



LeetCode 101: 和你一起你轻松刷题 (C++)

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

作者: 高畅 Chang Gao

版本: 预览版 0.01.1

101 Questions, 15 topics, 0 worries.

序

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

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

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

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

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

由于本书的目的是分享和教学，因此本书永久免费，也禁止任何商业利用。欢迎学术目的分享和传阅。由于我不对 LeetCode 的任何题目拥有版权，一切题目版权以 LeetCode 官方为准。同时感谢 GitHub 用户 CyC2018 的 LeetCode 题解，它对于我早期的整理起到了很大的帮助作用。

在这里感谢 ElegantBook 提供的精美 L^AT_EX 模版，使得我可以轻松地把 Markdown 笔记变成看起来更专业的电子书。另外，书的封面图片是我于 2019 年元月，在尼亚加拉大瀑布的加拿大侧拍摄的风景；在此感谢海澄兄同我一起旅行拍照。

为了艾泽拉斯！

目 录

1 LeetCode 入门	1	8.4 分割类型题	40
2 题目分类	2	8.5 子序列问题	42
3 最易懂的贪心算法	3	8.6 背包问题	44
3.1 算法解释	3	8.7 字符串编辑	46
3.2 分配问题	3	8.8 股票交易	48
3.3 区间问题	5	8.9 练习	50
3.4 练习	6	9 化繁为简的分治法	52
4 玩转双指针	7	9.1 算法解释	52
4.1 算法解释	7	9.2 表达式问题	52
4.2 Two Sum	7	9.3 练习	54
4.3 归并两个有序数组	8	10 巧解数学问题	55
4.4 快慢指针	9	10.1 引言	55
4.5 练习	10	10.2 公倍数与公因数	55
5 居合斩！二分查找	12	10.3 质数	55
5.1 算法解释	12	10.4 数字处理	56
5.2 求开方	12	10.5 随机与取样	59
5.3 查找区间	13	10.6 练习	62
5.4 旋转数组查找数字	14	11 神奇的位运算	63
5.5 练习	15	11.1 算法解释	63
6 千奇百怪的排序算法	17	11.2 位运算基础问题	63
6.1 常用排序算法	17	11.3 二进制特性	65
6.2 快速选择	19	11.4 练习	67
6.3 桶排序	20	12 妙用数据结构	68
6.4 练习	21	12.1 C++ STL	68
7 一切皆可搜索	22	12.2 数组	69
7.1 算法解释	22	12.3 栈和队列	72
7.2 深度优先搜索	22	12.4 单调栈	74
7.3 回溯法	26	12.5 优先队列	75
7.4 广度优先搜索	30	12.6 哈希表	77
7.5 练习	33	12.7 多重集合和映射	79
8 深入浅出动态规划	34	12.8 前缀和与积分图	80
8.1 算法解释	34	12.9 练习	82
8.2 基本动态规划：一维	34	13 令人头大的字符串	84
8.3 基本动态规划：二维	37	13.1 字符串比较	84
		13.2 字符串理解	86

13.3 字符串匹配	88	18.5 关于提交	116
13.4 练习	89	18.6 协作人员招募	116
14 指针三剑客之一：链表	90	18.7 致谢	116
14.1 数据结构介绍	90	18.8 捐赠	117
14.2 链表的基本操作	90	19 ElegantBook 设置说明	118
14.3 其它链表技巧	92	19.1 语言模式	118
14.4 练习	94	19.2 设备选项	118
15 指针三剑客之二：树	95	19.3 颜色主题	118
15.1 树的递归	95	19.4 章标题显示风格	119
15.2 层次遍历	98	19.5 数学环境简介	119
15.3 前中后序遍历	99	19.6 装饰物	120
15.4 二叉查找树	101	19.7 封面和徽标	120
15.5 字典树	103	19.8 列表环境	121
15.6 练习	104	19.9 参考文献	121
16 指针三剑客之三：图	106	19.10 添加序章	121
16.1 二分图	106	19.11 章节摘要	121
16.2 拓扑排序	107	19.12 章后习题	122
16.3 最小生成树	108	第 19 章 习题	122
16.4 练习	109	19.13 旁注	122
17 更加复杂的数据结构	110	19.14 连字符	123
17.1 并查集	110	19.15 符号字体	123
17.2 复合数据结构	111	20 ElegantBook 写作示例	125
17.3 练习	112	20.1 Lebesgue 积分	125
18 ElegantL^AT_EX 系列模板介绍	114	第 20 章 习题	127
18.1 ElegantBook 更新说明	114	21 常见问题集	128
18.2 模板安装与更新	114	A 基本数学工具	131
18.3 在线使用模板	115	A.1 求和算子与描述统计量	131
18.4 用户作品计划	115	B 最小示例	132

第 1 章 LeetCode 入门



第2章 题目分类

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

 **注意 TODO:** 思维导图。

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

第二个大分类是数学，包括偏向纯数学的数学问题，和偏向计算机知识的位运算问题。通常来说，面试官在面试时更喜欢出算法和数据结构类型的题，以考察面试者的基本能力；不过考虑到也有一些面试官喜欢出一些数学问题来测试面试者是否聪敏，本书也会讲解一些常见的数学问题。

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

第3章 最易懂的贪心算法

3.1 算法解释

顾名思义，贪心算法或贪心思想采用贪心的策略，以保证每次操作都是局部最优的，并且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。已知果园里有吃不完的苹果，求小明和小王一共最多吃多少个苹果。在这个例子中，我们可以选用的贪心策略为，每个人吃自己能吃的最多数量的苹果，这在每个人身上都是局部最优的。又因为全局结果是局部结果的简单求和，且局部结果互不相干，因此局部最优的策略也同样是全局最优的策略。

3.2 分配问题

455. Assign Cookies (Easy)

题目描述

有一群孩子和一堆饼干，每个孩子有一个饥饿度，每个饼干都有一个大小。每个孩子只能吃最多一个饼干，且只有饼干的大小大于孩子的饥饿度时，这个孩子才能吃饱。求解最多有多少孩子可以吃饱。

输入输出样例

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


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

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


题解

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

简而言之，这里的贪心策略就是，给剩余孩子里最小饥饿度的孩子分配最小的饼干。

 **注意** TODO: 给予贪心策略的严格证明。

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

 **注意** 对数组或字符串排序是常见的操作，方便之后的大小比较。



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

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

135. Candy (Hard)

题目描述

一群孩子站成一排，每一个孩子有自己的评分。现在需要给这些孩子发糖果，规则是如果一个孩子的评分比自己身旁的一个孩子要高，那么这个孩子就必须得到比身旁孩子更多的糖果；所有孩子至少要有有一个糖果。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，大家是不是认为存在比较关系的贪心策略需要排序或是选择？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 size = ratings.size();
    if (size < 2) {
        return size;
    }
    vector<int> num(size, 1);
```



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

3.3 区间问题

435. Non-overlapping Intervals (Medium)

题目描述

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

输入输出样例

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

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


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

题解

在选择要保留区间时，区间的结尾十分重要：选择的区间结尾越小，余留给其它区间的空间就越大，就越能保留更多的区间。因此，我们采取的贪心策略为，优先保留结尾小且不相交的区间。

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

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

 **注意** 需要根据实际情况判断按区间开头排序还是按区间结尾排序。

```
int eraseOverlapIntervals(vector<vector<int>>& intervals) {
    if (intervals.empty()) {
        return 0;
    }
    int n = intervals.size();
```

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

3.4 练习

基础难度

605. Can Place Flowers (Easy)


采取什么样的贪心策略，可以种植最多的花朵呢？

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

这道题和题目 435 十分类似，但是稍有不同，具体是哪里不同呢？

763. Partition Labels (Medium)

为了满足你的贪心策略，是否需要一些预处理？

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

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

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

进阶难度

406. Queue Reconstruction by Height (Medium)

温馨提示，这道题可能同时需要排序和插入操作。

665. Non-decreasing Array (Easy)

需要仔细思考你的贪心策略在各种情况下，是否仍然是最优解。

第4章 玩转双指针

4.1 算法解释

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

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

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

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

指针与常量

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

指针函数与函数指针

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

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

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

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

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

4.2 Two Sum

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

题目描述

在一个增序的整数数组里找到两个数，使它们的和为给定值。题目默认对于数组和给定值，有且只有一个解。

输入输出样例

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

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

在这个样例中，第一个数字 (2) 和第二个数字 (7) 的和等于给定值 (9)。

题解

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

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

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

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

4.3 归并两个有序数组

88. Merge Sorted Array (Easy)

题目描述

给定两个有序数组，把两个数组合并为一个。

输入输出样例


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

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

题解

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

在以下的代码里，我们直接利用 m 和 n 当作两个数组的指针，再额外创建一个 pos 指针，起始位置为 $m + n - 1$ 。每次向前移动 m 或 n 的时候，也要向前移动 pos。这里需要注意，如果 nums1 的数字已经复制完，不要忘记把 nums2 的数字继续复制；如果 nums2 的数字已经复制完，剩余 nums1 的数字不需要改变，因为它们已经被排好序。

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

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

4.4 快慢指针

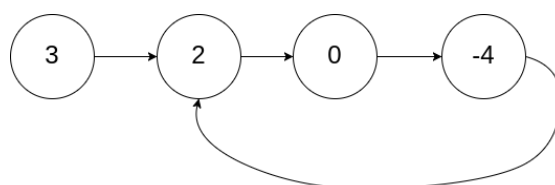
142. Linked List Cycle II (Medium)

题目描述

给定一个链表，如果有环路，找出环路的开始点。

输入输出样例

输入是一个链表，输出是链表的一个节点。如果没有环路，返回一个空指针。



在这个样例中，值为 2 的节点即为环路的开始点。

如果没有特殊说明，LeetCode 采用如下的数据结构表示链表。

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

题解

对于链表找环路的问题，有一个通用的解法——快慢指针（又叫 Floyd 判圈法）¹。给定两个指针，分别命名为 slow 和 fast，起始位置在链表的开头。每次 fast 前进两步，slow 前进一步。如果 fast 可以走到尽头，那么说明没有环路；如果 fast 可以无限走下去，那么说明一定有环路，且一定存在一个时刻 slow 和 fast 相遇。当 slow 和 fast 第一次相遇时，我们将 fast 重新移动到链表开头，并让 slow 和 fast 每次都前进一步。当 slow 和 fast 第二次相遇时，相遇的节点即为环路的开始点。

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

 **注意** TODO: 加一道 sliding window 的题。

4.5 练习

 **注意** TODO: 加一道 sliding window 的题。

基础难度

633. Sum of Square Numbers (Easy)

Two Sum 题目的变形题之一。

¹en.wikipedia.org/wiki/Cycle_detection

680. Valid Palindrome II (Easy)

Two Sum 题目的变形题之二。

524. Longest Word in Dictionary through Deleting (Medium)

归并两个有序数组的变形题。

进阶难度**340. Longest Substring with At Most K Distinct Characters (Hard)**

需要利用其它数据结构方便统计当前的字符状态。



第 5 章 居合斩！二分查找

5.1 算法解释

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

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

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

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

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

5.2 求开方

69. Sqrt(x) (Easy)

题目描述

给定一个非负整数，求它的开方，向下取整。

输入输出样例

输入一个整数，输出一个整数。

Input: 8
Output: 2

8 的开方结果是 2.82842...，向下取整即是 2。

题解

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

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

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

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



注意 这里为了防止 int 超上界，我们使用 long 来存储乘法结果。

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

5.3 查找区间

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

题目描述

给定一个增序的整数数组和一个值，查找该值第一次和最后一次出现的位置。

输入输出样例

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

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

数字 8 在第 3 位第一次出现，在第 4 位最后一次出现。

题解

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

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

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

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

5.4 旋转数组查找数字

81. Search in Rotated Sorted Array II (Medium)

题目描述

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

输入输出样例

输入是一个数组和一个值，输出是一个布尔值，表示数组中是否存在该值。

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

题解

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

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

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

5.5 练习

基础难度

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

旋转数组的变形题之一。

540. Single Element in a Sorted Array (Medium)

在出现独立数之前和之后，奇偶位数的值发生了什么变化？

进阶难度**4. Median of Two Sorted Arrays (Hard)**

需要对两个数组同时进行二分搜索。



第 6 章 千奇百怪的排序算法

6.1 常用排序算法

以下是一些最基本的排序算法。虽然在 C++ 里可以通过 `std::sort()` 快速排序，而且刷题时很少需要自己手写排序算法，但是熟习各种排序算法可以加深自己对算法的基本理解，也可以完美应对不怀好意、让你手推各种排序的面试官。

快速排序 (Quicksort)

我们采用左闭右闭的二分写法。

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

归并排序 (Merge Sort)

```
void merge_sort(vector<int> &nums, int l, int r, vector<int> &temp) {
    if (l + 1 >= r) {
        return;
    }
    // divide
    int m = l + (r - l) / 2;
    merge_sort(nums, l, m, temp);
    merge_sort(nums, m, r, temp);
    // conquer
    int p = l, q = m, i = l;
    while (p < m || q < r) {
        if (q >= r || (p < m && nums[p] <= nums[q])) {
            temp[i++] = nums[p++];
        } else {
            temp[i++] = nums[q++];
        }
    }
```

```
    }  
  }  
  for (i = 1; i < r; ++i) {  
    nums[i] = temp[i];  
  }  
}
```

插入排序 (Insertion Sort)

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

冒泡排序 (Bubble Sort)

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

选择排序 (Selection Sort)

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

以上排序代码调用方法为

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

6.2 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 Kth Element 问题，可以在 $O(n)$ 时间复杂度， $O(1)$ 空间复杂度完成求解工作。快速选择的实现和快速排序相似，不过只需要找到第 K 大的枢 (pivot) 即可，不需要对其左右再进行排序。与快速排序一样，快速选择一般需要先打乱数组，否则最坏情况下时间复杂度为 $O(n^2)$ 。

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



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

6.3 桶排序

347. Top K Frequent Elements (Medium)

题目描述

给定一个数组，求前 K 个最频繁的数字。

输入输出样例

输入是一个数组和一个目标值 K。输出是一个长度为 K 的数组。

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

在这个样例中，最频繁的两个数是 1 和 2。

题解

顾名思义，桶排序的意思是为每个值设立一个桶，桶内记录这个值出现的次数（或其它属性），然后对桶进行排序。针对样例来说，我们先通过桶排序得到三个桶 [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 个大桶即是前 K 个频繁的数字。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],[1]]，表示不同数字出现的频率。最后，我们从后往前遍历，直到找到 K 个旧桶。

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

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

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

6.4 练习

基础难度

451. Sort Characters By Frequency (Medium)

桶排序的变形题。

进阶难度

75. Sort Colors (Medium)

很经典的荷兰国旗问题，考察如何对三个重复且打乱的值进行排序。

第 7 章 一切皆可搜索

7.1 算法解释

TODO。

7.2 深度优先搜索

深度优先搜索 (DFS) TODO。

695. Max Area of Island (Easy)

题目描述

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

输入输出样例

输入是一个二维数组，输出是一个整数，表示最大的岛屿面积。

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

最大的岛屿面积为 6，位于最右侧。

题解

此题是十分标准的搜索题，我们可以拿来练手 DFS。一般来说，DFS 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 DFS 的递归调用。当然，我们也可以使用栈 (stack) 实现 DFS，但因为栈与递归的调用原理相同，而递归相对便于实现，因此刷题时个人推荐使用递归式写法，同时也方便进行回溯（见下节）。不过在实际工程上，直接使用栈可能才是最好的选择，一是因为便于理解，二是更不易出现递归栈满的情况。我们先展示使用栈的写法。

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

int maxAreaOfIsland(vector<vector<int>>& grid) {
    int m = grid.size(), n = m? grid[0].size(): 0, local_area, area = 0, x, y;
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (grid[i][j]) {
                local_area = 1;
```

```

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

```

这里我们使用了一个小技巧，对于四个方向的遍历，可以创建一个数组 [-1, 0, 1, 0, -1]，每相邻两位即为上下左右四个方向之一。

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

第一种递归写法为：

```

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

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

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

```

```
}
```

第二种递归写法为：

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

int dfs(vector<vector<int>>& grid, int r, int c) {
    if (r < 0 || r >= grid.size() ||
        c < 0 || c >= grid[0].size() || grid[r][c] == 0) {
        return 0;
    }
    grid[r][c] = 0;
    return 1 + dfs(grid, r + 1, c) + dfs(grid, r - 1, c) +
        dfs(grid, r, c + 1) + dfs(grid, r, c - 1);
}
```

547. Friend Circles (Medium)

题目描述

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

输入输出样例

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

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

在这个样例中， $[1,2]$ 处于一个朋友圈， $[3]$ 处于一个朋友圈。

题解

对于题目 695，图的表示方法是，每个位置代表一个节点，每个节点与上下左右四个节点相邻。而在这一道题里面，每一行（列）表示一个节点，它的每列（行）表示是否存在一个相邻节点。因此题目 695 拥有 $m \times n$ 个节点，每个节点有 4 条边；而本题拥有 n 个节点，每个节点最多有 n 条边，表示和所有人都是朋友，最少可以有 1 条边，表示自己与自己相连。当清楚了图的表

示方法后，这道题与题目 695 本质上是同一道题：搜索朋友圈（岛屿）的个数（最大面积）。我们这里采用递归的第一种写法。

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

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

417. Pacific Atlantic Water Flow (Medium)

题目描述

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

输入输出样例

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

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

在这个样例中，有括号的区域为满足条件的位置。

题解

虽然题目要求的是满足向下流能到达两个大洋的位置，如果我们对所有的位置进行搜索，那么在不剪枝的情况下复杂度会很高。因此我们可以反过来想，从两个大洋开始向上流，这样我们

只需要对矩形四条边进行搜索。搜索完成后，只需遍历一遍矩阵，满足条件的位置即为两个大洋向上流都能到达的位置。

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

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

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

7.3 回溯法

回溯法 (Backtracking) TODO。

46. Permutations (Medium)

题目描述

输入输出样例

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

题解

```
vector<vector<int>> permute(vector<int>& nums) {  
    vector<vector<int>> ans;  
    backtracking(nums, 0, ans);  
    return ans;  
}  
  
void backtracking(vector<int> &nums, int level, vector<vector<int>> &ans) {  
    if (level == nums.size() - 1) {  
        ans.push_back(nums);  
        return;  
    }  
    for (int i = level; i < nums.size(); i++) {  
        swap(nums[i], nums[level]);  
        backtracking(nums, level+1, ans);  
        swap(nums[i], nums[level]);  
    }  
}
```

77. Combinations (Medium)

题目描述

输入输出样例

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

题解

```
vector<vector<int>> combine(int n, int k) {  
    vector<vector<int>> ans;  
    vector<int> comb(k, 0);  
    backtracking(ans, comb, 1, 0, k, n);  
    return ans;  
}  
  
void backtracking(vector<vector<int>>& ans, vector<int>& comb, int pos, int  
count, int k, int n) {  
    if (!k) {
```

```

        ans.push_back(comb);
        return;
    }
    for (int i = pos; i <= n; ++i) {
        comb[count] = i;
        backtracking(ans, comb, i + 1, count + 1, k - 1, n);
    }
}

```

79. Word Search (Medium)

题目描述

输入输出样例

Input: word = "ABCCED", board =
 [['A','B','C','E'],
 ['S','F','C','S'],
 ['A','D','E','E']]
 Output: true

题解

```

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

void backtracking(int i, int j, vector<vector<char>>& board, string& word, int pos, bool& find, vector<vector<bool>>& visited) {
    if (i < 0 || i >= board.size() || j < 0 || j >= board[0].size()) {
        return;
    }
    if (visited[i][j] || find || board[i][j] != word[pos]) {
        return;
    }
    if (pos == word.size() - 1) {
        find = true;
        return;
    }
    visited[i][j] = true;
    backtracking(i + 1, j, board, word, pos + 1, find, visited);
    backtracking(i - 1, j, board, word, pos + 1, find, visited);
    backtracking(i, j + 1, board, word, pos + 1, find, visited);
}

```

```
    backtracking(i, j - 1, board, word, pos + 1, find, visited);  
    visited[i][j] = false;  
}
```

51. N-Queens (Hard)

题目描述

输入输出样例

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

题解

```
vector<vector<string>> solveNQueens(int n) {  
    vector<vector<string>> ans;  
    if (n == 0) {  
        return ans;  
    }  
    vector<string> board(n, string(n, '.'));  
    vector<bool> column(n, false), ldiag(2*n-1, false), rdiag(2*n-1, false);  
    backtracking(ans, board, column, ldiag, rdiag, 0, n);  
    return ans;  
}  
  
void backtracking(vector<vector<string>> &ans, vector<string> &board, vector<  
    bool> &column, vector<bool> &ldiag, vector<bool> &rdiag, int row, int n) {  
    if (row == n) {  
        ans.push_back(board);  
        return;  
    }  
    for (int i = 0; i < n; ++i) {  
        if (column[i] || ldiag[n-row+i-1] || rdiag[row+i+1]) {  
            continue;  
        }  
        board[row][i] = 'Q';  
        column[i] = ldiag[n-row+i-1] = rdiag[row+i+1] = true;  
        backtracking(ans, board, column, ldiag, rdiag, row+1, n);  
        board[row][i] = '.';  
        column[i] = ldiag[n-row+i-1] = rdiag[row+i+1] = false;  
    }  
}
```

7.4 广度优先搜索

934. Shortest Bridge (Medium)

题目描述

输入输出样例

```
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

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

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

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

```

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

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

```

126. Word Ladder II (Hard)

题目描述

输入输出样例

```

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

```

vector<vector<string>> findLadders(string beginWord, string endWord, vector<
string>& wordList) {
    vector<vector<string>> ans;
    unordered_set<string> dict;
    for (const auto &w: wordList){
        dict.insert(w);
    }
    if (!dict.count(endWord)) {
        return ans;
    }
    dict.erase(beginWord);

```



```

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

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

```

7.5 练习

基础难度

130. Surrounded Regions (Medium)

先从最外侧填充，然后再考虑里侧。

257. Binary Tree Paths (Easy)

输出二叉树中所有从根到叶子的路径，回溯法使用与否有什么区别？

进阶难度

47. Permutations II (Medium)

排列题的 follow-up，如何处理重复元素？

40. Combination Sum II (Medium)

组合题的 follow-up，如何处理重复元素？

37. Sudoku Solver (Hard)

十分经典的数独题，可以利用回溯法求解。事实上对于数独类型的题，有很多进阶的搜索方法和剪枝策略可以提高速度，如启发式搜索¹。

310. Minimum Height Trees (Medium)

如何将这道题转为搜索类型题？是使用深度优先还是广度优先呢？

¹www.sudokuwiki.org/Naked_Candidates

第 8 章 深入浅出动态规划

8.1 算法解释

TODO。

动态规划 (Dynamic Programming, DP) 和其它遍历算法 (如深度/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, 动态规划保存了子问题的解, 避免重复计算。

解决动态规划问题, 关键是找到状态转移方程。TODO。

空间压缩 TODO。

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。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。

解释 memoization, TODO。

8.2 基本动态规划: 一维

70. Climbing Stairs (Easy)

题目描述

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

输入输出样例

输入是一个数字, 表示台阶数量; 输出是爬台阶的总方式。

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

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

题解

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

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

```
    dp[i] = dp[i-1] + dp[i-2];
}
return dp[n];
}
```

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

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

198. House Robber (Easy)

题目描述

假如你是一个劫匪，并且决定抢劫一条街上的房子，每个房子内的钱财数量各不相同。如果你抢了两栋相邻的房子，则会触发警报机关。求在不触发警报机关的情况下，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 $dp[i] = \max(dp[i-1], \text{nums}[i-1] + dp[i-2])$ 。

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

```
    dp[i] = max(dp[i-1], nums[i-1] + dp[i-2]);
}
return dp[n];
}
```

同样的，我们可以像题目 70 那样，对空间进行压缩。

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

413. Arithmetic Slices (Medium)

题目描述

给定一个数组，求这个数组中连续且等差的子数组一共有多少个。

输入输出样例

输入是一个一维数组，输出是满足等差条件的连续子数组个数。

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

8.3 基本动态规划：二维

64. Minimum Path Sum (Medium)

题目描述

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

输入输出样例

输入是一个二维数组，输出是最优路径的数字和。

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


在这个样例中，最短路径为 $1 \rightarrow 3 \rightarrow 1 \rightarrow 1 \rightarrow 1$ 。

题解

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

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

同样的，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 $dp[i]$ ，在遍历到第 j 列的时候， $dp[i-1]$ 因为已经更新过了，所以代表 $dp[i-1][j]$ 的值；而 $dp[i]$ 待更新，当前存储的值是在第 $j-1$ 列的时候计算的，所以代表 $dp[i][j-1]$ 的值。

 **注意** 如果不是很熟悉空间压缩技巧，则推荐在面试时优先给出非空间压缩的解法，如果有剩余时间且力所能及再进行空间压缩。

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

542. 01 Matrix (Medium)

题目描述

给定一个由 0 和 1 组成的二维矩阵，求每个位置到最近的 0 的距离。

输入输出样例

输入是一个二维 0/1 矩阵，输出是一个同样大小的非负整数矩阵，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搜索，所以很多人的第一反应可能会是优先搜索。但是对于一个大小 $O(mn)$ 的二维数组，对每个位置进行四向搜索，最坏情况的时间复杂度（即全是 1）会达到恐怖的 $O(m^2n^2)$ 。一种办法是使用一个 dp 数组做 memoization，使得优先搜索不会重复遍历；另一种更简单的方法是，我们从左上到右下进行一次动态搜索，再从右下到左上进行一次动态搜索。两次动态搜索即可完成四个方向上的查找。

```
vector<vector<int>> updateMatrix(vector<vector<int>>& matrix) {
    if (matrix.empty()) return {};
    // ...
```

```

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

```

221. Maximal Square (Medium)

题目描述

给定一个二维的 0/1 矩阵，求全由 1 构成的最大正方形面积。

输入输出样例


输入是一个二维 0/1 数组，输出是最大正方形面积。

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

题解

对于在矩阵内搜索正方形或长方形的题型，一种常见的做法是定义一个二维 dp 数组，其中 $dp[i][j]$ 表示满足题目条件的、以 (i, j) 为右下角的正方形或者长方形的属性。对于本题，则表示以 (i, j) 为右下角的全由 1 构成的最大正方形面积。如果当前位置是 0，那么 $dp[i][j]$ 即为 0；如果当前位置是 1，我们假设 $dp[i][j] = k^2$ ，其充分条件为 $dp[i-1][j-1]$ 、 $dp[i][j-1]$ 和 $dp[i-1][j]$ 的值必须

都不小于 $(k-1)^2$ ，否则 (i, j) 位置不可以构成一个边长为 k 的正方形。同理，如果这三个值中的最小值为 $k-1$ ，则 (i, j) 位置一定且最大可以构成一个边长为 k 的正方形。

 **注意 TODO:** 画个图

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

8.4 分割类型题

279. Perfect Squares (Medium)

题目描述

给定一个正整数，求其最少可以由几个完全平方数相加构成。

输入输出样例

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

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

在这个样例中，13 的最少构成方法为 $4+9$ 。

题解

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

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

```
        dp[i] = min(dp[i], dp[i-j*j] + 1);
    }
}
return dp[n];
}
```

91. Decode Ways (Medium)

题目描述

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

输入输出样例

输入是一个数字串，输出是满足条件的字符串个数，即解码方式。

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

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

题解

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

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

139. Word Break (Medium)

题目描述

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

输入输出样例

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

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

题解

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


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

8.5 子序列问题

300. Longest Increasing Subsequence (Medium)

题目描述

给定一个未排序的整数数组，求最长的递增子序列。

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

输入输出样例

输入是一个一维数组，输出是一个正整数，表示最长递增子序列的长度。

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

在这个样例中，最长递增子序列之一是 [2,3,7,101]。

题解

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

二分 +dp, TODO。

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

1143. Longest Common Subsequence (Medium)

题目描述

输入输出样例

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

在这个样例中，最长公共子序列是 “ace”。

题解

```
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];
}
```

8.6 背包问题

背包问题介绍 TODO。

416. Partition Equal Subset Sum (Medium)

题目描述

输入输出样例

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

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

空间压缩 TODO。

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

474. Ones and Zeroes (Medium)

题目描述

输入输出样例

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 & str: strs) {
        auto [count0, count1] = count(str);
        for (int i = m; i >= count0; --i) {
            for (int j = n; j >= count1; --j) {
                dp[i][j] = max(dp[i][j], 1 + dp[i-count0][j-count1]);
            }
        }
    }
    return dp[m][n];
}

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

```
    return make_pair(count0, count1);  
}
```

322. Coin Change (Medium)

题目描述

输入输出样例

```
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 + 2` 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 + 1`，而最多的组成方式是只用 1 元硬币，因此 `amount + 2` 一定大于所有可能的组合方式，取最小值时一定不会是它。在动态规划完成后，若结果仍然是此值，则说明不存在满足条件的组合方法，返回 `-1`。

```
int coinChange(vector<int>& coins, int amount) {  
    if (coins.empty()) return -1;  
    vector<int> dp(amount + 1, amount + 2);  
    dp[0] = 0;  
    for (int i = 1; i <= amount; ++i) {  
        for (const int & coin : coins) {  
            if (i >= coin) {  
                dp[i] = min(dp[i], dp[i-coin] + 1);  
            }  
        }  
    }  
    return dp[amount] == amount + 2 ? -1 : dp[amount];  
}
```

8.7 字符串编辑

72. Edit Distance (Hard)

题目描述

输入输出样例

```
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

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

650. 2 Keys Keyboard (Medium)

题目描述

输入输出样例

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

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


10. Regular Expression Matching (Hard)

题目描述

输入输出样例

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

在个样例中，我们可以重复 c 零次，重复 a 两次。

题解

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

8.8 股票交易

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

题目描述

输入输出样例

```
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

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

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

题目描述

输入输出样例

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

```
int maxProfit(int k, vector<int>& prices) {
    int days = prices.size();
    if (days < 2) {
        return 0;
    }
    if (k >= days) {
        return maxProfitUnlimited(prices);
    }
    vector<int> buy(k + 1, INT_MIN), sell(k + 1, 0);
    for (int i = 0; i < days; ++i) {
        for (int j = 1; j <= k; ++j) {
            buy[j] = max(buy[j], sell[j-1] - prices[i]);
            sell[j] = max(sell[j], buy[j] + prices[i]);
        }
    }
    return sell[k];
}

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

```
}
```

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

题目描述

输入输出样例

```
Input: [1,2,3,0,2]
Output: 3
Explanation: transactions = [buy, sell, cooldown, buy, sell]
```

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

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

8.9 练习

基础难度

213. House Robber II (Medium)

强盗抢劫题目的 follow-up，如何处理环形数组呢？

53. Maximum Subarray (Easy)

经典的一维动态规划题目，试着把一维空间优化为常量吧。

343. Integer Break (Medium)

分割类型题，先尝试用动态规划求解，再思考是否有更简单的解法。

583. Delete Operation for Two Strings (Medium)

最长公共子序列的变种题。

进阶难度**646. Maximum Length of Pair Chain (Medium)**

最长递增子序列的变种题，同样的，尝试用二分进行加速。

376. Wiggle Subsequence (Medium)

最长摆动子序列，通项公式比较特殊，需要仔细思考。

494. Target Sum (Medium)

如果告诉你这道题是 0/1 背包，你是否会有一些思路？

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

建立状态机，股票交易类问题就会迎刃而解。



第 9 章 化繁为简的分治法

9.1 算法解释

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

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

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

定理 9.1. 主定理

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

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

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

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

9.2 表达式问题

241. Different Ways to Add Parentheses (Medium)

题目描述

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

输入输出样例

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

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

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

题解

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

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

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

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

```
        dp[j][i].push_back(val);
    }
}
}
}
}
return dp[0][n-1];
}
```

9.3 练习

基础难度

932. Beautiful Array (Medium)

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

进阶难度

312. Burst Balloons (Hard)

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

第 10 章 巧解数学问题

10.1 引言

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

10.2 公倍数与公因数

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

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

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

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

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

10.3 质数

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

204. Count Primes (Easy)

题目描述

给定一个数字 n，求小于 n 的质数的个数。

输入输出样例

输入一个整数，输出也是一个整数，表示小于输入数的质数的个数。

Input: 10
Output: 4

在这个样例中，小于 10 的质数只有 [2,3,5,7]。

题解

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

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

利用质数的一些性质，我们可以进一步优化该算法。

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

10.4 数字处理



504. Base 7 (Easy)

题目描述

给定一个十进制整数，求它在七进制下的表示。

输入输出样例

输入一个整数，输出一个字符串，表示其七进制。

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

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

题解

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

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

172. Factorial Trailing Zeroes

题目描述

给定一个非负整数，判断它的阶乘结果的结尾有几个 0。

输入输出样例

输入一个非负整数，输出一个非负整数，表示输入的阶乘结果的结尾有几个 0。

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

在这个样例中， $12! = 479001600$ 的结尾有两个 0。

题解

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

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

415. Add Strings (Easy)

题目描述

给定两个由数字组成的字符串，求它们相加的结果。

输入输出样例

输入是两个字符串，输出是一个整数，表示输入的数字和。

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

题解

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

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

```
    return output;  
}
```

326. Power of Three (Easy)

题目描述

判断一个数字是否是 3 的次方。

输入输出样例

输入一个整数，输出一个布尔值。

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

题解

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

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

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

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

10.5 随机与取样

384. Shuffle an Array (Medium)

题目描述

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

输入输出样例

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

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

在这个样例中，前两次打乱的结果只要是随机生成即可。



题解

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

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

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

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

528. Random Pick with Weight (Medium)

题目描述

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

输入输出样例


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

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

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

题解

思路是先使用 `partial_sum` 求前缀和，然后直接使用 `lower_bound` 获得随机位置。

 **注意** 考虑一下哪里先讲前缀和。

```

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

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

```

382. Linked List Random Node (Medium)

题目描述

给定一个单向链表，要求设计一个算法，可以随机取得其中的一个数字。

输入输出样例

输入是一个单向链表，输出是一个数字，表示链表里其中一个节点的值。

```

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

```

在这个样例中，我们有均等的概率得到任意一个节点，比如 3。

题解

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

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

```

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

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

```

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

10.6 练习

基础难度

168. Excel Sheet Column Title (Easy)

7 进制转换的变种题，需要注意的一点是从 1 开始而不是 0。

67. Add Binary (Easy)

字符串加法的变种题。

238. Product of Array Except Self (Medium)

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

进阶难度

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

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

169. Majority Element (Easy)

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

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

有些面试官很喜欢考这种随机数生成器的问题，如果用一个随机数生成器生成另一个随机数生成器？你可能需要利用原来的生成器多次。

202. Happy Number (Easy)

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

第 11 章 神奇的位运算

11.1 算法解释

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

$x \wedge 0s = x$	$x \& 0s = 0$	$x 0s = x$
$x \wedge 1s = \sim x$	$x \& 1s = x$	$x 1s = 1s$
$x \wedge x = 0$	$x \& x = x$	$x x = x$

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

11.2 位运算基础问题

461. Hamming Distance (Easy)

题目描述

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

输入输出样例

输入是两个十进制整数，输出是一个十进制整数，表示两个输入数字的汉明距离。

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

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

题解

对两个数进行按位异或操作，统计有多少个 1 即可。

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



```
}
```

190. Reverse Bits (Easy)

题目描述

给定一个十进制整数，输出它在二进制下的反转结果。

输入输出样例

输入和输出都是十进制整数。

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

题解

使用算术左移和右移，可以很轻易地实现二进制的反转。

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

136. Single Number (Easy)

题目描述

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

输入输出样例

输入是一个一维整数数组，输出是该数组内的一个整数。

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

题解

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



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

11.3 二进制特性

利用二进制的一些特性，我们可以把位运算使用到更多问题上。

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

342. Power of Four (Easy)

题目描述

给定一个整数，判断它是否是 4 的次方。

输入输出样例

输入是一个整数，输出是一个布尔值，表示判断结果。

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

在这个样例中，16 是 4 的二次方，因此返回值为真。

题解

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

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

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

318. Maximum Product of Word Lengths (Medium)

题目描述

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



输入输出样例

输入一个包含多个字母串的一维数组，输出一个整数，表示长度乘积的最大值。

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

在这个样例中，一种最优的选择是“ab”和“cd”。

题解

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

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

338. Counting Bits (Medium)

题目描述

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

输入输出样例

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

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

题解

本题可以利用动态规划和位运算进行快速的求解。定义一个数组 dp ，其中 $dp[i]$ 表示数字 i 的二进制含有 1 的个数。对于第 i 个数字，如果它二进制的最后一位为 1，那么它含有 1 的个数

则为 $dp[i-1] + 1$ ；如果它二进制的最后一位为 0，那么它含有 1 的个数和其算术右移结果相同，即 $dp[i >> 1]$ 。

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

11.4 练习

基础难度

268. Missing Number (Easy)

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

693. Binary Number with Alternating Bits (Easy)

利用位运算判断一个数的二进制是否会出现连续的 0 和 1。

476. Number Complement (Easy)

二进制翻转的变种题。

进阶难度

260. Single Number III (Medium)

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

第 12 章 妙用数据结构

12.1 C++ STL

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

1. Sequence Containers：维持顺序的容器。

- (a). `vector`：动态数组，是我们最常使用的数据结构之一，用于 $O(1)$ 的随机读取。因为大部分算法的时间复杂度都会大于 $O(n)$ ，因此我们经常新建 `vector` 来存储各种数据或中间变量。因为在尾部增删的复杂度是 $O(1)$ ，我们也可以把它当作 `stack` 来用。
- (b). `list`：双向链表，也可以当作 `stack` 和 `queue` 来使用。由于 LeetCode 的题目多用 `Node` 来表示链表，且链表不支持快速随机读取，因此我们很少用到这个数据结构。一个例外是经典的 LRU 问题，我们需要利用链表的特性来解决，我们在后文会遇到这个问题。
- (c). `deque`：这是一个非常强大的数据结构，既支持 $O(1)$ 随机读取，又支持 $O(1)$ 时间的头部增删和尾部增删，不过有一定的额外开销。
- (d). `array`：固定大小的数组，一般在刷题时我们不使用。
- (e). `forward_list`：单向链表，一般在刷题时我们不使用。

2. Container Adaptors：基于其它容器实现的数据结构。

- (a). `stack`：后入先出（LIFO）的数据结构，默认基于 `deque` 实现。`stack` 常用于深度优先搜索、一些字符串匹配问题以及单调栈问题。
- (b). `queue`：先入先出（FIFO）的数据结构，默认基于 `deque` 实现。`queue` 常用于广度优先搜索。
- (c). `priority_queue`：最大值先出的数据结构，默认基于 `vector` 实现堆结构。它可以在 $O(n \log n)$ 的时间排序数组， $O(\log n)$ 的时间插入任意值， $O(1)$ 的时间获得最大值， $O(\log n)$ 的时间删除最大值。`priority_queue` 常用于维护数据结构并快速获取最大或最小值。

3. Associative Containers：实现了排好序的数据结构。

- (a). `set`：有序集合，元素不可重复，底层实现默认为红黑树，即一种特殊的二叉查找树（BST）。它可以在 $O(n \log n)$ 的时间排序数组， $O(\log n)$ 的时间插入、删除、查找任意值， $O(\log n)$ 的时间获得最小或最大值。这里注意，`set` 和 `priority_queue` 都可以用于维护数据结构并快速获取最大最小值，但是它们的时间复杂度和功能略有区别，如 `priority_queue` 默认不支持删除任意值，而 `set` 获得最大或最小值的时间复杂度略高，具体使用哪个根据需求而定。
- (b). `multiset`：支持重复元素的 `set`。
- (c). `map`：在 `set` 的基础上加上映射关系，可以对每个元素 `key` 存一个值 `value`。
- (d). `multimap`：支持重复元素的 `map`。

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

- (a). `unordered_set`：哈希集合，可以在 $O(1)$ 的时间快速插入、查找、删除元素，常用于快速的查询一个元素是否在这个容器内。
- (b). `unordered_multiset`：支持重复元素的 `unordered_set`。
- (c). `unordered_map`：哈希表，在 `unordered_set` 的基础上加上映射关系，可以对每一个元素 `key` 存一个值 `value`。在某些情况下，如果 `key` 的范围已知且较小，我们也可以用 `vector`

代替 `unordered_map`，用位置表示 `key`，用每个位置的值表示 `value`。

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

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

12.2 数组

48. Rotate Image (Medium)

题目描述

输入输出样例

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

题解

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

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

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

题目描述

输入输出样例

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

题解

建立 n 个桶，把所有重复出现的位置设为负数，仍然为正数的位置即为消失的数。

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

240. Search a 2D Matrix II (Medium)

题目描述

输入输出样例

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

题解

从右上角开始查找。

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

```
    return false;
}
```

280. Wiggle Sort (Medium)

题目描述

不开新空间地修改一个数组，使得最终顺序为 $\text{nums}[0] \leq \text{nums}[1] \geq \text{nums}[2] \leq \text{nums}[3] \dots$

输入输出样例

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

在这个样例中，有多个正确答案，其中一个如输出所示。

题解

```
void wiggleSort(vector<int>& nums) {
    if (nums.empty()) return;
    for (int i = 0; i < nums.size() - 1; ++i) {
        if ((i % 2) == (nums[i] < nums[i+1])) {
            swap(nums[i], nums[i+1]);
        }
    }
}
```

769. Max Chunks To Make Sorted (Medium)

题目描述

输入输出样例

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

在这个样例中，最多分割是 [1, 0], [2], [3], [4]。

题解

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



```
    }  
    return chunks;  
}
```

12.3 栈和队列

232. Implement Queue using Stacks (Easy)

题目描述

输入输出样例

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

题解

我们可以用两个 stack 实现一个 queue。

```
class MyQueue {  
    stack<int> in, out;  
public:  
    MyQueue() {}  
  
    void push(int x) {  
        in.push(x);  
    }  
  
    int pop() {  
        in2out();  
        int x = out.top();  
        out.pop();  
        return x;  
    }  
  
    int peek() {  
        in2out();  
        return out.top();  
    }  
  
    void in2out() {  
        if (out.empty()) {  
            while (!in.empty()) {  
                int x = in.top();  
                in.pop();  
                out.push(x);  
            }  
        }  
    }  
}
```

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

155. Min Stack (Easy)

题目描述

输入输出样例

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

题解

```
class MinStack {  
    stack<int> s, min_s;  
public:  
    MinStack() {}  
  
    void push(int x) {  
        s.push(x);  
        if (min_s.empty() || min_s.top() >= x) {  
            min_s.push(x);  
        }  
    }  
  
    void pop() {  
        if (!min_s.empty() && min_s.top() == s.top()) {  
            min_s.pop();  
        }  
        s.pop();  
    }  
  
    int top() {  
        return s.top();  
    }  
  
    int getMin() {  
        return min_s.top();  
    }  
};
```

20. Valid Parentheses (Easy)

题目描述

输入输出样例

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

题解

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

12.4 单调栈

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

739. Daily Temperatures (Medium)

题目描述

输入输出样例

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

题解

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

12.5 优先队列

heap 实现 TODO。

23. Merge k Sorted Lists (Hard)

题目描述

输入输出样例

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

题解

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

ListNode* mergeKLists(vector<ListNode*>& lists) {
    if (lists.empty()) return nullptr;
    priority_queue<ListNode*, vector<ListNode*>, Comp> q;
    for (ListNode* list: lists) {
        if (list) {
```

```

        q.push(list);
    }
}
ListNode* dummy = new ListNode(0), *cur = dummy;
while (!q.empty()) {
    cur->next = q.top();
    q.pop();
    cur = cur->next;
    if (cur->next) {
        q.push(cur->next);
    }
}
return dummy->next;
}

```

218. The Skyline Problem (Hard)

题目描述

输入输出样例

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

放图 TODO。

题解

```

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

```

```
}
```

12.6 哈希表

实现哈希表。

1. Two Sum (Easy)

题目描述

输入输出样例

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

在这个样例中，第零个位置的值 2 和第一个位置的值 7 的和为 9。

题解

```
vector<int> twoSum(vector<int>& nums, int target) {  
    unordered_map<int, int> hash;  
    vector<int> ans;  
    for (int i = 0; i < nums.size(); ++i) {  
        int num = nums[i];  
        auto pos = hash.find(target - num);  
        if (pos == hash.end()) {  
            hash[num] = i;  
        } else {  
            ans.push_back(pos->second);  
            ans.push_back(i);  
            break;  
        }  
    }  
    return ans;  
}
```

128. Longest Consecutive Sequence (Hard)

题目描述

输入输出样例

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

在这个样例中，最长连续序列是 [1,2,3,4]。

题解

```

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

```

149. Max Points on a Line (Hard)

题目描述

输入输出样例

```

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

```

这个样例中， $y = 5 - x$ 上有四个点。

题解

```

int maxPoints(vector<vector<int>>& points) {
    unordered_map<double, int> hash; // <斜率, 点个数>
    int max_count = 0, same = 1, same_y = 1;
    for (int i = 0; i < points.size(); ++i) {
        same = 1, same_y = 1;
        for (int j = i + 1; j < points.size(); ++j) {
            if (points[i][1] == points[j][1]) {

```

```

        ++same_y;
        if (points[i][0] == points[j][0]) {
            ++same;
        }
    } else {
        double dx = points[i][0] - points[j][0], dy = points[i][1] -
            points[j][1];
        ++hash[dx/dy];
    }
}
max_count = max(max_count, same_y);
for (auto item : hash) {
    max_count = max(max_count, same + item.second);
}
hash.clear();
}
return max_count;
}

```

12.7 多重集合和映射

332. Reconstruct Itinerary (Medium)

题目描述

给定一些机票的起止机场，问如果从 JFK 起飞，那么这个人是按什么顺序飞的；如果存在多种可能性，返回字母序最小的那种。

输入输出样例

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

题解

本题可以用 hashmap+multiset，用 stack 遍历，最后反转输出。

```

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

```



```

        s.pop();
    } else {
        s.push(*hash[next].begin());
        hash[next].erase(hash[next].begin());
    }
}
reverse(ans.begin(), ans.end());
return ans;
}

```

12.8 前缀和与积分图

一维的前缀和，二维的积分图，都是把每个位置之前的一维线段或二维矩形预先存储，方便加速计算。如果需要对前缀和或积分图的值做寻址，则要在哈希表里；如果要对每个位置记录前缀和或积分图的值，则可以储存在一维或二维数组里，也常常伴随着动态规划。

303. Range Sum Query - Immutable (Easy)

题目描述

输入输出样例

Input: nums = [-2,0,3,-5,2,-1], actions: ["sumRange(0,2)","sumRange(1,5)"]
Output: [1,-1]

题解

```

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

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

```

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

题目描述

输入输出样例

```
Input: matrix =
[[3,0,1,4,2],
 [5,6,3,2,1],
 [1,2,0,1,5],
 [4,1,0,1,7],
 [1,0,3,0,5]]
], actions: ["sumRegion(2,1,4,3)","sumRegion(1,1,2,2)"]
Output: [8,11]
```

题解

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

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

560. Subarray Sum Equals K (Medium)

题目描述

寻找和为 k 的连续区间个数。

输入输出样例

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

题解

```
int subarraySum(vector<int>& nums, int k) {
    int count = 0, sum = 0;
    unordered_map<int, int> hash;
    hash[0] = 1; // 初始化很重要
```

```
for (int i: nums) {  
    sum += i;  
    count += hash[sum-k];  
    ++hash[sum];  
}  
return count;  
}
```

12.9 练习

基础难度

566. Reshape the Matrix (Easy)

没有什么难度，只是需要一点耐心。

225. Implement Stack using Queues (Easy)

利用相似的方法，我们也可以用 stack 实现 queue。

503. Next Greater Element II (Medium)

Daily Temperature 的变种题。

217. Contains Duplicate (Easy)

使用什么数据结构可以快速判断重复呢？

697. Degree of an Array (Easy)

如何对数组进行预处理才能正确并快速地计算子数组的长度？

594. Longest Harmonious Subsequence (Easy)

最长连续序列的变种题。


进阶难度

287. Find the Duplicate Number (Medium)

寻找丢失数字的变种题。除了标负位置，你还有没有其它算法可以解决这个问题？

313. Super Ugly Number (Medium)

尝试使用优先队列解决这一问题。

 **注意** 把诸如优先队列（priority_queue）这样的中英文对应找个方法介绍一下。

870. Advantage Shuffle (Medium)

如果我们需要比较关系，而且同一数字可能出现多次，那么应该用什么数据结构呢？

307. Range Sum Query - Mutable (Medium)

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



第 13 章 令人头大的字符串

13.1 字符串比较

242. Valid Anagram (Easy)

题目描述

判断两个字符串包含的字符是否完全相同。

输入输出样例

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

题解

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

205. Isomorphic Strings (Easy)

题目描述

输入输出样例

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

题解

记录一个字符上次出现的位置，如果两个字符串中的字符上次出现的位置一样，那么就属于同构。

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

647. Palindromic Substrings (Medium)

题目描述

输入输出样例

Input: "aaa"
Output: 6

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

题解

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

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

696. Count Binary Substrings (Easy)

题目描述

输入输出样例

Input: "00110011"
Output: 6

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

题解

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

13.2 字符串理解

227. Basic Calculator II (Medium)

题目描述

输入输出样例

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

在这个样例中，因为除法的优先度高于加法，所以结果是 5 而非 4。

题解

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

```

}

int parseExpr(const string& s, int& i) {
    char op = '+';
    long base = 0, temp = 0;
    while (i < s.length()) {
        if (s[i] != ' ') {
            long n = parseNum(s, i);
            switch (op) {
                case '+': base += temp; temp = n; break;
                case '-': base += temp; temp = -n; break;
                case '*': temp *= n; break;
                case '/': temp /= n; break;
            }
            if (i < s.length()) {
                op = s[i];
            }
        }
        ++i;
    }
    return base + temp;
}

long parseNum(const string& s, int& i) {
    long n = 0;
    while (i < s.length() && isdigit(s[i])) {
        n = 10 * n + (s[i++] - '0');
    }
    return n;
}

```

772. Basic Calculator III (Hard)

题目描述

输入输出样例

Input: "(2+6* 3+5- (3*14/7+2)*5)+3"
Output: -12

题解

```

int calculate(string s) {
    int i = 0;
    return parseExpr(s, i);
}

int parseExpr(const string& s, int& i) {
    char op = '+';
    long base = 0, temp = 0;
    while (i < s.length() && op != ')') { // 注意不同处
        if (s[i] != ' ') {

```



```

        long n = s[i] == '(' ? parseExpr(s, ++i) : parseNum(s, i); // 注意不
        同处
        switch (op) {
            case '+': base += temp; temp = n; break;
            case '-': base += temp; temp = -n; break;
            case '*': temp *= n; break;
            case '/': temp /= n; break;
        }
        if (i < s.length()) {
            op = s[i];
        }
    }
    ++i;
}
return base + temp;
}

long parseNum(const string& s, int& i) {
    long n = 0;
    while (i < s.length() && isdigit(s[i])) {
        n = 10 * n + (s[i++] - '0');
    }
    return n;
}

```

13.3 字符串匹配

28. Implement strStr() (Easy)

题目描述

输入输出样例

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

题解

使用 Knuth-Morris-Pratt (KMP) 算法，可以在 $O(m + n)$ 时间完成匹配。

```

int strStr(string haystack, string needle) {
    int k = -1, n = haystack.length(), p = needle.length();
    if (p == 0) return 0;
    vector<int> next(p, -1); // -1表示不存在相同的最大前缀和后缀
    calNext(needle, next); // 计算next数组
    for (int i = 0; i < n; ++i) {
        while (k > -1 && needle[k+1] != haystack[i]) {
            k = next[k]; // 有部分匹配，往前回溯
        }
        if (needle[k+1] == haystack[i]) {
            ++k;
        }
    }
}

```

```
        if (k == p-1) {
            return i - p + 1; // 说明k移动到needle的最末端，返回相应的位置
        }
    }
    return -1;
}

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

13.4 练习

基础难度

409. Longest Palindrome (Easy)

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

3. Longest Substring Without Repeating Characters (Medium)

计算最长无重复子字符串，同样的，可以利用其它数据结构进行辅助统计。

进阶难度

5. Longest Palindromic Substring (Medium)

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

第 14 章 指针三剑客之一：链表

14.1 数据结构介绍

链表是空节点，或者有一个值和一个指向下一个链表的指针，因此很多链表问题可以用递归来处理。

注意：一般来说，算法题不需要删除内存。实际面试时，对于无用的内存，则尽量显式回收。一个常用的小技巧是，dummy，TODO。

14.2 链表的基本操作

206. Reverse Linked List (Easy)

题目描述

输入输出样例

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

题解

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

递归的写法为：

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

非递归的写法为：

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

21. Merge Two Sorted Lists (Easy)

题目描述

输入输出样例

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

题解

我们提供了递归和非递归，共两种写法。递归的写法为：

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

非递归的写法为：

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

24. Swap Nodes in Pairs (Medium)

题目描述

输入输出样例

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

题解

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

14.3 其它链表技巧

160. Intersection of Two Linked Lists (Easy)

题目描述

输入输出样例

Input:
A: a1 a2

 c1 c2 c3

B: b1 b2 b3
Output: c1

题解

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

```
    return l1;  
}
```

234. Palindrome Linked List (Easy)

题目描述

以 $O(1)$ 的空间复杂度，判断链表是否回文。

输入输出样例

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

题解

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

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

14.4 练习

基础难度

83. Remove Duplicates from Sorted List (Easy)

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

328. Odd Even Linked List (Medium)

这道题其实很简单，千万不要把题目复杂化。

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

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

进阶难度

148. Sort List (Medium)

可以对链表进行 Merge Sort，利用快慢指针找到中点



第 15 章 指针三剑客之二：树

15.1 树的递归

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

104. Maximum Depth of Binary Tree (Easy)

题目描述

输入输出样例

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

题解

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

110. Balanced Binary Tree (Easy)

题目描述

输入输出样例

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


题解

```
bool isBalanced(TreeNode* root) {  
    return helper(root) != -1;  
}  
int helper(TreeNode* root) {  
    if (!root) {  
        return 0;  
    }  
    int left = helper(root->left), right = helper(root->right);  
    if (left == -1 || right == -1 || abs(left - right) > 1) {  
        return -1;  
    }  
    return 1 + max(left, right);  
}
```

543. Diameter of Binary Tree (Easy)

题目描述

输入输出样例

Input:

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

Output: 3

在这个样例中，最长直径是 [4,2,1,3] 和 [5,2,1,3]。

题解

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

437. Path Sum III (Easy)

题目描述

输入输出样例

Input: sum = 8, tree =

```
    10
   /  \
  5    -3
 / \   \
3  2   11
/ \   \
3 -2  1
Output: 3
```

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

题解

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

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

101. Symmetric Tree (Easy)

题目描述

输入输出样例

Input:

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

题解

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

15.2 层次遍历

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

637. Average of Levels in Binary Tree (Easy)

题目描述

输入输出样例

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

题解

```
vector<double> averageOfLevels(TreeNode* root) {
    vector<double> ans;
    if (!root) {
        return ans;
    }
    queue<TreeNode*> q;
    q.push(root);
```

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

15.3 前中后序遍历

前中后序遍历 TODO。

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

题目描述

输入输出样例

Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]

Output:

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

题解

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

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

144. Binary Tree Preorder Traversal (Medium)

题目描述

输入输出样例

Input:

```
1
 \
  2
 /
3
```

Output: [1,2,3]

题解

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

15.4 二叉查找树

二叉查找树 (Binary Search Tree, BST) 实现 TODO。

因为二叉查找树是有序的，因此中序遍历的结果即为排好序的数组。

99. Recover Binary Search Tree (Hard)

题目描述

输入输出样例

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

题解

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

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

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

```
        } else {
            mistake2 = root;
        }
        cout << mistake1->val;
        cout << mistake2->val;
    }
    prev = root;
    if (root->right) {
        inorder(root->right, mistake1, mistake2, prev);
    }
}
```

669. Trim a Binary Search Tree (Easy)

题目描述

输入输出样例

Input: L = 1, R = 3, tree =

```
    3
   / \
  0   4
   \
    2
   /
  1
Output:
    3
   /
  2
 /
1
```

题解

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

15.5 字典树

字典树（Trie）用于判断字符串是否存在或者是否具有某种字符串前缀。

208. Implement Trie (Prefix Tree) (Medium)

题目描述

输入输出样例

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

题解

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

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

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

    // 判断字典树里是否有一个词
    bool search(string word) {
        TrieNode* temp = root;
        for (int i = 0; i < word.size(); ++i) {
```



```
        if (!temp) {
            break;
        }
        temp = temp->childNode[word[i]-'a'];
    }
    return temp? temp->isVal: false;
}

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

15.6 练习

基础难度

226. Invert Binary Tree (Easy)

巧用递归，你可以在五行内完成这道题。

617. Merge Two Binary Trees (Easy)

同样的，利用递归可以轻松搞定。

572. Subtree of Another Tree (Easy)

子树是对称树的姊妹题，写法也十分类似。

404. Sum of Left Leaves (Easy)

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

513. Find Bottom Left Tree Value (Easy)

最左下角的节点满足什么条件？针对这种条件，我们该如何找到它？

538. Convert BST to Greater Tree (Easy)

尝试利用某种遍历方式来解决此题，每个节点只需遍历一次。

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

利用 BST 的独特性质，这道题可以很轻松完成。

530. Minimum Absolute Difference in BST (Easy)

还记得我们所说的，对于 BST 应该利用哪种遍历吗？

进阶难度**889. Construct Binary Tree from Preorder and Postorder Traversal (Medium)**

给定任意两种遍历结果，我们都可以重建树的结构。

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

给定任意两种遍历结果，我们都可以重建树的结构。

94. Binary Tree Inorder Traversal (Medium)

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

145. Binary Tree Postorder Traversal (Medium)

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

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

现在不是 BST，而是普通的二叉树了，该怎么办？

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

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

897. Increasing Order Search Tree (Easy)

把 BST 压成一个链表，务必考虑清楚指针操作的顺序，否则可能会出现环路。

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

啊哈，这道题可能会把你骗到。

450. Delete Node in a BST (Medium)

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



第 16 章 指针三剑客之三：图

图的两种表示方法 TODO。

16.1 二分图

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

785. Is Graph Bipartite? (Medium)

题目描述

输入输出样例

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

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

题解

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

```
    }  
  }  
}  
return true;  
}
```

16.2 拓扑排序

介绍 TODO。

210. Course Schedule II (Medium)

题目描述

输入输出样例

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

在这个样例中，另一种可行的结果是 [0,2,1,3]。

题解

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

```
}  
    return res;  
}
```

16.3 最小生成树

最小生成树 (minimum spanning tree, MST), 指的是一个连接图内所有点的树, 且这种连接是所有可能连接里消耗最小的方式。

1135. Connecting Cities With Minimum Cost (Medium)

题目描述

给定 N 个城市, 和一些 $\langle \text{城市 A}, \text{城市 B}, \text{消耗值} \rangle$ 的连接, 求出能让所有城市相连的最小总消耗值。

输入输出样例

```
Input: N = 3, connections = [[1,2,5],[1,3,6],[2,3,1]]  
Output: 6
```

在这个样例中, 选择任意两条路都可以把它们连接起来, 因此我们选择消耗最小的 1 和 5。

题解

最小生成树的问题通常有两种方法 (1) Prim's Algorithm: 利用优先队列选择最小的消耗 (2) Kruskal's Algorithm: 排序后使用并查集。鉴于我们会在最后一章讲解并查集, 我们先卖个关子, 这里只展示第一种写法。

```
int minimumCost(int N, vector<vector<int>>& connections) {  
    vector<vector<pair<int, int>>> graph(N, vector<pair<int, int>>());  
    // pair表示<城市距离, 城市编号>  
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;  
    unordered_set<int> visited;  
    int costs = 0;  
    for (const auto& conn : connections) {  
        graph[conn[0]-1].push_back(make_pair(conn[2], conn[1] - 1));  
        graph[conn[1]-1].push_back(make_pair(conn[2], conn[0] - 1));  
    }  
    pq.push(make_pair(0, 0));  
    while (!pq.empty()) {  
        auto [cost, city2] = pq.top();  
        pq.pop();  
        if (!visited.count(city2)) {  
            costs += cost;  
            visited.insert(city2);  
            for (const auto & v: graph[city2]) {  
                pq.push(v);  
            }  
        }  
    }  
}
```

```
    }  
  }  
  return visited.size() == N? costs: -1;  
}
```

16.4 练习

基础难度

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

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

进阶难度

882. Reachable Nodes In Subdivided Graph (Hard)

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



第 17 章 更加复杂的数据结构

17.1 并查集

并查集 (Union-Find) 可以动态地连通两个点, 并且可以非常快速地判断两个点是否连通。并查集通常会使用按秩合并和路径压缩进行加速。

684. Redundant Connection (Medium)

题目描述

在无向图找出一条边, 移除它之后该图能够成为一棵树 (即无向无环图)。

输入输出样例

如果有多个解, 返回在原数组中位置最靠后的那条边。

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

题解

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

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

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

```

        size[i] += size[j];
    }
}

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

};

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

```

17.2 复合数据结构

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

146. LRU Cache (Medium)

题目描述

设计一个固定大小的，least recently used cache。

输入输出样例

```

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

```


题解

我们采用复合数据结构 `list<pair<int, int>>` 和 `unordered_map<int, list<pair<int, int>>::iterator>`, 存 iterator 的原因是方便调用 list 的 `splice` 函数来直接更新 cash hit 的 pair。

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

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

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

17.3 练习

基础难度

1135. Connecting Cities With Minimum Cost (Medium)

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

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

设计一个插入、删除和随机取值均为 $O(1)$ 时间复杂度的数据结构。

进阶难度

432. All O‘one Data Structure (Hard)

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

716. Max Stack (Easy)

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



第 18 章 Elegant \LaTeX 系列模板介绍

Elegant \LaTeX 项目组致力于打造一系列美观、优雅、简便的模板方便用户使用。目前由 **ElegantNote**, **ElegantBook**, **ElegantPaper** 组成, 分别用于排版笔记, 书籍和工作论文。强烈推荐使用最新正式版本! 本文将介绍本模板的一些设置内容以及基本使用方法。如果您有其他问题, 建议或者意见, 欢迎在 Github 上给我们提交 **issues** 或者邮件联系我们。


我们的联系方式:

- 官网: <https://elegantlatex.org/>
- Github 网址: <https://github.com/ElegantLaTeX/>
- CTAN 地址: <https://ctan.org/pkg/elegantbook>
- 文档 Wiki: <https://github.com/ElegantLaTeX/ElegantBook/wiki>
- 下载地址: 正式发行版, 最新版
- 微博: ElegantLaTeX
- 微信公众号: ElegantLaTeX
- 用户 QQ 群: 692108391
- 邮件: elegantlatex2e@gmail.com

18.1 ElegantBook 更新说明

此次更新主要有

1. 删除 `\elegantpar` 命令;
2. 修复符号字体设置;
3. 增加双栏目录选项;
4. 修改脚注格式;
5. 其他。

 **注意** 2.x 版本的用户请仔细查看跨版本转换。

18.2 模板安装与更新

你可以通过免安装的方式使用本模板, 也可以通过安装模板的方式使用。

免安装使用方法如下, 从 Github 或者 CTAN 下载最新 (正式) 版文件, 严格意义上只需要类文件 `elegantbook.cls`。然后将模板文件放在你的工作目录下即可使用。这样使用的好处是, 无需安装, 简便; 缺点是, 当模板进行更新之后, 你需要手动替换 `cls` 文件。

如果你是 \TeX Live 2019 用户, 我们非常推荐你直接进行安装和更新。你可以通过 \TeX Live 2019 自带的 `tlshello`¹ 进行安装。安装非常简单, 步骤如下, 搜索并打开 `tlshello`, 然后通过 `File -> Load Default Repository` 加载远程仓库, 如果你不想使用默认的仓库, 你可以通过 `Options` 下的菜单设置远程仓库。设置好仓库之后, 等待仓库加载完毕, 你可以在下面的搜索栏搜索 `elegantbook`, 然后选择进行安装与更新。

如果你是 \TeX Live 2018 的用户, 由于 2018 无法直接更新到 2019, 所以你想更新的话, 需要卸载 2018 重装 2019。如果你实在不想折腾, 那么你仍然可以使用本模板。你可以手动安装模板,

¹也叫 \TeX Live Manager

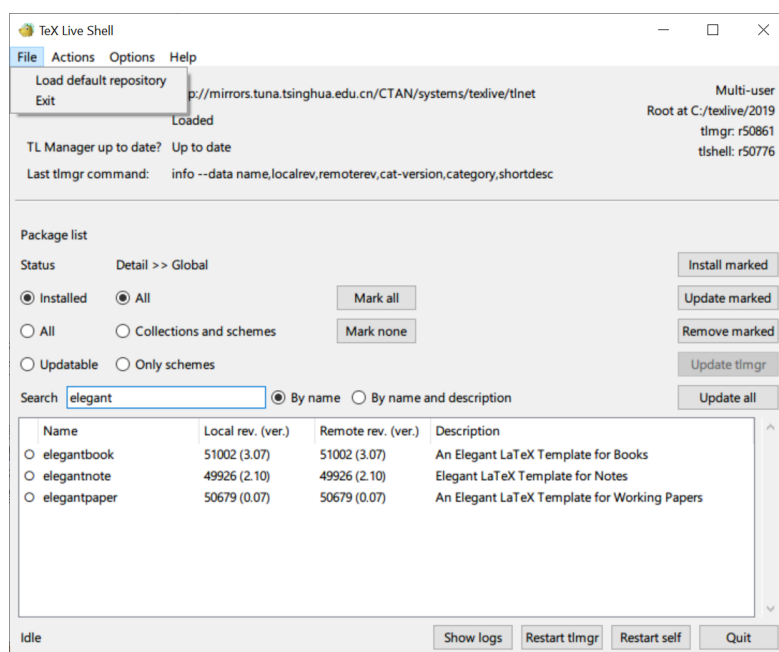


图 18.1: 使用 TeX Live Shell 安装 ElegantBook 模板

将 `elegantbook.cls` 复制到你的 TeX Live 目录下，默认安装目录为 `C:\texlive\2019\texmf-dist\tex\latex\``elegantbook`，然后通过命令行（管理员权限），运行 `texhash` 即可。

啥？你是 CTeX 用户？Sorry，本模板不提供支持。

更多关于 TeX Live 2019 的安装使用以及 CTeX 与 TeX Live 的兼容、系统路径问题，请参考官方文档以及嘯行的一份简短的安装 LaTeX 的介绍。

18.3 在线使用模板

我们把三套模板全部上传到 Overleaf 上了，网络便利的用户可以直接通过 Overleaf 在线使用我们的模板。使用 Overleaf 的好处是无需安装 TeX Live 2019，可以随时随地访问自己的文件。查找模板，请在 Overleaf 模板库里面搜索 `elegantlatex` 即可，你也可以直接访问搜索结果。选择适当的模板之后，将其 `Open as Template`，即可把模板存到自己账户下，然后可以自由编辑以及与他人一起协作。更多关于 Overleaf 的介绍和使用，请参考 Overleaf 的官方文档。

注 Overleaf 上，中文需要使用 XeLaTeX 进行编译，英文可以使用 PDFLaTeX 与 XeLaTeX 进行编译。

18.4 用户作品计划

ElegantLaTeX 系列模板从创立至今已经有 8 年了，我们的模板也受到了很多用户的喜爱，在此，为了促进模板用户之间的交流，了解用户需求，完善本模板，我们将建立一个区域专门展示用户的文档，包括但不限于 Github 和官网等。如果你愿意将自己的作品展示出来，请邮件或者其他方式联系我们。如果自己代码已经传到 Github 或者 Gitee 等网站，可以提供对应网址。

广告位招租！

18.5 关于提交

出于某些因素的考虑，Elegant \LaTeX 项目自 2019 年 5 月 20 日开始，**不再接受任何非作者预约性质的提交**（pull request）！如果你想改进模板，你可以给我们提交 issues，或者可以在遵循协议（LPPL-1.3c）的情况下，克隆到自己仓库下进行修改。

18.6 协作人员招募

招募 Elegant \LaTeX 的协作人员，没有工资。工作内容：翻译 Elegant \LaTeX 系列模板相关的文稿（中文 \rightarrow 英文），维护模板的 wiki（主要涉及 Markdown 语法），如果有公众号文稿写作经历的话，也可以帮忙写微信稿。本公告长期有效。

目前 Elegant \LaTeX 共有 4 名协作人员，分别是

- 官方文档翻译: **YPY**;
- Github 维基维护: **Ingo Zinngo**、**追寻原风景**;
- QQ 群管理员: **Sikouhju**.

在此感谢他们无私的奉献！

18.7 致谢

2019 年 5 月 20 日，ElegantBook 模板在 Github 上的收藏数（star）达到了 100²。

在此特别感谢 China \LaTeX 以及 **L \LaTeX 工作室** 对于本系列模板的大力宣传与推广。L \LaTeX 工作室网站上有很多精彩的帖子和精致的模板，欢迎大家去挖掘里面的宝藏。这也是国内最全面的 L \LaTeX 相关的网站。也非常感谢 **muzimuzhi** 对于模板的完善。

如果你喜欢我们的模板，你可以在 Github 上收藏我们的模板。

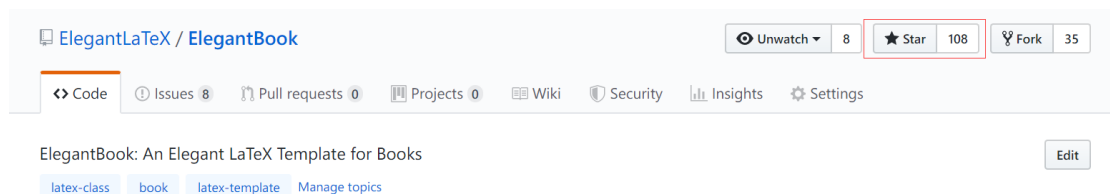


图 18.2: 一键三连求赞

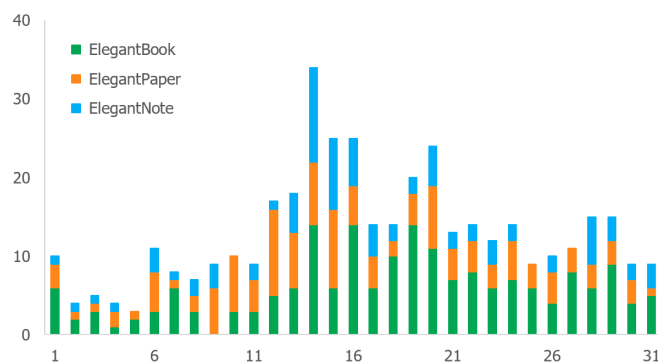


图 18.3: Elegant \LaTeX 系列模板上线 Github 之后上每周 Star 数变化

²截止 2019 年 8 月 18 日 3.09 版本正式发布，star 数为 189。

18.8 捐赠

如果您非常喜爱我们的模板或者我，你还可以选择捐赠以表达您对我们模板和我的支持。本模板自 3.08 版本发布了捐赠信息之后，收到了近千元的捐赠（四舍五入就是一个亿），非常感谢！



微信



支付宝

赞赏费用的使用解释权归 **Elegant \LaTeX** 所有，并且不接受监督，请自愿理性打赏。10 元以上的赞赏，我们将列入捐赠榜，谢谢各位金主！

表 18.1: 捐赠榜

捐赠者	金额	时间	渠道
Lerh	10 元	2019/05/15	微信
越过地平线	10 元	2019/05/15	微信
大熊	20 元	2019/05/27	微信
佚名	10 元	2019/05/30	微信
latexstudio.net	666 元	2019/06/05	支付宝
Cassis	11 元	2019/06/30	微信
佚名	10 元	2019/07/23	微信



第 19 章 ElegantBook 设置说明

本模板基于基础的 book 文类，所以 book 的选项对于本模板也是有效的（纸张无效，因为模板有设备选项）。默认编码为 UTF-8，推荐使用 \TeX Live 编译。本文编写环境为 Win10 (64bit) + \TeX Live 2019，支持 $\text{\textcolor{red}{PDFLaTeX}}$ 以及 $\text{\textcolor{red}{XeLaTeX}}$ 编译。

19.1 语言模式

本模板内含两套语言环境，改变语言环境会改变图表标题的引导词（图，表），文章结构词（比如目录，参考文献等），以及定理环境中的引导词（比如定理，引理等）。不同语言模式的启用如下：

```
\documentclass[cn]{elegantbook}
\documentclass[lang=cn]{elegantbook}
```

注 只有中文环境（ $\text{\textcolor{red}{lang}}=\text{\textcolor{red}{cn}}$ ）才可以输入中文。另外如果抄录环境（ $\text{\textcolor{red}{lstlisting}}$ ）中有中文字符，请务必使用 $\text{\textcolor{red}{XeLaTeX}}$ 编译。

19.2 设备选项

最早我们在 ElegantNote 模板中加入了设备选项（ $\text{\textcolor{red}{device}}$ ），后来，我们觉得这个设备选项的设置可以应用到 ElegantBook 中¹，而且 Book 一般内容比较多，如果在 iPad 上看无需切边，放大，那用户的阅读体验将会得到巨大提升。你可以使用下面的选项将版面设置为 iPad 设备模式²

```
\documentclass[pad]{elegantbook} %or
\documentclass[device=pad]{elegantbook}
```

19.3 颜色主题

本模板内置 5 组颜色主题，分别为 $\text{\textcolor{green}{green}}$ ³、 $\text{\textcolor{cyan}{cyan}}$ 、 $\text{\textcolor{blue}{blue}}$ （默认）、 $\text{\textcolor{gray}{gray}}$ 、 $\text{\textcolor{black}{black}}$ 。另外还有一个自定义的选项 $\text{\textcolor{red}{nocolor}}$ 。调用颜色主题 $\text{\textcolor{green}{green}}$ 的方法为

```
\documentclass[green]{elegantbook} %or
\documentclass[color=green]{elegantbook}
```

如果需要自定义颜色的话请选择 $\text{\textcolor{red}{nocolor}}$ 选项或者使用 $\text{\textcolor{red}{color}}=\text{\textcolor{red}{none}}$ ，然后在导言区定义 $\text{\textcolor{red}{structurecolor}}$ 、 $\text{\textcolor{red}{main}}$ 、 $\text{\textcolor{red}{second}}$ 、 $\text{\textcolor{red}{third}}$ 颜色，具体方法如下：

```
\definecolor{structurecolor}{RGB}{0,0,0}
\definecolor{main}{RGB}{70,70,70}
\definecolor{second}{RGB}{115,45,2}
\definecolor{third}{RGB}{0,80,80}
```

¹不过因为 ElegantBook 模板封面图片的存在，在修改页面设计时，需要对图片进行裁剪。

²默认为 normal 模式，也即 A4 纸张大小。

³为原先默认主题。

表 19.1: ElegantBook 模板中的颜色主题

	green	cyan	blue	gray	black	主要使用的环境
structure	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	chapter section subsection
main	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	definition exercise problem
second	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	theorem lemma corollary
third	<div></div>	<div></div>	<div></div>	<div></div>	<div></div>	proposition

19.4 章标题显示风格

本模板内置 2 套章标题显示风格，包含 `hang`（默认）与 `display` 两种风格，区别在于章标题单行显示（`hang`）与双行显示（`display`），本说明使用了 `hang`。调用方式为

```
\documentclass[hang]{elegantbook} %or
\documentclass[titlestyle=hang]{elegantbook}
```

19.5 数学环境简介

在我们这个模板中，我们定义了两种不同的定理模式 `mode`，包括简单模式（`simple`）和炫彩模式（`fancy`），默认为 `fancy` 模式，不同模式的选择为

```
\documentclass[simple]{elegantbook} %or
\documentclass[mode=simple]{elegantbook}
```

本模板定义了四大类环境

- 定理类环境，包含标题和内容两部分，全部定理类环境的编号均以章节编号。根据格式的不同分为 3 种
 - `definition` 环境，颜色为 `main`；
 - `theorem`、`lemma`、`corollary` 环境，颜色为 `second`；
 - `proposition` 环境，颜色为 `third`。
- 示例类环境，有 `example`、`problem`、`exercise` 环境（对应于例、例题、练习），自动编号，编号以章节为单位，其中 `exercise` 有提示符。
- 提示类环境，有 `note` 环境，特点是：无编号，有引导符。
- 结论类环境，有 `conclusion`、`assumption`、`property`、`remark`、`solution` 环境⁴，三者均以粗体的引导词为开头，和普通段落格式一致。

19.5.1 定理类环境的使用

由于本模板使用了 `tcolorbox` 宏包来定制定理类环境，所以和普通的定理环境的使用有些许区别，定理的使用方法如下：

⁴本模板还添加了一个 `result` 选项，用于隐藏 `solution` 和 `proof` 环境，默认为显示（`result=answer`），隐藏使用 `result=noanswer`。


```
\begin{theorem}{theorem name}{label}
The content of theorem.
\end{theorem}
```

第一个必选项 `theorem name` 是定理的名字，第二个必选项 `label` 是交叉引用时所用到的标签，交叉引用的方法为 `\ref{thm:label}`。请注意，交叉引用时必须加上前缀 `thm:`。

其他相同用法的定理类环境有：

表 19.2: 定理类环境

环境名	标签名	前缀	交叉引用
definition	label	def	<code>\ref{def:label}</code>
theorem	label	thm	<code>\ref{thm:label}</code>
lemma	label	lem	<code>\ref{lem:label}</code>
corollary	label	cor	<code>\ref{cor:label}</code>
proposition	label	pro	<code>\ref{pro:label}</code>

19.5.2 其他环境的使用

其他三种环境没有选项，可以直接使用，比如 `example` 环境的使用方法与效果：

```
\begin{example}
This is the content of example environment.
\end{example}
```

例 19.1 This is the content of example environment.

这几个都是同一类环境，区别在于

- 示例环境 (`example`)、练习 (`exercise`) 与例题 (`problem`) 章节自动编号；
- 注意 (`note`)，练习 (`exercise`) 环境有提醒引导符；
- 结论 (`conclusion`) 等环境都是普通段落环境，引导词加粗。

19.6 装饰物

本模板为章节后和页面下方的装饰物 (`base`) 添加了隐藏选项，有 `show` 和 `hide` 两个选项。

```
\documentclass[hide]{elegantbook} %or
\documentclass[base=hide]{elegantbook}
```

19.7 封面和徽标

本模板使用的封面图片来源于 pixabay.com⁵，图片完全免费，可用于任何场景。封面图片的尺寸为 1280 × 1024，更换图片的时候请严格按照封面图片尺寸进行裁剪。推荐一个免费的在线图片裁剪网站 fotor.com。用户 QQ 群内有一些合适尺寸的封面，欢迎取用。

本文用到的 Logo 比例为 1:1，也即正方形图片，在更换图片的时候请选择合适的图片进行替换。

⁵感谢 ChinaTeX 提供免费图源网站，另外还推荐 pexels.com。

19.8 列表环境

本模板借助于 `tikz` 定制了 `itemize` 和 `enumerate` 环境，其中 `itemize` 环境修改了 3 层嵌套，而 `enumerate` 环境修改了 4 层嵌套（仅改变颜色）。示例如下

- | | |
|---------------------------|-----------------------------|
| • first item of nesti; | 1. first item of nesti; |
| • second item of nesti; | 2. second item of nesti; |
| • first item of nestii; | (a). first item of nestii; |
| • second item of nestii; | (b). second item of nestii; |
| • first item of nestiii; | I. first item of nestiii; |
| • second item of nestiii. | II. second item of nestiii. |

19.9 参考文献

此模板使用了 `BibTeX` 来生成参考文献，在中文示例中，使用了 `gbt7714` 宏包。参考文献示例：[1-3] 使用了中国一个大型的 P2P 平台（人人贷）的数据来检验男性投资者和女性投资者在投资表现上是否有显著差异。

你可以在谷歌学术，Mendeley，Endnote 中获得文献条目（bib item），然后把它们添加到 `reference.bib` 中。在文中引用的时候，引用它们的键值（bib key）即可。注意需要在编译的过程中添加 `BibTeX` 编译。如果你想添加未引用的文献，可以使用

```
\nocite{EINAV2010,Havrylchyk2018} %or include some bibitems
\nocite{*} %include all the bibitems
```

本模板还添加了 `cite=numbers`、`cite=super` 和 `cite=authoryear` 三个参考文献选项，用于设置参考文献格式的设置，默认为 `numbers`。据我们所知，理工科类一般使用数字形式 `numbers` 或者上标形式 `super`，而文科类使用作者--年份 `authoryear` 比较多，所以我们将 `numbers` 作为默认格式。如果需要改为 `cite=super` 或者 `authoryear`，可以使用

```
\documentclass[cite=super]{elegantbook} % set cite for super style ref style
\documentclass[super]{elegantbook}
\documentclass[cite=authoryear]{elegantbook} % set cite for author year ref style
\documentclass[authoryear]{elegantbook}
```

19.10 添加序章

如果你想在第一章前面添序章，不改变原本章节序号，可以在第一章内容前面使用

```
\chapter*{Introduction}
\addcontentsline{toc}{chapter}{Introduction}
\markboth{Introduction}{}
The content of introduction.
```

19.11 章节摘要

模板新增了一个章节摘要环境（introduction），使用示例

```
\begin{introduction}
  \item Definition of Theorem
  \item Ask for help
  \item Optimization Problem
  \item Property of Cauchy Series
  \item Angle of Corner
\end{introduction}
```

效果如下：

内容提要

- | | |
|--|--|
| <input type="checkbox"/> Definition of Theorem | <input type="checkbox"/> Property of Cauchy Series |
| <input type="checkbox"/> Ask for help | <input type="checkbox"/> Angle of Corner |
| <input type="checkbox"/> Optimization Problem | |

环境的标题文字可以通过这个环境的可选参数进行修改，修改方法为：

```
\begin{introduction}[Brief Introduction]
...
\end{introduction}
```

19.12 章后习题

前面我们介绍了例题和练习两个环境，这里我们再加一个，章后习题（`problemset`）环境，用于在每一章结尾，显示本章的练习。使用方法如下

```
\begin{problemset}
  \item exercise 1
  \item exercise 2
  \item exercise 3
\end{problemset}
```

效果如下：

第 19 章 习题

1. exercise 1
2. exercise 2
3. exercise 3

注 如果你想把 `problemset` 环境的标题改为其他文字，你可以类似于 `introduction` 环境修改 `problemset` 的可选参数。另外，目前这个环境会自动出现在目录中，但是不会出现在页眉页脚信息中（待解决）。

19.13 旁注

在 3.08 版本中，我们引入了旁注设置选项 `marginpar=margintrue` 以及测试命令 `\elegantpar`，但是由此带来一堆问题。我们决定在 3.09 版本中将其删除，并且，在旁注命令得到大幅度优化之前，不会将此命令再次引入书籍模板中。对此造成各位用户的不方便，非常抱歉！不过我们保

留了 `marginpar` 这个选项，你可以使用 `marginpar=margintrue` 获得保留右侧旁注的版面设计。然后使用系统自带的 `\marginpar` 或者 `marginnote` 宏包的 `\marginnote` 命令。

注 在使用旁注的时候，需要注意的是，文本和公式可以直接在旁注中使用。

```
% text
\marginpar{margin paragraph text}

% equation
\marginpar{
  \begin{equation}
    a^2 + b^2 = c^2
  \end{equation}
}
```

但是浮动体（表格、图片）需要注意，不能用浮动体环境，需要使用直接插图命令或者表格命令环境。然后使用 `\captionof` 为其设置标题。为了得到居中的图表，可以使用 `\centerline` 命令或者 `center` 环境。更多详情请参考：[Caption of Figure in Marginpar](#)。

```
% graph with centerline command
\marginpar{
  \centerline{
    \includegraphics[width=0.2\textwidth]{logo.png}
  }
  \captionof{figure}{your figure caption}
}

% graph with center environment
\marginpar{
  \begin{center}
    \includegraphics[width=0.2\textwidth]{logo.png}
    \captionof{figure}{your figure caption}
  \end{center}
}
```

19.14 连字符

由于模板使用了 `newtx` 系列字体宏包，所以在使用本模板的时候，需要注意下连字符的问题。

$$\int_{R^q} f(x,y)dy.off \quad (19.1)$$

的代码为

```
\begin{equation}
  \int_{R^q} f(x,y) dy.\emph{of \kern0pt f}
\end{equation}
```

19.15 符号字体

在 3.08 版本中，用户反馈模板在和 `yhmath` 以及 `esvect` 等宏包搭配使用的时候会出现报错：

LaTeX Error:

Too many symbol fonts declared.

原因是模板重新定义了数学字体，达到了**最多 16 个数学字体**的上限，在调用其他宏包的时候，无法新增数学字体。为了减少调用非常用宏包，在此给出如何调用 `yhmath` 以及 `esvect` 宏包的方法。

请在 `elegantbook.cls` 内搜索 `yhmath` 或者 `esvect`，将你所需要的宏包加载语句取消注释即可。

```
%%% use yhmath pkg, uncomment following code
% \let\oldwidering\widering
% \let\widering\undefined
% \RequirePackage{yhmath}
% \let\widering\oldwidering

%%% use esvect pkg, uncomment following code
% \RequirePackage{esvect}
```

第 20 章 ElegantBook 写作示例

内容提要

□ 积分定义 20.1

□ Fubini 定理 20.1

□ 最优性原理 20.1

□ 柯西列性质 20.1.1

□ 韦达定理

20.1 Lebesgue 积分

在前面各章做了必要的准备后,本章开始介绍新的积分。在 Lebesgue 测度理论的基础上建立了 Lebesgue 积分,其被积函数和积分域更一般,可以对有界函数和无界函数统一处理。正是由于 Lebesgue 积分的这些特点,使得 Lebesgue 积分比 Riemann 积分具有在更一般条件下的极限定理和累次积分交换积分顺序的定理,这使得 Lebesgue 积分不仅在理论上更完善,而且在计算上更灵活有效。

Lebesgue 积分有几种不同的定义方式。我们将采用逐步定义非负简单函数,非负可测函数和一般可测函数积分的方式。

由于现代数学的许多分支如概率论、泛函分析、调和分析等常常用到一般空间上的测度与积分理论,在本章最后一节将介绍一般的测度空间上的积分。

20.1.1 积分的定义

我们将通过三个步骤定义可测函数的积分。首先定义非负简单函数的积分。以下设 E 是 \mathcal{R}^n 中的可测集。

定义 20.1. 可积性

设 $f(x) = \sum_{i=1}^k a_i \chi_{A_i}(x)$ 是 E 上的非负简单函数,其中 $\{A_1, A_2, \dots, A_k\}$ 是 E 上的一个可测分割, a_1, a_2, \dots, a_k 是非负实数。定义 f 在 E 上的积分为 $\int_a^b f(x)$

$$\int_E f dx = \sum_{i=1}^k a_i m(A_i) \quad (20.1)$$

一般情况下 $0 \leq \int_E f dx \leq \infty$ 。若 $\int_E f dx < \infty$, 则称 f 在 E 上可积。

一个自然的问题是, Lebesgue 积分与我们所熟悉的 Riemann 积分有什么联系和区别? 在 4.4 我们将详细讨论 Riemann 积分与 Lebesgue 积分的关系。这里只看一个简单的例子。设 $D(x)$ 是区间 $[0, 1]$ 上的 Dirichlet 函数。即 $D(x) = \chi_{Q_0}(x)$, 其中 Q_0 表示 $[0, 1]$ 中的有理数的全体。根据非负简单函数积分的定义, $D(x)$ 在 $[0, 1]$ 上的 Lebesgue 积分为

$$\int_0^1 D(x) dx = \int_0^1 \chi_{Q_0}(x) dx = m(Q_0) = 0 \quad (20.2)$$

即 $D(x)$ 在 $[0, 1]$ 上是 Lebesgue 可积的并且积分值为零。但 $D(x)$ 在 $[0, 1]$ 上不是 Riemann 可积的。

有界变差函数是与单调函数有密切联系的一类函数。有界变差函数可以表示为两个单调递增函数之差。与单调函数一样,有界变差函数几乎处处可导。与单调函数不同,有界变差函数类对

线性运算是封闭的, 它们构成一线空间。练习题 20.1 是一个性质的证明。

 **练习 20.1** 设 $f \notin L(\mathcal{R}^1)$, g 是 \mathcal{R}^1 上的有界可测函数。证明函数

$$I(t) = \int_{\mathcal{R}^1} f(x+t)g(x)dx \quad t \in \mathcal{R}^1 \quad (20.3)$$

是 \mathcal{R}^1 上的连续函数。

例题 20.1 即 $D(x)$ 在 $[0, 1]$ 上是 Lebesgue 可积的并且积分值为零。但 $D(x)$ 在 $[0, 1]$ 上不是 Riemann 可积的。

例 20.1 即 $D(x)$ 在 $[0, 1]$ 上是 Lebesgue 可积的并且积分值为零。但 $D(x)$ 在 $[0, 1]$ 上不是 Riemann 可积的。

证明 测试证明环境


解 即 $D(x)$ 在 $[0, 1]$ 上是 Lebesgue 可积的并且积分值为零。但 $D(x)$ 在 $[0, 1]$ 上不是 Riemann 可积的。

定理 20.1. Fubini 定理

(1) 若 $f(x, y)$ 是 $\mathcal{R}^p \times \mathcal{R}^q$ 上的非负可测函数, 则对几乎处处的 $x \in \mathcal{R}^p$, $f(x, y)$ 作为 y 的函数是 \mathcal{R}^q 上的非负可测函数, $g(x) = \int_{\mathcal{R}^q} f(x, y)dy$ 是 \mathcal{R}^p 上的非负可测函数。并且

$$\int_{\mathcal{R}^p \times \mathcal{R}^q} f(x, y)dx dy = \int_{\mathcal{R}^p} \left(\int_{\mathcal{R}^q} f(x, y)dy \right) dx. \quad (20.4)$$

(2) 若 $f(x, y)$ 是 $\mathcal{R}^p \times \mathcal{R}^q$ 上的可积函数, 则对几乎处处的 $x \in \mathcal{R}^p$, $f(x, y)$ 作为 y 的函数是 \mathcal{R}^q 上的可积函数, 并且 $g(x) = \int_{\mathcal{R}^q} f(x, y)dy$ 是 \mathcal{R}^p 上的可积函数。而且 20.4 成立。

 **注意** 在本模板中, 引理 (lemma), 推论 (corollary) 的样式和定理 20.1 的样式一致, 包括颜色, 仅仅只有计数器的设置不一样。

我们说一个实变或者复变量的实值或者复值函数是在区间上平方可积的, 如果其绝对值的平方在该区间上的积分是有限的。所有在勒贝格积分意义下平方可积的可测函数构成一个希尔伯特空间, 也就是所谓的 L^2 空间, 几乎处处相等的函数归为同一等价类。形式上, L^2 是平方可积函数的空间和几乎处处为 0 的函数空间的商空间。

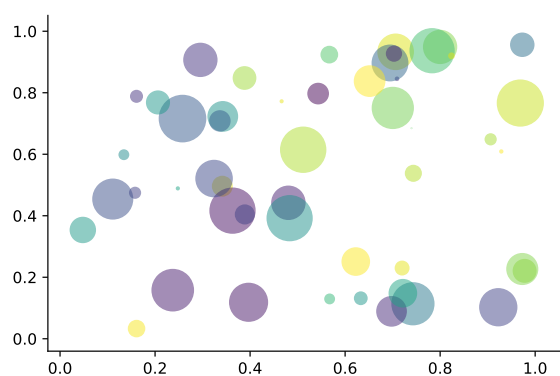
命题 20.1. 最优性原理

如果 u^* 在 $[s, T]$ 上为最优解, 则 u^* 在 $[s, T]$ 任意子区间都是最优解, 假设区间为 $[t_0, t_1]$ 的最优解为 u^* , 则 $u(t_0) = u^*(t_0)$, 即初始条件必须还是在 u^* 上。

我们知道最小二乘法可以用来处理一组数据, 可以从一组测定的数据中寻求变量之间的依赖关系, 这种函数关系称为经验公式。本课题将介绍最小二乘法的精确定义及如何寻求点与点之间近似成线性关系时的经验公式。假定实验测得变量之间的 n 个数据, 则在平面上, 可以得到 n 个点, 这种图形称为“散点图”, 从图中可以粗略看出这些点大致散落在某直线近旁, 我们认为其近似为一线性函数, 下面介绍求解步骤。

以最简单的一元线性模型来解释最小二乘法。什么是一元线性模型呢? 监督学习中, 如果预测的变量是离散的, 我们称其为分类 (如决策树, 支持向量机等), 如果预测的变量是连续的, 我们称其为回归。回归分析中, 如果只包括一个自变量和一个因变量, 且二者的关系可用一条直线近似表示, 这种回归分析称为一元线性回归分析。如果回归分析中包括两个或两个以上的自变量, 且因变量和自变量之间是线性关系, 则称为多元线性回归分析。对于二维空间线性是一条直线; 对于三维空间线性是一个平面, 对于多维空间线性是一个超平面。

性质 柯西列的性质

图 20.1: 散点图示例 $\hat{y} = a + bx$

1. $\{x_k\}$ 是柯西列, 则其子列 $\{x_k^i\}$ 也是柯西列。
2. $x_k \in \mathcal{R}^n$, $\rho(x, y)$ 是欧几里得空间, 则柯西列收敛, (\mathcal{R}^n, ρ) 空间是完备的。

结论 回归分析 (regression analysis) 是确定两种或两种以上变量间相互依赖的定量关系的一种统计分析方法。运用十分广泛, 回归分析按照涉及的变量的多少, 分为一元回归和多元回归分析; 按照因变量的多少, 可分为简单回归分析和多重回归分析; 按照自变量和因变量之间的关系类型, 可分为线性回归分析和非线性回归分析。

第 20 章 习题

1. 设 A 为数域 K 上的 n 级矩阵。证明: 如果 K^n 中任意非零列向量都是 A 的特征向量, 则 A 一定是数量矩阵。
2. 证明: 不为零矩阵的幂零矩阵不能对角化。
3. 设 $A = (a_{ij})$ 是数域 K 上的一个 n 级上三角矩阵, 证明: 如果 $a_{11} = a_{22} = \cdots = a_{nn}$, 并且至少有一个 $a_{kl} \neq 0 (k < l)$, 则 A 一定不能对角化。

第 21 章 常见问题集

问题 有没有办法章节用“第一章，第一节，(一)”这种？

解 你可以修改模板中对于章节的设置，利用 `ctex` 宏集的 `\zhnumber` 命令可以把计数器的数字形式转为中文。

问题 3.07 版本的 `cls` 的 `natbib` 加了 `numbers` 编译完了没变化，群主设置了不可更改了？

解 3.07 中在 `gbt7714` 宏包使用时，加入了 `authoryear` 选项，这个使得 `natbib` 设置了 `numbers` 也无法生效。3.08 和 3.09 版本中，模板新增加了 `numbers`、`super` 和 `authoryear` 文献选项，你可以参考前文设置说明。

问题 大佬，我想把正文字体改为亮色，背景色改为黑灰色。

解 页面颜色可以使用 `\pagecolor` 命令设置，文本命令可以参考[这里](#)进行设置。

问题 Package `ctex` Error: CTeX fontset ‘Mac’ is unavailable.

解 在 Mac 系统下，中文编译请使用 `XeLaTeX`。

问题 ! LaTeX Error: Unknown option ‘scheme=plain’ for package ‘ctex’.

解 你用的 CTeX 套装吧？这个里面的 `ctex` 宏包已经是已经是 10 年前的了，与本模板使用的 `ctex` 宏集有很大区别。不建议 CTeX 套装了，请卸载并安装 T_EX Live 2019。

问题 我该使用什么版本？

解 请务必使用**最新正式发行版**，发行版间不定期可能会有更新（修复 bug 或者改进之类），如果你在使用过程中没有遇到问题，不需要每次更新**最新版**，但是在发行版更新之后，请尽可能使用最新版（发行版）！最新发行版可以在 Github 或者 T_EX Live 2019 内获取。

问题 我该使用什么编辑器？

解 你可以使用 T_EX Live 2019 自带的编辑器 T_EXworks 或者使用 T_EXstudio，T_EXworks 的自动补全，你可以参考我们的总结 [T_EXworks 自动补全](#)。推荐使用 T_EX Live 2019 + T_EXStudio。我自己用 VS Code 和 Sublime Text，相关的配置说明，请参考 [L^AT_EX 编译环境配置：Visual Studio Code 配置简介](#) 和 [Sublime Text 搭建 L^AT_EX 编写环境](#)。

问题 您好，我们想用您的 ElegantBook 模板写一本书。关于机器学习的教材，希望获得您的授权，谢谢您的宝贵时间。

解 模板的使用修改都是自由的，你们声明模板来源以及模板地址（github 地址）即可，其他未尽事宜按照开源协议 LPPL-1.3c。做好之后，如果方便的话，可以给我们一个链接，我把你们的教材放在 ElegantLaTeX 用户作品集里。

问题 我想要原来的封面！

解 我们计划在未来版本加入封面选择，让用户可以选择旧版封面。

问题 我想修改中文字体！

解 首先，我们**强烈建议你不要去修改字体**！如果你一定坚持修改字体，请在 `newtexttext` 宏包加载前加入中文字体设置（`xeCJK` 宏包）。

如果你选择自定义字体，请设置好 `\kaishu`，`\heiti` 等命令，否则会报错。如果你看不懂我现在说的，请停止你的字体自定义行为。

问题 请问交叉引用是什么？

解 本群和本模板适合有一定 \LaTeX 基础的用户使用，新手请先学习 \LaTeX 的基础，理解各种概念，否则你将寸步难行。

问题 定义等环境中无法使用加粗命令么？

解 是这样的，默认中文并没加粗命令，如果你想在定义等环境中使用加粗命令，请使用 `\heiti` 等字体命令，而不要使用 `\textbf`。或者，你可以将 `\textbf` 重新定义为 `\heiti`。英文模式不存在这个问题。

问题 代码高亮环境能用其他语言吗？

解 可以的，ElegantBook 模板用的是 `listings` 宏包，你可以在环境 (`\lstlisting`) 之后加上语言（比如 Python 使用 `language=Python` 选项），全局语言修改请使用 `\lstset` 命令，更多信息请参考宏包文档。

问题 群主，什么时候出 Beamer 的模板（主题），ElegantSlide 或者 ElegantBeamer？

解 这个问题问的人比较多，我这里给个明确的答案。由于 Beamer 中有一个很优秀的主题 **Metropolis**。我觉得在我们找到非常好的创意之前不会发布正式的 Beamer 主题，如果你非常希望得到 Elegant \LaTeX “官方”的主题，请在用户 QQ 群内下载我们测试主题 PreElegantSlide（未来不一定按照这个制作）。正式版制作计划在 2020 年之后。

参考文献

- [1] QUADRINI V. Financial Frictions in Macroeconomic Fluctuations[J]. FRB Richmond Economic Quarterly, 2011, 97(3): 209-254.
- [2] CARLSTROM C T, FUERST T S. Agency Costs, Net Worth, and Business Fluctuations: A Computable General Equilibrium Analysis[J]. The American Economic Review, 1997:893-910.
- [3] LI Q, CHEN L, ZENG Y. The Mechanism and Effectiveness of Credit Scoring of P2P Lending Platform: Evidence from Renrendai.com[J]. China Finance Review International, 2018, 8(3):256-274.
- [4] 方军雄. 所有制、制度环境与信贷资金配置[J]. 经济研究, 2007(12):82-92.
- [5] 刘凤良, 章潇萌, 于泽. 高投资、结构失衡与价格指数二元分化[J]. 金融研究, 2017(02):54-69.
- [6] 吕捷, 王高望. CPI 与 PPI “背离”的结构性解释[J]. 经济研究, 2015, 50(04):136-149.

附录 基本数学工具

本附录包括了计量经济学中用到的一些基本数学，我们扼要论述了求和算子的各种性质，研究了线性和某些非线性方程的性质，并复习了比例和百分数。我们还介绍了一些在应用计量经济学中常见的特殊函数，包括二次函数和自然对数，前4节只要求基本的代数技巧，第5节则对微分学进行了简要回顾；虽然要理解本书的大部分内容，微积分并非必需，但在一些章末附录和第3篇某些高深专题中，我们还是用到了微积分。

A.1 求和算子与描述统计量

求和算子是用以表达多个数求和运算的一个缩略符号，它在统计学和计量经济学分析中扮演着重要作用。如果 $\{x_i : i = 1, 2, \dots, n\}$ 表示 n 个数的一个序列，那么我们就把这 n 个数的和写为：

$$\sum_{i=1}^n x_i \equiv x_1 + x_2 + \cdots + x_n \quad (\text{A.1})$$

附录 最小示例

```
\documentclass[lang=cn,11pt]{elegantbook}
% title info
\title{Title}
\subtitle{Subtitle is here}
% bio info
\author{Your Name}
\institute{XXX University}
\date{\today}
% extra info
\version{1.00}
\extrainfo{Victory won\rq t come to us unless we go to it. --- M. Moore}
\logo{logo.png}
\cover{cover.jpg}

\begin{document}

\maketitle
\tableofcontents
\mainmatter
\hypersetup{pageanchor=true}
% add preface chapter here if needed
\chapter{Example Chapter Title}
The content of chapter one.

\bibliography{reference}

\end{document}
```