



LeetCode 101: 力扣刷题指南 (第二版)

LeetCode 101: A Grinding Guide (Second Edition)

作者: 高畅 Chang Gao

版本: 预览版 1.90b (未完成, 正式版 2.00 即将发布)

一个面向有一定的编程基础, 但缺乏刷题经验的读者的教科书和工具书。

序

2019 年底，本书第一版在 GitHub 上正式发布，反响十分热烈。过去的五年，作者积累了大量的工作经验，同时也越发觉得这本书存在很多纰漏和不完善之处。于是在 2024 年底，作者重新整理了本书的内容，发布了第二版。

本书分为算法、数学和数据结构三大部分，详细讲解了刷 LeetCode 时常用的技巧。在第一版时，为了强行呼应 101（在英文里是入门的意思），作者把题目精简到了 101 道。但现如今面试中可能考察的题型越来越多，所以第二版增加了一些高频题目，方便读者查漏补缺。

本书所有常规题目均附有 C++ 和 Python3 题解。由于本书的目的不是学习 C++ 和 Python 语言，作者在行文时不会过多解释语法细节，而且会适当使用一些新 C++ 标准的语法。截止 2024 年底，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。

感谢 GitHub 用户 CyC2018 的 LeetCode 题解，它对于作者早期的整理起到了很大的帮助作用。感谢 ElegantBook 提供的精美 L^AT_EX 模版，使得作者可以轻松地把笔记变成看起来更专业的电子书。另外，第二版书的封面图片是作者于 2024 年 7 月，在旧金山金门大桥北的 Sausalito 小镇海边拍摄。当时还买了一筒三球超大份冰淇淋，可惜实在是太甜了，吃了几口实在是吃不下去，只能扔掉。

如果您觉得这本书对您有帮助，不妨打赏一下作者哟：



图 1：微信打赏二维码

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，作者会将您加入鸣谢名单（见后记）。

另外，如果您有任何建议和咨询，欢迎前往作者的[个人网站](#)联系作者。如果需要私信，请通过邮箱或领英发送消息，作者会尽快回复。

请谨记，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。

重要声明

本书 GitHub 地址：github.com/changgyhub/leetcode_101

由于本书的目的是分享和教学，本书永久免费，也禁止任何营利性利用。欢迎以学术为目的的分享和传阅。由于作者不对 LeetCode 的任何题目拥有版权，一切题目版权以 LeetCode 官方为准，且本书不会展示 LeetCode 付费会员专享题目的内容或题解。

简介

打开 LeetCode 网站，如果我们按照题目类型数量分类，最多的几个题型包括数组、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，在按照类型进行分类的同时，也按照难易程度进行由浅入深的讲解。

第一个大分类是算法。本书先从最简单的贪心算法讲起，然后逐渐进阶到双指针、排序算法、二分查找和搜索算法，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，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。

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、Python3 自带的常见数据结构、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。

目 录

1 最易懂的贪心算法	1	4.2 快速选择	22
1.1 算法解释	1	4.3 桶排序	23
1.2 分配问题	1	4.4 练习	25
1.3 区间问题	3		
1.4 练习	4		
2 玩转双指针	6	5 一切皆可搜索	26
2.1 算法解释	6	5.1 算法解释	26
2.2 Two Sum	7	5.2 深度优先搜索	26
2.3 归并两个有序数组	8	5.3 回溯法	32
2.4 快慢指针	9	5.4 广度优先搜索	39
2.5 滑动窗口	11	5.5 练习	45
2.6 练习	12		
3 居合斩！二分查找	14	6 深入浅出动态规划	46
3.1 算法解释	14	6.1 算法解释	46
3.2 求开方	14	6.2 基本动态规划：一维	46
3.3 查找区间	16	6.3 基本动态规划：二维	50
3.4 旋转数组查找数字	18	6.4 分割类型题	54
3.5 练习	19	6.5 子序列问题	60
4 千奇百怪的排序算法	20	6.6 背包问题	63
4.1 常用排序算法	20	6.7 字符串编辑	70
		6.8 股票交易	72
		6.9 练习	75

第1章 最易懂的贪心算法

内容提要

- 算法解释
- 分配问题

- 区间问题

1.1 算法解释

顾名思义，贪心算法或贪心思想(greedy algorithm)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同样是全局最优的。

证明一道题能用贪心算法解决，有时远比用贪心算法解决该题更复杂。一般情况下，在简单操作后，具有明显的从局部到整体的递推关系，或者可以通过数学归纳法推测结果时，我们才会使用贪心算法。

1.2 分配问题

455. Assign Cookies

题目描述

有一群孩子和一堆饼干，每个孩子有一个饥饿度，每个饼干都有一个饱腹度。每个孩子只能吃一个饼干，且只有饼干的饱腹度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饱腹度。输出可以吃饱的孩子的最大数量。

```
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。排序顺序默认是从小到大。



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_i = 0, cookie_i = 0;
    int n_children = children.size(), n_cookies = cookies.size();
    while (child_i < n_children && cookie_i < n_cookies) {
        if (children[child_i] <= cookies[cookie_i]) {
            ++child_i;
        }
        ++cookie_i;
    }
    return child_i;
}
```

```
def findContentChildren(children: List[int], cookies: List[int]) -> int:
    children.sort()
    cookies.sort()
    child_i, cookie_i = 0, 0
    n_children, n_cookies = len(children), len(cookies)
    while child_i < n_children and cookie_i < n_cookies:
        if children[child_i] <= cookies[cookie_i]:
            child_i += 1
            cookie_i += 1
    return child_i
```

135. Candy

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]。

```
int candy(vector<int>& ratings) {
    int n = ratings.size();
    vector<int> candies(n, 1);
    for (int i = 1; i < n; ++i) {
        if (ratings[i] > ratings[i - 1]) {
            candies[i] = candies[i - 1] + 1;
        }
    }
    for (int i = n - 1; i > 0; --i) {
        if (ratings[i] < ratings[i - 1]) {
            candies[i - 1] = max(candies[i - 1], candies[i] + 1);
        }
    }
    return accumulate(candies.begin(), candies.end(), 0);
}
```

```
def candy(ratings_list: List[int]) -> int:
    n = len(ratings_list)
    candies = [1] * n
    for i in range(1, n):
        if ratings_list[i] > ratings_list[i - 1]:
            candies[i] = candies[i - 1] + 1
    for i in range(n - 1, 0, -1):
        if ratings_list[i] < ratings_list[i - 1]:
            candies[i - 1] = max(candies[i - 1], candies[i] + 1)
    return sum(candies)
```

1.3 区间问题

435. Non-overlapping Intervals

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，包含多个长度固定为 2 的子数组，表示每个区间的开始和结尾。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（利用 lambda 函数），每次选择结尾最小且和前一个选择的区间不重叠的区间。

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

```
def eraseOverlapIntervals(intervals: List[List[int]]) -> int:
    intervals.sort(key=lambda x: x[1])
    removed, prev_end = 0, intervals[0][1]
    for i in range(1, len(intervals)):
        if prev_end > intervals[i][0]:
            removed += 1
        else:
            prev_end = intervals[i][1]
    return removed
```

1.4 练习



基础难度

605. Can Place Flowers

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

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

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

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

温馨提示，这道题可能同时需要排序和插入操作。

665. Non-decreasing Array

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2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

2.1 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。

由于 C++ 相比 Python 可以更方便地操作指针符，在 C++ 中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);
```

2.2 Two Sum

167. Two Sum II - Input array is sorted

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, two_sum;
    while (l < r) {
        two_sum = numbers[l] + numbers[r];
        if (two_sum == target) {
            break;
        }
        if (two_sum < target) {
            ++l;
        } else {
            --r;
        }
    }
}
```



```

    return vector<int>{l + 1, r + 1};
}

```

```

def twoSum(numbers: List[int], target: int) -> List[int]:
    l, r = 0, len(numbers) - 1
    while l < r:
        two_sum = numbers[l] + numbers[r]
        if two_sum == target:
            break
        if two_sum < target:
            l += 1
        else:
            r -= 1
    return [l + 1, r + 1]

```

2.3 归并两个有序数组

88. Merge Sorted Array

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 $m + n - 1$ 。每次向左移动 m 或 n 的时候，也要向左移动 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** 在 C++ 题解里我们使用了 `++` 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，则两个写法都可以（`++a` 在未经编译器优化的情况下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--];
    }
}

```

```

def merge(nums1: List[int], m: int, nums2: List[int], n: int) -> None:
    pos = m + n - 1
    m -= 1
    n -= 1
    while m >= 0 and n >= 0:
        if nums1[m] > nums2[n]:
            nums1[pos] = nums1[m]
            m -= 1
        else:
            nums1[pos] = nums2[n]
            n -= 1
        pos -= 1
    nums1[:n + 1] = nums2[:n + 1]

```

2.4 快慢指针

142. Linked List Cycle II

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。

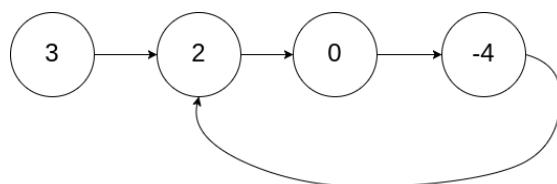


图 2.1: 题目 142 - 输入样例

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) {}
};

```

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None
```

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。

 **注意** 对于某些只需要判断是否存在环路的题目，也可以通过建造哈希表来查重。

```
ListNode *detectCycle(ListNode *head) {
    ListNode *slow = head, *fast = head;
    bool is_fisrt_cycle = true;
    // 检查环路。
    while (fast != slow || is_fisrt_cycle) {
        if (fast == nullptr || fast->next == nullptr) {
            return nullptr;
        }
        fast = fast->next->next;
        slow = slow->next;
        is_fisrt_cycle = false;
    }
    // 寻找节点。
    fast = head;
    while (fast != slow) {
        slow = slow->next;
        fast = fast->next;
    }
    return fast;
}
```

```
def detectCycle(head: Optional[ListNode]) -> Optional[ListNode]:
    slow = head
    fast = head
    is_fisrt_cycle = True
    # 检查环路。
    while fast != slow or is_fisrt_cycle:
        if fast is None or fast.next is None:
            return None
        fast = fast.next.next
        slow = slow.next
        is_fisrt_cycle = False
    # 寻找节点。
    fast = head
    while fast != slow:
        fast = fast.next
        slow = slow.next
    return fast
```



2.5 滑动窗口

76. Minimum Window Substring

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。如果不存在解，则输出一个空字符串。

```
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。C++ 题解中使用了两个长度为 128 的数组， $valid$ 和 $freq$ ，来映射字符（ASCII 只包含 128 个字符）。其中 $valid$ 表示每个字符在 t 中是否存在，而 $freq$ 表示目前 t 中每个字符在 s 的滑动窗口中缺少的数量：如果为正，则说明还缺少；如果为负，则说明有盈余。Python 题解则直接使用 Counter 数据结构同时统计 t 中存在的字符和其缺少的数量（也可以用 dict 替代）。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 $O(n)$ 。

```
string minWindow(string s, string t) {
    vector<bool> valid(128, false);
    vector<int> freq(128, 0);
    // 统计t中的字符情况。
    for (int i = 0; i < t.length(); ++i) {
        valid[t[i]] = true;
        ++freq[t[i]];
    }
    // 移动滑动窗口，不断更改统计数据。
    int count = 0;
    int min_l = -1, min_length = -1;
    for (int l = 0, r = 0; r < s.length(); ++r) {
        if (!valid[s[r]]) {
            continue;
        }
        // 把r位置的字符加入频率统计，并检查是否补充了t中缺失的字符。
        if (--freq[s[r]] >= 0) {
            ++count;
        }
        // 滑动窗口已包含t中全部字符，尝试右移l，在不影响结果的情况下寻找最短串。
        while (count == t.length()) {
            if (min_length == -1 || r - l + 1 < min_length) {
                min_l = l;
                min_length = r - l + 1;
            }
            if (valid[s[l]]) {
                --freq[s[l]];
                --count;
            }
            l++;
        }
    }
    return min_l == -1 ? "" : s.substr(min_l, min_length);
}
```

```

        min_l = l;
        min_length = r - l + 1;
    }
    // 把l位置的字符移除频率统计，并检查t中对应字符是否重新缺失。
    if (valid[s[l]] && ++freq[s[l]] > 0) {
        --count;
    }
    ++l;
}
return min_length == -1 ? "" : s.substr(min_l, min_length);
}

```

```

def minWindow(s: str, t: str) -> str:
    # 统计t中的字符情况，等价于：
    # freq = dict()
    # for c in t:
    #     freq[c] = freq.get(c, 0) + 1
    freq = Counter(t)
    # 移动滑动窗口，不断更改统计数据。
    count = 0
    min_l, min_length = None, None
    l = 0
    for r in range(len(s)):
        if s[r] not in freq:
            continue
        # 把r位置的字符加入频率统计，并检查是否补充了t中缺失的字符。
        freq[s[r]] -= 1
        if freq[s[r]] >= 0:
            count += 1
        # 滑动窗口已包含t中全部字符，尝试右移l，在不影响结果的情况下寻找最短串。
        while count == len(t):
            if min_length == None or r - l + 1 < min_length:
                min_l = l
                min_length = r - l + 1
            # 把l位置的字符移除频率统计，并检查t中对应字符是否重新缺失。
            if s[l] in freq:
                freq[s[l]] += 1
                if freq[s[l]] > 0:
                    count -= 1
            l += 1
    return "" if min_length is None else s[min_l: min_l + min_length]
}

```

2.6 练习

基础难度

633. Sum of Square Numbers

Two Sum 题目的变形题之一。

680. Valid Palindrome II

Two Sum 题目的变形题之二。



524. Longest Word in Dictionary through Deleting

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**

需要利用其它数据结构方便统计当前的字符状态。



第3章 居合斩！二分查找

内容提要

□ 算法解释
□ 求开方

□ 查找区间
□ 旋转数组查找数字

3.1 算法解释

二分查找也常被称为二分法或者折半查找 (binary search, bisect)，每次查找时通过将待查找的单调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 $f(t)$ ，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， $f(t)$ 是离散函数 $f(t) = t + 2$ ，查找 4 是否存在等价于求 $f(t) - 4 = 0$ 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。这里笔者提供两个小诀窍，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，指针通常每次移动半个区间长度。

3.2 求开方

69. Sqrt(x)

题目描述

给定一个非负整数 x ，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 x ，求 $f(t) = t^2 - x = 0$ 的解。因为我们只考虑 $t \geq 0$ ，所以 $f(t)$ 在定义域上是单调递增的。考虑到 $f(0) = -x \leq 0$ ， $f(x) = x^2 - x \geq 0$ ，我们可以对 $[0, x]$ 区间使用二分法找到 $f(t) = 0$ 的解。这里笔者使用了左闭右闭的写法。

 **注意** 在 C++ 题解中， $mid = (l + r)/2$ 可能会因为 $l + r$ 溢出而错误，因而采用 $mid = l + (r - l)/2$ 的写法；直接计算 $mid * mid$ 也有可能溢出，因此我们比较 mid 和 x/mid 。

```
int mySqrt(int x) {
    int l = 1, r = x, mid, x_div_mid;
    while (l <= r) {
        mid = l + (r - 1) / 2;
        x_div_mid = x / mid;
        if (mid == x_div_mid) {
            return mid;
        }
        if (mid < x_div_mid) {
            l = mid + 1;
        } else {
            r = mid - 1;
        }
    }
    return r;
}
```

```
def mySqrt(x: int) -> int:
    l, r = 1, x
    while l <= r:
        mid = (l + r) // 2
        mid_sqr = mid**2
        if mid_sqr == x:
            return mid
        if mid_sqr < x:
            l = mid + 1
        else:
            r = 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 $t_{n+1} = t_n - f(t_n)/f'(t_n)$ 。给定 $f(t) = t^2 - x = 0$ ，这里的迭代公式为 $t_{n+1} = (t_n + x/t_n)/2$ ，其代码如下。

 **注意** 在 C++ 题解中，为了防止 int 超上界，除了上面提到的方法外，我们也可以使用 long 来存储乘法结果。

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

```
def mySqrt(x: int) -> int:
    t = x
    while t**2 > x:
        t = (t + x // t) // 2
    return t
```

3.3 查找区间

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

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，或者 Python 里的 `bisect_left` 和 `bisect_right` 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。

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



```

}

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

vector<int> searchRange(vector<int> &nums, int target) {
    if (nums.empty()) {
        return vector<int>{-1, -1};
    }
    int lower = lowerBound(nums, target);
    int upper = upperBound(nums, target) - 1;
    if (lower == nums.size() || nums[lower] != target) {
        return vector<int>{-1, -1};
    }
    return vector<int>{lower, upper};
}

```

```

def lowerBound(nums: List[int], target: int) -> int:
    l, r = 0, len(nums)
    while l < r:
        mid = (l + r) // 2
        if nums[mid] < target:
            l = mid + 1
        else:
            r = mid
    return l

def upperBound(nums: List[int], target: int) -> int:
    l, r = 0, len(nums)
    while l < r:
        mid = (l + r) // 2
        if nums[mid] <= target:
            l = mid + 1
        else:
            r = mid
    return l

def searchRange(nums: List[int], target: int) -> List[int]:
    if not nums:
        return [-1, -1]
    lower = lowerBound(nums, target)
    upper = upperBound(nums, target) - 1
    if lower == len(nums) or nums[lower] != target:
        return [-1, -1]
    return [lower, upper]

```



3.4 旋转数组查找数字

81. Search in Rotated Sorted Array II

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 $[1,2,3,4,5] \rightarrow [2,3,4,5,1,2]$ ，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。本题我们采用左闭右闭的写法。

```
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[mid] == nums[l]) {
            // 无法判断哪个区间是增序的，但l位置一定不是target。
            ++l;
        } else if (nums[mid] == nums[r]) {
            // 无法判断哪个区间是增序的，但r位置一定不是target。
            --r;
        } 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;
}
```

```
def search(nums: List[int], target: int) -> bool:
    l, r = 0, len(nums) - 1
    while l <= r:
        mid = (l + r) // 2
        if nums[mid] == target:
            return True
        if nums[mid] == nums[l]:
            # 无法判断哪个区间是增序的，但l位置一定不是target。
            l += 1
        elif nums[mid] == nums[r]:
            # 无法判断哪个区间是增序的，但r位置一定不是target。
            r -= 1
        elif nums[mid] < nums[r]:
            # 右区间是增序的。
            if nums[mid] < target <= nums[r]:
                l = mid + 1
            else:
                r = mid - 1
        else:
            # 左区间是增序的。
            if nums[l] <= target < nums[mid]:
                r = mid - 1
            else:
                l = mid + 1
    return False
```

3.5 练习

基础难度

154. Find Minimum in Rotated Sorted Array II

旋转数组的变形题之一。

540. Single Element in a Sorted Array

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

需要对两个数组同时进行二分搜索。

第4章 千奇百怪的排序算法

内容提要

- 常用排序算法
- 桶排序
- 快速排序

4.1 常用排序算法

虽然在 C++ 和 Python 里都可以通过 `sort` 函数排序，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。这里我们展示两种复杂度为 $O(n \log(n))$ 的排序算法：快速排序和归并排序，其中前者实际速度较快，后者可以保证相同值的元素在数组中的相对位置不变（即“稳定排序”）。

快速排序（Quicksort）

快速排序的原理并不复杂：对于当前一个未排序片段，我们先随机选择一个位置当作中枢点，然后通过遍历操作，将所有比中枢点小的数字移动到其左侧，再将所有比中枢点大的数字移动到其右侧。操作完成后，我们再次对中枢点左右侧的片段再次进行快速排序即可。可证明，如果中枢点选取是随机的，那么该算法的平均复杂度可以达到 $O(n \log(n))$ ，最差情况下复杂度则为 $O(n^2)$ 。

我们采用左闭右闭的二分写法，初始化条件为 $l = 0, r = n - 1$ 。

```
void quickSort(vector<int> &nums, int l, int r) {
    if (l >= r) {
        return;
    }
    // 在当前的[l, r]区间中，随机选择一个位置当作pivot。
    int pivot = l + (rand() % (r - l + 1));
    int pivot_val = nums[pivot];
    // 将pivot与l交换。
    swap(nums[1], nums[pivot]);
    // 从[l+1, r]两端向内遍历，查找是否有位置不满足递增关系。
    int i = l + 1, j = r;
    while (true) {
        while (i < j && nums[j] >= pivot_val) {
            --j;
        }
        while (i < j && nums[i] <= pivot_val) {
            ++i;
        }
        if (i == j) {
            break;
        }
        // i位置的值大于pivot值，j位置的值小于pivot值，将二者交换。
        swap(nums[i], nums[j]);
    }
}
```

```
// i和j相遇的位置即为新的pivot，我们将pivot与l重新换回来。
// 此时相遇位置左侧一定比pivot值小，右侧一定比pivot值大。
int new_pivot = nums[i] <= nums[1] ? i : i - 1;
swap(nums[1], nums[new_pivot]);
quickSort(nums, l, new_pivot - 1);
quickSort(nums, new_pivot + 1, r);
}
```

```
def quickSort(nums: List[int], l: int, r: int) -> bool:
    if l >= r:
        return
    # 在当前的[l, r]区间中，随机选择一个位置当作pivot。
    pivot = random.randint(l, r)
    pivot_val = nums[pivot]
    # 将pivot与l交换。
    nums[1], nums[pivot] = nums[pivot], nums[1]
    # 从[l+1, r]两端向内遍历，查找是否有位置不满足递增关系。
    i, j = l + 1, r
    while True:
        while i < j and nums[j] >= pivot_val:
            j -= 1
        while i < j and nums[i] <= pivot_val:
            i += 1
        if i == j:
            break
        # i位置的值大于pivot值，j位置的值小于pivot值，将二者交换。
        nums[i], nums[j] = nums[j], nums[i]
    # i和j相遇的位置即为新的pivot，我们将pivot与l重新换回来。
    # 此时相遇位置左侧一定比pivot值小，右侧一定比pivot值大。
    new_pivot = i if nums[i] <= nums[1] else i - 1
    nums[1], nums[new_pivot] = nums[new_pivot], nums[1]
    quickSort(nums, l, new_pivot - 1)
    quickSort(nums, new_pivot + 1, r)
```

归并排序 (Merge Sort)

归并排序是典型的分治法，我们在之后的章节会展开讲解。简单来讲，对于一个未排序片段，我们可以先分别排序其左半侧和右半侧，然后将两侧重新组合（“治”）；排序每个半侧时可以通过递归再次把它切分成两侧（“分”）。

我们采用左闭右闭的二分写法，初始化条件为 $l = 0$, $r = n - 1$ ，另外提前建立一个和 $nums$ 大小相同的数组 $cache$ ，用来存储临时结果。

```
void mergeSort(vector<int> &nums, vector<int> &cache, int l, int r) {
    if (l >= r) {
        return;
    }
    // 分。
    int mid = l + (r - 1) / 2;
    mergeSort(nums, cache, l, mid);
    mergeSort(nums, cache, mid + 1, r);
    // 治。
    // i和j同时向右前进，i的范围是[l, mid]，j的范围是[mid+1, r]。
    int i = l, j = mid + 1;
```



```

for (int pos = l; pos <= r; ++pos) {
    if (j > r || (i <= mid && nums[i] <= nums[j])) {
        cache[pos] = nums[i++];
    } else {
        cache[pos] = nums[j++];
    }
}
for (int pos = l; pos <= r; ++pos) {
    nums[pos] = cache[pos];
}
}

```

```

def mergeSort(nums: List[int], cache: List[int], l: int, r: int) -> bool:
    if l >= r:
        return
    # 分。
    mid = (l + r) // 2
    mergeSort(nums, cache, l, mid)
    mergeSort(nums, cache, mid + 1, r)
    # 合。
    # i和j同时向右前进，i的范围是[l, mid]，j的范围是[mid+1, r]。
    i, j = l, mid + 1
    for pos in range(l, r + 1):
        if j > r or (i <= mid and nums[i] <= nums[j]):
            cache[pos] = nums[i]
            i += 1
        else:
            cache[pos] = nums[j]
            j += 1
    nums[l:r+1] = cache[l:r+1]

```

4.2 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，可以在平均 $O(n)$ 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 $O(n^2)$ 。

 **注意** 这道题如果直接用上面的快速排序原代码运行，会导致在 LeetCode 上接近超时。我们可以用空间换取时间，直接存储比中枢点小和大的值，尽量避免进行交换位置的操作。

```
int findKthLargest(vector<int> nums, int k) {
    int pivot = rand() % nums.size();
    int pivot_val = nums[pivot];
    vector<int> larger, equal, smaller;
    for (int num : nums) {
        if (num > pivot_val) {
            larger.push_back(num);
        } else if (num < pivot_val) {
            smaller.push_back(num);
        } else {
            equal.push_back(num);
        }
    }
    if (k <= larger.size()) {
        return findKthLargest(larger, k);
    }
    if (k > larger.size() + equal.size()) {
        return findKthLargest(smaller, k - larger.size() - equal.size());
    }
    return pivot_val;
}
```

```
def findKthLargest(nums: List[int], k: int) -> int:
    pivot_val = random.choice(nums)
    larger, equal, smaller = [], [], []
    for num in nums:
        if num > pivot_val:
            larger.append(num)
        elif num < pivot_val:
            smaller.append(num)
        else:
            equal.append(num)
    if k <= len(larger):
        return findKthLargest(larger, k)
    if k > len(larger) + len(equal):
        return findKthLargest(smaller, k - len(larger) - len(equal))
    return pivot_val
```

4.3 桶排序

347. Top K Frequent Elements

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。

 **注意** C++ 中，我们用 `unordered_map` 实现哈希表，即 Python 中的 `dict`。

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

    unordered_map<int, vector<int>> buckets;
    for (auto [num, count] : counts) {
        buckets[count].push_back(num);
    }

    vector<int> top_k;
    for (int count = nums.size(); count >= 0; --count) {
        if (buckets.contains(count)) {
            for (int num : buckets[count]) {
                top_k.push_back(num);
                if (top_k.size() == k) {
                    return top_k;
                }
            }
        }
    }
    return top_k;
}
```

```
def topKFrequent(nums: List[int], k: int) -> List[int]:
    counts = Counter(nums)
    buckets = dict()
    for num, count in counts.items():
        if count in buckets:
            buckets[count].append(num)
```

```
        else:
            buckets[count] = [num]
    top_k = []
    for count in range(len(nums), 0, -1):
        if count in buckets:
            top_k += buckets[count]
            if len(top_k) >= k:
                return top_k[:k]
    return top_k[:k]
```

4.4 练习

基础难度

451. Sort Characters By Frequency

桶排序的变形题。

进阶难度

75. Sort Colors

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5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

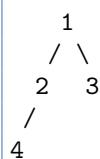
5.1 算法解释

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。

5.2 深度优先搜索

深度优先搜索 (depth-first search, 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，则遍历过程为 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。



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

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。

 **注意** 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。当然您也可以显式写成 [-1, 0]、[1, 0]、[0, 1] 和 [0, -1]，以便理解。

```
int maxAreaOfIsland(vector<vector<int>>& grid) {  
    vector<int> direction{-1, 0, 1, 0, -1};  
    int m = grid.size(), n = grid[0].size(), max_area = 0;  
    for (int i = 0; i < m; ++i) {  
        for (int j = 0; j < n; ++j) {  
            if (grid[i][j] == 1) {  
                stack<pair<int, int>> island;  
                // 初始化第一个节点。  
                int local_area = 1;  
                grid[i][j] = 0;  
                island.push({i, j});  
                // DFS.  
                while (!island.empty()) {  
                    auto [r, c] = island.top();  
                    island.pop();  
                    for (int k = 0; k < 4; ++k) {  
                        int x = r + direction[k], y = c + direction[k + 1];  
                        // 放入满足条件的相邻节点。  
                        if (x >= 0 && x < m && y >= 0 && y < n &&  
                            grid[x][y] == 1) {  
                            ++local_area;  
                            grid[x][y] = 0;  
                            island.push({x, y});  
                        }  
                    }  
                }  
                max_area = max(max_area, local_area);  
            }  
        }  
    }  
}
```



```

        }
    }
    return max_area;
}

```

```

def maxAreaOfIsland(grid: List[List[int]]) -> int:
    direction = [-1, 0, 1, 0, -1]
    m, n, max_area = len(grid), len(grid[0]), 0
    for i in range(m):
        for j in range(n):
            if grid[i][j] == 1:
                island = []
                # 初始化第一个节点。
                local_area = 1
                grid[i][j] = 0
                island.append((i, j))
                # DFS.
                while len(island) > 0:
                    r, c = island.pop()
                    for k in range(4):
                        x, y = r + direction[k], c + direction[k + 1]
                        # 放入满足条件的相邻节点。
                        if 0 <= x < m and 0 <= y < n and grid[x][y] == 1:
                            local_area += 1
                            grid[x][y] = 0
                            island.append((x, y))
                max_area = max(max_area, local_area)
    return max_area

```

下面我们展示递归写法，注意进行递归搜索时，一定要检查边界条件。可以在每次调用辅函数之前检查，也可以在辅函数的一开始进行检查。

```

// 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));
}

// 主函数。
int maxAreaOfIsland(vector<vector<int>>& grid) {
    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。
def dfs(grid: List[List[int]], r: int, c: int) -> int:
    if r < 0 or r >= len(grid) or c < 0 or c >= len(grid[0]) or 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))

# 主函数。
def maxAreaOfIsland(grid: List[List[int]]) -> int:
    max_area = 0
    for i in range(len(grid)):
        for j in range(len(grid[0])):
            max_area = max(max_area, dfs(grid, i, j))
    return max_area

```

547. Number of Provinces

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 个城市和第 j 个城市处于同一城市圈。已知城市的相邻关系是可以传递的，即如果 a 和 b 相邻， b 和 c 相邻，那么 a 和 c 也相邻，换言之这三个城市处于同一个城市圈之内。求一共有多少个城市圈。

输入输出样例

输入是一个二维数组，输出是一个整数，表示城市圈数量。因为城市相邻关系具有对称性，该二维数组为对称矩阵。同时，因为自己也处于自己的城市圈，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] 处于一个城市圈，[3] 处于一个城市圈。

题解

在上一道题目中，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。上一道题目拥有 $m \times n$ 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，表示和所有城市都相邻，最少可以有 1 条边，表示当前城市圈只有自己。当清楚了图的表示方法后，这道题目与上一道题目本质上是同一道题：搜索朋友圈（城市圈）的个数。我们这里采用递归的写法。

 **注意** 对于节点连接类问题，我们也可以利用并查集来进行快速的连接和搜索。我们将会在之后的章节讲解。

```
// 辅函数。
void dfs(vector<vector<int>>& isConnected, int i, vector<bool>& visited) {
    visited[i] = true;
    for (int j = 0; j < isConnected.size(); ++j) {
        if (isConnected[i][j] == 1 && !visited[j]) {
            dfs(isConnected, j, visited);
        }
    }
}

// 主函数。
int findCircleNum(vector<vector<int>>& isConnected) {
    int n = isConnected.size(), count = 0;
    // 防止重复搜索已被搜索过的节点。
    vector<bool> visited(n, false);
    for (int i = 0; i < n; ++i) {
        if (!visited[i]) {
            dfs(isConnected, i, visited);
            ++count;
        }
    }
    return count;
}
```

```
# 辅函数。
def dfs(isConnected: List[List[int]], city: int, visited: Set[int]):
    visited.add(city)
    for i in range(len(isConnected)):
        if isConnected[city][i] == 1 and i not in visited:
            dfs(isConnected, i, visited)

# 主函数。
def findCircleNum(isConnected: List[List[int]]) -> int:
    count = 0
    # 防止重复搜索已被搜索过的节点。
    visited = set()
    for i in range(len(isConnected)):
        if i not in visited:
            dfs(isConnected, i, visited)
            count += 1
    return count
```

417. Pacific Atlantic Water Flow

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};

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

// 主函数。
vector<vector<int>> pacificAtlantic(vector<vector<int>>& heights) {
    int m = heights.size(), n = heights[0].size();
    vector<vector<bool>> can_reach_p(m, vector<bool>(n, false));
    vector<vector<bool>> can_reach_a(m, vector<bool>(n, false));
    vector<vector<int>> can_reach_p_and_a;
    for (int i = 0; i < m; ++i) {
        dfs(heights, can_reach_p, i, 0);
        dfs(heights, can_reach_a, i, n - 1);
    }
    for (int i = 0; i < n; ++i) {
        dfs(heights, can_reach_p, 0, i);
    }
}
```

```

        dfs(heights, 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]) {
                can_reach_p_and_a.push_back({i, j});
            }
        }
    }
    return can_reach_p_and_a;
}

```

```

direction = [-1, 0, 1, 0, -1]

# 辅函数。
def dfs(heights: List[List[int]], can_reach: List[List[int]], r: int, c: int):
    if can_reach[r][c]:
        return
    can_reach[r][c] = True
    for i in range(4):
        x, y = r + direction[i], c + direction[i + 1]
        if (x >= 0 and x < len(heights) and y >= 0 and y < len(heights[0])) and
           heights[x][y] >= heights[r][c]:
            dfs(heights, can_reach, x, y)

# 主函数。
def pacificAtlantic(heights: List[List[int]]) -> List[List[int]]:
    m, n = len(heights), len(heights[0])
    can_reach_p = [[False for _ in range(n)] for _ in range(m)]
    can_reach_a = [[False for _ in range(n)] for _ in range(m)]
    for i in range(m):
        dfs(heights, can_reach_p, i, 0)
        dfs(heights, can_reach_a, i, n - 1)
    for j in range(n):
        dfs(heights, can_reach_p, 0, j)
        dfs(heights, can_reach_a, m - 1, j)
    return [
        [i, j] for i in range(m) for j in range(n)
        if can_reach_p[i][j] and can_reach_a[i][j]
    ]

```

5.3 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

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。

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

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

```

# 辅函数。
def backtracking(nums: List[int], level: int, permutations: List[List[int]]):
    if level == len(nums) - 1:
        permutations.append(nums[:]) # int为基本类型，可以浅拷贝
        return
    for i in range(level, len(nums)):
        nums[i], nums[level] = nums[level], nums[i] # 修改当前节点状态
        backtracking(nums, level + 1, permutations) # 递归子节点
        nums[i], nums[level] = nums[level], nums[i] # 回改当前节点状态

# 主函数。
def permute(nums: List[int]) -> List[List[int]]:
    permutations = []
    backtracking(nums, 0, permutations)
    return permutations

```

77. Combinations

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，而组合回溯的是是否把当前的数字加入结果中。

```

// 辅函数。
void backtracking(vector<vector<int>>& combinations, vector<int>& pick,
                  int pos, int n, int k) {
    if (pick.size() == k) {
        combinations.push_back(pick);
        return;
    }
    for (int i = pos; i <= n; ++i) {
        pick.push_back(i); // 修改当前节点状态
        backtracking(combinations, pick, i + 1, n, k); // 递归子节点
        pick.pop_back(); // 回改当前节点状态
    }
}

// 主函数。

```

```
vector<vector<int>> combine(int n, int k) {
    vector<vector<int>> combinations;
    vector<int> pick;
    backtracking(combinations, pick, 1, n, k);
    return combinations;
}
```

```
# 辅函数。
def backtracking(
    combinations: List[List[int]], pick: List[int], pos: int, n: int, k: int):
    if len(pick) == k:
        combinations.append(pick[:]) # int为基本类型，可以浅拷贝
        return
    for i in range(pos, n + 1):
        pick.append(i) # 修改当前节点状态
        backtracking(combinations, pick, i + 1, n, k) # 递归子节点
        pick.pop() # 回改当前节点状态

# 主函数。
def combine(n: int, k: int) -> List[List[int]]:
    combinations = []
    pick = []
    backtracking(combinations, pick, 1, n, k)
    return combinations
```

79. Word Search

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 = "ABCED", 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"ABCED"。

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。使用回溯法时，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。

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

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

```
# 辅函数。
def backtracking(board: List[List[str]], word: str,
                  visited: List[List[bool]], i: int, j: int, word_pos: int):
    if (i < 0 or i >= len(board) or j < 0 or j >= len(board[0]))
        or visited[i][j] or board[i][j] != word[word_pos]):
        return False
    if word_pos == len(word) - 1:
        return True
    visited[i][j] = True # 修改当前节点状态
    if (backtracking(board, word, visited, i + 1, j, word_pos + 1) or
        backtracking(board, word, visited, i - 1, j, word_pos + 1) or
        backtracking(board, word, visited, i, j + 1, word_pos + 1) or
        backtracking(board, word, visited, i, j - 1, word_pos + 1)):
        return True # 递归子节点
    visited[i][j] = False # 回改当前节点状态
    return False
```

```
# 主函数。
def exist(board: List[List[str]], word: str) -> bool:
    m, n = len(board), len(board[0])
    visited = [[False for _ in range(n)] for _ in range(m)]
    return any([
        backtracking(board, word, visited, i, j, 0)
        for i in range(m) for j in range(n)
    ])
}
```

51. N-Queens

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。

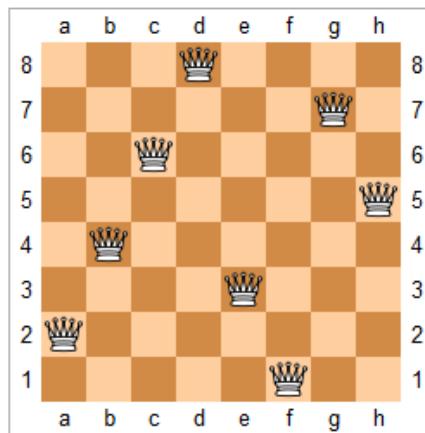


图 5.1: 题目 51 - 八皇后的一种解法

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.", // 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。这里如果我们通过对每一行/列遍历来插入皇后，我们就不需要对行/列建立访问数组了。

```
// 辅函数。
void backtracking(vector<vector<string>> &solutions, vector<string> &board,
                  vector<bool> &column, vector<bool> &ldiag,
                  vector<bool> &rdiag, int row) {
    int n = board.size();
    if (row == n) {
        solutions.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(solutions, board, column, ldiag, rdiag, row + 1);
        // 回改当前节点状态。
        board[row][i] = '.';
        column[i] = ldiag[n - row + i - 1] = rdiag[row + i] = false;
    }
}

// 主函数。
vector<vector<string>> solveNQueens(int n) {
    vector<vector<string>> solutions;
    vector<string> board(n, string(n, '.'));
    vector<bool> column(n, false);
    vector<bool> ldiag(2 * n - 1, false);
    vector<bool> rdiag(2 * n - 1, false);
    backtracking(solutions, board, column, ldiag, rdiag, 0);
    return solutions;
}
```

```
# 辅函数。
def backtracking(solutions: List[List[str]], board: List[List[str]],
                  column: List[bool], ldiag: List[bool], rdiag: List[bool], row: int):
    n = len(board)
    if row == n:
        solutions.append(["".join(row) for row in board])
        return
    for i in range(n):
        if column[i] or ldiag[n - row + i - 1] or 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(solutions, board, column, ldiag, rdiag, row + 1)
# 回改当前节点状态。
board[row][i] = "."
column[i] = ldiag[n - row + i - 1] = rdiag[row + i] = False

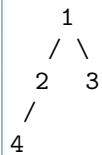
# 主函数。
def solveNQueens(n: int) -> List[List[str]]:
    solutions = []
    board = [["." for _ in range(n)] for _ in range(n)]
    column = [False] * n
    ldiag = [False] * (2 * n - 1)
    rdiag = [False] * (2 * n - 1)
    backtracking(solutions, board, column, ldiag, rdiag, 0)
    return solutions

```

5.4 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 (queue) 而非先入后出的栈 (stack) 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。



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

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。这里我们展示利用深度优先搜索查找第一个岛屿。

```
vector<int> direction{-1, 0, 1, 0, -1};  
  
// 辅函数。  
void dfs(queue<pair<int, int>>& points, vector<vector<int>>& grid,  
        int i, int j) {  
    int m = grid.size(), n = grid[0].size();  
    if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] == 2) {  
        return;  
    }  
    if (grid[i][j] == 0) {  
        points.push({i, j});  
        return;  
    }  
    grid[i][j] = 2;  
    for (int k = 0; k < 4; ++k) {  
        dfs(points, grid, i + direction[k], j + direction[k + 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 && !flipped; ++i) {  
        for (int j = 0; j < n && !flipped; ++j) {  
            if (grid[i][j] == 1) {  
                dfs(points, grid, i, j);  
                flipped = true;  
            }  
        }  
    }  
    // BFS寻找第二个岛屿，并把过程中经过的0赋值为2。  
    int level = 0;  
    while (!points.empty()) {  
        ++level;  
        int n_points = points.size();  
        while (n_points--) {  
            auto [r, c] = points.front();  
            points.pop();  
            grid[r][c] = 2;  
            for (int k = 0; k < 4; ++k) {  
                int x = r + direction[k], y = c + direction[k + 1];  
            }  
        }  
    }  
}
```

```

        if (x >= 0 && x < m && y >= 0 && y < n) {
            if (grid[x][y] == 2) {
                continue;
            }
            if (grid[x][y] == 1) {
                return level;
            }
            grid[x][y] = 2;
            points.push({x, y});
        }
    }
}
return 0;
}

```

```

direction = [-1, 0, 1, 0, -1]

# 辅函数。
def dfs(points: Deque[Tuple[int, int]], grid: List[List[int]], i: int, j: int):
    m, n = len(grid), len(grid[0])
    if i < 0 or i >= m or j < 0 or j >= n or grid[i][j] == 2:
        return
    if grid[i][j] == 0:
        points.append((i, j))
        return
    grid[i][j] = 2
    for k in range(4):
        dfs(points, grid, i + direction[k], j + direction[k + 1])

def shortestBridge(grid: List[List[int]]) -> int:
    m, n = len(grid), len(grid[0])
    points = deque()
    # DFS寻找第一个岛屿，并把1全部赋值为2。
    flipped = False
    for i in range(m):
        if flipped:
            break
        for j in range(n):
            if grid[i][j] == 1:
                dfs(points, grid, 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。
    level = 0
    while len(points) > 0:
        level += 1
        points_at_current_level = len(points)
        for _ in range(points_at_current_level):
            r, c = points.popleft()
            grid[r][c] = 2
            for k in range(4):
                x, y = r + direction[k], c + direction[k + 1]
                if x >= 0 and x < m and y >= 0 and y < n:
                    if grid[x][y] == 2:

```

```

        continue
if grid[x][y] == 1:
    return level
grid[x][y] = 2
points.append((x, y))
return level

```

126. Word Ladder II

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。

这道题略微复杂，需要读者耐心思考和实现代码。

```

// 辅函数。
void backtracking(const string &src, const string &dst,
                  unordered_map<string, vector<string>> &next_words,
                  vector<string> &path, vector<vector<string>> &ladder) {
    if (src == dst) {
        ladder.push_back(path);
        return;
    }
    if (!next_words.contains(src)) {
        return;
    }

```



```
for (const auto &w : next_words[src]) {
    path.push_back(w); // 修改当前节点状态
    backtracking(w, dst, next_words, path, ladder); // 递归子节点
    path.pop_back(); // 回改当前节点状态
}
}

// 主函数。
vector<vector<string>> findLadders(string beginWord, string endWord,
                                         vector<string> &wordList) {
    vector<vector<string>> ladder;
    // 用哈希集合存储字典，方便查找。
    unordered_set<string> word_dict;
    for (const auto &w : wordList) {
        word_dict.insert(w);
    }
    if (!word_dict.contains(endWord)) {
        return ladder;
    }
    word_dict.erase(beginWord);
    word_dict.erase(endWord);
    // 建立两个queue，从beginWord和endWord同时延展，每次延展最小的。
    // 因为之后的去重操作需要遍历queue，我们这里用哈希表实现它，
    // 只要保证是分层次遍历即可。
    unordered_set<string> q_small{beginWord}, q_large{endWord};
    unordered_map<string, vector<string>> next_words;
    bool reversed_path = false, found_path = false;
    while (!q_small.empty()) {
        unordered_set<string> q;
        for (const auto &w : q_small) {
            string s = w;
            for (int i = 0; i < s.size(); ++i) {
                for (int j = 0; j < 26; ++j) {
                    s[i] = j + 'a';
                    if (q_large.contains(s)) {
                        reversed_path ? next_words[s].push_back(w)
                                      : next_words[w].push_back(s);
                        found_path = true;
                    }
                    if (word_dict.contains(s)) {
                        reversed_path ? next_words[s].push_back(w)
                                      : next_words[w].push_back(s);
                        q.insert(s);
                    }
                }
                s[i] = w[i];
            }
        }
        if (found_path) {
            break;
        }
        // 环路一定不是最短解，所以这里需要去重和避免无限循环。
        for (const auto &w : q) {
            word_dict.erase(w);
        }
        // 更新两个queue，并维持大小关系。
        if (q.size() <= q_large.size()) {
            q_small = q;
        }
    }
}
```

```

    } else {
        reversed_path = !reversed_path;
        q_small = q_large;
        q_large = q;
    }
}

if (found_path) {
    vector<string> path{beginWord};
    backtracking(beginWord, endWord, next_words, path, ladder);
}
return ladder;
}

```

```

# 辅函数。
def backtracking(src: str, dst: str, next_words: Dict[str, List[str]],
                 path: List[str], ladder: List[List[str]]):
    if src == dst:
        ladder.append(path[:])
        return
    if src not in next_words:
        return
    for w in next_words[src]:
        path.append(w) # 修改当前节点状态
        backtracking(w, dst, next_words, path, ladder) # 递归子节点
        path.pop() # 回改当前节点状态

# 主函数。
def findLadders(beginWord: str, endWord: str,
                wordList: List[str]) -> List[List[str]]:
    ladder = []
    # 用哈希集合存储字典，方便查找。
    word_dict = set(wordList)
    if endWord not in word_dict:
        return ladder
    word_dict = word_dict.difference(set([beginWord, endWord]))
    # 建立两个queue，从beginWord和endWord同时延展，每次延展最小的。
    # 因为之后的去重操作需要遍历queue，我们这里用哈希表实现它，
    # 只要保证是分层次遍历即可。
    q_small, q_large = set([beginWord]), set([endWord])
    next_words = dict()
    reversed_path, found_path = False, False
    while len(q_small) > 0:
        q = set()
        for w in q_small:
            for i in range(len(w)):
                for j in range(26):
                    s = w[:i] + chr(ord("a") + j) + w[i + 1 :]
                    if s in q_large:
                        if reversed_path:
                            next_words[s] = next_words.get(s, []) + [w]
                        else:
                            next_words[w] = next_words.get(w, []) + [s]
                        found_path = True
                    if s in word_dict:
                        if reversed_path:
                            next_words[s] = next_words.get(s, []) + [w]

```

```

        else:
            next_words[w] = next_words.get(w, []) + [s]
            q.add(s)
    if found_path:
        break
# 环路一定不是最短解，所以这里需要去重和避免无限循环。
word_dict = word_dict.difference(q)
# 更新两个queue，并维持大小关系。
if len(q) <= len(q_large):
    q_small = q
else:
    reversed_path = not reversed_path
    q_small = q_large
    q_large = q
if found_path:
    path = [beginWord]
    backtracking(beginWord, endWord, next_words, path, ladder)
return ladder

```

5.5 练习

基础难度

130. Surrounded Regions

先从最外侧填充，然后再考虑里侧。

257. Binary Tree Paths

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

排列题的 follow-up，如何处理重复元素？

40. Combination Sum II

组合题的 follow-up，如何处理重复元素？

37. Sudoku Solver

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

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6章 深入浅出动态规划

内容提要

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

6.1 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。

6.2 基本动态规划：一维

70. Climbing Stairs

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。

 **注意** 有的时候为了方便处理边界情况，我们可以在构造 dp 数组时多留一个位置，用来处理初始状态。本题即多留了一个第 0 阶的初始位置。

```
int climbStairs(int 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];
}
```

```
def climbStairs(n: int) -> int:
    dp = [1] * (n + 1)
    for i in range(2, n + 1):
        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) {
    int prev_prev = 1, prev = 1, cur = 1;
    for (int i = 2; i <= n; ++i) {
        cur = prev_prev + prev;
        prev_prev = prev;
        prev = cur;
    }
    return cur;
}
```

```
def climbStairs(n: int) -> int:
    prev_prev = prev = cur = 1
    for _ in range(2, n + 1):
        cur = prev_prev + prev
        prev_prev = prev
        prev = cur
    return cur
```



198. House Robber

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) {
    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];
}
```

```
def rob(nums: List[int]) -> int:
    n = len(nums)
    dp = [0] * (n + 1)
    dp[1] = nums[0]
    for i in range(2, n + 1):
        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) {
    int prev_prev = 0, prev = 0, cur = 0;
    for (int i = 0; i < nums.size(); ++i) {
        cur = max(prev_prev + nums[i], prev);
        prev_prev = prev;
        prev = cur;
    }
    return cur;
```



}

```
def rob(nums: List[int]) -> int:
    prev_prev = prev = cur = 0
    for i in range(len(nums)):
        cur = max(prev_prev + nums[i], prev)
        prev_prev = prev
        prev = cur
    return cur
```

413. Arithmetic Slices

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 $\text{num}[i] - \text{num}[i-1] = \text{num}[i-1] - \text{num}[i-2]$ 。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();
    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);
}
```

```
def numberOfArithmeticSlices(nums: List[int]) -> int:
    n = len(nums)
    dp = [0] * n
    for i in range(2, n):
        if nums[i] - nums[i - 1] == nums[i - 1] - nums[i - 2]:
            dp[i] = dp[i - 1] + 1
    return sum(dp)
```



6.3 基本动态规划：二维

64. Minimum Path Sum

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。

 **注意** Python 语言中，多维数组多初始化比较特殊，直接初始化为 $[[val] * n] * m$ 会导致只是创造了 m 个 $[[val] * n]$ 的引用。正确的初始化方法为 $[[val for _ in range(n)] for _ in range(m)]$ 。

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

```
def minPathSum(grid: List[List[int]]) -> int:  
    m, n = len(grid), len(grid[0])
```



```

dp = [[0 for _ in range(n)] for _ in range(m)]
for i in range(m):
    for j in range(n):
        if i == j == 0:
            dp[i][j] = grid[i][j]
        elif i == 0:
            dp[i][j] = grid[i][j] + dp[i][j - 1]
        elif j == 0:
            dp[i][j] = grid[i][j] + dp[i - 1][j]
        else:
            dp[i][j] = grid[i][j] + min(dp[i][j - 1], dp[i - 1][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] = grid[i][j] + dp[j - 1];
            } else if (j == 0) {
                dp[j] = grid[i][j] + dp[j];
            } else {
                dp[j] = grid[i][j] + min(dp[j], dp[j - 1]);
            }
        }
    }
    return dp[n - 1];
}

```

```

def minPathSum(grid: List[List[int]]) -> int:
    m, n = len(grid), len(grid[0])
    dp = [0 for _ in range(n)]
    for i in range(m):
        for j in range(n):
            if i == j == 0:
                dp[j] = grid[i][j]
            elif i == 0:
                dp[j] = grid[i][j] + dp[j - 1]
            elif j == 0:
                dp[j] = grid[i][j] + dp[j]
            else:
                dp[j] = grid[i][j] + min(dp[j - 1], dp[j])
    return dp[n - 1]

```

542. 01 Matrix

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（即全为 1）会达到恐怖的 $O(m^2n^2)$ 。一种办法是使用一个二维布尔值数组做 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) {  
    int m = matrix.size(), n = matrix[0].size();  
    vector<vector<int>> dp(m, vector<int>(n, numeric_limits<int>::max() - 1));  
    for (int i = 0; i < m; ++i) {  
        for (int j = 0; j < n; ++j) {  
            if (matrix[i][j] != 0) {  
                if (i > 0) {  
                    dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1);  
                }  
                if (j > 0) {  
                    dp[i][j] = min(dp[i][j], dp[i][j - 1] + 1);  
                }  
            } else {  
                dp[i][j] = 0;  
            }  
        }  
    }  
    for (int i = m - 1; i >= 0; --i) {  
        for (int j = n - 1; j >= 0; --j) {  
            if (matrix[i][j] != 0) {  
                if (i < m - 1) {  
                    dp[i][j] = min(dp[i][j], dp[i + 1][j] + 1);  
                }  
            }  
        }  
    }  
}
```



```

        if (j < n - 1) {
            dp[i][j] = min(dp[i][j], dp[i][j + 1] + 1);
        }
    }
}
return dp;
}

```

```

def updateMatrix(matrix: List[List[int]]) -> List[List[int]]:
    m, n = len(matrix), len(matrix[0])
    dp = [[sys.maxsize - 1 for _ in range(n)] for _ in range(m)]
    for i in range(m):
        for j in range(n):
            if matrix[i][j] != 0:
                if i > 0:
                    dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1)
                if j > 0:
                    dp[i][j] = min(dp[i][j], dp[i][j - 1] + 1)
            else:
                dp[i][j] = 0
    for i in range(m - 1, -1, -1): # m-1 to 0, reversed
        for j in range(n - 1, -1, -1): # n-1 to 0, reversed
            if matrix[i][j] != 0:
                if i < m - 1:
                    dp[i][j] = min(dp[i][j], dp[i + 1][j] + 1)
                if j < n - 1:
                    dp[i][j] = min(dp[i][j], dp[i][j + 1] + 1)
    return dp

```

221. Maximal Square

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 构成的最大正方形边长。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 $dp[i][j] = k$ ，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 $k - 1$ ，否则 (i, j) 位置不可以构成一个面积为 k^2 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面积为 k^2 的正方形。

0	0	1	0
0	1	1	1
1	1	1	1
0	1	1	1

0	0	1	0
0	1	1	1
1	1	2	2
0	1	2	3

图 6.1：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) {
    int m = matrix.size(), n = matrix[0].size();
    int 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;
}
```

```
def maximalSquare(matrix: List[List[str]]) -> int:
    m, n = len(matrix), len(matrix[0])
    dp = [[0 for _ in range(n+1)] for _ in range(m+1)]
    for i in range(1, m+1):
        for j in range(1, n+1):
            if matrix[i-1][j-1] == "1":
                dp[i][j] = min(dp[i-1][j-1], dp[i][j-1], dp[i-1][j]) + 1
    return max(max(row) for row in dp) ** 2
```

6.4 分割类型题

279. Perfect Squares

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 $i - j^2$ 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 $1 + \min(dp[i-1], dp[i-4], dp[i-9] \dots)$ 。注意边界条件的处理。

```
int numSquares(int n) {
    vector<int> dp(n + 1, numeric_limits<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];
}
```

```
def numSquares(n: int) -> int:
    dp = [0] + [sys.maxsize] * n
    for i in range(1, n+1):
        for j in range(1, int(floor(sqrt(i)))+1):
            dp[i] = min(dp[i], dp[i - j * j] + 1)
    return dp[n]
```

91. Decode Ways

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();
    int prev = s[0] - '0';
    if (prev == 0) {
        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) {
            // 00, 30, 40, ..., 90, 非法。
            return 0;
        }
        if ((prev < 2 && prev > 0) || (prev == 2 && cur <= 6)) {
            // 10, 11, ..., 25, 26.
            if (cur == 0) {
                // 10, 20, 只能连续解码两位。
                dp[i] = dp[i - 2];
            } else {
                // 可以解码当前位，也可以连续解码两位。
                dp[i] = dp[i - 2] + dp[i - 1];
            }
        } else {
            // 合法，但只能解码当前位。
            dp[i] = dp[i - 1];
        }
        prev = cur;
    }
    return dp[n];
}
```

```
def numDecodings(s: str) -> int:
    n = len(s)
    prev = ord(s[0]) - ord('0')
    if prev == 0:
        return 0
    if n == 1:
        return 1
    dp = [1] * (n + 1)
    for i in range(2, n+1):
        cur = ord(s[i-1]) - ord('0')
        if (prev == 0 or prev > 2) and cur == 0:
            # 00, 30, 40, ..., 90, 非法。
            return 0
        if 0 < prev < 2 or (prev == 2 and cur <= 6):
            # 10, 11, ..., 25, 26.
```

```

if cur == 0:
    # 10, 20, 只能连续解码两位。
    dp[i] = dp[i - 2]
else:
    # 可以解码当前位置，也可以连续解码两位。
    dp[i] = dp[i - 2] + dp[i - 1]
else:
    # 合法，但只能解码当前位置。
    dp[i] = dp[i - 1]
prev = cur
return dp[n]

```

139. Word Break

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 m = word.length();
            if (i >= m && s.substr(i - m, m) == word) {
                dp[i] = dp[i - m];
            }
            // 提前剪枝，略微加速运算。
            //如果不剪枝，上一行代码需要变更为 dp[i] = dp[i] || dp[i - m];
            if (dp[i]) {
                break;
            }
        }
    }
    return dp[n];
}

```



```

def wordBreak(s: str, wordDict: List[str]) -> bool:
    n = len(s)
    dp = [True] + [False] * n
    for i in range(1, n+1):
        for word in wordDict:
            m = len(word)
            if i >= m and s[i-m:i] == word:
                dp[i] = dp[i-m]
            # 提前剪枝，略微加速运算。
            # 如果不剪枝，上一行代码需要变更为 dp[i] = dp[i] or dp[i-m]
            if dp[i]:
                break
    return dp[n]

```

1105. Filling Bookcase Shelves

题目描述

给定一个数组，每个元素代表一本书的厚度和高度。问对于一个固定宽度的书架，如果按照数组中书的顺序从左到右、从上到下摆放，最小总高度是多少。

输入输出样例

```

Input: books = [[1,1],[2,3],[2,3],[1,1],[1,1],[1,1],[1,2]], shelfWidth = 4
Output: 6

```

最小总高度放法如图所示。注意这里如果把 1 和 2 这两本书都放在第一排，则前两排总高度为 6，且放不下最后两本书。

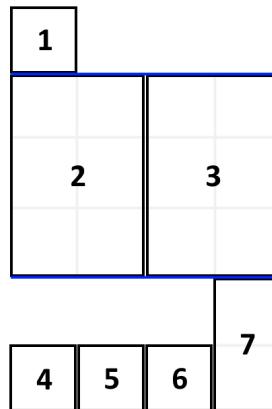


图 6.2: 书架摆放问题 - 样例图解

题解

令 $dp[i]$ 表示放置第 i 本书时的最小总高度，则 $dp[i]$ 可以是在第 $i-1$ 本书下面重新放一排，也可以是在满足不超过前一排宽度的情况下放在前一排。

```

int minHeightShelves(vector<vector<int>>& books, int shelfWidth) {
    int n = books.size();
    vector<int> dp(n + 1, 0);
    for (int i = 1; i <= n; ++i) {
        int w = books[i - 1][0], h = books[i - 1][1];
        dp[i] = dp[i - 1] + h;
        for (int j = i - 1; j > 0; --j) {
            int prev_w = books[j - 1][0], prev_h = books[j - 1][1];
            w += prev_w;
            if (w > shelfWidth) {
                break;
            }
            h = max(h, prev_h);
            dp[i] = min(dp[i], dp[j - 1] + h);
        }
    }
    return dp[n];
}

```

```

def minHeightShelves(books: List[List[int]], shelfWidth: int) -> int:
    n = len(books)
    dp = [0] * (n + 1)
    for i, (w, h) in enumerate(books, 1):
        dp[i] = dp[i-1] + h
        for j in range(i-1, 0, -1):
            prev_w, prev_h = books[j-1]
            w += prev_w
            if w > shelfWidth:
                break
            h = max(h, prev_h)
            dp[i] = min(dp[i], dp[j-1] + h)
    return dp[n]

```

377. Combination Sum IV

题目描述

给定一个不重复数字的数组和一个目标数，求加起来是目标数的所有排列的总数量。（虽然这道题叫做 Combination Sum，但是不同顺序的组合会被当作不同答案，因此本质上是排列。）

输入输出样例

```

Input: nums = [1,2,3], target = 4
Output: 7

```

七种不同的排列为 (1, 1, 1, 1)、(1, 1, 2)、(1, 2, 1)、(1, 3)、(2, 1, 1)、(2, 2) 和 (3, 1)。

题解

令 $dp[i]$ 表示加起来和为 i 时，满足条件的排列数量。在内循环中我们可以直接对所有合法数字进行拿取。这里注意，在 C++ 题解中，因为求和时很容易超过 int 上界，我们这里用 double 存



储 dp 数组。

```
int combinationSum4(vector<int>& nums, int target) {
    vector<double> dp(target + 1, 0);
    dp[0] = 1;
    for (int i = 1; i <= target; ++i) {
        for (int num : nums) {
            if (num <= i) {
                dp[i] += dp[i - num];
            }
        }
    }
    return dp[target];
}
```

```
def combinationSum4(nums: List[int], target: int) -> int:
    dp = [1] + [0] * target
    for i in range(1, target+1):
        dp[i] = sum(dp[i-num] for num in nums if i >= num)
    return dp[target]
```

6.5 子序列问题

300. Longest Increasing Subsequence

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

在这个样例中，最长递增子序列之一是 [2,3,7,101]。

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_len = 0, n = nums.size();
    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_len = max(max_len, dp[i]);
    }
    return max_len;
}

```

```

def lengthOfLIS(nums: List[int]) -> int:
    n = len(nums)
    dp = [1] * n
    for i in range(n):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)

```

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 $\text{dp}[k]$ 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 [10,9,2,5,3,7,101,4]，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]
4	[2, 3, 4, 101]

最终我们知道最长递增子序列的长度是 4。注意 dp 数组最终的形式并不一定是合法的排列形式，如 [2,3,4,101] 并不是子序列；但之前覆盖掉的 [2,3,7,101] 是最优解之一。

类似的，对于其他题目，如果状态转移方程的结果递增或递减，且需要进行插入或查找操作，我们也可以使用二分法进行加速。

```

int lengthOfLIS(vector<int>& nums) {
    vector<int> dp{nums[0]};
    for (int num: nums) {
        if (dp.back() < num) {
            dp.push_back(num);
        }
    }
    return dp.size();
}

```



```

    } else {
        *lower_bound(dp.begin(), dp.end(), num) = num;
    }
}
return dp.size();
}

```

```

def lengthOfLIS(nums: List[int]) -> int:
    dp = [nums[0]]
    for num in nums:
        if dp[-1] < num:
            dp.append(num)
        else:
            dp[bisect.bisect_left(dp, num, 0, len(dp))] = num
    return len(dp)
}

```

1143. Longest Common Subsequence

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];
}

```



```

    }
}

return dp[m][n];
}

```

```

def longestCommonSubsequence(text1: str, text2: str) -> int:
    m, n = len(text1), len(text2)
    dp = [[0 for _ in range(n+1)] for _ in range(m+1)]
    for i in range(1, m+1):
        for j in range(1, n+1):
            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]

```

6.6 背包问题

背包问题 (knapsack problem) 是一种组合优化的 NP 完全问题：有 n 个物品和载重为 w 的背包，每个物品都有自己的重量 weight 和价值 value，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 (0-1 knapsack)；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 (unbounded knapsack)。

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 件物品重量不超过 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，在当前背包总载重为 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 件物品重量为 $weight$ ，价值为 $value$ ，那么我们得到 $dp[i][j] = dp[i-1][j-weight] + value$ 。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 weight = weights[i - 1], value = values[i - 1];
        for (int j = 1; j <= w; ++j) {
            if (j >= weight) {
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight] + value);
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
    return dp[n][w];
}

```

```

def knapsack(weights: List[int], values: List[int], n: int, w: int) -> int:
    dp = [[0 for _ in range(w+1)] for _ in range(n+1)]
    for i in range(1, n+1):

```

```

    weight, value = weights[i - 1], values[i - 1]
    for j in range(1, w+1):
        if j >= weight:
            dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight] + value)
        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 6.3: 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 $i = 2$ ，且其重量为 $weight = 2$ ，价值为 $value = 3$ ；对于背包载重 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 $dp[j] = \max(dp[j], dp[j-weight] + value)$ 。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 $dp[j-weight]$ 的值；若按照从左往右的顺序进行正向遍历，则 $dp[j-weight]$ 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 weight = weights[i - 1], value = values[i - 1];
        for (int j = w; j >= weight; --j) {
            dp[j] = max(dp[j], dp[j - weight] + value);
        }
    }
    return dp[w];
}

```

```

def knapsack(weights: List[int], values: List[int], n: int, w: int) -> int:
    dp = [0] * (w + 1)
    for i in range(1, n+1):
        weight, value = weights[i - 1], values[i - 1]
        for j in range(w, weight-1, -1):
            dp[j] = max(dp[j], [j - weight] + value)
    return dp[w]

```

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 $i = 2$ ，且其重量为 $weight = 2$ ，价值为 $value = 3$ ；对于背包载重 $j = 5$ ，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

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

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

图 6.4: 完全背包问题 - 状态转移矩阵样例

方法，假设背包载重无穷大而物体的重量无穷小，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 weight = weights[i - 1], value = values[i - 1];
        for (int j = 1; j <= w; ++j) {
            if (j >= weight) {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - weight] + value);
            } else {
                dp[i][j] = dp[i - 1][j];
            }
        }
    }
    return dp[n][w];
}
```

```
def knapsack(weights: List[int], values: List[int], n: int, w: int) -> int:
    dp = [[0 for _ in range(w+1)] for _ in range(n+1)]
    for i in range(1, n+1):
        weight, value = weights[i - 1], values[i - 1]
        for j in range(1, w+1):
            if j >= weight:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - weight] + value)
            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 -weight 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 weight = weights[i - 1], value = values[i - 1];
        for (int j = weight; j <= w; ++j) {
            dp[j] = max(dp[j], dp[j - weight] + value);
        }
    }
    return dp[w];
}

```

```

def knapsack(weights: List[int], values: List[int], n: int, w: int) -> int:
    dp = [0] * (w + 1)
    for i in range(1, n+1):
        weight, value = weights[i - 1], values[i - 1]
        for j in range(weight, w+1):
            dp[j] = max(dp[j], [j - weight] + value)
    return dp[w]

```



注意 压缩空间时到底需要正向还是逆向遍历呢？物品和重量哪个放在外层，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，以及压缩哪个维度更省空间。

416. Partition Equal Subset Sum

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 nums_sum = accumulate(nums.begin(), nums.end(), 0);
    if (nums_sum % 2 != 0) {
        return false;
    }
    int target = nums_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];
}

```

```

def canPartition(nums: List[int]) -> bool:
    nums_sum = sum(nums)
    if nums_sum % 2 != 0:
        return False
    target, n = nums_sum // 2, len(nums)
    dp = [[False for _ in range(target + 1)] for _ in range(n + 1)]
    dp[0][0] = True
    for i in range(1, n+1):
        for j in range(target+1):
            if j < nums[i-1]:
                dp[i][j] = dp[i-1][j]
            else:
                dp[i][j] = dp[i-1][j] or 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 nums_sum = accumulate(nums.begin(), nums.end(), 0);
    if (nums_sum % 2 != 0) {
        return false;
    }
    int target = nums_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];
}

```



```

def canPartition(nums: List[int]) -> bool:
    nums_sum = sum(nums)
    if nums_sum % 2 != 0:
        return False
    target, n = nums_sum // 2, len(nums)
    dp = [True] + [False] * target
    for i in range(1, n+1):
        for j in range(target, nums[i-1]-1, -1):
            dp[j] = dp[j] or dp[j - nums[i - 1]]
    return dp[target]

```

474. Ones and Zeroes

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。

```

int findMaxForm(vector<string>& strs, int m, int n) {
    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));
    for (const string & s: strs) {
        int zeros = 0, ones = 0;
        for (char c: s) {
            if (c == '0') {
                ++zeros;
            } else {
                ++ones;
            }
        }
        for (int i = m; i >= zeros; --i) {
            for (int j = n; j >= ones; --j) {
                dp[i][j] = max(dp[i][j], dp[i-zeros][j-ones] + 1);
            }
        }
    }
    return dp[m][n];
}

```



```

def findMaxForm(strs: List[str], m: int, n: int) -> int:
    dp = [[0 for _ in range(n+1)] for _ in range(m+1)]
    for s in strs:
        zeros = len(list(filter(lambda c: c == '0', s)))
        ones = len(s) - zeros
        for i in range(m, zeros - 1, -1):
            for j in range(n, ones - 1, -1):
                dp[i][j] = max(dp[i][j], dp[i-zeros][j-ones] + 1)
    return dp[m][n]

```

322. Coin Change

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) {
    vector<int> dp(amount + 1, amount + 1);
    dp[0] = 0;
    for (int i = 1; i <= amount; ++i) {
        for (int coin : coins) {
            if (i >= coin) {
                dp[i] = min(dp[i], dp[i - coin] + 1);
            }
        }
    }
    return dp[amount] != amount + 1 ? dp[amount] : -1;
}

```

```
def coinChange(coins: List[int], amount: int) -> int:
    dp = [0] + [amount + 1] * amount
    for i in range(1, amount+1):
        for coin in coins:
            if i >= coin:
                dp[i] = min(dp[i], dp[i-coin] + 1)
    return dp[amount] if dp[amount] != amount + 1 else -1
```

6.7 字符串编辑

72. Edit Distance

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 || j == 0) {
                dp[i][j] = i + j;
            } else {
                dp[i][j] = dp[i - 1][j - 1] + (word1[i - 1] != word2[j - 1]);
                dp[i][j] = min(dp[i][j], dp[i - 1][j] + 1);
                dp[i][j] = min(dp[i][j], dp[i][j - 1] + 1);
            }
        }
    }
    return dp[m][n];
}
```

```

def minDistance(word1: str, word2: str) -> int:
    m, n = len(word1), len(word2)
    dp = [[0 for _ in range(n+1)] for _ in range(m+1)]
    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0:
                dp[i][j] = i + j
            else:
                dp[i][j] = min(
                    dp[i-1][j-1] + int(word1[i - 1] != word2[j - 1]),
                    dp[i][j-1] + 1,
                    dp[i-1][j] + 1,
                )
    return dp[m][n]

```

650. 2 Keys Keyboard

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 j 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, 0);
    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];
                // 提前剪枝，因为j从小到大，因此操作数量一定最小。
                break;
            }
        }
    }
    return dp[n];
}

```



}

```

def minSteps(n: int) -> int:
    dp = [0] * 2 + list(range(2, n+1))
    for i in range(2, n+1):
        for j in range(2, floor(sqrt(i))+1):
            if i % j == 0:
                dp[i] = dp[j] + dp[i//j]
                # 提前剪枝，因为j从小到大，因此操作数量一定最小。
                break
    return dp[n]

```

6.8 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

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。注意本题中以及之后题目中的 buy 和 sell 表示买卖操作时，用户账户的收益。因此买时为负，卖时为正。

```

int maxProfit(vector<int>& prices) {
    int buy = numeric_limits<int>::lowest(), sell = 0;
    for (int price : prices) {
        buy = max(buy, -price);
        sell = max(sell, buy + price);
    }
    return sell;
}

```



```
def maxProfit(prices: List[int]) -> int:
    buy, sell = -sys.maxsize, 0
    for price in prices:
        buy = max(buy, -price)
        sell = max(sell, buy + price)
    return sell
```

188. Best Time to Buy and Sell Stock IV

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

类似地，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， $\text{buy}[j]$ 表示在第 j 次买入时的最大收益， $\text{sell}[j]$ 表示在第 j 次卖出时的最大收益。

```
int maxProfit(int k, vector<int>& prices) {
    int days = prices.size();
    vector<int> buy(k + 1, numeric_limits<int>::lowest()), 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];
}
```

```
def maxProfit(k: int, prices: List[int]) -> int:
    days = len(prices)
    buy, sell = [-sys.maxsize] * (k+1), [0] * (k+1)
    for i in range(days):
        for j in range(1, k+1):
            buy[j] = max(buy[j], sell[j-1] - prices[i])
            sell[j] = max(sell[j], buy[j] + prices[i])
    return sell[k]
```



309. Best Time to Buy and Sell Stock with Cooldown

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，以及它们之间的转移方式。



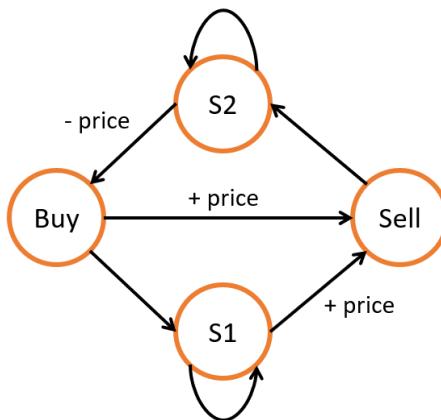


图 6.5: 题目 309 - 状态机状态转移

```

int maxProfit(vector<int>& prices) {
    int n = prices.size();
    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]);
}

```

```

def maxProfit(prices: List[int]) -> int:
    n = len(prices)
    buy, sell, s1, s2 = [0] * n, [0] * n, [0] * n, [0] * n
    s1[0] = buy[0] = -prices[0];
    sell[0] = s2[0] = 0;
    for i in range(1, n):
        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])

```

6.9 练习

基础难度

213. House Robber II

强盗抢劫题目的 follow-up，如何处理环形数组呢？

53. Maximum Subarray

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

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

最长公共子序列的变种题。

进阶难度

646. Maximum Length of Pair Chain

最长递增子序列的变种题，同样的，尝试用二分进行加速。

10. Regular Expression Matching

正则表达式匹配，非常考验耐心。需要根据正则表达式的不同情况，即字符、星号，点号等，分情况讨论。

376. Wiggle Subsequence

最长摆动子序列，通项公式比较特殊，需要仔细思考。

494. Target Sum

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

建立状态机，股票交易类问题就会迎刃而解。



后记

鸣谢名单

GitHub 用户: quweikoala, szlghl1, mag1cianag, woodpenker, yangCoder96, cluckl, shaoshuai-luo, KivenGood, alicegeong, hitYunhongXu, shengchen1998, jihchi, hapoyige, hitYunhongXu, h82258652, conan81412B, Mirrorigin, Light-young, XiangYG, xnervwang, sabaizzz, PhoenixChen98, zhangwang997, corbg118, tracyqwerty, yorhaha, DeckardZ46, Ricardo-609, namu-guwal, hkulyc, jacklanda, View0, HelloYJohn, Corezczy, MoyuST, lutengda, fengshansi 等