不能含有相同字母。

输入输出样例

输入一个包含多个字母串的一维数组，输出一个整数，表示长度乘积的最大值。

```
Input: ["a","ab","abc","d","cd","bcd","abcd"]
Output: 4
```

在这个样例中，一种最优的选择是“ab”和“cd”。

题解

怎样快速判断两个字母串是否含有重复数字呢？可以为每个字母串建立一个长度为 26 的二进制数字，每个位置表示是否存在该字母。如果两个字母串含有重复数字，那它们的二进制表示的按位与不为 0。同时，我们可以建立一个哈希表来存储字母串（在数组的位置）到二进制数字的映射关系，方便查找调用。

```
int maxProduct(vector<string>& words) {
    unordered_map<int, int> hash;
    int ans = 0;
    for (const string & word : words) {
        int mask = 0, size = word.size();
        for (const char & c : word) {
            mask |= 1 << (c - 'a');
        }
        hash[mask] = max(hash[mask], size);
        for (const auto& [h_mask, h_len]: hash) {
            if (!(mask & h_mask)) {
                ans = max(ans, size * h_len);
            }
        }
    }
    return ans;
}
```

338. Counting Bits (Medium)

题目描述

给定一个非负整数 n，求从 0 到 n 的所有数字的二进制表达中，分别有多少个 1。

输入输出样例

输入是一个非负整数 n，输出是长度为 $n+1$ 的非负整数数组，每个位置 m 表示 m 的二进制里有多少个 1。

```
Input: 5
Output: [0,1,1,2,1,2]
```

题解

本题可以利用动态规划和位运算进行快速的求解。定义一个数组 dp，其中 dp[i] 表示数字 i 的二进制含有 1 的个数。对于第 i 个数字，如果它二进制的最后一位为 1，那么它含有 1 的个数则为 dp[i-1] + 1；如果它二进制的最后一位为 0，那么它含有 1 的个数和其算术右移结果相同，即 dp[i>>1]。

```
vector<int> countBits(int num) {
    vector<int> dp(num+1, 0);
    for (int i = 1; i <= num; ++i)
        dp[i] = i & 1? dp[i-1] + 1: dp[i>>1];
        // 等价于dp[i] = dp[i&(i-1)] + 1;
    return dp;
}
```

10.4 练习

基础难度

268. Missing Number (Easy)

Single Number 的变种题。除了利用二进制，也可以使用高斯求和公式。

693. Binary Number with Alternating Bits (Easy)

利用位运算判断一个数的二进制是否会出现连续的 0 和 1。

476. Number Complement (Easy)

二进制翻转的变种题。

进阶难度

260. Single Number III (Medium)

Single Number 的 follow-up，需要认真思考如何运用位运算求解。

第 11 章 妙用数据结构

内容提要

- C++ STL
- 数组
- 栈和队列
- 单调栈
- 优先队列
- 双端队列
- 哈希表
- 多重集合和映射
- 前缀和与积分图

11.1 C++ STL

在刷题时，我们几乎一定会用到各种数据结构来辅助我们解决问题，因此我们必须熟悉各种数据结构的特点。C++ STL 提供的数据结构包括（实际底层细节可能因编译器而异）：

1. Sequence Containers：维持顺序的容器。
   (a). vector：动态数组，是我们最常使用的数据结构之一，用于 $O(1)$ 的随机读取。因为大部分算法的时间复杂度都会大于 $O(n)$，因此我们经常新建 vector 来存储各种数据或中间变量。因为在尾部增删的复杂度是 $O(1)$，我们也可以把它当作 stack 来用。
   (b). list：双向链表，也可以当作 stack 和 queue 来使用。由于 LeetCode 的题目多用 Node 来表示链表，且链表不支持快速随机读取，因此我们很少用到这个数据结构。一个例外是经典的 LRU 问题，我们需要利用链表的特性来解决，我们在后文会遇到这个问题。
   (c). deque：双端队列，这是一个非常强大的数据结构，既支持 $O(1)$ 随机读取，又支持 $O(1)$ 时间的头部增删和尾部增删，不过有一定的额外开销。
   (d). array：固定大小的数组，一般在刷题时我们不使用。
   (e). forward_list：单向链表，一般在刷题时我们不使用。
2. Container Adaptors：基于其它容器实现的数据结构。
   (a). stack：后入先出（LIFO）的数据结构，默认基于 deque 实现。stack 常用于深度优先搜索、一些字符串匹配问题以及单调栈问题。
   (b). queue：先入先出（FIFO）的数据结构，默认基于 deque 实现。queue 常用于广度优先搜索。
   (c). priority_queue：最大值先出的数据结构，默认基于 vector 实现堆结构。它可以在 $O(n \log n)$ 的时间排序数组，$O(\log n)$ 的时间插入任意值，$O(1)$ 的时间获得最大值，$O(\log n)$ 的时间删除最大值。priority_queue 常用于维护数据结构并快速获取最大或最小值。
3. Associative Containers：实现了排好序的数据结构。
   (a). set：有序集合，元素不可重复，底层实现默认为红黑树，即一种特殊的二叉查找树（BST）。它可以在 $O(n \log n)$ 的时间排序数组，$O(\log n)$ 的时间插入、删除、查找任意值，$O(\log n)$ 的时间获得最小或最大值。这里注意，set 和 priority_queue 都可以用于维护数据结构并快速获取最大最小值，但是它们的时间复杂度和功能略有区别，如 priority_queue 默认不支持删除任意值，而 set 获得最大或最小值的时间复杂度略高，具体使用哪个根据需求而定。
   (b). multiset：支持重复元素的 set。

(c). map：有序映射或有序表，在 set 的基础上加上映射关系，可以对每个元素 key 存一个值 value。

(d). multimap：支持重复元素的 map。

4. Unordered Associative Containers：对每个 Associative Containers 实现了哈希版本。

(a). unordered_set：哈希集合，可以在 $O(1)$ 的时间快速插入、查找、删除元素，常用于快速的查询一个元素是否在这个容器内。

(b). unordered_multiset：支持重复元素的 unordered_set。

(c). unordered_map：哈希映射或哈希表，在 unordered_set 的基础上加上映射关系，可以对每一个元素 key 存一个值 value。在某些情况下，如果 key 的范围已知且较小，我们也可以用 vector 代替 unordered_map，用位置表示 key，用每个位置的值表示 value。

(d). unordered_multimap：支持重复元素的 unordered_map。

因为这并不是一本讲解 C++ 原理的书，更多的 STL 细节请读者自行搜索。只有理解了这些数据结构的原理和使用方法，才能够更加游刃有余地解决算法和数据结构问题。

11.2 数组

448. Find All Numbers Disappeared in an Array (Easy)

题目描述

给定一个长度为 n 的数组，其中包含范围为 1 到 n 的整数，有些整数重复了多次，有些整数没有出现，求 1 到 n 中没有出现过的整数。

输入输出样例

输入是一个一维整数数组，输出也是一个一维整数数组，表示输入数组内没出现过的数字。

```
Input: [4,3,2,7,8,2,3,1]
Output: [5,6]
```

题解

利用数组这种数据结构建立 n 个桶，把所有重复出现的位置进行标记，然后再遍历一遍数组，即可找到没有出现过的数字。进一步地，我们可以直接对原数组进行标记：把重复出现的数字在原数组出现的位置设为负数，最后仍然为正数的位置即为没有出现过的数。

```
vector<int> findDisappearedNumbers(vector<int>& nums) {
    vector<int> ans;
    for (const int & num: nums) {
        int pos = abs(num) - 1;
        if (nums[pos] > 0) {
            nums[pos] = -nums[pos];
        }
    }
    for (int i = 0; i < nums.size(); ++i) {
        if (nums[i] > 0) {
            ans.push_back(i + 1);
```

```
        }
    }
    return ans;
}
```

48. Rotate Image (Medium)

题目描述

给定一个 $n \times n$ 的矩阵，求它顺时针旋转 90 度的结果，且必须在原矩阵上修改（in-place）。怎样能够尽量不创建额外储存空间呢?

输入输出样例

输入和输出都是一个二维整数矩阵。

```
Input:
[[1,2,3],
 [4,5,6],
 [7,8,9]]
Output:
[[7,4,1],
 [8,5,2],
 [9,6,3]]
```

题解

每次只考虑四个间隔 90 度的位置，可以进行 $O(1)$ 额外空间的旋转。

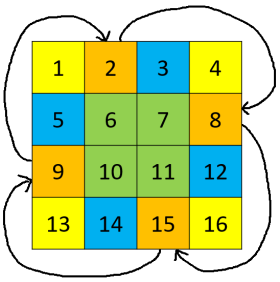

图 11.1: 题目 48 - $O(1)$ 空间旋转样例，相同颜色代表四个互相交换的位置

```
void rotate(vector<vector<int>>& matrix) {
    int temp = 0, n = matrix.size()-1;
    for (int i = 0; i <= n / 2; ++i) {
        for (int j = i; j < n - i; ++j) {
            temp = matrix[j][n-i];
            matrix[j][n-i] = matrix[i][j];
            matrix[i][j] = matrix[n-j][i];
            matrix[n-j][i] = matrix[n-i][n-j];
            matrix[n-i][n-j] = temp;
        }
    }
}
```

240. Search a 2D Matrix II (Medium)

题目描述

给定一个二维矩阵，已知每行和每列都是增序的，尝试设计一个快速搜索一个数字是否在矩阵中存在的算法。

输入输出样例

输入是一个二维整数矩阵，和一个待搜索整数。输出是一个布尔值，表示这个整数是否存在于矩阵中。

```
Input: matrix =
[ [1,  4,  7, 11, 15],
  [2,  5,  8, 12, 19],
  [3,  6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]], target = 5
Output: true
```

题解

这道题有一个简单的技巧：我们可以从右上角开始查找，若当前值大于待搜索值，我们向左移动一位；若当前值小于待搜索值，我们向下移动一位。如果最终移动到左下角时仍不等于待搜索值，则说明待搜索值不存在于矩阵中。

```
bool searchMatrix(vector<vector<int>>& matrix, int target) {
    int m = matrix.size();
    if (m == 0) {
        return false;
    }
    int n = matrix[0].size();
    int i = 0, j = n - 1;
    while (i < m && j >= 0) {
        if (matrix[i][j] == target) {
            return true;
        } else if (matrix[i][j] > target) {
            --j;
        } else {
            ++i;
        }
    }
    return false;
}
```

769. Max Chunks To Make Sorted (Medium)

题目描述

给定一个含有 0 到 n 整数的数组，每个整数只出现一次，求这个数组最多可以分割成多少个子数组，使得对每个子数组进行增序排序后，原数组也是增序的。

输入输出样例

输入一个一维整数数组，输出一个整数，表示最多的分割数。

```
Input: [1,0,2,3,4]
Output: 4
```

在这个样例中，最多分割是 [1, 0], [2], [3], [4]。

题解

从左往右遍历，同时记录当前的最大值，每当当前最大值等于数组位置时，我们可以多一次分割。

为什么可以通过这个算法解决问题呢？如果当前最大值大于数组位置，则说明右边一定有小于数组位置的数字，需要把它也加入待排序的子数组；又因为数组只包含不重复的 0 到 n，所以当前最大值一定不会小于数组位置。所以每当当前最大值等于数组位置时，假设为 p，我们可以成功完成一次分割，并且其与上一次分割位置 q 之间的值一定是 $q+1$ 到 p 的所有数字。

```
int maxChunksToSorted(vector<int>& arr) {
    int chunks = 0, cur_max = 0;
    for (int i = 0; i < arr.size(); ++i) {
        cur_max = max(cur_max, arr[i]);
        if (cur_max == i) {
            ++chunks;
        }
    }
    return chunks;
}
```

11.3 栈和队列

232. Implement Queue using Stacks (Easy)

题目描述

尝试使用栈（stack）来实现队列（queue）。

输入输出样例

以下是数据结构的调用样例。

```
MyQueue queue = new MyQueue();
queue.push(1);
queue.push(2);
queue.peek(); // returns 1
queue.pop();  // returns 1
queue.empty(); // returns false
```

题解

我们可以用两个栈来实现一个队列：因为我们需要得到先入先出的结果，所以必定要通过一个额外栈翻转一次数组。这个翻转过程既可以在插入时完成，也可以在取值时完成。

```
class MyQueue {
    stack<int> in, out;
public:
    MyQueue() {}

    void push(int x) {
        in.push(x);
    }

    int pop() {
        in2out();
        int x = out.top();
        out.pop();
        return x;
    }

    int peek() {
        in2out();
        return out.top();
    }

    void in2out() {
        if (out.empty()) {
            while (!in.empty()) {
                int x = in.top();
                in.pop();
                out.push(x);
            }
        }
    }

    bool empty() {
        return in.empty() && out.empty();
    }
};
```

155. Min Stack (Easy)

题目描述

设计一个最小栈，除了需要支持常规栈的操作外，还需要支持在 $O(1)$ 时间内查询栈内最小值的功能。

输入输出样例

以下是数据结构的调用样例。

```
MinStack minStack = new MinStack();
```

```
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // Returns -3.
minStack.pop();
minStack.top();    // Returns 0.
minStack.getMin(); // Returns -2.
```

题解

我们可以额外建立一个新栈，栈顶表示原栈里所有值的最小值。每当在原栈里插入一个数字时，若该数字小于等于新栈栈顶，则表示这个数字在原栈里是最小值，我们将其同时插入新栈内。每当从原栈里取出一个数字时，若该数字等于新栈栈顶，则表示这个数是原栈里的最小值之一，我们同时取出新栈栈顶的值。

一个写起来更简单但是时间复杂度略高的方法是，我们每次插入原栈时，都向新栈插入一次原栈里所有值的最小值（新栈栈顶和待插入值中小的那一个）；每次从原栈里取出数字时，同样取出新栈的栈顶。这样可以避免判断，但是每次都要插入和取出。我们这里只展示第一种写法。

```
class MinStack {
    stack<int> s, min_s;
public:
    MinStack() {}

    void push(int x) {
        s.push(x);
        if (min_s.empty() || min_s.top() >= x) {
            min_s.push(x);
        }
    }

    void pop() {
        if (!min_s.empty() && min_s.top() == s.top()) {
            min_s.pop();
        }
        s.pop();
    }

    int top() {
        return s.top();
    }

    int getMin() {
        return min_s.top();
    }
};
```

20. Valid Parentheses (Easy)

题目描述

给定一个只由左右原括号、花括号和方括号组成的字符串，求这个字符串是否合法。合法的定义是每一个类型的左括号都有一个右括号一一对应，且括号内的字符串也满足此要求。

输入输出样例

输入是一个字符串，输出是一个布尔值，表示字符串是否合法。

```
Input: "{[]}()"
Output: true
```

题解

括号匹配是典型的使用栈来解决的问题。我们从左往右遍历，每当遇到左括号便放入栈内，遇到右括号则判断其和栈顶的括号是否是统一类型，是则从栈内取出左括号，否则说明字符串不合法。

```
bool isValid(string s) {
    stack<char> parsed;
    for (int i = 0; i < s.length(); ++i) {
        if (s[i] == '{' || s[i] == '[' || s[i] == '(') {
            parsed.push(s[i]);
        } else {
            if (parsed.empty()) {
                return false;
            }
            char c = parsed.top();
            if ((s[i] == '}' && c == '{') ||
                (s[i] == ']' && c == '[') ||
                (s[i] == ')' && c == '(')) {
                parsed.pop();
            } else {
                return false;
            }
        }
    }
    return parsed.empty();
}
```

11.4 单调栈

单调栈通过维持栈内值的单调递增（递减）性，在整体 $O(n)$ 的时间内处理需要大小比较的问题。

739. Daily Temperatures (Medium)

题目描述

给定每天的温度，求对于每一天需要等几天才可以等到更暖和的一天。如果该天之后不存在更暖和的天气，则记为 0。

输入输出样例

输入是一个一维整数数组，输出是同样长度的整数数组，表示对于每天需要等待多少天。

```
Input: [73, 74, 75, 71, 69, 72, 76, 73]
Output: [1, 1, 4, 2, 1, 1, 0, 0]
```

题解

我们可以维持一个单调递减的栈，表示每天的温度；为了方便计算天数差，我们这里存放位置（即日期）而非温度本身。我们从左向右遍历温度数组，对于每个日期 p，如果 p 的温度比栈顶存储位置 q 的温度高，则我们取出 q，并记录 q 需要等待的天数为 $p-q$；我们重复这一过程，直到 p 的温度小于等于栈顶存储位置的温度（或空栈）时，我们将 p 插入栈顶，然后考虑下一天。在这个过程中，栈内数组永远保持单调递减，避免了使用排序进行比较。最后若栈内剩余一些日期，则说明它们之后都没有出现更暖和的日期。

```
vector<int> dailyTemperatures(vector<int>& temperatures) {
    int n = temperatures.size();
    vector<int> ans(n);
    stack<int> indices;
    for (int i = 0; i < n; ++i) {
        while (!indices.empty()) {
            int pre_index = indices.top();
            if (temperatures[i] <= temperatures[pre_index]) {
                break;
            }
            indices.pop();
            ans[pre_index] = i - pre_index;
        }
        indices.push(i);
    }
    return ans;
}
```

11.5 优先队列

优先队列（priority queue）可以在 $O(1)$ 时间内获得最大值，并且可以在 $O(\log n)$ 时间内取出最大值或插入任意值。

优先队列常常用堆（heap）来实现。堆是一个完全二叉树，其每个节点的值总是大于等于子节点的值。实际实现堆时，我们通常用一个数组而不是用指针建立一个树。这是因为堆是完全二叉树，所以用数组表示时，位置 i 的节点的父节点位置一定为 i/2，而它的两个子节点的位置又一定分别为 2i 和 2i+1。

以下是堆的实现方法，其中最核心的两个操作是上浮和下沉：如果一个节点比父节点大，那么需要交换这个两个节点；交换后还可能比它新的父节点大，因此需要不断地进行比较和交换操作，我们称之为上浮；类似地，如果一个节点比父节小，也需要不断地向下进行比较和交换操作，我们称之为下沉。如果一个节点有两个子节点，我们总是交换最大的子节点。

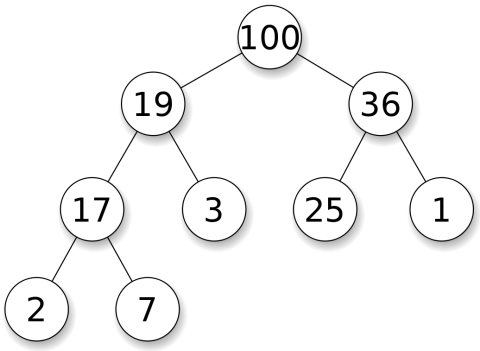

图 11.2: （最大）堆，维护的是数据结构中的大于关系

```
vector<int> heap;

// 获得最大值
void top() {
    return heap[0];
}

// 插入任意值：把新的数字放在最后一位，然后上浮
void push(int k) {
    heap.push_back(k);
    swim(heap.size() - 1);
}

// 删除最大值：把最后一个数字挪到开头，然后下沉
void pop() {
    heap[0] = heap.back();
    heap.pop_back();
    sink(0);
}

// 上浮
void swim(int pos) {
    while (pos > 1 && heap[pos/2] < heap[pos])) {
        swap(heap[pos/2], heap[pos]);
        pos /= 2;
    }
}

// 下沉
void sink(int pos) {
    while (2 * pos <= N) {
        int i = 2 * pos;
        if (i < N && heap[i] < heap[i+1]) ++i;
        if (heap[pos] >= heap[i]) break;
        swap(heap[pos], heap[i]);
        pos = i;
    }
}
```

通过将算法中的大于号和小于号互换，我们也可以得到一个快速获得最小值的优先队列。另外，正如我们在 STL 章节提到的那样，如果我们需要在维持大小关系的同时，还需要支持

查找任意值、删除任意值、维护所有数字的大小关系等操作，可以考虑使用 set 或 map 来代替优先队列。

23. Merge k Sorted Lists (Hard)

题目描述

给定 k 个增序的链表，试将它们合并成一条增序链表。

输入输出样例

输入是一个一维数组，每个位置存储链表的头节点；输出是一条链表。

```
Input:
[1->4->5,
 1->3->4,
 2->6]
Output: 1->1->2->3->4->4->5->6
```

题解

本题可以有很多中解法，比如类似于归并排序进行两两合并。我们这里展示一个速度比较快的方法，即把所有的链表存储在一个优先队列中，每次提取所有链表头部节点值最小的那个节点，直到所有链表都被提取完为止。注意因为 Comp 函数默认是对最大堆进行比较并维持递增关系，如果我们想要获取最小的节点值，则我们需要实现一个最小堆，因此比较函数应该维持递减关系，所以 operator() 中返回时用大于号而不是等增关系时的小于号进行比较。

```
struct Comp {
    bool operator() (ListNode* l1, ListNode* l2) {
        return l1->val > l2->val;
    }
};

ListNode* mergeKLists(vector<ListNode*>& lists) {
    if (lists.empty()) return nullptr;
    priority_queue<ListNode*, vector<ListNode*>, Comp> q;
    for (ListNode* list: lists) {
        if (list) {
            q.push(list);
        }
    }
    ListNode* dummy = new ListNode(0), *cur = dummy;
    while (!q.empty()) {
        cur->next = q.top();
        q.pop();
        cur = cur->next;
        if (cur->next) {
            q.push(cur->next);
        }
    }
    return dummy->next;
}
```

218. The Skyline Problem (Hard)

题目描述

给定建筑物的起止位置和高度，返回建筑物轮廓（天际线）的拐点。

输入输出样例

输入是一个二维整数数组，表示每个建筑物的 [左端, 右端, 高度]；输出是一个二维整数数组，表示每个拐点的横纵坐标。

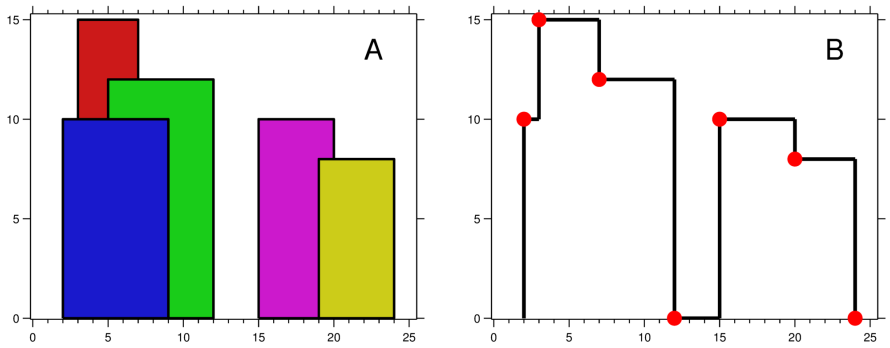

图 11.3: 题目 218 - 建筑物及其天际线样例

```
Input: [[2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8]]
Output: [[2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0]]
```

题解

我们可以使用优先队列储存每个建筑物的高度和右端（这里使用 pair，其默认比较函数是先比较第一个值，如果相等则再比较第二个值），从而获取目前会拔高天际线、且妨碍到前一个建筑物（的右端端点）的下一个建筑物。

```
vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
    vector<vector<int>> ans;
    priority_queue<pair<int, int>> max_heap; // <高度, 右端>
    int i = 0, len = buildings.size();
    int cur_x, cur_h;
    while (i < len || !max_heap.empty()) {
        if (max_heap.empty() || i < len && buildings[i][0] <= max_heap.top().
            second) {
            cur_x = buildings[i][0];
            while (i < len && cur_x == buildings[i][0]) {
                max_heap.emplace(buildings[i][2], buildings[i][1]);
                ++i;
            }
        } else {
            cur_x = max_heap.top().second;
            while (!max_heap.empty() && cur_x >= max_heap.top().second) {
                max_heap.pop();
            }
        }
```

```
        cur_h = (max_heap.empty()) ? 0 : max_heap.top().first;
        if (ans.empty() || cur_h != ans.back()[1]) {
            ans.push_back({cur_x, cur_h});
        }
    }
    return ans;
}
```

11.6 双端队列

239. Sliding Window Maximum (Hard)

题目描述

给定一个整数数组和一个滑动窗口大小，求在这个窗口的滑动过程中，每个时刻其包含的最大值。

输入输出样例

输入是一个一维整数数组，和一个表示滑动窗口大小的整数；输出是一个一维整数数组，表示每个时刻时的窗口内最大值。

```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
```

在这个样例中，滑动窗口在每个位置的最大包含值取法如下：

```
     Window position           Max
------------------------     -----
[1  3  -1] -3  5  3  6  7       3
 1 [3  -1  -3] 5  3  6  7       3
 1  3 [-1  -3  5] 3  6  7       5
 1  3  -1 [-3  5  3] 6  7       5
 1  3  -1  -3 [5  3  6] 7       6
 1  3  -1  -3  5 [3  6  7]      7
```

题解

我们可以利用双端队列进行操作：每当向右移动时，把窗口左端的值从队列左端剔除，把队列右边小于窗口右端的值全部剔除。这样双端队列的最左端永远是当前窗口内的最大值。另外，这道题也是单调栈的一种延申：该双端队列利用从左到右递减来维持大小关系。

```
vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    deque<int> dq;
    vector<int> ans;
    for (int i = 0; i < nums.size(); ++i) {
        if (!dq.empty() && dq.front() == i - k) {
            dq.pop_front();
        }
```

```
        while (!dq.empty() && nums[dq.back()] < nums[i]) {
            dq.pop_back();
        }
        dq.push_back(i);
        if (i >= k - 1) {
            ans.push_back(nums[dq.front()]);
        }
    }
    return ans;
}
```

11.7 哈希表

哈希表，又称散列表，使用 $O(n)$ 空间复杂度存储数据，通过哈希函数映射位置，从而实现近似 $O(1)$ 时间复杂度的插入、查找、删除等操作。

C++ 中的哈希集合为 `unordered_set`，可以查找元素是否在集合中。如果需要同时存储键和值，则需要用 `unordered_map`，可以用来统计频率，记录内容等等。如果元素有穷，并且范围不大，那么可以用一个固定大小的数组来存储或统计元素。例如我们需要统计一个字符串中所有字母的出现次数，则可以用一个长度为 26 的数组来进行统计，其哈希函数即为字母在字母表的位置，这样空间复杂度就可以降低为常数。

一个简单的哈希表的实现如下。

```
// 一个简单的对整数实现的哈希函数
int MyHash(const int& key, const int& tableSize) {
    return key % tableSize;
}

template <typename T>
class HashTable {
public:
    // size最好是质数
    HashTable(int size=31) {
        hash_table_.reserve(size);
        hash_table_.resize(size);
    }

    ~HashTable() {}

    // 查找哈希表是否存在该值
    bool Contains(const T& obj) {
        const list<T>& slot = hash_table_[Hash(obj)];
        auto it = slot.cbegin();
        for (; it != slot.cend() && *it != obj; ++it);
        return it != slot.cend();
    }

    // 插入值
    bool Insert(const T& obj) {
        if (Contains(obj)) {
            return false;
        }
        hash_table_[Hash(obj)].push_front(obj);
        return true;
```

```
    }

    // 删除值
    bool Remove(const T& obj) {
        list<T>& slot = hash_table_[Hash(obj)];
        auto it = find(slot.begin(), slot.end(), obj);
        if (it == slot.end()) {
            return false;
        }
        slot.erase(it);
        return true;
    }
private:
    vector<list<T>> hash_table_;
    // 哈希函数
    int Hash(const T& obj) const {
        return MyHash(obj, hash_table_.size());
    }
};
```

如果需要大小关系的维持，且插入查找并不过于频繁，则可以使用有序的 set/map 来代替 unordered_set/unordered_map。

1. Two Sum (Easy)

题目描述

给定一个整数数组，已知有且只有两个数的和等于给定值，求这两个数的位置。

输入输出样例

输入一个一维整数数组和一个目标值，输出是一个大小为 2 的一维数组，表示满足条件的两个数字的位置。

```
Input: nums = [2, 7, 11, 15], target = 9
Output: [0, 1]
```

在这个样例中，第 0 个位置的值 2 和第 1 个位置的值 7 的和为 9。

题解

我们可以利用哈希表存储遍历过的值以及它们的位置，每次遍历到位置 i 的时候，查找哈希表里是否存在 target - nums[i]，若存在，则说明这两个值的和为 target。

```
vector<int> twoSum(vector<int>& nums, int target) {
    // 键是数字，值是该数字在数组的位置
    unordered_map<int, int> hash;
    vector<int> ans;
    for (int i = 0; i < nums.size(); ++i) {
        int num = nums[i];
        auto pos = hash.find(target - num);
        if (pos == hash.end()) {
            hash[num] = i;
```

```
        } else {
            ans.push_back(pos->second);
            ans.push_back(i);
            break;
        }
    }
    return ans;
}
```

128. Longest Consecutive Sequence (Hard)

题目描述

给定一个整数数组，求这个数组中的数字可以组成的最长连续序列有多长。

输入输出样例

输入一个整数数组，输出一个整数，表示连续序列的长度。

```
Input: [100, 4, 200, 1, 3, 2]
Output: 4
```

在这个样例中，最长连续序列是 [1,2,3,4]。

题解

我们可以把所有数字放到一个哈希表，然后不断地从哈希表中任意取一个值，并删除掉其之前之后的所有连续数字，然后更新目前的最长连续序列长度。重复这一过程，我们就可以找到所有的连续数字序列。

```
int longestConsecutive(vector<int>& nums) {
    unordered_set<int> hash;
    for (const int & num: nums) {
        hash.insert(num);
    }
    int ans = 0;
    while (!hash.empty()) {
        int cur = *(hash.begin());
        hash.erase(cur);
        int next = cur + 1, prev = cur - 1;
        while (hash.count(next)){
            hash.erase(next++);
        }
        while (hash.count(prev)) {
            hash.erase(prev--);
        }
        ans = max(ans, next - prev - 1);
    }
    return ans;
}
```

149. Max Points on a Line (Hard)

题目描述

给定一些二维坐标中的点，求同一条线上最多由多少点。

输入输出样例

输入是一个二维整数数组，表示每个点的横纵坐标；输出是一个整数，表示满足条件的最多点数。

```
Input: [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]
^
|
|  o
|     o       o
|        o
|  o       o
+------------------->
0  1  2  3  4  5  6
Output: 4
```

这个样例中，$y = 5 - x$ 上有四个点。

题解

对于每个点，我们对其它点建立哈希表，统计同一斜率的点一共有多少个。这里利用的原理是，一条线可以由一个点和斜率而唯一确定。另外也要考虑斜率不存在和重复坐标的情况。

本题也利用了一个小技巧：在遍历每个点时，对于数组中位置 i 的点，我们只需要考虑 i 之后的点即可，因为 i 之前的点已经考虑过 i 了。

```
int maxPoints(vector<vector<int>>& points) {
    unordered_map<double, int> hash; // <斜率，点个数>
    int max_count = 0, same = 1, same_y = 1;
    for (int i = 0; i < points.size(); ++i) {
        same = 1, same_y = 1;
        for (int j = i + 1; j < points.size(); ++j) {
            if (points[i][1] == points[j][1]) {
                ++same_y;
                if (points[i][0] == points[j][0]) {
                    ++same;
                }
            } else {
                double dx = points[i][0] - points[j][0], dy = points[i][1] -
                    points[j][1];
                ++hash[dx/dy];
            }
        }
        max_count = max(max_count, same_y);
        for (auto item : hash) {
            max_count = max(max_count, same + item.second);
        }
        hash.clear();
```

```
    }
    return max_count;
}
```

11.8 多重集合和映射

332. Reconstruct Itinerary (Medium)

题目描述

给定一个人坐过的一些飞机的起止机场，已知这个人从 JFK 起飞，那么这个人是按什么顺序飞的；如果存在多种可能性，返回字母序最小的那种。

输入输出样例

输入是一个二维字符串数组，表示多个起止机场对子；输出是一个一维字符串数组，表示飞行顺序。

```
Input: [["MUC", "LHR"], ["JFK", "MUC"], ["SFO", "SJC"], ["LHR", "SFO"]]
Output: ["JFK", "MUC", "LHR", "SFO", "SJC"]
```

题解

本题可以先用哈希表记录起止机场，其中键是起始机场，值是一个多重集合，表示对应的终止机场。因为一个人可能坐过重复的线路，所以我们需要使用多重集合储存重复值。储存完成之后，我们可以利用栈来恢复从终点到起点飞行的顺序，再将结果逆序得到从起点到终点的顺序。

```
vector<string> findItinerary(vector<vector<string>>& tickets) {
    vector<string> ans;
    if (tickets.empty()) {
        return ans;
    }
    unordered_map<string, multiset<string>> hash;
    for (const auto & ticket: tickets) {
        hash[ticket[0]].insert(ticket[1]);
    }
    stack<string> s;
    s.push("JFK");
    while (!s.empty()) {
        string next = s.top();
        if (hash[next].empty()) {
            ans.push_back(next);
            s.pop();
        } else {
            s.push(*hash[next].begin());
            hash[next].erase(hash[next].begin());
        }
    }
    reverse(ans.begin(), ans.end());
    return ans;
```

```
}
```

11.9 前缀和与积分图

一维的前缀和，二维的积分图，都是把每个位置之前的一维线段或二维矩形预先存储，方便加速计算。如果需要对前缀和或积分图的值做寻址，则要存在哈希表里；如果要对每个位置记录前缀和或积分图的值，则可以储存到一维或二维数组里，也常常伴随着动态规划。

303. Range Sum Query - Immutable (Easy)

题目描述

设计一个数据结构，使得其能够快速查询给定数组中，任意两个位置间所有数字的和。

输入输出样例

以下是数据结构的调用样例。

```
vector<int> nums{-2,0,3,-5,2,-1};
NumArray num_array = new NumArray(nums);
num_array.sumRange(0,2); // Result = -2+0+3 = 1.
num_array.sunRange(1,5); // Result = 0+3-5+2-1 = -1.
```

题解

对于一维的数组，我们可以使用前缀和来解决此类问题。先建立一个与数组 nums 长度相同的新数组 psum，表示 nums 每个位置之前前所有数字的和。psum 数组可以通过 C++ 自带的 partial_sum 函数建立，也可以直接遍历一遍 nums 数组，并利用状态转移方程 psum[i] = psum[i-1] + nums[i] 完成统计。如果我们需要获得位置 i 和 j 之间的数字和，只需计算 psum[j+1] - psum[i] 即可。

```
class NumArray {
    vector<int> psum;
public:
    NumArray(vector<int> nums): psum(nums.size() + 1, 0) {
        partial_sum(nums.begin(), nums.end(), psum.begin() + 1);
    }

    int sumRange(int i, int j) {
        return psum[j+1] - psum[i];
    }
};
```

304. Range Sum Query 2D - Immutable (Medium)

题目描述

设计一个数据结构，使得其能够快速查询给定矩阵中，任意两个位置包围的长方形中所有数字的和。

输入输出样例

以下是数据结构的调用样例。其中 sumRegion 函数的四个输入分别是第一个点的横、纵坐标，和第二个点的横、纵坐标。

```
vector<int> matrix{{3,0,1,4,2},
 {5,6,3,2,1},
 {1,2,0,1,5},
 {4,1,0,1,7},
 {1,0,3,0,5}
};
NumMatrix num_matrix = new NumMatrix(matrix);
num_matrix.sumRegion(2,1,4,3); // Result = 8.
num_matrix.sumRegion(1,1,2,2); // Result = 11.
```

题解

类似于前缀和，我们可以把这种思想拓展到二维，即积分图（image integral）。我们可以先建立一个 intergral 矩阵，intergral[i][j] 表示以位置 (0, 0) 为左上角、位置 (i-1, j-1) 为右下角的长方形中所有数字的和。

如图 1 所示，我们可以用动态规划来计算 integral 矩阵：intergral[i][j] = matrix[i-1][j-1] + integral[i-1][j] + integral[i][j-1] - integral[i-1][j-1]，即当前坐标的数字 + 上面长方形的数字和 + 左边长方形的数字和 - 上面长方形和左边长方形重合面积（即左上一格的长方形）中的数字和。

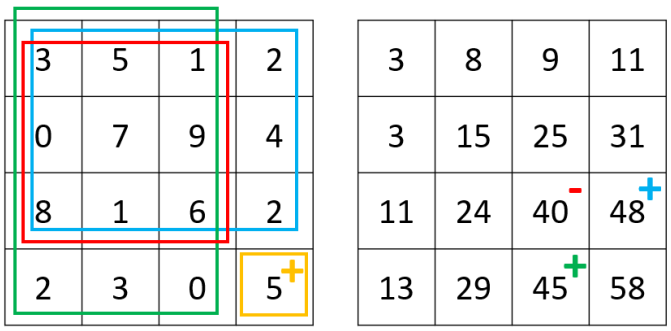

图 11.4: 题目 304 - 图 1 - 左边为给定矩阵，右边为积分图结果，右下角位置的积分图值为 5 + 48 + 45 − 40 = 58

如图 2 所示，假设我们要查询长方形 E 的数字和，因为 $E = D - B - C + A$，我们发现 E 其实可以由四个位置的积分图结果进行加减运算得到。因此这个算法在预处理时的时间复杂度为 $O(mn)$，而在查询时的时间复杂度仅为 $O(1)$。

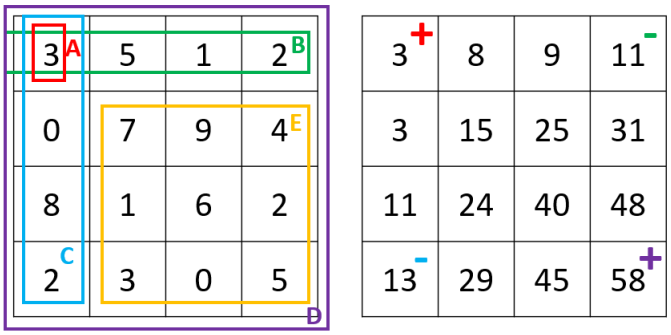

图 11.5: 题目 304 - 图 2 - 左边为给定矩阵，右边为积分图结果，长方形 E 的数字和等于 $58 - 11 - 13 + 3 = 37$

```
class NumMatrix {
    vector<vector<int>> integral;
public:
    NumMatrix(vector<vector<int>> matrix) {
        int m = matrix.size(), n = m > 0? matrix[0].size(): 0;
        integral = vector<vector<int>>(m + 1, vector<int>(n + 1, 0));
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {
                integral[i][j] = matrix[i-1][j-1] + integral[i-1][j] +
                    integral[i][j-1] - integral[i-1][j-1];
            }
        }
    }

    int sumRegion(int row1, int col1, int row2, int col2) {
        return integral[row2+1][col2+1] - integral[row2+1][col1] -
            integral[row1][col2+1] + integral[row1][col1];
    }
};
```

560. Subarray Sum Equals K (Medium)

题目描述

给定一个数组，寻找和为 k 的连续区间个数。

输入输出样例

输入一个一维整数数组和一个整数值 k；输出一个整数，表示满足条件的连续区间个数。

```
Input: nums = [1,1,1], k = 2
Output: 2
```

在这个样例中，我们可以找到两个 [1,1] 连续区间满足条件。

题解

本题同样是利用前缀和，不同的是这里我们使用一个哈希表 hashmap，其键是前缀和，而值是该前缀和出现的次数。在我们遍历到位置 i 时，假设当前的前缀和是 psum，那么 hashmap[psum-k] 即为以当前位置结尾、满足条件的区间个数。

```
int subarraySum(vector<int>& nums, int k) {
    int count = 0, psum = 0;
    unordered_map<int, int> hashmap;
    hashmap[0] = 1; // 初始化很重要
    for (int i: nums) {
        psum += i;
        count += hashmap[psum-k];
        ++hashmap[psum];
    }
    return count;
}
```

11.10 练习

基础难度

566. Reshape the Matrix (Easy)

没有什么难度，只是需要一点耐心。

225. Implement Stack using Queues (Easy)

利用相似的方法，我们也可以用 stack 实现 queue。

503. Next Greater Element II (Medium)

Daily Temperature 的变种题。

217. Contains Duplicate (Easy)

使用什么数据结构可以快速判断重复呢?

697. Degree of an Array (Easy)

如何对数组进行预处理才能正确并快速地计算子数组的长度?

594. Longest Harmonious Subsequence (Easy)

最长连续序列的变种题。

进阶难度

287. Find the Duplicate Number (Medium)

寻找丢失数字的变种题。除了标负位置，你还有没有其它算法可以解决这个问题?

313. Super Ugly Number (Medium)

尝试使用优先队列解决这一问题。

870. Advantage Shuffle (Medium)

如果我们需要比较大小关系，而且同一数字可能出现多次，那么应该用什么数据结构呢?

307. Range Sum Query - Mutable (Medium)

前缀和的变种题。好吧我承认，这道题可能有些超纲，你或许需要搜索一下什么是线段树。

第 12 章 令人头大的字符串

内容提要

- 引言
- 字符串比较
- 字符串理解
- 字符串匹配

12.1 引言

字符串可以看成是字符组成的数组。由于字符串是程序里经常需要处理的数据类型，因此有很多针对字符串处理的题目，以下是一些常见的类型。

12.2 字符串比较

242. Valid Anagram (Easy)

题目描述

判断两个字符串包含的字符是否完全相同。

输入输出样例

输入两个字符串，输出一个布尔值，表示两个字符串是否满足条件。

```
Input: s = "anagram", t = "nagaram"
Output: true
```

题解

我们可以利用哈希表或者数组统计两个数组中每个数字出现的频次，若频次相同，则说明它们包含的字符完全相同。

```
bool isAnagram(string s, string t) {
    if (s.length() != t.length()) {
        return false;
    }
    vector<int> counts(26, 0);
    for (int i = 0; i < s.length(); ++i) {
        ++counts[s[i]-'a'];
        --counts[t[i]-'a'];
    }
    for (int i = 0; i < 26; ++i) {
        if (counts[i]) {
            return false;
        }
```

```
    }
    return true;
}
```

205. Isomorphic Strings (Easy)

题目描述

判断两个字符串是否同构。同构的定义是，可以通过把一个字符串的某些相同的字符转换成另一些相同的字符，使得两个字符串相同，且两种不同的字符不能够被转换成同一种字符。

输入输出样例

输入两个字符串，输出一个布尔值，表示两个字符串是否满足条件。

```
Input: s = "paper", t = "title"
Output: true
```

在这个样例中，通过把 *s* 中的 p、a、e、r 字符转换成 t、i、l、e 字符，可以使得两个字符串相同，

题解

我们可以将问题转化一下：记录两个字符串每个位置的字符第一次出现的位置，如果两个字符串中相同位置的字符与它们第一次出现的位置一样，那么这两个字符串同构。举例来说，对于“paper”和“title”，假设我们现在遍历到第三个字符“p”和“t”，发现它们第一次出现的位置都在第一个字符，则说明目前位置满足同构。

```
bool isIsomorphic(string s, string t) {
    vector<int> s_first_index(256, 0), t_first_index(256, 0);
    for (int i = 0; i < s.length(); ++i) {
        if (s_first_index[s[i]] != t_first_index[t[i]]) {
            return false;
        }
        s_first_index[s[i]] = t_first_index[t[i]] = i + 1;
    }
    return true;
}
```

647. Palindromic Substrings (Medium)

题目描述

给定一个字符，求其有多少个回文子字符串。回文的定义是左右对称。

输入输出样例

输入是一个字符串，输出一个整数，表示回文子字符串的数量。

```
Input: "aaa"
Output: 6
```

六个回文子字符串分别是 ["a","a","a","aa","aa","aaa"]。

题解

我们可以从字符串的每个位置开始，向左向右延长，判断存在多少以当前位置为中轴的回文子字符串。

```
int countSubstrings(string s) {
    int count = 0;
    for (int i = 0; i < s.length(); ++i) {
        count += extendSubstrings(s, i, i); // 奇数长度
        count += extendSubstrings(s, i, i + 1); // 偶数长度
    }
    return count;
}

int extendSubstrings(string s, int l, int r) {
    int count = 0;
    while (l >= 0 && r < s.length() && s[l] == s[r]) {
        --l;
        ++r;
        ++count;
    }
    return count;
}
```

696. Count Binary Substrings (Easy)

题目描述

给定一个 0-1 字符串，求有多少非空子字符串的 0 和 1 数量相同。

输入输出样例

输入是一个字符串，输出一个整数，表示满足条件的子字符串的数量。

```
Input: "00110011"
Output: 6
```

在这个样例中，六个 0 和 1 数量相同的子字符串是 ["0011","01","1100","10","0011","01"]。

题解

从左往右遍历数组，记录和当前位置数字相同且连续的长度，以及其之前连续的不同数字的长度。举例来说，对于 00110 的最后一位，我们记录的相同数字长度是 1，因为只有一个连续 0；我们记录的不同数字长度是 2，因为在 0 之前有两个连续的 1。若不同数字的连续长度大于等于当前数字的连续长度，则说明存在一个且只存在一个以当前数字结尾的满足条件的子字符串。

```
int countBinarySubstrings(string s) {
    int pre = 0, cur = 1, count = 0;
    for (int i = 1; i < s.length(); ++i) {
        if (s[i] == s[i-1]) {
            ++cur;
        } else {
            pre = cur;
            cur = 1;
        }
        if (pre >= cur) {
            ++count;
        }
    }
    return count;
}
```

12.3 字符串理解

227. Basic Calculator II (Medium)

题目描述

给定一个包含加减乘除整数运算的字符串，求其运算结果，只保留整数。

输入输出样例

输入是一个合法的运算字符串，输出是一个整数，表示其运算结果。

```
Input: " 3+5 / 2 "
Output: 5
```

在这个样例中，因为除法的优先度高于加法，所以结果是 5 而非 4。

题解

如果我们在字符串左边加上一个加号，可以证明其并不改变运算结果，且字符串可以分割成多个 < 一个运算符，一个数字 > 对子的形式；这样一来我们就可以从左往右处理了。由于乘除的优先级高于加减，因此我们需要使用一个中间变量来存储高优先度的运算结果。

此类型题也考察很多细节处理，如无运算符的情况，和多个空格的情况等等。

```
// 主函数
int calculate(string s) {
    int i = 0;
    return parseExpr(s, i);
}

// 辅函数 - 递归parse从位置i开始的剩余字符串
int parseExpr(const string& s, int& i) {
    char op = '+';
    long left = 0, right = 0;
    while (i < s.length()) {
```

```
        if (s[i] != ' ') {
            long n = parseNum(s, i);
            switch (op) {
                case '+' : left += right; right = n; break;
                case '-' : left += right; right = -n; break;
                case '*' : right *= n; break;
                case '/' : right /= n; break;
            }
            if (i < s.length()) {
                op = s[i];
            }
        }
        ++i;
    }
    return left + right;
}

// 辅函数 - parse从位置i开始的一个数字
long parseNum(const string& s, int& i) {
    long n = 0;
    while (i < s.length() && isdigit(s[i])) {
        n = 10 * n + (s[i++] - '0');
    }
    return n;
}
```

12.4 字符串匹配

28. Implement strStr() (Easy)

题目描述

判断一个字符串是不是另一个字符串的子字符串，并返回其位置。

输入输出样例

输入一个母字符串和一个子字符串，输出一个整数，表示子字符串在母字符串的位置，若不存在则返回-1。

```
Input: haystack = "hello", needle = "ll"
Output: 2
```

题解

使用著名的Knuth-Morris-Pratt（KMP）算法，可以在 $O(m+n)$ 时间利用动态规划完成匹配。

```
// 主函数
int strStr(string haystack, string needle) {
    int k = -1, n = haystack.length(), p = needle.length();
    if (p == 0) return 0;
    vector<int> next(p, -1); // -1表示不存在相同的最大前缀和后缀
```

```
    calNext(needle, next); // 计算next数组
    for (int i = 0; i < n; ++i) {
        while (k > -1 && needle[k+1] != haystack[i]) {
            k = next[k]; // 有部分匹配，往前回溯
        }
        if (needle[k+1] == haystack[i]) {
            ++k;
        }
        if (k == p-1) {
            return i - p + 1; // 说明k移动到needle的最末端，返回相应的位置
        }
    }
    return -1;
}

// 辅函数 - 计算next数组
void calNext(const string &needle, vector<int> &next) {
    for (int j = 1, p = -1; j < needle.length(); ++j) {
        while (p > -1 && needle[p+1] != needle[j]) {
            p = next[p]; // 如果下一位不同，往前回溯
        }
        if (needle[p+1] == needle[j]) {
            ++p; // 如果下一位相同，更新相同的最大前缀和最大后缀长
        }
        next[j] = p;
    }
}
```

12.5 练习

基础难度

409. Longest Palindrome (Easy)

计算一组字符可以构成的回文字符串的最大长度，可以利用其它数据结构进行辅助统计。

3. Longest Substring Without Repeating Characters (Medium)

计算最长无重复子字符串，同样的，可以利用其它数据结构进行辅助统计。

进阶难度

772. Basic Calculator III (Hard)

题目 227 的 follow-up，十分推荐练习。

5. Longest Palindromic Substring (Medium)

类似于我们讲过的子序列问题，子数组或子字符串问题常常也可以用动态规划来解决。先使用动态规划写出一个 $O(n^2)$ 时间复杂度的算法，再搜索一下 Manacher's Algorithm，它可以在 $O(n)$ 时间解决这一问题。

第 13 章 指针三剑客之一：链表

内容提要

- 数据结构介绍
- 链表的基本操作
- 其它链表技巧

13.1 数据结构介绍

（单）链表是由节点和指针构成的数据结构，每个节点存有一个值，和一个指向下一个节点的指针，因此很多链表问题可以用递归来处理。不同于数组，链表并不能直接获取任意节点的值，必须要通过指针找到该节点后才能获取其值。同理，在未遍历到链表结尾时，我们也无法知道链表的长度，除非依赖其他数据结构储存长度。LeetCode 默认的链表表示方法如下。

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(nullptr) {}
};
```

由于在进行链表操作时，尤其是删除节点时，经常会因为对当前节点进行操作而导致内存或指针出现问题。有两个小技巧可以解决这个问题：一是尽量处理当前节点的下一个节点而非当前节点本身，二是建立一个虚拟节点 (dummy node)，使其指向当前链表的头节点，这样即使原链表所有节点全被删除，也会有一个 dummy 存在，返回 dummy->next 即可。

注意 一般来说，算法题不需要删除内存。在刷 LeetCode 的时候，如果想要删除一个节点，可以直接进行指针操作而无需回收内存。实际做软件工程时，对于无用的内存，笔者建议尽量显式回收，或利用智能指针。

13.2 链表的基本操作

206. Reverse Linked List (Easy)

题目描述

翻转一个链表。

输入输出样例

输入一个链表，输出该链表翻转后的结果。

```
Input: 1->2->3->4->5->nullptr
Output: 5->4->3->2->1->nullptr
```

题解

链表翻转是非常基础也一定要掌握的技能。我们提供了两种写法——递归和非递归，且我们建议你同时掌握这两种写法。

递归的写法为：

```
ListNode* reverseList(ListNode* head, ListNode* prev=nullptr) {
    if (!head) {
        return prev;
    }
    ListNode* next = head->next;
    head->next = prev;
    return reverseList(next, head);
}
```

非递归的写法为：

```
ListNode* reverseList(ListNode* head) {
    ListNode *prev = nullptr, *next;
    while (head) {
        next = head->next;
        head->next = prev;
        prev = head;
        head = next;
    }
    return prev;
}
```

21. Merge Two Sorted Lists (Easy)

题目描述

给定两个增序的链表，试将其合并成一个增序的链表。

输入输出样例

输入两个链表，输出一个链表，表示两个链表合并的结果。

```
Input: 1->2->4, 1->3->4
Output: 1->1->2->3->4->4
```

题解

我们提供了递归和非递归，共两种写法。递归的写法为：

```
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    if (!l2) {
        return l1;
    }
    if (!l1) {
        return l2;
    }
    if (l1->val > l2->val) {
```

```
        l2->next = mergeTwoLists(l1, l2->next);
        return l2;
    }
    l1->next = mergeTwoLists(l1->next, l2);
    return l1;
}
```

非递归的写法为：

```
ListNode* mergeTwoLists(ListNode *l1, ListNode *l2) {
    ListNode *dummy = new ListNode(0), *node = dummy;
    while (l1 && l2) {
        if (l1->val <= l2->val) {
            node->next = l1;
            l1 = l1->next;
        } else {
            node->next = l2;
            l2 = l2->next;
        }
        node = node->next;
    }
    node->next = l1? l1: l2;
    return dummy->next;
}
```

24. Swap Nodes in Pairs (Medium)

题目描述

给定一个矩阵，交换每个相邻的一对节点。

输入输出样例

输入一个链表，输出该链表交换后的结果。

```
Input: 1->2->3->4
Output: 2->1->4->3
```

题解

利用指针进行交换操作，没有太大难度，但一定要细心。

```
ListNode* swapPairs(ListNode* head) {
    ListNode *p = head, *s;
    if (p && p->next) {
        s = p->next;
        p->next = s->next;
        s->next = p;
        head = s;
        while (p->next && p->next->next) {
            s = p->next->next;
            p->next->next = s->next;
            s->next = p->next;
```

```
            p->next = s;
            p = s->next;
        }
    }
    return head;
}
```

13.3 其它链表技巧

160. Intersection of Two Linked Lists (Easy)

题目描述

给定两个链表，判断它们是否相交于一点，并求这个相交节点。

输入输出样例

输入是两条链表，输出是一个节点。如无相交节点，则返回一个空节点。

```
Input:
A:        a1 -> a2
                |
                v
                c1 -> c2 -> c3
                ^
                |
B: b1 -> b2 -> b3
Output: c1
```

题解

假设链表 A 的头节点到相交点的距离是 a，链表 B 的头节点到相交点的距离是 b，相交点到链表终点的距离为 c。我们使用两个指针，分别指向两个链表的头节点，并以相同的速度前进，若到达链表结尾，则移动到另一条链表的头节点继续前进。按照这种前进方法，两个指针会在 $a + b + c$ 次前进后同时到达相交节点。

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    ListNode *l1 = headA, *l2 = headB;
    while (l1 != l2) {
        l1 = l1? l1->next: headB;
        l2 = l2? l2->next: headA;
    }
    return l1;
}
```

234. Palindrome Linked List (Easy)

题目描述

以 $O(1)$ 的空间复杂度，判断链表是否回文。

输入输出样例

输入是一个链表，输出是一个布尔值，表示链表是否回文。

```
Input: 1->2->3->2->1
Output: true
```

题解

先使用快慢指针找到链表中点，再把链表切成两半；然后把后半段翻转；最后比较两半是否相等。

```cpp
// 主函数
bool isPalindrome(ListNode* head) {
    if (!head || !head->next) {
        return true;
    }
    ListNode *slow = head, *fast = head;
    while (fast->next && fast->next->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    slow->next = reverseList(slow->next);
    slow = slow->next;
    while (slow) {
        if (head->val != slow->val) {
            return false;
        }
        head = head->next;
        slow = slow->next;
    }
    return true;
}

// 辅函数
ListNode* reverseList(ListNode* head) {
    ListNode *prev = nullptr, *next;
    while (head) {
        next = head->next;
        head->next = prev;
        prev = head;
        head = next;
    }
    return prev;
}
```

13.4 练习

基础难度

83. Remove Duplicates from Sorted List (Easy)

虽然 LeetCode 没有强制要求，但是我们仍然建议你回收内存，尤其当题目要求你删除的时候。

328. Odd Even Linked List (Medium)

这道题其实很简单，千万不要把题目复杂化。

19. Remove Nth Node From End of List (Medium)

既然我们可以使用快慢指针找到中点，也可以利用类似的方法找到倒数第 n 个节点，无需遍历第二遍。

进阶难度

148. Sort List (Medium)

利用快慢指针找到链表中点后，可以对链表进行归并排序。

第 14 章 指针三剑客之二：树

内容提要

- 数据结构介绍
- 树的递归
- 层次遍历
- 前中后序遍历
- 二叉查找树
- 字典树

14.1 数据结构介绍

作为（单）链表的升级版，我们通常接触的树都是二叉树（binary tree），即每个节点最多有两个子节点；且除非题目说明，默认树中不存在循环结构。LeetCode 默认的树表示方法如下。

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

可以看出，其与链表的主要差别就是多了一个子节点的指针。

14.2 树的递归

对于一些简单的递归题，某些 LeetCode 达人喜欢写 one-line code，即用一行代码解决问题，把 if-else 判断语句压缩成问号冒号的形式。我们也会展示一些这样的代码，但是对于新手，笔者仍然建议您使用 if-else 判断语句。

在很多时候，树递归的写法与深度优先搜索的递归写法相同，因此本书不会区分二者。

104. Maximum Depth of Binary Tree (Easy)

题目描述

求一个二叉树的最大深度。

输入输出样例

输入是一个二叉树，输出是一个整数，表示该树的最大深度。

```
Input:
    3
   / \
  9  20
    /  \
   15   7
```

```
Output: 3
```

题解

利用递归，我们可以很方便地求得最大深度。

```
int maxDepth(TreeNode* root) {
    return root? 1 + max(maxDepth(root->left), maxDepth(root->right)): 0;
}
```

110. Balanced Binary Tree (Easy)

题目描述

判断一个二叉树是否平衡。树平衡的定义是，对于树上的任意节点，其两侧节点的最大深度的差值不得大于 1。

输入输出样例

输入是一个二叉树，输出一个布尔值，表示树是否平衡。

```
Input:
       1
      / \
     2   2
    / \
   3   3
  / \
 4   4
Output: false
```

题解

解法类似于求树的最大深度，但有两个不同的地方：一是我们需要先处理子树的深度再进行比较，二是如果我们在处理子树时发现其已经不平衡了，则可以返回一个-1，使得所有其长辈节点可以避免多余的判断（本题的判断比较简单，做差后取绝对值即可；但如果此处是一个开销较大的比较过程，则避免重复判断可以节省大量的计算时间）。

```
// 主函数
bool isBalanced(TreeNode* root) {
    return helper(root) != -1;
}

// 辅函数
int helper(TreeNode* root) {
    if (!root) {
        return 0;
    }
    int left = helper(root->left), right = helper(root->right);
```

```
    if (left == -1 || right == -1 || abs(left - right) > 1) {
        return -1;
    }
    return 1 + max(left, right);
}
```

543. Diameter of Binary Tree (Easy)

题目描述

求一个二叉树的最长直径。直径的定义是二叉树上任意两节点之间的无向距离。

输入输出样例

输入是一个二叉树，输出一个整数，表示最长直径。

```
Input:
      1
     / \
    2   3
   / \
  4   5
Output: 3
```

在这个样例中，最长直径是 [4,2,1,3] 和 [5,2,1,3]。

题解

同样的，我们可以利用递归来处理树。解题时要注意，在我们处理某个子树时，我们更新的最长直径值和递归返回的值是不同的。这是因为待更新的最长直径值是经过该子树根节点的最长直径（即两侧长度）；而函数返回值是以该子树根节点为端点的最长直径值（即一侧长度），使用这样的返回值才可以通过递归更新父节点的最长直径值）。

```
// 主函数
int diameterOfBinaryTree(TreeNode* root) {
    int diameter = 0;
    helper(root, diameter);
    return diameter;
}

// 辅函数
int helper(TreeNode* node, int& diameter) {
    if (!node) {
        return 0;
    }
    int l = helper(node->left, diameter), r = helper(node->right, diameter);
    diameter = max(l + r, diameter);
    return max(l, r) + 1;
}
```

437. Path Sum III (Easy)

题目描述

给定一个整数二叉树，求有多少条路径节点值的和等于给定值。

输入输出样例

输入一个二叉树和一个给定整数，输出一个整数，表示有多少条满足条件的路径。

```
Input: sum = 8, tree =
      10
     /  \
    5   -3
   / \    \
  3   2   11
 / \   \
3  -2   1
Output: 3
```

在这个样例中，和为 8 的路径一共有三个：[[5,3],[5,2,1],[-3,11]]。

题解

递归每个节点时，需要分情况考虑：（1）如果选取该节点加入路径，则之后必须继续加入连续节点，或停止加入节点（2）如果不选取该节点加入路径，则对其左右节点进行重新进行考虑。因此一个方便的方法是我们创建一个辅函数，专门用来计算连续加入节点的路径。

```
// 主函数
int pathSum(TreeNode* root, int sum) {
    return root? pathSumStartWithRoot(root, sum) +
        pathSum(root->left, sum) + pathSum(root->right, sum): 0;
}

// 辅函数
int pathSumStartWithRoot(TreeNode* root, int sum) {
    if (!root) {
        return 0;
    }
    int count = root->val == sum? 1: 0;
    count += pathSumStartWithRoot(root->left, sum - root->val);
    count += pathSumStartWithRoot(root->right, sum - root->val);
    return count;
}
```

101. Symmetric Tree (Easy)

题目描述

判断一个二叉树是否对称。

输入输出样例

输入一个二叉树，输出一个布尔值，表示该树是否对称。

```
Input:
    1
   / \
  2   2
 / \ / \
3  4 4  3
Output: true
```

题解

判断一个树是否对称等价于判断左右子树是否对称。笔者一般习惯将判断两个子树是否相等或对称类型的题的解法叫做“四步法”：（1）如果两个子树都为空指针，则它们相等或对称（2）如果两个子树只有一个为空指针，则它们不相等或不对称（3）如果两个子树根节点的值不相等，则它们不相等或不对称（4）根据相等或对称要求，进行递归处理。

```
// 主函数
bool isSymmetric(TreeNode *root) {
    return root? isSymmetric(root->left, root->right): true;
}

// 辅函数
bool isSymmetric(TreeNode* left, TreeNode* right) {
    if (!left && !right) {
        return true;
    }
    if (!left || !right) {
        return false;
    }
    if (left->val != right->val) {
        return false;
    }
    return isSymmetric(left->left, right->right) && isSymmetric(left->right,
        right->left);
}
```

1110. Delete Nodes And Return Forest (Medium)

题目描述

给定一个整数二叉树和一些整数，求删掉这些整数对应的节点后，剩余的子树。

输入输出样例

输入是一个整数二叉树和一个一维整数数组，输出一个数组，每个位置存储一个子树（的根节点）。

```
Input: to_delete = [3,5], tree =
    1
   / \
  2   3
 / \ / \
4  5 6  7
Output: [
    1
   /
  2
 /
4   ,6 ,7]
```

题解

这道题最主要需要注意的细节是如果通过递归处理原树，以及需要在什么时候断开指针。同时，为了便于寻找待删除节点，可以建立一个哈希表方便查找。笔者强烈建议读者在看完题解后，自己写一遍本题，加深对于递归的理解和运用能力。

```
// 主函数
vector<TreeNode*> delNodes(TreeNode* root, vector<int>& to_delete) {
    vector<TreeNode*> forest;
    unordered_set<int> dict(to_delete.begin(), to_delete.end());
    root = helper(root, dict, forest);
    if (root) {
        forest.push_back(root);
    }
    return forest;
}

// 辅函数
TreeNode* helper(TreeNode* root, unordered_set<int> & dict, vector<TreeNode*> &
     forest) {
    if (!root) {
        return root;
    }
    root->left = helper(root->left, dict, forest);
    root->right = helper(root->right, dict, forest);
    if (dict.count(root->val)) {
        if (root->left) {
            forest.push_back(root->left);
        }
        if (root->right) {
            forest.push_back(root->right);
        }
        root = NULL;
    }
    return root;
}
```

14.3 层次遍历

我们可以使用广度优先搜索进行层次遍历。注意，不需要使用两个队列来分别存储当前层的节点和下一层的节点，因为在开始遍历一层的节点时，当前队列中的节点数就是当前层的节点数，只要控制遍历这么多节点数，就能保证这次遍历的都是当前层的节点。

637. Average of Levels in Binary Tree (Easy)

题目描述

给定一个二叉树，求每一层的节点值的平均数。

输入输出样例

输入是一个二叉树，输出是一个一维数组，表示每层节点值的平均数。

```
Input:
    3
   / \
  9  20
    /  \
   15   7
Output: [3, 14.5, 11]
```

题解

利用广度优先搜索，我们可以很方便地求取每层的平均值。

```
vector<double> averageOfLevels(TreeNode* root) {
    vector<double> ans;
    if (!root) {
        return ans;
    }
    queue<TreeNode*> q;
    q.push(root);
    while (!q.empty()) {
        int count = q.size();
        double sum = 0;
        for (int i = 0; i < count; ++i) {
            TreeNode* node = q.front();
            q.pop();
            sum += node->val;
            if (node->left) {
                q.push(node->left);
            }
            if (node->right) {
                q.push(node->right);
            }
        }
        ans.push_back(sum / count);
    }
    return ans;
```

```
}
```

14.4 前中后序遍历

前序遍历、中序遍历和后序遍历是三种利用深度优先搜索遍历二叉树的方式。它们是在对节点访问的顺序有一点不同，其它完全相同。考虑如下一棵树，

```
    1
   / \
  2   3
 / \   \
4   5   6
```

前序遍历先遍历父结点，再遍历左结点，最后遍历右节点，我们得到的遍历顺序是 [1 2 4 5 3 6]。

```
void preorder(TreeNode* root) {
    visit(root);
    preorder(root->left);
    preorder(root->right);
}
```

中序遍历先遍历左节点，再遍历父结点，最后遍历右节点，我们得到的遍历顺序是 [4 2 5 1 3 6]。

```
void inorder(TreeNode* root) {
    inorder(root->left);
    visit(root);
    inorder(root->right);
}
```

后序遍历先遍历左节点，再遍历右结点，最后遍历父节点，我们得到的遍历顺序是 [4 5 2 6 3 1]。

```
void postorder(TreeNode* root) {
    postorder(root->left);
    postorder(root->right);
    visit(root);
}
```

105. Construct Binary Tree from Preorder and Inorder Traversal (Medium)

题目描述

给定一个二叉树的前序遍历和中序遍历结果，尝试复原这个树。已知树里不存在重复值的节点。

输入输出样例

输入是两个一维数组，分别表示树的前序遍历和中序遍历结果；输出是一个二叉树。

```
Input: preorder = [4,9,20,15,7], inorder = [9,4,15,20,7]
Output:
    4
   / \
  9  20
    /  \
   15   7
```

题解

我们通过本题的样例讲解一下本题的思路。前序遍历的第一个节点是 4，意味着 4 是根节点。我们在中序遍历结果里找到 4 这个节点，根据中序遍历的性质可以得出，4 在中序遍历数组位置的左子数组为左子树，节点数为 1，对应的是前序排列数组里 4 之后的 1 个数字（9）；4 在中序遍历数组位置的右子数组为右子树，节点数为 3，对应的是前序排列数组里最后的 3 个数字。有了这些信息，我们就可以对左子树和右子树进行递归复原了。为了方便查找数字的位置，我们可以用哈希表预处理中序遍历的结果。

```cpp
// 主函数
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    if (preorder.empty()) {
        return nullptr;
    }
    unordered_map<int, int> hash;
    for (int i = 0; i < preorder.size(); ++i) {
        hash[inorder[i]] = i;
    }
    return buildTreeHelper(hash, preorder, 0, preorder.size() - 1, 0);
}

// 辅函数
TreeNode* buildTreeHelper(unordered_map<int, int> & hash, vector<int>& preorder
    , int s0, int e0, int s1) {
    if (s0 > e0) {
        return nullptr;
    }
    int mid = preorder[s1], index = hash[mid], leftLen = index - s0 - 1;
    TreeNode* node = new TreeNode(mid);
    node->left = buildTreeHelper(hash, preorder, s0, index - 1, s1 + 1);
    node->right = buildTreeHelper(hash, preorder, index + 1, e0, s1 + 2 +
        leftLen);
    return node;
}
```

144. Binary Tree Preorder Traversal (Medium)

题目描述

不使用递归，实现二叉树的前序遍历。

输入输出样例

输入一个二叉树，输出一个数组，为二叉树前序遍历的结果，

```
Input:
   1
    \
     2
    /
   3
Output: [1,2,3]
```

题解

因为递归的本质是栈调用，因此我们可以通过栈来实现前序遍历。注意入栈的顺序。

```
vector<int> preorderTraversal(TreeNode* root) {
    vector<int> ret;
    if (!root) {
        return ret;
    }
    stack<TreeNode*> s;
    s.push(root);
    while (!s.empty()) {
        TreeNode* node = s.top();
        s.pop();
        ret.push_back(node->val);
        if (node->right) {
            s.push(node->right); // 先右后左，保证左子树先遍历
        }
        if (node->left) {
            s.push(node->left);
        }
    }
    return ret;
}
```

14.5 二叉查找树

二叉查找树（Binary Search Tree, BST）是一种特殊的二叉树：对于每个父节点，其左子树中所有节点的值小于等于父结点的值，其右子树中所有节点的值大于等于父结点的值。因此对于一个二叉查找树，我们可以在 $O(\log n)$ 的时间内查找一个值是否存在：从根节点开始，若当前节点的值大于查找值则向左下走，若当前节点的值小于查找值则向右下走。同时因为二叉查找树是有序的，对其中序遍历的结果即为排好序的数组。

一个二叉查找树的实现如下。

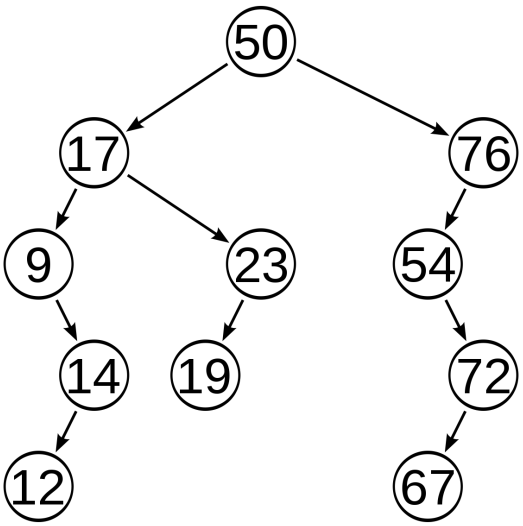

图 14.1: 二叉查找树样例

```
template <class T>
class BST {
    struct Node {
        T data;
        Node* left;
        Node* right;
    };

    Node* root;

    Node* makeEmpty(Node* t) {
        if (t == NULL) return NULL;
        makeEmpty(t->left);
        makeEmpty(t->right);
        delete t;
        return NULL;
    }

    Node* insert(Node* t, T x) {
        if (t == NULL) {
            t = new Node;
            t->data = x;
            t->left = t->right = NULL;
        } else if (x < t->data) {
            t->left = insert(t->left, x);
        } else if (x > t->data) {
            t->right = insert(t->right, x);
        }
        return t;
    }

    Node* find(Node* t, T x) {
        if (t == NULL) return NULL;
        if (x < t->data) return find(t->left, x);
        if (x > t->data) return find(t->right, x);
        return t;
    }
```

```
    Node* findMin(Node* t) {
        if (t == NULL || t->left == NULL) return t;
        return findMin(t->left);
    }

    Node* findMax(Node* t) {
        if (t == NULL || t->right == NULL) return t;
        return findMax(t->right);
    }

    Node* remove(Node* t, T x) {
        Node* temp;
        if (t == NULL) return NULL;
        else if (x < t->data) t->left = remove(t->left, x);
        else if (x > t->data) t->right = remove(t->right, x);
        else if (t->left && t->right) {
            temp = findMin(t->right);
            t->data = temp->data;
            t->right = remove(t->right, t->data);
        } else {
            temp = t;
            if (t->left == NULL) t = t->right;
            else if (t->right == NULL) t = t->left;
            delete temp;
        }
        return t;
    }

public:
    BST(): root(NULL) {}

    ~BST() {
        root = makeEmpty(root);
    }

    void insert(T x) {
        insert(root, x);
    }

    void remove(T x) {
        remove(root, x);
    }
};
```

99. Recover Binary Search Tree (Hard)

题目描述

给定一个二叉查找树，已知有两个节点被不小心交换了，试复原此树。

输入输出样例

输入是一个被误交换两个节点的二叉查找树，输出是改正后的二叉查找树。

```
Input:
  3
 / \
1   4
   /
  2
Output:
  2
 / \
1   4
   /
  3
```

在这个样例中，2 和 3 被不小心交换了。

题解

我们可以使用中序遍历这个二叉查找树，同时设置一个 prev 指针，记录当前节点中序遍历时的前节点。如果当前节点大于 prev 节点的值，说明需要调整次序。有一个技巧是如果遍历整个序列过程中只出现了一次次序错误，说明就是这两个相邻节点需要被交换；如果出现了两次次序错误，那就需要交换这两个节点。

```
// 主函数
void recoverTree(TreeNode* root) {
    TreeNode *mistake1 = nullptr, *mistake2 = nullptr, *prev = nullptr;
    inorder(root, mistake1, mistake2, prev);
    if (mistake1 && mistake2) {
        int temp = mistake1->val;
        mistake1->val = mistake2->val;
        mistake2->val = temp;
    }
}

// 辅函数
void inorder(TreeNode* root, TreeNode*& mistake1, TreeNode*& mistake2, TreeNode
    *& prev) {
    if (!root) {
        return;
    }
    if (root->left) {
        inorder(root->left, mistake1, mistake2, prev);
    }
    if (prev && root->val < prev->val) {
        if (!mistake1) {
            mistake1 = prev;
            mistake2 = root;
        } else {
            mistake2 = root;
        }
        cout << mistake1->val;
        cout << mistake2->val;
    }
    prev = root;
    if (root->right) {
        inorder(root->right, mistake1, mistake2, prev);
```

```
    }
}
```

669. Trim a Binary Search Tree (Easy)

题目描述

给定一个二叉查找树和两个整数 L 和 R，且 $L < R$，试修剪此二叉查找树，使得修剪后所有节点的值都在 $[L, R]$ 的范围内。

输入输出样例

输入是一个二叉查找树和两个整数 L 和 R，输出一个被修剪好的二叉查找树。

```
Input: L = 1, R = 3, tree =
    3
   / \
  0   4
   \
    2
   /
  1
Output:
      3
     /
   2
  /
 1
```

题解

利用二叉查找树的大小关系，我们可以很容易地利用递归进行树的处理。

```
TreeNode* trimBST(TreeNode* root, int L, int R) {
    if (!root) {
        return root;
    }
    if (root->val > R) {
        return trimBST(root->left, L, R);
    }
    if (root->val < L) {
        return trimBST(root->right, L, R);
    }
    root->left = trimBST(root->left, L, R);
    root->right = trimBST(root->right, L, R);
    return root;
}
```

14.6 字典树

字典树（Trie）用于判断字符串是否存在或者是否具有某种字符串前缀。

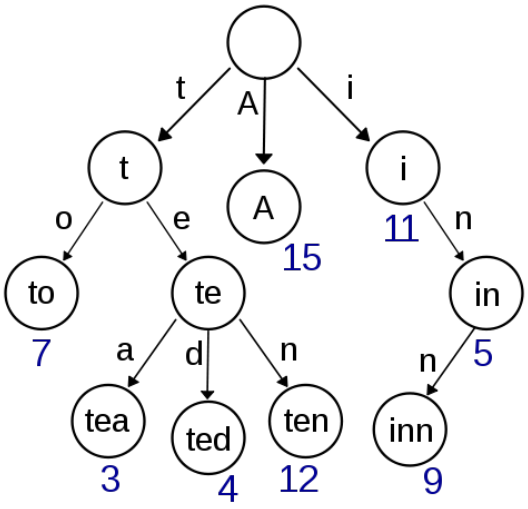

图 14.2: 字典树，存储了单词 A、to、tea、ted、ten、i、in 和 inn，以及它们的频率

为什么需要用字典树解决这类问题呢？假如我们有一个储存了近万个单词的字典，即使我们使用哈希，在其中搜索一个单词的实际开销也是非常大的，且无法轻易支持搜索单词前缀。然而由于一个英文单词的长度 n 通常在 10 以内，如果我们使用字典树，则可以在 $O(n)$——近似 $O(1)$ 的时间内完成搜索，且额外开销非常小。

208. Implement Trie (Prefix Tree) (Medium)

题目描述

尝试建立一个字典树，支持快速插入单词、查找单词、查找单词前缀的功能。

输入输出样例

以下是数据结构的调用样例。

```
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");  // true
trie.search("app");    // false
trie.startsWith("app"); // true
trie.insert("app");
trie.search("app");    //  true
```

题解

以下是字典树的典型实现方法。

```
class TrieNode {
public:
    TrieNode* childNode[26];
    bool isVal;
    TrieNode(): isVal(false) {
        for (int i = 0; i < 26; ++i) {
```

```
            childNode[i] = nullptr;
        }
    }
};

class Trie {
    TrieNode* root;
public:
    Trie(): root(new TrieNode()) {}

    // 向字典树插入一个词
    void insert(string word) {
        TrieNode* temp = root;
        for (int i = 0; i < word.size(); ++i) {
            if (!temp->childNode[word[i]-'a']) {
                temp->childNode[word[i]-'a'] = new TrieNode();
            }
            temp = temp->childNode[word[i]-'a'];
        }
        temp->isVal = true;
    }

    // 判断字典树里是否有一个词
    bool search(string word) {
        TrieNode* temp = root;
        for (int i = 0; i < word.size(); ++i) {
            if (!temp) {
                break;
            }
            temp = temp->childNode[word[i]-'a'];
        }
        return temp? temp->isVal: false;
    }

    // 判断字典树是否有一个以词开始的前缀
    bool startsWith(string prefix) {
        TrieNode* temp = root;
        for (int i = 0; i < prefix.size(); ++i) {
            if (!temp) {
                break;
            }
            temp = temp->childNode[prefix[i]-'a'];
        }
        return temp;
    }
};
```

14.7 练习

基础难度

226. Invert Binary Tree (Easy)

巧用递归，你可以在五行内完成这道题。

617. Merge Two Binary Trees (Easy)

同样的，利用递归可以轻松搞定。

572. Subtree of Another Tree (Easy)

子树是对称树的姊妹题，写法也十分类似。

404. Sum of Left Leaves (Easy)

怎么判断一个节点是不是左节点呢？一种可行的方法是，在辅函数里多传一个参数，表示当前节点是不是父节点的左节点。

513. Find Bottom Left Tree Value (Easy)

最左下角的节点满足什么条件？针对这种条件，我们该如何找到它？

538. Convert BST to Greater Tree (Easy)

尝试利用某种遍历方式来解决此题，每个节点只需遍历一次。

235. Lowest Common Ancestor of a Binary Search Tree (Easy)

利用 BST 的独特性质，这道题可以很轻松完成。

530. Minimum Absolute Difference in BST (Easy)

还记得我们所说的，对于 BST 应该利用哪种遍历吗？

进阶难度

889. Construct Binary Tree from Preorder and Postorder Traversal (Medium)

给定任意两种遍历结果，我们都可以重建树的结构。

106. Construct Binary Tree from Inorder and Postorder Traversal (Medium)

给定任意两种遍历结果，我们都可以重建树的结构。

94. Binary Tree Inorder Traversal (Medium)

因为前中序后遍历是用递归实现的，而递归的底层实现是栈操作，因此我们总能用栈实现。

145. Binary Tree Postorder Traversal (Medium)

因为前中序后遍历是用递归实现的，而递归的底层实现是栈操作，因此我们总能用栈实现。

236. Lowest Common Ancestor of a Binary Tree (Medium)

现在不是 BST，而是普通的二叉树了，该怎么办？

109. Convert Sorted List to Binary Search Tree (Medium)

把排好序的链表变成 BST。为了使得 BST 尽量平衡，我们需要寻找链表的中点。

897. Increasing Order Search Tree (Easy)

把 BST 压成一个链表，务必考虑清楚指针操作的顺序，否则可能会出现环路。

653. Two Sum IV - Input is a BST (Easy)

啊哈，这道题可能会把你骗到。

450. Delete Node in a BST (Medium)

当寻找到待删节点时，你可以分情况考虑——当前节点是叶节点、只有一个子节点和有两个子节点。建议同时回收内存。

第 15 章 指针三剑客之三：图

内容提要

- 数据结构介绍
- 二分图
- 拓扑排序

15.1 数据结构介绍

作为指针三剑客之三，图是树的升级版。图通常分为有向（directed）或无向（undirected），有循环（cyclic）或无循环（acyclic），所有节点相连（connected）或不相连（disconnected）。树即是一个相连的无向无环图，而另一种很常见的图是有向无环图（Directed Acyclic Graph，DAG）。

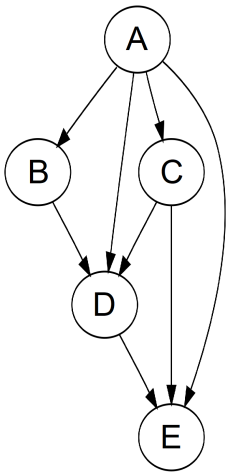

图 15.1: 有向无环图样例

图通常有两种表示方法。假设图中一共有 n 个节点、m 条边。第一种表示方法是邻接矩阵（adjacency matrix）：我们可以建立一个 $n \times n$ 的矩阵 G，如果第 i 个节点连向第 j 个节点，则 G[i][j] = 1，反之为 0；如果图是无向的，则这个矩阵一定是对称矩阵，即 G[i][j] = G[j][i]。第二种表示方法是邻接链表（adjacency list）：我们可以建立一个大小为 n 的数组，每个位置 i 储存一个数组或者链表，表示第 i 个节点连向的其它节点。邻接矩阵空间开销比邻接链表大，但是邻接链表不支持快速查找 i 和 j 是否相连，因此两种表示方法可以根据题目需要适当选择。除此之外，我们也可以直接用一个 $m \times 2$ 的矩阵储存所有的边。

15.2 二分图

二分图算法也称为染色法，是一种广度优先搜索。如果可以用两种颜色对图中的节点进行着色，并且保证相邻的节点颜色不同，那么图为二分。

785. Is Graph Bipartite? (Medium)

题目描述

给定一个图，判断其是否可以二分，

输入输出样例

输入是邻接链表表示的图（如位置 0 的邻接链表为 [1,3]，表示 0 与 1、0 与 3 相连）；输出是一个布尔值，表示图是否二分。

```
Input: [[1,3], [0,2], [1,3], [0,2]]
0----1
|    |
|    |
3----2
Output: true
```

在这个样例中，我们可以把 {0,2} 分为一组，把 {1,3} 分为另一组。

题解

利用队列和广度优先搜索，我们可以对未染色的节点进行染色，并且检查是否有颜色相同的相邻节点存在。注意在代码中，我们用 0 表示未检查的节点，用 1 和 2 表示两种不同的颜色。

```
bool isBipartite(vector<vector<int>>& graph) {
    int n = graph.size();
    if (n == 0) {
        return true;
    }
    vector<int> color(n, 0);
    queue<int> q;
    for (int i = 0; i < n; ++i) {
        if (!color[i]) {
            q.push(i);
            color[i] = 1;
        }
        while (!q.empty()) {
            int node = q.front();
            q.pop();
            for (const int & j: graph[node]) {
                if (color[j] == 0) {
                    q.push(j);
                    color[j] = color[node] == 2 ? 1 : 2;
                } else if (color[node] == color[j]) {
                    return false;
                }
            }
        }
    }
    return true;
}
```

15.3 拓扑排序

拓扑排序（topological sort）是一种常见的，对有向无环图排序的算法。给定有向无环图中的 N 个节点，我们把它们排序成一个线性序列；若原图中节点 i 指向节点 j，则排序结果中 i 一定在 j 之前。拓扑排序的结果不是唯一的，只要满足以上条件即可。

210. Course Schedule II (Medium)

题目描述

给定 N 个课程和这些课程的前置必修课，求可以一次性上完所有课的顺序。

输入输出样例

输入是一个正整数，表示课程数量，和一个二维矩阵，表示所有的有向边（如 [1,0] 表示上课程 1 之前必须先上课程 0）。输出是一个一维数组，表示拓扑排序结果。

```
Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: [0,1,2,3]
```

在这个样例中，另一种可行的顺序是 [0,2,1,3]。

题解

我们可以先建立一个邻接矩阵表示图，方便进行直接查找。这里注意我们将所有的边反向，使得如果课程 i 指向课程 j，那么课程 i 需要在课程 j 前面先修完。这样更符合我们的直观理解。

拓扑排序也可以被看成是广度优先搜索的一种情况：我们先遍历一遍所有节点，把入度为 0 的节点（即没有前置课程要求）放在队列中。在每次从队列中获得节点时，我们将该节点放在目前排序的末尾，并且把它指向的课程的入度各减 1；如果在这个过程中有课程的所有前置必修课都已修完（即入度为 0），我们把这个节点加入队列中。当队列的节点都被处理完时，说明所有的节点都已排好序，或因图中存在循环而无法上完所有课程。

```
vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {
    vector<vector<int>> graph(numCourses, vector<int>());
    vector<int> indegree(numCourses, 0), res;
    for (const auto & prerequisite: prerequisites) {
        graph[prerequisite[1]].push_back(prerequisite[0]);
        ++indegree[prerequisite[0]];
    }
    queue<int> q;
    for (int i = 0; i < indegree.size(); ++i) {
        if (!indegree[i]) {
            q.push(i);
        }
    }
    while (!q.empty()) {
        int u = q.front();
        res.push_back(u);
        q.pop();
        for (auto v: graph[u]) {
```

```
            --indegree[v];
            if (!indegree[v]) {
                q.push(v);
            }
        }
    }
    for (int i = 0; i < indegree.size(); ++i) {
        if (indegree[i]) {
            return vector<int>();
        }
    }
    return res;
}
```

15.4 练习

基础难度

1059. All Paths from Source Lead to Destination (Medium)

虽然使用深度优先搜索可以解决大部分的图遍历问题，但是注意判断是否陷入了环路。

进阶难度

1135. Connecting Cities With Minimum Cost (Medium)

笔者其实已经把这道题的题解写好了，才发现这道题是需要解锁才可以看的题目。为了避免版权纠纷，故将其移至练习题内。本题考察最小生成树（minimum spanning tree, MST）的求法，通常可以用两种方式求得：Prim's Algorithm，利用优先队列选择最小的消耗；以及 Kruskal's Algorithm，排序后使用并查集。

882. Reachable Nodes In Subdivided Graph (Hard)

这道题笔者考虑了很久，最终决定把它放在练习题而非详细讲解。本题是经典的节点最短距离问题，常用的算法有 Bellman-Ford 单源最短路算法，以及 Dijkstra 无负边单源最短路算法。虽然经典，但是 LeetCode 很少有相关的题型，因此这里仅供读者自行深入学习。

第 16 章 更加复杂的数据结构

内容提要

- 引言
- 并查集
- 复合数据结构

16.1 引言

目前为止，我们接触了大量的数据结构，包括利用指针实现的三剑客和 C++ 自带的 STL 等。对于一些题目，我们不仅需要利用多个数据结果解决问题，还需要把这些数据结构进行嵌套和联动，进行更为复杂、更为快速的操作。

16.2 并查集

并查集（union-find, 或 disjoint set）可以动态地连通两个点，并且可以非常快速地判断两个点是否连通。假设存在 n 个节点，我们先将所有节点的父亲标为自己；每次要连接节点 i 和 j 时，我们可以将 i 的父亲标为 j；每次要查询两个节点是否相连时，我们可以查找 i 和 j 的祖先是否最终为同一个人。

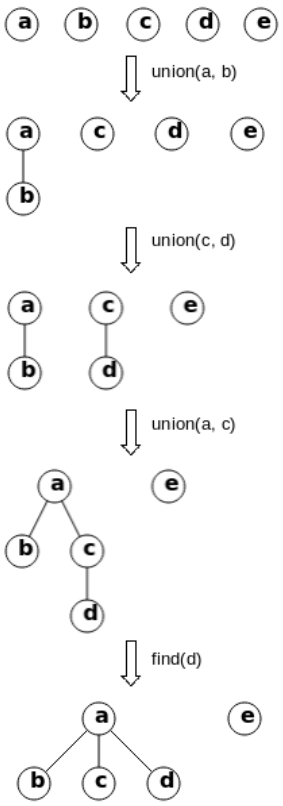

图 16.1: 并查集样例，其中 union 操作可以将两个集合连在一起，find 操作可以查找给定节点的祖先，并且如果可以的话，将集合的层数/高度降低

684. Redundant Connection (Medium)

题目描述

在无向图找出一条边，移除它之后该图能够成为一棵树（即无向无环图）。如果有多个解，返回在原数组中位置最靠后的那条边。

输入输出样例

输入是一个二维数组，表示所有的边（对应的两个节点）；输出是一个一维数组，表示需要移除的边（对应的两个节点）。

```
Input: [[1,2], [1,3], [2,3]]
  1
 / \
2 - 3
Output: [2,3]
```

题解

因为需要判断是否两个节点被重复连通，所以我们可以使用并查集来解决此类问题。具体实现算法如下所示。

```
class UF {
    vector<int> id;
public:
    UF(int n): id(n){
        iota(id.begin(), id.end(), 0); // iota函数可以把数组初始化为0到n-1
    }

    int find(int p) {
        while (p != id[p]) {
            p = id[p];
        }
        return p;
    }

    void connect(int p, int q) {
        id[find(p)] = find(q);
    }

    bool isConnected(int p, int q) {
        return find(p) == find(q);
    }
};

class Solution {
public:
    vector<int> findRedundantConnection(vector<vector<int>>& edges) {
        int n = edges.size();
        UF uf(n + 1);
        for (auto e: edges) {
            int u = e[0], v = e[1];
```

```
            if (uf.isConnected(u, v)) {
                return e;
            }
            uf.connect(u, v);
        }
        return vector<int>{-1, -1};
    }
};
```

为了加速查找，我们可以使用路径压缩和按秩合并来优化并查集。其具体写法如下所示。

```
class UF {
    vector<int> id, size;
public:
    UF(int n): id(n), size(n, 1) {
        iota(id.begin(), id.end(), 0); // iota函数可以把数组初始化为0到n-1
    }

    int find(int p) {
        while (p != id[p]) {
            id[p] = id[id[p]]; // 路径压缩，使得下次查找更快
            p = id[p];
        }
        return p;
    }

    void connect(int p, int q) {
        int i = find(p), j = find(q);
        if (i != j) {
            // 按秩合并：每次合并都把深度较小的集合合并在深度较大的集合下面
            if (size[i] < size[j]) {
                id[i] = j;
                size[j] += size[i];
            } else {
                id[j] = i;
                size[i] += size[j];
            }
        }
    }

    bool isConnected(int p, int q) {
        return find(p) == find(q);
    }
};
```

16.3 复合数据结构

这一类题通常采用 unordered_map 或 map 辅助记录，从而加速寻址；再配上 vector 或者 list 进行数据储存，从而加速连续选址或删除值。

146. LRU Cache (Medium)

题目描述

设计一个固定大小的，最少最近使用缓存 (least recently used cache, LRU)。每次将信息插入未满的缓存的时候，以及更新或查找一个缓存内存在的信息的时候，将该信息标为最近使用。在缓存满的情况下将一个新信息插入的时候，需要移除最旧的信息，插入新信息，并将该信息标为最近使用。

输入输出样例

以下是数据结构的调用样例。给定一个大小为 n 的缓存，我们希望其使用最少最近使用策略进行数据储存。

```
LRUCache cache = new LRUCache( 2 /* capacity */ );
cache.put(1, 1);
cache.put(2, 2);
cache.get(1);      // returns 1
cache.put(3, 3);   // evicts key 2
cache.get(2);      // returns -1 (not found)
cache.put(4, 4);   // evicts key 1
cache.get(1);      // returns -1 (not found)
cache.get(3);      // returns 3
cache.get(4);      // returns 4
```

题解

我们采用一个链表 list<pair<int, int>> 来储存信息的 key 和 value，链表的链接顺序即为最近使用的新旧顺序，最新的信息在链表头节点。同时我们需要一个嵌套着链表的迭代器的 unordered_map<int, list<pair<int, int>>::iterator> 进行快速搜索，存迭代器的原因是方便调用链表的 splice 函数来直接更新查找成功（cash hit）时的信息，即把迭代器对应的节点移动为链表的头节点。

```
class LRUCache{
    unordered_map<int, list<pair<int, int>>::iterator> hash;
    list<pair<int, int>> cache;
    int size;
public:
    LRUCache(int capacity):size(capacity) {}

    int get(int key) {
        auto it = hash.find(key);
        if (it == hash.end()) {
            return -1;
        }
        cache.splice(cache.begin(), cache, it->second);
        return it->second->second;
    }

    void put(int key, int value) {
        auto it = hash.find(key);
        if (it != hash.end()) {
            it->second->second = value;
```

```
            return cache.splice(cache.begin(), cache, it->second);
        }
        cache.insert(cache.begin(), make_pair(key, value));
        hash[key] = cache.begin();
        if (cache.size() > size) {
            hash.erase(cache.back().first);
            cache.pop_back();
        }
    }
};
```

16.4 练习

基础难度

1135. Connecting Cities With Minimum Cost (Medium)

使用并查集，按照 Kruskal's Algorithm 把这道题再解决一次吧。

380. Insert Delete GetRandom O(1) (Medium)

设计一个插入、删除和随机取值均为 $O(1)$ 时间复杂度的数据结构。

进阶难度

432. All O‘one Data Structure (Hard)

设计一个 increaseKey，decreaseKey，getMaxKey，getMinKey 均为 $O(1)$ 时间复杂度的数据结构。

716. Max Stack (Easy)

设计一个支持 push，pop，top，getMax 和 popMax 的 stack。可以用类似 LRU 的方法降低时间复杂度，但是因为想要获得的是最大值，我们应该把 unordered_map 换成哪一种数据结构呢?

第 17 章 后记

本书节选和改编于作者的个人笔记整理，其含有更多内容，但缺少相关的题解，作者一般用其来巩固基础知识。

如果您觉得这本书对您有帮助，不妨打赏一下作者哟 ~~~

图 17.1: 微信打赏二维码

另外，如果您有任何建议和咨询，也可以加作者的微信 synchro9z。我会尽快回复您的信息。