LeetCode 101：刷题答案（C++）

LeetCode 101: Homework Answer (C++ Version)

作者：Austin

组织：USTC

时间：October 6, 2022

版本：0.6

封面图源：Muse Dash

我所犯下的大罪，光是懒惰就以足够。——折木奉太郎

目录

第 1 章 题目分类

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

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

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

第 2 章 贪心算法

前情提要

- 排序
- 简单遍历
- 寻找递推式

605. Can Place Flowers (Easy)

题目描述

假设有一个很长的花坛，一部分地块种植了花，另一部分却没有。可是，花不能种植在相邻的地块上，它们会争夺水源，两者都会死去。

给你一个整数数组表示花坛，另有一个数表示需要种植的花的数目，能否在不打破种植规则的情况下种入这么多朵花?

输入输出样例

输入是一个数组和一个整数，表示花坛和需要种植的花的数目。输出表示能否不打破规则种植的布尔值。

```
Input: flowerbed = [1,0,0,0,1], n = 1
Output: true
```

题解

方法一

利用条件判断解决边界问题。

```
bool canPlaceFlowers(vector<int>& flowerbed, int n) {
    int m = flowerbed.size(), total = 0;
    if (m == 1) {
        if (!flowerbed[0]) {
            return n <= 1;
        }
        return n == 0;
    }
    if ((!flowerbed[0]) && (!flowerbed[1])) {
        flowerbed[0] = 1;
        total++;
    }
    if ((!flowerbed[m-2]) && (!flowerbed[m-1])) {
        flowerbed[m-1] = 1;
        total++;
    }
    for (int i = 1; i < m - 1; i++) {
        if ((!flowerbed[i-1]) && (!flowerbed[i]) && (!flowerbed[i+1])) {
```

```
            flowerbed[i] = 1;
            total++;
        }
    }
    return n <= total;
}
```

方法二

五次条件判断显得过于冗杂，采用“跳格子”的方法或许清晰易懂一些。从开始位置判断，如果是 1 则跳过两个格子，如果是 0 则判断其后的格子为 1 还是为 0。由于花坛中的花符合规则，所以为 0 的格子前必然是 0，因为如果为 1 的话则这个格子肯定被跳过了。如果 0 后的格子为 0，就在当前位置种植花；如果为 1，就跳过三个格子，相当于跳到 1 处格子后再跳两格。

```
bool canPlaceFlowers(vector<int>& flowerbed, int n) {
    int i = 0, m = flowerbed.size();
    while (i < flowerbed.size()) {
        if (flowerbed[i] == 1) {
            i += 2;
        } else if (i == flowerbed.size() - 1 || flowerbed[i + 1] == 0) {
            i += 2;
            n--;
        } else {
            i += 3;
        }
    }
    return n <= 0;
}
```

方法三

由于花只能种在花坛的空位处，所以可以用数学方法计算两个相邻的 1 之间有多少个 0，进而求出可种植的花的数目。

```
bool canPlaceFlowers(vector<int>& flowerbed, int n) {
    int count = 0;
    int m = flowerbed.size();
    int prev = -1;
    for (int i = 0; i < m; i++) {
        if (flowerbed[i] == 1) {
            if (prev < 0) {
                count += i / 2;
            } else {
                count += (i - prev - 2) / 2;
            }
            if (count >= n) {
                return true;
            }
            prev = i;
```

```
        }
    }
    if (prev < 0) {
        count += (m + 1) / 2;
    } else {
        count += (m - prev - 1) / 2;
    }
    return count >= n;
}
```

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

题目描述

有一些球形气球贴在一堵墙面上。墙面上的气球记录在整数数组中。一支弓箭可以沿着 x 轴从不同点完全垂直地射出。在坐标 x 处射出一支箭，若有一个气球的直径的开始和结束坐标包含 x ，则该气球会被引爆。可以射出的弓箭的数量没有限制。弓箭一旦被射出之后，可以无限地前进。

给你一个数组，返回引爆所有气球所必须射出的最小弓箭数 。

输入输出样例

```
Input: points = [[10,16],[2,8],[1,6],[7,12]]
Output: 2
```

题解

一定存在一种最优（射出的箭数最小）的方法，使得每一支箭的射出位置都恰好对应着某一个气球的右边界。考虑所有气球中右边界位置最靠左的那一个，那么一定有一支箭的射出位置就是它的右边界（否则就没有箭可以将其引爆了）。将这支箭引爆的所有气球移除，并从剩下未被引爆的气球中，再选择右边界位置最靠左的那一个，确定下一支箭，直到所有的气球都被引爆。

```
int findMinArrowShots(vector<vector<int>>& points) {
    if (points.empty()) {
        return 0;
    }
    sort(points.begin(), points.end(), [](const vector<int>& vec1, const vector<int>& vec2) {
        return vec1[1] < vec2[1];
    });
    int right = points[0][1];
    int ans = 1;
    for (const vector<int>& balloon: points) {
        if (balloon[0] > right) {
            right = balloon[1];
            ++ans;
        }
    }
    return ans;
```

```
}
```

763. Partition Labels (Medium)

题目描述

字符串由小写字母组成。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。返回一个表示每个字符串片段的长度的列表。

输入输出样例

```
Input: S = "ababcbacadefegdehijhklij"
Output: [9,7,8]
```

题解

遍历字符串，得到每个字母最后一次出现的下标位置。在得到每个字母最后一次出现的下标位置之后，可以使用贪心的方法将字符串划分为尽可能多的片段，寻找每个片段可能的最小结束下标。

```
vector<int> partitionLabels(string s) {
    int last[26];
    int length = s.size();
    for (int i = 0; i < length; i++) {
        last[s[i] - 'a'] = i;
    }
    vector<int> partition;
    int start = 0, end = 0;
    for (int i = 0; i < length; i++) {
        end = max(end, last[s[i] - 'a']);
        if (i == end) {
            partition.push_back(end - start + 1);
            start = end + 1;
        }
    }
    return partition;
}
```

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

题目描述

给你一个整数数组 prices ， 其中 prices[i] 表示某支股票第 i 天的价格。

在每一天，你可以决定是否购买和/或出售股票。你在任何时候最多只能持有一股股票。你也可以先购买，然后在同一天出售。

返回你能获得的最大利润 。

输入输出样例

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

题解

方法一

写出动态转移方程。第一个变量表示交易完后手里没有股票的最大利润，另一个变量表示交易完后手里持有一支股票的最大利润。

```
int maxProfit(vector<int>& prices) {
    int days = prices.size();
    int dp0 = 0, dp1 = -prices[0];
    for (int i = 1; i < days; ++i) {
        int newDp0 = max(dp0, dp1 + prices[i]);
        int newDp1 = max(dp1, dp0 - prices[i]);
        dp0 = newDp0;
        dp1 = newDp1;
    }
    return dp0;
}
```

方法二

将数组分成数个利润大于 0 的区间，利润即是这些区间的价值和。亏损可以视作利润为 0 而舍弃。

```
int maxProfit(vector<int>& prices) {
    int ans = 0;
    int days = prices.size();
    for (int i = 1; i < days; ++i) {
        ans += max(0, prices[i] - prices[i - 1]);
    }
    return ans;
}
```

第 3 章 双指针

前情提要

- ❑ 快慢指针
- ❑ 滑动窗口
- ❑ 搜索

633. Sum of Square Numbers (Easy)

题目描述

给定一个非负整数，你要判断是否存在两个整数，使得这两个数的平方和等于此非负整数。

输入输出样例

输入一个非负整数，输出布尔值。

```
Input: c = 5
Output: true
```

题解

```
bool judgeSquareSum(int c) {
    long left = 0, right = (long)sqrt(c);
    while (left <= right) {
        if (left * left + right * right == c) {
            return true;
        } else if (left * left + right * right < c) {
            left++;
        } else {
            right--;
        }
    }
    return false;
}
```

680. Valid Palindrome II (Easy)

题目描述

给定一个非空字符串，最多删除一个字符。判断是否能成为回文字符串。

输入输出样例

输入一个非负整数，输出布尔值。

```
Input: s = "aba"
Output: true
```

题解

朴素的暴力验证会超出时间限制，通过双指针可以解决问题。两个指针分别指向字符串的头和尾，如果指向的字符相同则两个同时向中间移动一位直至无法移动；如果不相同则需要删除一个字符，返回判断删去左边字符或是右边字符后剩余的字符串是否为回文串。

```
bool checkPalindrome(const string& s, int left, int right) {
    for (int i = left, j = right; i < j; ++i, --j) {
        if (s[i] != s[j]) {
            return false;
        }
    }
    return true;
}

bool validPalindrome(string s) {
    int left = 0, right = s.size() - 1;
    while (left < right) {
        if (s[left] == s[right]) {
            ++left;
            --right;
        } else {
            return checkPalindrome(s, left, right - 1) || checkPalindrome(s, left + 1, right);
        }
    }
    return true;
}
```

524. Longest Word in Dictionary through Deleting (Medium)

题目描述

给你一个字符串 s 和一个字符串数组 dictionary ，找出并返回 dictionary 中最长的字符串，该字符串可以通过删除 s 中的某些字符得到。

如果答案不止一个，返回长度最长且字母序最小的字符串。如果答案不存在，则返回空字符串。

输入输出样例

```
Input: s = "abpcplea", dictionary = ["ale","apple","monkey","plea"]
Output: "apple"
```

题解

t 为字典的字符串。初始化两个指针 i 和 j，分别指向 t 和 s 的初始位置。每次贪心地匹配，匹配成功则 i 和 j 同时右移，匹配 t 的下一个位置，匹配失败则 j 右移，i 不变，尝试用 s 的下一个字符匹配 t。

最终如果 i 移动到 t 的末尾，则说明 t 是 s 的子序列。

遍历 dictionary 中的字符串，并维护当前长度最长且字典序最小的字符串。

```
string findLongestWord(string s, vector<string>& dictionary) {
    string res;
    for (string t: dictionary) {
        int j = 0;
        for (int i = 0; i < s.size() && j < t.size(); i++) {
            if (t[j] == s[i]) {
                j++;
            }
        }
        if (j == t.size()) {
            if (t.size() > res.size()) {
                res = t;
            } else if (t.size() == res.size()) {
                res = t < res ? t : res;
            }
        }
    }
    return res;
}
```

第 4 章 二分查找

前情提要

- 对数时间

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

题目描述

给你一个可能存在重复元素值的数组，它原来是一个升序排列的数组，并进行了多次旋转。请你找出并返回数组中的最小元素。

输入输出样例

输入一个数组，输出其中的最小值。

```
Input: nums = [1,3,5]
Output: 1
```

题解

由于数组中可能存在重复元素，那么在原来的二分查找方法的基础上加上一个判断：如果中间端点对应的值与右端点对应的值相等，无论何种情况，右端点都是可忽略的。

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

540. Single Element in a Sorted Array (Medium)

题目描述

给你一个仅由整数组成的有序数组，其中每个元素都会出现两次，唯有一个数只会出现一次。请找出并返回只出现一次的那个数。

输入输出样例

```
Input: nums = [1,1,2,3,3,4,4,8,8]
Output: 2
```

题解

方法一

利用整个数组的奇偶性，顺便沾了异或的语法糖。

```
int singleNonDuplicate(vector<int>& nums) {
    int left = 0, right = nums.size() - 1;
    while (left < right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] == nums[mid ^ 1]) {
            left = mid + 1;
        } else {
            right = mid;
        }
    }
    return nums[left];
}
```

方法二

利用偶数下标查找，按位与是亮点。

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

第 5 章 排序算法

前情提要

- ❑ 快速排序
- ❑ 归并排序
- ❑ 桶排序

451. Sort Characters By Frequency (Medium)

题目描述

给定一个字符串 s，根据字符出现的频率对其进行降序排序。一个字符出现的频率是它出现在字符串中的次数。

返回已排序的字符串。如果有多个答案，返回其中任何一个。

输入输出样例

```
Input: s = "tree"
Output: "eert"
```

题解

方法一

日常哈希表。

```
string frequencySort(string s) {
    unordered_map<char, int> hash_table;
    int length = s.length();
    for (const auto &ch : s) {
        ++hash_table[ch];
    }
    vector<pair<char, int>> vec;
    for (const auto &it : hash_table) {
        vec.emplace_back(it);
    }
    sort(vec.begin(), vec.end(), [](const pair<char, int> &a, const pair<char, int> &b) {
        return a.second > b.second;
    });
    string ret;
    for (const auto &[ch, num] : vec) {
        for (int i = 0; i < num; i++) {
            ret.push_back(ch);
        }
    }
    return ret;
```

```
}
```

方法二

桶排序。遍历字符串，统计每个字符出现的频率，同时记录最高频率 maxFreq；创建桶，存储从 1 到 maxFreq 的每个出现频率的字符；按照出现频率从大到小的顺序遍历桶，对于每个出现频率，获得对应的字符，然后将每个字符按照出现频率拼接到排序后的字符串。

```
string frequencySort(string s) {
    unordered_map<char, int> hash_table;
    int maxFreq = 0;
    int length = s.length();
    for (const auto &ch : s) {
        maxFreq = max(maxFreq, ++hash_table[ch]);
    }
    vector<string> buckets(maxFreq + 1);
    for (const auto &[ch, num] : hash_table) {
        buckets[num].push_back(ch);
    }
    string ret;
    for (int i = maxFreq; i > 0; i--) {
        string &bucket = buckets[i];
        for (const auto &ch : bucket) {
            for (int k = 0; k < i; k++) {
                ret.push_back(ch);
            }
        }
    }
    return ret;
}
```

75. Sort Colors (Medium)

题目描述

给定一个包含红色、白色和蓝色、共 n 个元素的数组 nums ，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

必须在不使用库的 sort 函数的情况下解决这个问题。

输入输出样例

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

题解

方法一

单指针，指哪打哪。

```
void sortColors(vector<int>& nums) {
    int n = nums.size();
    int ptr = 0;
    for (int i = 0; i < n; ++i) {
        if (nums[i] == 0) {
            swap(nums[i], nums[ptr]);
            ++ptr;
        }
    }
    for (int i = ptr; i < n; ++i) {
        if (nums[i] == 1) {
            swap(nums[i], nums[ptr]);
            ++ptr;
        }
    }
}
```

方法二

双指针，并不复杂。

```
void sortColors(vector<int>& nums) {
    int n = nums.size();
    int p0 = 0, p1 = 0;
    for (int i = 0; i < n; ++i) {
        if (nums[i] == 1) {
            swap(nums[i], nums[p1]);
            ++p1;
        } else if (nums[i] == 0) {
            swap(nums[i], nums[p0]);
            if (p0 < p1) {
                swap(nums[i], nums[p1]);
            }
            ++p0;
            ++p1;
        }
    }
}
```

第 6 章 搜索算法

前情提要

- ❑ 深度优先搜索
- ❑ 广度优先搜索
- ❑ 回溯法

130. Surrounded Regions (Medium)

题目描述

给你一个 m x n 的矩阵 board ，由若干字符'X' 和'O' ，找到所有被'X' 围绕的区域，并将这些区域里所有的'O' 用'X' 填充。

输入输出样例

```
Input: board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]
Output: [["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]
```

题解

方法一

深度优先搜索。所有的不被包围的 O 都直接或间接与边界上的 O 相连，通过标记这些 O 可以区分。

```
int n, m;

void dfs(vector<vector<char>>& board, int x, int y) {
    if (x < 0 || x >= n || y < 0 || y >= m || board[x][y] != 'O') {
        return;
    }
    board[x][y] = 'A';
    dfs(board, x + 1, y);
    dfs(board, x - 1, y);
    dfs(board, x, y + 1);
    dfs(board, x, y - 1);
}

void solve(vector<vector<char>>& board) {
    n = board.size();
    if (n == 0) {
        return;
    }
    m = board[0].size();
    for (int i = 0; i < n; i++) {
        dfs(board, i, 0);
```

```
        dfs(board, i, m - 1);
    }
    for (int i = 1; i < m - 1; i++) {
        dfs(board, 0, i);
        dfs(board, n - 1, i);
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (board[i][j] == 'A') {
                board[i][j] = 'O';
            } else if (board[i][j] == 'O') {
                board[i][j] = 'X';
            }
        }
    }
}
```

方法二

广度优先搜索。

```
const int dx[4] = {1, -1, 0, 0};
const int dy[4] = {0, 0, 1, -1};

void solve(vector<vector<char>>& board) {
    int n = board.size();
    if (n == 0) {
        return;
    }
    int m = board[0].size();
    queue<pair<int, int>> que;
    for (int i = 0; i < n; i++) {
        if (board[i][0] == 'O') {
            que.emplace(i, 0);
            board[i][0] = 'A';
        }
        if (board[i][m - 1] == 'O') {
            que.emplace(i, m - 1);
            board[i][m - 1] = 'A';
        }
    }
    for (int i = 1; i < m - 1; i++) {
        if (board[0][i] == 'O') {
            que.emplace(0, i);
            board[0][i] = 'A';
        }
        if (board[n - 1][i] == 'O') {
            que.emplace(n - 1, i);
            board[n - 1][i] = 'A';
```

```
        }
    }
    while (!que.empty()) {
        int x = que.front().first, y = que.front().second;
        que.pop();
        for (int i = 0; i < 4; i++) {
            int mx = x + dx[i], my = y + dy[i];
            if (mx < 0 || my < 0 || mx >= n || my >= m || board[mx][my] != 'O') {
                continue;
            }
            que.emplace(mx, my);
            board[mx][my] = 'A';
        }
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            if (board[i][j] == 'A') {
                board[i][j] = 'O';
            } else if (board[i][j] == 'O') {
                board[i][j] = 'X';
            }
        }
    }
}
```

257. Binary Tree Paths (Easy)

题目描述

给你一个二叉树的根节点 root ，按任意顺序，返回所有从根节点到叶子节点的路径。

输入输出样例

```
Input: root = [1,2,3,null,5]
Output: ["1->2->5","1->3"]
```

题解

方法一

深度优先搜索。如果当前节点不是叶子节点，则在当前的路径末尾添加该节点，并继续递归遍历该节点的每一个孩子节点。如果当前节点是叶子节点，则在当前路径末尾添加该节点后我们就得到了一条从根节点到叶子节点的路径，将该路径加入到答案即可。

```
void construct_paths(TreeNode* root, string path, vector<string>& paths) {
    if (root != nullptr) {
        path += to_string(root->val);
        if (root->left == nullptr && root->right == nullptr) {
```

```
            paths.push_back(path);
        } else {
            path += "->";
            construct_paths(root->left, path, paths);
            construct_paths(root->right, path, paths);
        }
    }
}

vector<string> binaryTreePaths(TreeNode* root) {
    vector<string> paths;
    construct_paths(root, "", paths);
    return paths;
}
```

方法二

广度优先搜索。维护一个队列，存储节点以及根到该节点的路径。一开始这个队列里只有根节点。在每一步迭代中，我们取出队列中的首节点，如果它是叶子节点，则将它对应的路径加入到答案中。如果它不是叶子节点，则将它的所有孩子节点加入到队列的末尾。当队列为空时广度优先搜索结束。

```
vector<string> binaryTreePaths(TreeNode* root) {
    vector<string> paths;
    if (root == nullptr) {
        return paths;
    }
    queue<TreeNode*> node_queue;
    queue<string> path_queue;

    node_queue.push(root);
    path_queue.push(to_string(root->val));

    while (!node_queue.empty()) {
        TreeNode* node = node_queue.front();
        string path = path_queue.front();
        node_queue.pop();
        path_queue.pop();

        if (node->left == nullptr && node->right == nullptr) {
            paths.push_back(path);
        } else {
            if (node->left != nullptr) {
                node_queue.push(node->left);
                path_queue.push(path + "->" + to_string(node->left->val));
            }

            if (node->right != nullptr) {
                node_queue.push(node->right);
                path_queue.push(path + "->" + to_string(node->right->val));
```

```
            }
        }
    }
    return paths;
}
```

第 7 章 动态规划

前情提要

- ❑ 一维二维基本动态规划
- ❑ 分割问题
- ❑ 子序列问题
- ❑ 背包问题
- ❑ 字符串
- ❑ 股票交易

213. House Robber II (Medium)

题目描述

你是一个专业的小偷，计划偷窃沿街的房屋，每间房内都藏有一定的现金。这个地方所有的房屋都围成一圈，这意味着第一个房屋和最后一个房屋是紧挨着的。同时，相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下，今晚能够偷窃到的最高金额。

输入输出样例

输入一个其值为金额的数组，输出一个正整数。

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

题解

第一个房屋和最后一个房屋不可兼得。可将把环拆成两个数组，一个是从 0 到 n - 1，另一个是从 1 到 n，然后返回两个结果最大的。注意房屋数为 1 或 2 时需要额外判断。

```
int robRange(vector<int>& nums, int start, int end) {
    int first = nums[start], second = max(nums[start], nums[start + 1]);
    for (int i = start + 2; i <= end; ++i) {
        int temp = second;
        second = max(first + nums[i], second);
        first = temp;
    }
    return second;
}

int rob(vector<int>& nums) {
    int n = nums.size();
    if (n == 1) {
        return nums[0];
    } else if (n == 2) {
        return max(nums[0], nums[1]);
    }
```

```
    return max(robRange(nums, 0, n - 2), robRange(nums, 1, n - 1));
}
```

53. Maximum Subarray (Easy)

题目描述

给你一个整数数组，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。子数组是数组中的一个连续部分。

输入输出样例

输入一个数组，输出整数。

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

题解

暴力日常超时，此时动态规划是一个不错的思路。理解了状态转移方程就能写出程序。

```
int maxSubArray(vector<int>& nums) {
    int prev = 0, maxSum = nums[0];
    for (auto num : nums) {
        prev = max(prev + num, num);
        maxSum = max(prev, maxSum);
    }
    return maxSum;
}
```

第 8 章 分治法

前情提要

- ❑ 表达式问题
- ❑ 主定理

932. Beautiful Array (Medium)

题目描述

对于某些固定的 N，如果数组 A 是整数 1, 2, ..., N 组成的排列，使得：对于每个 i < j，都不存在 k 满足 i < k < j 使得 A[k] * 2 = A[i] + A[j]。那么数组 A 是漂亮数组。

给定 N，返回任意漂亮数组 A（保证存在一个）。

输入输出样例

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

题解

将数组分成两部分 left 和 right，分别求出一个漂亮的数组，然后将它们进行仿射变换，使得不存在满足条件的三元组。一个简单的办法就是让 left 部分的数都是奇数，right 部分的数都是偶数。

因此我们将所有的奇数放在 left 部分，所有的偶数放在 right 部分，这样可以保证等式恒不成立。

对于 left 部分，我们进行 k = 1/2, b = 1/2 的仿射变换，把这些奇数一一映射到不超过 (N + 1) / 2 的整数。对于 right 部分，我们进行 k = 1/2, b = 0 的仿射变换，把这些偶数一一映射到不超过 N / 2 的整数。

```
unordered_map<int, vector<int>> memo;

vector<int> beautifulArray(int n) {
    if (n == 1) {
        return {1};
    }
    if (memo.count(n) != 0) {
        return memo[n];
    }
    vector<int> ans;
    int mid = (n + 1) / 2;
    vector<int> leftArray = beautifulArray(mid);
    vector<int> rightArray = beautifulArray(n - mid);
    for (auto &val : leftArray) {
        ans.push_back(val * 2 - 1);
    }
    for (auto & val : rightArray) {
        ans.push_back(val * 2);
    }
```

```
    memo[n] = ans;
    return ans;
}
```

312. Burst Balloons (Hard)

题目描述

有 n 个气球，编号为 0 到 n - 1，每个气球上都标有一个数字，这些数字存在数组 nums 中。

现在要求你戳破所有的气球。戳破第 i 个气球，你可以获得 nums[i - 1] * nums[i] * nums[i + 1] 枚硬币。 这里的 i - 1 和 i + 1 代表和 i 相邻的两个气球的序号。如果 i - 1 或 i + 1 超出了数组的边界，那么就当它是一个数字为 1 的气球。

求所能获得硬币的最大数量。

输入输出样例

```
Input: nums = [3,1,5,8]
Output: 167
```

题解

方法一

戳破气球可能比较难想，但是我们可以反过来，全过程看作是每次添加一个气球。

```
vector<vector<int>> rec;
vector<int> val;

int solve(int left, int right) {
    if (left >= right - 1) {
        return 0;
    }
    if (rec[left][right] != -1) {
        return rec[left][right];
    }
    for (int i = left + 1; i < right; i++) {
        int sum = val[left] * val[i] * val[right];
        sum += solve(left, i) + solve(i, right);
        rec[left][right] = max(rec[left][right], sum);
    }
    return rec[left][right];
}

int maxCoins(vector<int>& nums) {
    int n = nums.size();
    val.resize(n + 2);
    for (int i = 1; i <= n; i++) {
        val[i] = nums[i - 1];
```

```
    }
    val[0] = val[n + 1] = 1;
    rec.resize(n + 2, vector<int>(n + 2, -1));
    return solve(0, n + 1);
}
```

方法二

同上，只不过整合了一下代码成为动态规划。

```
int maxCoins(vector<int>& nums) {
    int n = nums.size();
    vector<vector<int>> rec(n + 2, vector<int>(n + 2));
    vector<int> val(n + 2);
    val[0] = val[n + 1] = 1;
    for (int i = 1; i <= n; i++) {
        val[i] = nums[i - 1];
    }
    for (int i = n - 1; i >= 0; i--) {
        for (int j = i + 2; j <= n + 1; j++) {
            for (int k = i + 1; k < j; k++) {
                int sum = val[i] * val[k] * val[j];
                sum += rec[i][k] + rec[k][j];
                rec[i][j] = max(rec[i][j], sum);
            }
        }
    }
    return rec[0][n + 1];
}
```

第 9 章 数学问题

前情提要

- ❑ 公倍数与公因数
- ❑ 质数
- ❑ 数字处理
- ❑ 随机与取样

168. Excel Sheet Column Title (Easy)

题目描述

给你一个整数 columnNumber ，返回它在 Excel 表中相对应的列名称。

输入输出样例

```
Input: columnNumber = 2147483647
Output: "FXSHRXW"
```

题解

```
string convertToTitle(int columnNumber) {
    string ans;
    while (columnNumber > 0) {
        --columnNumber;
        ans += columnNumber % 26 + 'A';
        columnNumber /= 26;
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

67. Add Binary (Easy)

题目描述

给你两个二进制字符串，返回它们的和（用二进制表示）。输入为非空字符串且只包含数字 1 和 0。

输入输出样例

```
Input: a = "11", b = "1"
Output: "100"
```

题解

方法一

一开始朴素的想法是：先将 a 和 b 转化成十进制数，求和后再转化为二进制数。然而这种方法只在特定不限制整数长度的语言（比如 Python ）中才有效。

不如换种思路，模拟十进制“逢十进一”，二进制字符串可以采用“逢二进一”的方式。小学时学的“列竖式”还记得吗？末尾对齐，逐位相加。

```
string addBinary(string a, string b) {
    string ans;
    reverse(a.begin(), a.end());
    reverse(b.begin(), b.end());

    int n = max(a.size(), b.size()), carry = 0;
    for (size_t i = 0; i < n; ++i) {
        carry += i < a.size() ? (a.at(i) == '1') : 0;
        carry += i < b.size() ? (b.at(i) == '1') : 0;
        ans.push_back((carry % 2) ? '1' : '0');
        carry /= 2;
    }

    if (carry) {
        ans.push_back('1');
    }
    reverse(ans.begin(), ans.end());

    return ans;
}
```

方法二

位运算。（程序仅限思路展示，实际中会超过整数限制范围）

```
string addBinary(string a, string b) {
    int x = binary_to_decimal_int(a), y = binary_to_decimal_int(b);
    while (y) {
        int ans = x ^ y;
        int carry = (x & y) << 1;
        x = ans;
        y = carry;
    }
    return decimal_to_binary_str(x);
}

int binary_to_decimal_int(string s) {
    int ans = 0;
    for (int i = s.size() - 1, j = 1; i >= 0; --i, j *= 2) {
        ans += (s[i] - '0') * j;
    }
```

```
    return ans;
}

string decimal_to_binary_str(int x) {
    string str = "";
    do {
        str += to_string(x % 2);
        x /= 2;
    } while (x);
    reverse(str.begin(), str.end());
    return str;
}
```

238. Product of Array Except Self (Medium)

题目描述

给你一个整数数组 nums，返回数组 answer ，其中 answer[i] 等于 nums 中除 nums[i] 之外其余各元素的乘积 。题目数据保证数组 nums 之中任意元素的全部前缀元素和后缀的乘积都在 32 位整数范围内。

请不要使用除法，且在 O(n) 时间复杂度内完成此题。

输入输出样例

```
Input: nums = [1,2,3,4]
Output: [24,12,8,6]
```

题解

方法一

利用索引左侧所有数字的乘积和右侧所有数字的乘积（即前缀与后缀）相乘得到答案。

```
vector<int> productExceptSelf(vector<int>& nums) {
    int n = nums.size();
    vector<int> left(n, 0), right(n, 0);
    vector<int> ans(n);
    left[0] = 1;
    right[n - 1] = 1;
    for (int i = 1; i < n; i++) {
        left[i] = nums[i - 1] * left[i - 1];
    }
    for (int i = n - 2; i >= 0; i--) {
        right[i] = nums[i + 1] * right[i + 1];
    }
    for (int i = 0; i < n; i++) {
        ans[i] = left[i] * right[i];
    }
    return ans;
```

```
}
```

方法二

动态构造右侧数组优化方法一。

```
vector<int> productExceptSelf(vector<int>& nums) {
    int n = nums.size();
    vector<int> ans(n);
    ans[0] = 1;
    for (int i = 1; i < n; i++) {
        ans[i] = nums[i - 1] * ans[i - 1];
    }
    int right = 1;
    for (int i = n - 1; i >= 0; i--) {
        ans[i] = ans[i] * right;
        right *= nums[i];
    }
    return ans;
}
```

第 10 章 位运算

前情提要

- ❑ 二进制特性

268. Missing Number (Easy)

题目描述

给定一个包含 [0, n] 中 n 个数的数组 nums ， 找出 [0, n] 这个范围内没有出现在数组中的那个数。

输入输出样例

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

题解

方法一

```
int missingNumber(vector<int>& nums) {
    int n = nums.size();
    int total = n * (n + 1) / 2;
    int arraySum = 0;
    for (int i = 0; i < n; i++) {
        arraySum += nums[i];
    }
    return total - arraySum;
}
```

方法二

```
int missingNumber(vector<int>& nums) {
    int res = 0;
    int n = nums.size();
    for (int i = 0; i < n; i++) {
        res ^= nums[i];
    }
    for (int i = 0; i <= n; i++) {
        res ^= i;
    }
    return res;
}
```

693. Binary Number with Alternating Bits (Easy)

题目描述

给定一个正整数，检查它的二进制表示是否总是 0、1 交替出现：换句话说，就是二进制表示中相邻两位的数字永不相同。

输入输出样例

```
Input: n = 5
Output: true
```

题解

方法一

```
bool hasAlternatingBits(int n) {
    int prev = 2;
    while (n != 0) {
        int cur = n % 2;
        if (cur == prev) {
            return false;
        }
        prev = cur;
        n /= 2;
    }
    return true;
}
```

方法二

```
bool hasAlternatingBits(int n) {
    long a = n ^ (n >> 1);
    return (a & (a + 1)) == 0;
}
```

476. Number Complement (Easy)

题目描述

对整数的二进制表示取反（0 变 1 ，1 变 0）后，再转换为十进制表示，可以得到这个整数的补数。

输入输出样例

给你一个整数 num ，输出它的补数。

```
Input: num = 5
Output: 2
```

题解

```
int findComplement(int num) {
    int highbit = 0;
    for (int i = 1; i <= 30; ++i) {
        if (num >= (1 << i)) {
            highbit = i;
        }
        else {
            break;
        }
    }
    int mask = (highbit == 30 ? 0x7fffffff : (1 << (highbit + 1)) - 1);
    return num ^ mask;
}
```

260. Single Number III (Medium)

题目描述

给定一个整数数组 nums，其中恰好有两个元素只出现一次，其余所有元素均出现两次。找出只出现一次的那两个元素。你可以按任意顺序返回答案。

输入输出样例

```
Input: nums = [1,2,1,3,2,5]
Output: [3, 5]
```

题解

方法一

日常哈希表，不过空间复杂度为 O(n) 。

```
vector<int> singleNumber(vector<int>& nums) {
    unordered_map<int, int> freq;
    for (int num: nums) {
        ++freq[num];
    }
    vector<int> ans;
    for (const auto& [num, occ]: freq) {
        if (occ == 1) {
            ans.push_back(num);
```

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

方法二

数组经过异或运算后得到两个只出现一次的数字的异或和 x 。使用位运算 x & -x 取出 x 的二进制表示中最低位那个 1，设其为第 l 位，所以两个只出现一次的数字的第 l 位不同。

```
vector<int> singleNumber(vector<int>& nums) {
    int xorsum = 0;
    for (int num: nums) {
        xorsum ^= num;
    }
    int lsb = (xorsum == INT_MIN ? xorsum : xorsum & (-xorsum));
    int type1 = 0, type2 = 0;
    for (int num: nums) {
        if (num & lsb) {
            type1 ^= num;
        }
        else {
            type2 ^= num;
        }
    }
    return {type1, type2};
}
```

第 11 章 数据结构

前情提要

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

566. Reshape the Matrix (Easy)

题目描述

给你一个由二维数组表示的 m x n 矩阵，以及两个正整数 r 和 c ，分别表示想要的重构的矩阵的行数和列数。

重构后的矩阵需要将原始矩阵的所有元素以相同的行遍历顺序填充。

如果具有给定参数的 reshape 操作是可行且合理的，则输出新的重塑矩阵；否则，输出原始矩阵。

输入输出样例

```
Input: mat = [[1,2],[3,4]], r = 1, c = 4
Output: [[1,2,3,4]]
```

题解

```
vector<vector<int>> matrixReshape(vector<vector<int>>& mat, int r, int c) {
    int m = mat.size(), n = mat[0].size();
    if (m * n != r * c) {
        return mat;
    }
    vector<vector<int>> reshapedMat(r, vector<int>(c));
    for (int i = 0; i < r * c; ++i) {
        reshapedMat[i / c][i % c] = mat[i / n][i % n];
    }
    return reshapedMat;
}
```

225. Implement Stack using Queues (Easy)

题目描述

请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通栈的全部四种操作（push、top、pop 和 empty）。

输入输出样例

```
Input: ["MyStack", "push", "push", "top", "pop", "empty"]
[[], [1], [2], [], [], []]
Output: [null, null, null, 2, 2, false]
```

题解

方法一

使用两个队列实现栈的操作，其中第一个队列用于存储栈内的元素，第二个队列作为入栈操作的辅助队列。

```
queue<int> mainQueue;
queue<int> supQueue;

MyStack() {

}

void push(int x) {
    supQueue.push(x);
    while (!mainQueue.empty()) {
        supQueue.push(mainQueue.front());
        mainQueue.pop();
    }
    swap(mainQueue, supQueue);
}

int pop() {
    int top = mainQueue.front();
    mainQueue.pop();
    return top;
}

int top() {
    int top = mainQueue.front();
    return top;
}

bool empty() {
    return mainQueue.empty();
}
```

方法二

也可以使用一个队列实现栈的操作。

```
queue<int> stackQueue;
```

```
MyStack() {

}

void push(int x) {
    int n = stackQueue.size();
    stackQueue.push(x);
    for (int i = 0; i < n; i++) {
        stackQueue.push(stackQueue.front());
        stackQueue.pop();
    }
}

int pop() {
    int top = stackQueue.front();
    stackQueue.pop();
    return top;
}

int top() {
    int top = stackQueue.front();
    return top;
}

bool empty() {
    return stackQueue.empty();
}
```

503. Next Greater Element II (Medium)

题目描述

给定一个循环数组 nums （ nums[nums.length - 1] 的下一个元素是 nums[0] ），返回 nums 中每个元素的下一个更大元素。

数字 x 的下一个更大的元素是按数组遍历顺序，这个数字之后的第一个比它更大的数，这意味着你应该循环地搜索它的下一个更大的数。如果不存在，则输出 -1 。

输入输出样例

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

题解

寻找下一个最大/最小的问题，第一直觉应该想到单调栈。如果没有想到，那说明你缺乏 “统栈思维”。单调栈中保存的是下标，从栈底到栈顶的下标在数组 nums 中对应的值是单调不升的。

注意，光遍历一遍数组是不够的，因为遍历一次之后不知道最后一个元素的下一个更大元素。

```
vector<int> nextGreaterElements(vector<int>& nums) {
    int n = nums.size();
    vector<int> ret(n, -1);
    stack<int> stk;
    for (int i = 0; i < n * 2 - 1; i++) {
        while (!stk.empty() && nums[stk.top()] < nums[i % n]) {
            ret[stk.top()] = nums[i % n];
            stk.pop();
        }
        stk.push(i % n);
    }
    return ret;
}
```

217. Contains Duplicate (Easy)

题目描述

给你一个整数数组 nums 。如果任一值在数组中出现至少两次，返回 true ；如果数组中每个元素互不相同，返回 false 。

输入输出样例

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

题解

方法一

排序。相同元素一定在相邻位置，可以节省空间。（小伙子你不讲武德）

```
bool containsDuplicate(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    int n = nums.size();
    for (int i = 0; i < n - 1; i++) {
        if (nums[i] == nums[i + 1]) {
            return true;
        }
    }
    return false;
}
```

方法二

日常哈希表。提示词：重复。

```cpp
bool containsDuplicate(vector<int>& nums) {
    unordered_set<int> hash_table;
    for (const int &num : nums) {
        if (hash_table.find(num) != hash_table.end()) {
            return true;
        }
        hash_table.insert(num);
    }
    return false;
}
```

第 12 章 字符串

前情提要

- ❑ 字符串比较
- ❑ 字符串匹配

409. Longest Palindrome (Easy)

题目描述

给定一个包含大写字母和小写字母的字符串，返回通过这些字母构造成的最长的回文串。在构造过程中，请注意区分大小写。

输入输出样例

```
Input: s = "abccccdd"
Output: 7
```

题解

```
int longestPalindrome(string s) {
    unordered_map<char, int> counter;
    for (char c : s) {
        counter[c]++;
    }
    int res = 0, odd = 0;
    for (auto kv : counter) {
        int count = kv.second;
        int rem = count % 2;
        res += count - rem;
        if (rem == 1) odd = 1;
    }
    return res + odd;
}
```

3. Longest Substring Without Repeating Characters (Medium)

题目描述

给定一个字符串，请你找出其中不含有重复字符的最长子串的长度。

输入输出样例

```
Input: s = "abcabcbb"
Output: 3
```

题解

解题思路是使用滑动窗口的方法。当左指针向右移动一格，可以不断地向右移动右指针，但需要保证这两个指针对应的子串中没有重复的字符。在移动结束后，这个子串就对应着以左指针开始的，不包含重复字符的最长子串。找到的最长的子串的长度即为答案。

```
int lengthOfLongestSubstring(string s) {
    unordered_set<char> hash_table;
    int n = s.size();
    int right_key = -1, ans = 0;
    for (int i = 0; i < n; ++i) {
        if (i != 0) {
            hash_table.erase(s[i - 1]);
        }
        while (right_key + 1 < n && !hash_table.count(s[right_key + 1])) {
            hash_table.insert(s[right_key + 1]);
            ++right_key;
        }
        ans = max(ans, right_key - i + 1);
    }
    return ans;
}
```

第 13 章 链表

前情提要

- 链表操作

83. Remove Duplicates from Sorted List (Easy)

题目描述

给定一个已排序的链表的头，删除所有重复的元素，使每个元素只出现一次。返回已排序的链表。

输入输出样例

```
Input: head = [1,1,2]
Output: [1,2]
```

题解

注意记得回收内存。

```
ListNode* deleteDuplicates(ListNode* head) {
    if (!head) {
        return head;
    }
    ListNode* cur = head;
    while (cur->next) {
        if (cur->val == cur->next->val) {
            ListNode* del = cur->next;
            cur->next = cur->next->next;
            delete del;
        } else {
            cur = cur->next;
        }
    }
    return head;
}
```

328. Odd Even Linked List (Medium)

题目描述

给定单链表的头节点，将所有索引为奇数的节点和索引为偶数的节点分别组合在一起，然后返回重新排序的列表。第一个节点的索引被认为是奇数，第二个节点的索引为偶数，以此类推。偶数组和奇数组内部的相对顺序应该与输入时保持一致。

输入输出样例

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

题解

奇偶节点交错，所以可以设置奇偶两个链表，最后将奇链表的末尾连接偶链表的头节点。

```
ListNode* oddEvenList(ListNode* head) {
    if (head == nullptr) {
        return head;
    }
    ListNode* evenHead = head->next;
    ListNode* odd = head;
    ListNode* even = evenHead;
    while (even != nullptr && even->next != nullptr) {
        odd->next = even->next;
        odd = odd->next;
        even->next = odd->next;
        even = even->next;
    }
    odd->next = evenHead;
    return head;
}
```

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

题目描述

给你一个链表，删除链表的倒数第 n 个结点，并且返回链表的头结点。

输入输出样例

```
Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]
```

题解

方法一

朴素想法，先遍历一遍列表得到其长度，然后再进行一次遍历以删除节点。可以设置一个指向头节点的哑节点以免去对头节点的特殊判断。

```
ListNode* removeNthFromEnd(ListNode* head, int n) {
    ListNode* dummy = new ListNode(0, head);
    int length = 0;
    while (head) {
```

```
        ++length;
        head = head->next;
    }
    ListNode* cur = dummy;
    for (int i = 1; i < length - n + 1; ++i) {
        cur = cur->next;
    }
    cur->next = cur->next->next;
    ListNode* ans = dummy->next;
    delete dummy;
    return ans;
}
```

方法二

双指针可以帮助在同时间定位倒数第 n 个结点。快节点比慢节点超前 n 个节点，当快节点到达链表末尾时，慢节点正好处于倒数第 n 个结点。可以利用哑节点更好地删除节点。

```
ListNode* removeNthFromEnd(ListNode* head, int n) {
    ListNode* dummy = new ListNode(0, head);
    ListNode* fast = head;
    ListNode* slow = dummy;
    for (int i = 0; i < n; ++i) {
        fast = fast->next;
    }
    while (fast) {
        fast = fast->next;
        slow = slow->next;
    }
    slow->next = slow->next->next;
    ListNode* ans = dummy->next;
    delete dummy;
    return ans;
}
```

148. Sort List (Medium)

题目描述

给你链表的头结点，请将其按升序排列并返回排序后的链表。

输入输出样例

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

题解

方法一

利用快慢指针找到链表中点后，可以对链表进行归并排序。程序可通过递归实现。递归的终止条件是链表的节点个数小于或等于 1，即当链表为空或者链表只包含 1 个节点时，不需要对链表进行拆分和排序。

```
ListNode* sortList(ListNode* head) {
    if (head == nullptr || head->next == nullptr) {
        return head;
    }
    ListNode* fast = head, *slow = head;
    while (fast->next && fast->next->next) {
        fast = fast->next->next;
        slow = slow->next;
    }
    fast = slow->next;
    slow->next = nullptr;
    return merge(sortList(head), sortList(fast));
}

ListNode* merge(ListNode* list1, ListNode* list2) {
    ListNode* dummy = new ListNode(0);
    ListNode* temp = dummy;
    while (list1 != nullptr && list2 != nullptr) {
        if (list1->val <= list2->val) {
            temp->next = list1;
            list1 = list1->next;
        } else {
            temp->next = list2;
            list2 = list2->next;
        }
        temp = temp->next;
    }
    temp->next = list1 ? list1 : list2;
    return dummy->next;
}
```

方法二

也可以采用自底向上归并排序的方法。每次将链表拆分成若干个长度为 subLength 的子链表（最后一个子链表的长度可以小于 subLength），按照每两个子链表一组进行合并。最终得到一个有序的链表。

```
ListNode* sortList(ListNode* head) {
    if (head == nullptr) {
        return head;
    }
    int length = 0;
    ListNode* node = head;
    while (node != nullptr) {
```

```
        ++length;
        node = node->next;
    }
    ListNode* dummy = new ListNode(0, head);
    for (int subLength = 1; subLength < length; subLength <<= 1) {
        ListNode* prev = dummy, *cur = dummy->next;
        while (cur != nullptr) {
            ListNode* head1 = cur;
            for (int i = 1; i < subLength && cur->next != nullptr; ++i) {
                cur = cur->next;
            }
            ListNode* head2 = cur->next;
            cur->next = nullptr;
            cur = head2;
            for (int i = 1; i < subLength && cur != nullptr && cur->next != nullptr; ++i) {
                cur = cur->next;
            }
            ListNode* next = nullptr;
            if (cur != nullptr) {
                next = cur->next;
                cur->next = nullptr;
            }
            ListNode* merged = merge(head1, head2);
            prev->next = merged;
            while (prev->next != nullptr) {
                prev = prev->next;
            }
            cur = next;
        }
    }
    return dummy->next;
}

ListNode* merge(ListNode* list1, ListNode* list2) {
    ListNode* dummy = new ListNode(0);
    ListNode* temp = dummy;
    while (list1 != nullptr && list2 != nullptr) {
        if (list1->val <= list2->val) {
            temp->next = list1;
            list1 = list1->next;
        } else {
            temp->next = list2;
            list2 = list2->next;
        }
        temp = temp->next;
    }
    temp->next = list1 ? list1 : list2;
    return dummy->next;
}
```

第 14 章 树

前情提要

- 树的递归
- 层次遍历
- 前中后序遍历
- 二叉查找树
- 字典树

226. Invert Binary Tree (Easy)

题目描述

给你一棵二叉树的根节点 root ，翻转这棵二叉树，并返回其根节点。

输入输出样例

```
Input: root = [4,2,7,1,3,6,9]
Output: [4,7,2,9,6,3,1]
```

题解

```
TreeNode* invertTree(TreeNode* root) {
    if (root == nullptr) {
        return nullptr;
    }
    TreeNode* left = invertTree(root->left);
    TreeNode* right = invertTree(root->right);
    root->left = right;
    root->right = left;
    return root;
}
```

617. Merge Two Binary Trees (Easy)

题目描述

给你两棵二叉树：root1 和 root2 。

想象一下，当你将其中一棵覆盖到另一棵之上时，两棵树上的一些节点将会重叠（而另一些不会）。你需要将这两棵树合并成一棵新二叉树。合并的规则是：如果两个节点重叠，那么将这两个节点的值相加作为合并后节点的新值；否则，不为 null 的节点将直接作为新二叉树的节点。返回合并后的二叉树。

注意: 合并过程必须从两个树的根节点开始。

输入输出样例

```
Input: root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]
Output: [3,4,5,5,4,null,7]
```

题解

方法一

深度优先搜索。日常递归。

```
TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
    if (root1 == nullptr) {
        return root2;
    }
    if (root2 == nullptr) {
        return root1;
    }
    auto merged = new TreeNode(root1->val + root2->val);
    merged->left = mergeTrees(root1->left, root2->left);
    merged->right = mergeTrees(root1->right, root2->right);
    return merged;
}
```

方法二

广度优先搜索。使用三个队列分别存储合并后的二叉树的节点以及两个原始二叉树的节点。初始时将每个二叉树的根节点分别加入相应的队列。每次从每个队列中取出一个节点，判断两个原始二叉树的节点的左右子节点是否为空。如果两个原始二叉树的当前节点中至少有一个节点的左子节点不为空，则合并后的二叉树的对应节点的左子节点也不为空。对于右子节点同理。

如果两个原始二叉树的左子节点都不为空，则合并后的二叉树的左子节点的值为两个原始二叉树的左子节点的值之和，在创建合并后的二叉树的左子节点之后，将每个二叉树中的左子节点都加入相应的队列；如果两个原始二叉树的左子节点有一个为空，即有一个原始二叉树的左子树为空，则合并后的二叉树的左子树即为另一个原始二叉树的左子树，此时也不需要对非空左子树继续遍历，因此不需要将左子节点加入队列。对于右子节点和右子树，处理方法与左子节点和左子树相同。

步骤很长，请慢慢看。

```
TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
    if (root1 == nullptr) {
        return root2;
    }
    if (root2 == nullptr) {
        return root1;
    }
    auto merged = new TreeNode(root1->val + root2->val);
    auto q = queue<TreeNode*>();
    auto queue1 = queue<TreeNode*>();
    auto queue2 = queue<TreeNode*>();
```

```
    q.push(merged);
    queue1.push(root1);
    queue2.push(root2);
    while (!queue1.empty() && !queue2.empty()) {
        auto node = q.front(), node1 = queue1.front(), node2 = queue2.front();
        q.pop();
        queue1.pop();
        queue2.pop();
        auto left1 = node1->left, left2 = node2->left, right1 = node1->right, right2 = node2->right;
        if (left1 != nullptr || left2 != nullptr) {
            if (left1 != nullptr && left2 != nullptr) {
                auto left = new TreeNode(left1->val + left2->val);
                node->left = left;
                q.push(left);
                queue1.push(left1);
                queue2.push(left2);
            } else if (left1 != nullptr) {
                node->left = left1;
            } else if (left2 != nullptr) {
                node->left = left2;
            }
        }
        if (right1 != nullptr || right2 != nullptr) {
            if (right1 != nullptr && right2 != nullptr) {
                auto right = new TreeNode(right1->val + right2->val);
                node->right = right;
                q.push(right);
                queue1.push(right1);
                queue2.push(right2);
            } else if (right1 != nullptr) {
                node->right = right1;
            } else {
                node->right = right2;
            }
        }
    }
    return merged;
}
```

572. Subtree of Another Tree (Easy)

题目描述

给你两棵二叉树 root 和 subRoot 。检验 root 中是否包含和 subRoot 具有相同结构和节点值的子树。如果存在，返回 true ；否则，返回 false 。

二叉树 tree 的一棵子树包括 tree 的某个节点和这个节点的所有后代节点。tree 也可以看做它自身的一棵子树。

输入输出样例

```
Input: root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]
Output: false
```

题解

方法一

有一种暴力的美。

```
bool isSameTree(TreeNode *root1, TreeNode *root2) {
    if (!root1 && !root2) {
        return true;
    }
    if ((root1 && !root2) || (!root1 && root2) || (root1->val != root2->val)) {
        return false;
    }
    return isSameTree(root1->left, root2->left) && isSameTree(root1->right, root2->right);
}

bool isSubtree(TreeNode *root, TreeNode *subRoot) {
    if (!root && !subRoot) {
        return true;
    }
    if (!root && subRoot) {
        return false;
    }
    return isSameTree(root, subRoot) || isSubtree(root->left, subRoot) || isSubtree(root->right,
        subRoot);
}
```

方法二

```
vector <int> sOrder, tOrder;
int maxElement, lNull, rNull;

void getMaxElement(TreeNode *o) {
    if (!o) {
        return;
    }
    maxElement = max(maxElement, o->val);
    getMaxElement(o->left);
    getMaxElement(o->right);
}

void getDfsOrder(TreeNode *o, vector <int> &tar) {
    if (!o) {
```

```
        return;
    }
    tar.push_back(o->val);
    if (o->left) {
        getDfsOrder(o->left, tar);
    } else {
        tar.push_back(lNull);
    }
    if (o->right) {
        getDfsOrder(o->right, tar);
    } else {
        tar.push_back(rNull);
    }
}

bool kmp() {
    int sLen = sOrder.size(), tLen = tOrder.size();
    vector <int> fail(tOrder.size(), -1);
    for (int i = 1, j = -1; i < tLen; ++i) {
        while (j != -1 && tOrder[i] != tOrder[j + 1]) {
            j = fail[j];
        }
        if (tOrder[i] == tOrder[j + 1]) {
            ++j;
        }
        fail[i] = j;
    }
    for (int i = 0, j = -1; i < sLen; ++i) {
        while (j != -1 && sOrder[i] != tOrder[j + 1]) {
            j = fail[j];
        }
        if (sOrder[i] == tOrder[j + 1]) {
            ++j;
        }
        if (j == tLen - 1) {
            return true;
        }
    }
    return false;
}

bool isSubtree(TreeNode* s, TreeNode* t) {
    maxElement = INT_MIN;
    getMaxElement(s);
    getMaxElement(t);
    lNull = maxElement + 1;
    rNull = maxElement + 2;

    getDfsOrder(s, sOrder);
```

```
    getDfsOrder(t, tOrder);

    return kmp();
}
```

方法三

```
static constexpr int MAX_N = 1000 + 5;
static constexpr int MOD = int(1E9) + 7;

bool vis[MAX_N];
int p[MAX_N], tot;
void getPrime() {
    vis[0] = vis[1] = 1; tot = 0;
    for (int i = 2; i < MAX_N; ++i) {
        if (!vis[i]) p[++tot] = i;
        for (int j = 1; j <= tot && i * p[j] < MAX_N; ++j) {
            vis[i * p[j]] = 1;
            if (i % p[j] == 0) break;
        }
    }
}

struct Status {
    int f, s; // f 为哈希值 | s 为子树大小
    Status(int f_ = 0, int s_ = 0)
    : f(f_), s(s_) {}
};

unordered_map <TreeNode *, Status> hS, hT;

void dfs(TreeNode *o, unordered_map <TreeNode *, Status> &h) {
    h[o] = Status(o->val, 1);
    if (!o->left && !o->right) return;
    if (o->left) {
        dfs(o->left, h);
        h[o].s += h[o->left].s;
        h[o].f = (h[o].f + (31LL * h[o->left].f * p[h[o->left].s]) % MOD) % MOD;
    }
    if (o->right) {
        dfs(o->right, h);
        h[o].s += h[o->right].s;
        h[o].f = (h[o].f + (179LL * h[o->right].f * p[h[o->right].s]) % MOD) % MOD;
    }
}

bool isSubtree(TreeNode* s, TreeNode* t) {
    getPrime();
```

```
    dfs(s, hS);
    dfs(t, hT);

    int tHash = hT[t].f;
    for (const auto &[k, v]: hS) {
        if (v.f == tHash) {
            return true;
        }
    }

    return false;
}
```

404. Sum of Left Leaves (Easy)

题目描述

给定二叉树的根节点 root ，返回所有左叶子之和。

输入输出样例

```
Input: root = [3,9,20,null,null,15,7]
Output: 24
```

题解

方法一

伟大的递归之神，请聆听我虔诚的祈祷!

```
int sumOfLeftLeaves(TreeNode* root) {
    if (!root) {
        return 0;
    } else if (root->left && !root->left->left && !root->left->right) {
        return root->left->val + sumOfLeftLeaves(root->right);
    } else {
        return ans + sumOfLeftLeaves(root->left) + sumOfLeftLeaves(root->right);
    }
}
```

既然如此，为什么不用一行表示呢?

```
int sumOfLeftLeaves(TreeNode* root) {
    return root ? sumOfLeftLeaves(root->left) + sumOfLeftLeaves(root->right) + (root->left && !root->
        left->left && !root->left->right ? root->left->val : 0) : 0;
}
```

方法二

呜呜……人家广度优先搜索明明也……很优秀的。不要老是盯着姐姐……可以吗?

```
bool isLeafNode(TreeNode* node) {
    return !node->left && !node->right;
}

int sumOfLeftLeaves(TreeNode* root) {
    if (!root) {
        return 0;
    }

    queue<TreeNode*> q;
    q.push(root);
    int ans = 0;
    while (!q.empty()) {
        TreeNode* node = q.front();
        q.pop();
        if (node->left) {
            if (isLeafNode(node->left)) {
                ans += node->left->val;
            }
            else {
                q.push(node->left);
            }
        }
        if (node->right) {
            if (!isLeafNode(node->right)) {
                q.push(node->right);
            }
        }
    }
    return ans;
}
```

513. Find Bottom Left Tree Value (Easy)

题目描述

给定一个二叉树的根节点 root，请找出该二叉树的最底层最左边节点的值。
假设二叉树中至少有一个节点。

输入输出样例

```
Input: root = [1,2,3,4,null,5,6,null,null,7]
Output: 7
```

题解

方法一

多年以后，面对行刑队，奥雷里亚诺 · 布恩迪亚上校将会回想起刷 LeetCode 却没有用 DFS 的那个遥远的下午。

——加西亚 · 马尔克斯《百年孤独》

```
void dfs(TreeNode *root, int height, int &curVal, int &curHeight) {
    if (root == nullptr) {
        return;
    }
    height++;
    dfs(root->left, height, curVal, curHeight);
    dfs(root->right, height, curVal, curHeight);
    if (height > curHeight) {
        curHeight = height;
        curVal = root->val;
    }
}

int findBottomLeftValue(TreeNode* root) {
    int curVal, curHeight = 0;
    dfs(root, 0, curVal, curHeight);
    return curVal;
}
```

方法二

不可避免，BFS 总是让他想起刷 LeetCode 受阻后的命运。

——加西亚 · 马尔克斯《霍乱时期的爱情》

```
int findBottomLeftValue(TreeNode* root) {
    int ret;
    queue<TreeNode *> q;
    q.push(root);
    while (!q.empty()) {
        auto p = q.front();
        q.pop();
        if (p->right) {
            q.push(p->right);
        }
        if (p->left) {
            q.push(p->left);
        }
        ret = p->val;
    }
    return ret;
}
```

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

题目描述

给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。

输入输出样例

```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
```

题解

方法一

递归。

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (root == nullptr) {
        return nullptr;
    } else if (root->val > p->val && root->val > q->val) {
        return lowestCommonAncestor(root->left, p, q);
    } else if (root->val < p->val && root->val < q->val) {
        return lowestCommonAncestor(root->right, p, q);
    } else {
        return root;
    }
}
```

一行解决。

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    return (root->val - p->val) * (root->val - q->val) > 0 ? lowestCommonAncestor(root->val > p->val ?
        root->left : root->right, p, q) : root;
}
```

方法二

迭代。

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    TreeNode* ancestor = root;
    while (true) {
        if (p->val < ancestor->val && q->val < ancestor->val) {
            ancestor = ancestor->left;
        }
        else if (p->val > ancestor->val && q->val > ancestor->val) {
            ancestor = ancestor->right;
        }
        else {
```

```
            break;
        }
    }
    return ancestor;
}
```

第 15 章 图

前情提要

- 二分图
- 拓扑排序

第 16 章 更加复杂的数据结构

前情提要

- 并查集
- 复合数据结构