1. 两数之和

给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出 和为目标值 target 的那 两个整数，并返回它们的数组下标。

你可以假设每种输入只会对应一个答案，并且你不能使用两次相同的元素。

你可以按任意顺序返回答案

梦开始的地方/golang里面的map是map[KeyType]ValueType

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int,int> umap;
        for(int i{0};i<nums.size();i++){
            if(umap.find(target - nums[i]) != umap.end()){
                auto it = umap.find(target - nums[i]);
                return vector<int>{i,it->second};
            }
            else{
                umap[nums[i]] = i;
            }
        }
        return vector<int>{0,0};
    }
};
```

704. 二分查找

给定一个 n 个元素有序的（升序）整型数组 nums 和一个目标值 target ，写一个函数搜索 nums 中的 target，如果 target 存在返回下标，否则返回 -1。

你必须编写一个具有 O(log n) 时间复杂度的算法。

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int left=0;
        int right=0;
        int middle=0;
        right = size(nums)-1;
        while(left<=right) {
            middle =  left + (right -left) / 2 ;
            if (nums[middle] == target) {
                return middle;
            } else if (nums[middle] < target) {
                left = middle + 1;
            } else if (nums[middle] > target) {
                right = middle -1;
            } else {
                return -1;
            }
        }
        return -1;
    }
};
```

31. 下一个排列

整数数组的一个 排列 就是将其所有成员以序列或线性顺序排列。

例如，arr = [1,2,3] ，以下这些都可以视作 arr 的排列：[1,2,3]、[1,3,2]、[3,1,2]、[2,3,1] 。
整数数组的 下一个排列 是指其整数的下一个字典序更大的排列。更正式地，如果数组的所有排列根据其字典顺序从小到大排列在一个容器中，那么数组的 下一个排列 就是在这个有序容器中排在它后面的那个排列。如果不存在下一个更大的排列，那么这个数组必须重排为字典序最小的排列（即，其元素按升序排列）。

例如，arr = [1,2,3] 的下一个排列是 [1,3,2] 。
类似地，arr = [2,3,1] 的下一个排列是 [3,1,2] 。
而 arr = [3,2,1] 的下一个排列是 [1,2,3] ，因为 [3,2,1] 不存在一个字典序更大的排列。
给你一个整数数组 nums ，找出 nums 的下一个排列。

必须 原地 修改，只允许使用额外常数空间。

```cpp
class Solution {
public:
    void nextPermutation(vector<int>& nums) {
        int cur = nums.size()-2;
        while(cur>=0&&nums[cur]>=nums[cur+1]) cur--;
        if(cur<0) sort(nums.begin(),nums.end());
        else {
            int pos = nums.size()-1;
            while(pos>=0&&nums[pos]<=nums[cur]) pos--;
            swap(nums[pos],nums[cur]);
            reverse((nums.begin()+cur+1),nums.end());

        }
    }
};
```

146. LRU 缓存

```
struct Node {
    int key;
    int value;
    Node* next;
    Node* pre;
    Node() : key(0), value(0), next(nullptr), pre(nullptr) {};
    Node(int _key, int _value) : key(_key), value(_value), next(nullptr), pre(nullptr) {};
};

class LRUCache {
private:
    int capacity;
    int size;
    Node* head;
    Node* tail;
    unordered_map<int, Node*> umap;
public:
    LRUCache(int _capacity) : capacity(_capacity), size(0) {
        head = new Node();
        tail = new Node();
        head->next = tail;
        tail->pre = head;
    };
    void DeleteNode(Node* node) {
        node->pre->next = node->next;
        node->next->pre = node->pre;
    }
    void InsertHead(Node* node) {
        node->next = head->next;
        head->next->pre = node;
        node->pre = head;
        head->next = node;
    }
    int get(int key){ //如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1
        if(umap.find(key) == umap.end()) return -1; //没有找到返回-1
        Node* node = umap[key];
        DeleteNode(node);
        InsertHead(node);
        return node->value;
    }
    void put(int key, int value) { //如果关键字 key 已经存在，则变更其数据值 value ；如果不存在，则
        if(umap.find(key) != umap.end()) {
            // 找到了
```

```
            Node* node = umap[key];
            node->value = value;
            DeleteNode(node);
            InsertHead(node);
        } else {
            // 没有找到
            // 1.先插入
            Node* node = new Node(key, value);
            InsertHead(node);
            size++;
            umap[key] = node;
            // 判断是否找过了容量
            if(size > capacity) {
                Node* node = tail->pre;
                umap.erase(node->key);
                DeleteNode(node);
                size--;
                delete node;
                node = nullptr;
            }
        }
    }
};
```

739. 每日温度

给定一个整数数组 temperatures ，表示每天的温度，返回一个数组 answer ，其中 answer[i] 是指对于第 i 天，下一个更高温度出现在几天后。如果气温在这之后都不会升高，请在该位置用 0 来代替。

注意result数组的下标是st.top()/golang

```cpp
class Solution {
public:
    vector<int> dailyTemperatures(vector<int>& temperatures) {
        // 使用单调栈 从左到右维护这个栈
        // 存储的是下标 如果当前的元素比栈顶的要大 那么把栈顶的全部放出来 这个元素下标压进去
        // 貌似这个就是单调栈的东西了
        stack<int> sta;
        vector<int> result(temperatures.size(), 0);
        for(int i{0};i<temperatures.size();i++) {
            while(sta.empty() == false && temperatures[i] > temperatures[sta.top()]) {
                int preindex = sta.top();
                result[preindex] = i - preindex;
                sta.pop();
            }
            sta.push(i);
        }
        return result;
    }
};
```

215. 数组中的第K个最大元素

给定整数数组 nums 和整数 k，请返回数组中第 k 个最大的元素。

请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

你必须设计并实现时间复杂度为 O(n) 的算法解决此问题。

快排本质是分治-递归/golang函数参数永远是值传递，然而切片有可能会被函数体修改

```
class Solution {
public:
    // 快排复习一下
    int Quick(vector<int>& nums, int left, int right) {
        int base = nums[left];
        while(left < right) {
            while(left < right && nums[right] >= base) right--;
            nums[left] = nums[right];
            while(left < right && nums[left] <= base) left++;
            nums[right] = nums[left];
        }
        nums[left] = base;
        return left;
    }
    // 现在已经有一次实现的函数了 需要一个函数来实现分治-递归
    void QuickSort(vector<int>& nums, int left, int right, int y) {
        if(left > right) return ; //下标不合法
        if(left == right) return ;// 一个数字天然有序
        int base = Quick(nums,left,right);
        if(base == y) return;
        else if(base < y) QuickSort(nums,base + 1, right,y);
        else QuickSort(nums,left,base - 1,y);
    }
    int findKthLargest(vector<int>& nums, int k) {
        QuickSort(nums,0,nums.size() - 1, nums.size() - k); // 这里你当成及要找的数字就是y 然后去
        return nums[nums.size() -k];
    }
};
```

2. 两数相加

给你两个 非空 的链表，表示两个非负的整数。它们每位数字都是按照 逆序 的方式存储的，并且每个节点只能存储 一位 数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

写完加法器记得移动list1和list2链表！！

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        // 创建一个新的链表
        ListNode* dummy = new ListNode();
        ListNode* head = dummy;
        int temp{0};
        int sum{0};
        while(l1 != nullptr || l2 != nullptr || temp != 0) {
            int val1 = l1 == nullptr ? 0 : l1->val;
            int val2 = l2 != nullptr ? l2->val : 0;
            sum = val1 + val2 + temp;
            int low_num = 0;
            if(sum >= 10) {
                low_num = sum % 10;
                temp = sum / 10;
            }
            else {
                temp = 0;
                low_num = sum;
            }
            cout<<"temp is"<<temp<<endl;
            ListNode* node = new ListNode(low_num);
            head->next = node;
            head = head->next;
            l1 = l1 == nullptr ? nullptr : l1->next;
            l2 = l2 == nullptr ? nullptr : l2->next;
        }
        return dummy->next;
    }
};
```

15. 三数之和

给你一个整数数组 nums ，判断是否存在三元组 [nums[i], nums[j], nums[k]] 满足 i != j、i != k 且 j != k ，同时还满足 nums[i] + nums[j] + nums[k] == 0 。请你返回所有和为 0 且不重复的三元组。

注意：答案中不可以包含重复的三元组

求你了第三遍了还是不能完整写出来，双指针一定要使用while！！！/left和right一定要移动呀！！！

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        // 三数之和
        // 先定一个指针 之后再定义另外两个指针遍历后面的 需要排序这个数组 双指针
        sort(nums.begin(), nums.end());
        vector<vector<int>> result;
        for(int i{0};i<nums.size();i++) {
            if(i > 0 && nums[i] == nums[i - 1]) continue;
            int left = i + 1;
            int right = nums.size() - 1;
            while(left < right) {
                int sum = nums[i] + nums[left] + nums[right];
                if(sum < 0) left++;
                else if(sum > 0) right--;
                else {
                    result.push_back(vector<int> {nums[i],nums[left],nums[right]});
                    while(left<right && nums[left] == nums[left + 1]) left++;
                    while(left<right && nums[right] == nums[right - 1]) right--;
                    left++;
                    right--;
                }
            }
        }
        return result;
    }
};
```

152. 乘积最大子数组

给你一个整数数组 nums ，请你找出数组中乘积最大的非空连续 子数组（该子数组中至少包含一个数字），并返回该子数组所对应的乘积。

测试用例的答案是一个 32-位 整数。

请注意，一个只包含一个元素的数组的乘积是这个元素的值。

怎么截断i？这里给出了答案

```cpp
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        // 这个就是dp
        vector<int> dp_max(nums.size(), 0);
        vector<int> dp_min(nums.size(), 0);
        dp_max[0] = nums[0];
        dp_min[0] = nums[0];
        int man_mul{nums[0]};
        for(int i{1};i<nums.size();i++) {
            if(nums[i] >= 0) {
                dp_max[i] = max(dp_max[i - 1] * nums[i], nums[i]);
                dp_min[i] = min(dp_min[i - 1] * nums[i], nums[i]);
            }
            else {
                dp_max[i] = max(dp_min[i - 1] * nums[i], nums[i]);
                dp_min[i] = min(dp_max[i - 1] * nums[i], nums[i]);
            }
            man_mul = max(man_mul, dp_max[i]);
        }
        return man_mul;
    }
};
```

448. 找到所有数组中消失的数字

给你一个含 n 个整数的数组 nums ，其中 nums[i] 在区间 [1, n] 内。请你找出所有在 [1, n] 范围内但没有出现在 nums 中的数字，并以数组的形式返回结果。

负数标记法/记住下标是要-1的！！！

```
class Solution {
public:
    vector<int> findDisappearedNumbers(vector<int>& nums) {
        vector<int> result;
        int n = nums.size();
        for(int i{0};i<nums.size();i++) {
            int index = nums[i] > n ? nums[i] - n : nums[i];
            if(nums[index - 1] <= n) nums[index - 1] = nums[index - 1] + n;
            else nums[index - 1] = nums[index - 1];
        }
        for(int i{0};i<nums.size();i++) {
            if(nums[i] <= n) result.push_back(i + 1);
        }
        return result;
    }
};
```

198. 打家劫舍

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，影响你偷窃的唯一制约因素就是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 不触动警报装置的情况下 ，一夜之内能够偷窃到的最高金额。

```
class Solution {
public:
    int rob(vector<int>& nums) {
        // 01 背包问题
        // 使用一维数组dp尝试一下
        // dp[j]表示从下标0-j的房屋里面偷到的最大金额
        // dp[j] 等于 dp[j -2] + nums[j] 和 dp[j - 1] 之间max
        // dp[0] = nums[0];
        // dp[1] = max(num[0],num[1]);
        if(nums.size() == 1) return nums[0];
        vector<int> dp(nums.size(), 0);
        dp.at(-1) = nums[0];
        dp[1] = max(nums[0],nums[1]);
        for(int j{2};j<nums.size();j++) {
            dp[j] = max(dp[j -2] + nums[j],dp[j - 1]);
        }
        return dp[nums.size() - 1];
    }
};
```

79. 单词搜索

给定一个 m x n 二维字符网格 board 和一个字符串单词 word 。如果 word 存在于网格中，返回 true ；否则，返回 false 。

单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用

dfs刚进去的记得先判断当前单词是否满足的，之后在判断是否结束

```
class Solution {
private:
    vector<vector<int>> dir{{1,0}, {-1,0}, {0,1}, {0,-1}};
public:
//  dfs
    bool dfs(vector<vector<char>>& board, string word, int x, int y, vector<vector<int>>& vistie
        if(board[x][y] != word[k]) return false;
        if(k == word.size() - 1) return true;
        vistied[x][y] = 1;
        for(int i{0};i<dir.size();i++) {
            int nextx = x + dir[i][0];
            int nexty = y + dir[i][1];
            if(nextx >= 0 && nextx < board.size() && nexty >= 0 && nexty < board[0].size() && v
                bool fage = dfs(board,word,nextx,nexty,vistied,k + 1);
                if(fage == true) return true;
            }
        }
        vistied[x][y] = 0;
        return false;
    }
    bool exist(vector<vector<char>>& board, string word) {
        cout<<board.size()<<endl;
        cout<<board[0].size()<<endl;
        for(int i{0};i<board.size();i++) {
            for(int j{0};j<board[0].size();j++) {
                vector<vector<int>> vistied(board.size(), vector<int>(board[0].size(), 0));
                bool fage  = dfs(board,word,i,j,vistied,0);
                if(fage == true) return true;
            }
        }
        return false;
    }
};
```

139. 单词拆分

给你一个字符串 s 和一个字符串列表 wordDict 作为字典。如果可以利用字典中出现的一个或多个单词拼接出 s 则返回 true。

注意：不要求字典中出现的单词全部都使用，并且字典中的单词可以重复使用。

这里dp[i]一定要定义为长度为i的是否可以拼接！！！！！！！

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        // dp[i] 表示长度为i的字符串长度是否可以被拼接
        vector<bool> dp(s.size() + 1, false); // 这道题目涉及到下标和长度 很恶心 边界很难处理  所以
        dp[0] = true; //长度为0的当然可以拼接
        for(int i{1};i <= s.size();i++) { //自然i长度要从1开始
            for(const string& str : wordDict) {
                if(i < str.size()) continue;
                else {
                    int len = str.size();
                    int j = i - len; //j是前面的长度
                    string ss = s.substr(j, len);
                    if(ss == str && dp[j] == true) dp[i] = true;
                }
            }
        }
        return dp[s.size()];
    }
};
```

72. 编辑距离

给你两个单词 word1 和 word2， 请返回将 word1 转换成 word2 所使用的最少操作数 。

你可以对一个单词进行如下三种操作：

插入一个字符
删除一个字符
替换一个字符

全军复诵！/字符串的尽量使用长度dp，你要知道的是当前的数值依赖左上/左/上三个方向的数值/go

```
class Solution {
public:
    int minDistance(string word1, string word2) {
        vector<vector<int>> dp(word1.size() + 1, vector<int>(word2.size() + 1, 0));
        //dp[i][j]定义为word1长度为i和word2长度为j的子字符串的最小编辑距离数
        // dp[i][0]不就是一直给word2一直增加i个操作数吗
        // dp[0][j] 也就是一直给word1增加j个操作数
        dp[0][0] = 0;
        for(int i{1}; i < word1.size() + 1; i++) {
            dp[i][0] = i;
        }
        for(int j{0}; j < word2.size() + 1; j++) {
            dp[0][j] = j;
        }
        for(int i{1};i<word1.size() + 1;i++) {
            for(int j{1};j<word2.size() + 1;j++) {
                if(word1[i - 1] == word2[j - 1]) dp[i][j] = dp[i - 1][j - 1];
                //长度是i和j 那么对应当前的下标就是i-1和j-1
                else {
                    dp[i][j] = min(dp[i][j - 1] + 1, min(dp[i - 1][j] + 1, dp[i - 1][j - 1] + 1
                    // dp[i][j - 1] + 1是我给word2增加一个字母 操作数加1
                    // dp[i - 1][j] + 1 是我给word1增加一个字母 操作数加1
                    // dp[i - 1][j - 1] + 1是我给他们其中一个替换成相同的字母 操作数加1
                }
            }
        }
        return dp[word1.size()][word2.size()];
    }
};
```

236. 二叉树的最近公共祖先

给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

百度百科中最近公共祖先的定义为：“对于有根树 T 的两个节点 p、q，最近公共祖先表示为一个节点 x，满足 x 是 p、q 的祖先且 x 的深度尽可能大（一个节点也可以是它自己的祖先）。”

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* traversal(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root == nullptr) return nullptr;
        TreeNode* left = traversal(root->left,p,q);
        TreeNode* right = traversal(root->right,p,q);
        //当前节点处理逻辑
        if(root == p || root == q) return root;
        // return
        if(left != nullptr && right != nullptr) return root;
        if(left == nullptr && right != nullptr) return right;
        if(right == nullptr && left != nullptr) return left;
        if(right == nullptr && left == nullptr) return nullptr;
        return root; // 永远不会到这一步
    }
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        // 如果当前的节点等于p/q 就返回他本身
        // 后序遍历 如果一个节点的左右孩子返回值都不为空 那么他不就是祖宗节点嘛
        return traversal(root,p,q);
    }
};
```

20. 有效的括号

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串 s ，判断字符串是否有效。

有效字符串需满足：

左括号必须用相同类型的右括号闭合。
左括号必须以正确的顺序闭合。
每个右括号都有一个对应的相同类型的左括号。

遇到左括号就入栈，遇到右括号就看是否匹配，匹配就出栈，不匹配或者空栈出错，最终栈空就有效

```
class Solution {
public:
    bool isValid(string s) {
        // 最容易理解的一集
        // 遇到相应的左括号入栈 同时使用哈希表存储好括号对
        // 后面遇到右括号就需要查找对应的左括号是否在栈顶(栈顶是为了保证最近的顺序)
        // 如果不匹配或者栈顶本来就是空的 否则无效
        // 最后循环结束而且栈空了 有效
        stack<char> st;
        unordered_map<char,char> umap{{']', '['}, {'}', '{'}, {')', '('}};
        for(int i{0};i<s.size();i++) {
            char c = s[i];
            if(c == '(' || c== '[' || c == '{') st.push(c); //遇到左括号就入栈
            else {
                // 遇到右括号判断栈顶的左括号是否匹配
                // 如果不匹配或者栈顶本来就是空的
                auto it = umap.find(c);
                if(st.empty() == true) return false;
                if(st.empty() == false && it->second != st.top()) return false;
                else {
                    st.pop();
                }
            }
        }
        return st.empty();
    }
};
```

32. 最长有效括号

给你一个只包含 '(' 和 ')' 的字符串，找出最长有效（格式正确且连续）括号 子串 的长度。

左右括号匹配，即每个左括号都有对应的右括号将其闭合的字符串是格式正确的，比如 "(()())"。

全军复诵！/栈模拟，这个还需要是连续的/新思路就是加上一个start 这个很好理解 抓住连续的有效的必然是连续不相交的

```
class Solution {
public:
    int longestValidParentheses(string s) {
        // 这道题目无法使用回溯+剪枝+有效括号的思路去做
        // 因为需要连续 上述301的思路计算最终长度的时候得到的是不连续的最大长度 天然大于等于此题答案
        // 这道题目抓住连续的有效的必然是连续不相交的
        // 使用一个start记录有效左括号可能的位置
        // 并使用一个栈来记录 左括号入栈 右括号出栈 注意是操作的下标
        // 并判断 如果出栈之后空栈 那么start就是一个满足的左括号位置
        // 如果不是空栈 那么这个栈顶就是一个满足的左括号位置的前一个位置
        // 如果空栈之后还是遇到了右括号 没有办法出栈了 那么start要更新为i + 1 因为必然要从下一个位置
        stack<int> st;
        int start{0}, max_len{0};
        for(int i{0};i<s.size();i++) {
            char c = s[i];
            if(c == '(') st.push(i); //左括号
            else { //右括号
                if(st.empty() == false) {
                    st.pop();
                    if(st.empty() == true) max_len = max(max_len, i - start + 1);
                    else {
                        max_len = max(max_len, i - st.top()); //无需加1这里
                    }
                }
                else start = i + 1;
            }
        }
        return max_len;
    }
};
```

301. 删除无效的括号

给你一个由若干括号和字母组成的字符串 s ，删除最小数量的无效括号，使得输入的字符串有效。

返回所有可能的结果。答案可以按 任意顺序 返回。

组合型回溯 但是去重和剪枝 建议多多复习这括号三姐妹

```
class Solution {
private:
    string path;
    unordered_set<string> result;
public:
    bool IsVaild(string s) {
        // 判断一个字符串是否合法
        // 栈模拟 遇到左括号就入栈 遇到右括号就找相应的左括号(hasp map)，然后看栈顶是不是这个 不是的
        // 也就是说遇到右括号就无非判单栈顶的两种状态 空错 非空继续判断
        // 最终循环结束栈为空那就合法
        stack<char> st;
        for(char& c : s) {
            if(c != '('  && c != ')') continue;
            else if(c == '(') st.push(c);
            else if(c == ')') {
                if(st.empty() == true) return false; //栈是空的 说明第一个就是右括号 非法
                if(st.empty() == false && st.top() == '(') st.pop(); //暂时合法
                else return false; //错误
            }
        }
        if(st.empty() == true) return true;
        return false;
    }

    void traversal(string &s, int startindex, int last_lenth) {
        if(path.size() == last_lenth && IsVaild(path) == true) {
            result.insert(path);
        }
        if(startindex >= s.size()) return ;
        for(int i{startindex};i<s.size() ;i++) {
            if (i > startindex && s[i] == s[i-1] && (s[i] == '(' || s[i] == ')')) {
                continue;
            }
            // if (i > startindex && s[i] == '(' && s[i] == s[i] - 1) continue;
            // if (i > startindex && s[i] == ')' && s[i] == s[i] - 1) continue;
            // 遇上左右括号需要各自去重 但是字母不需要哦
            // 下面的去重无法通过 但是合在一起就可以了 6
            // i > startindex：确保是 当前递归层中，连续相同字符的第 2 个及以后。
            // s[i] == s[i-1]：当前字符和前一个字符相同（连续重复）。
            // s[i] == '(' || s[i] == ')'：只对括号生效，非括号字符（如 'a'、'f'）即使连续也不跳过。
            path.push_back(s[i]);
            traversal(s,i + 1,last_lenth);
            path.pop_back();
```

```
        }
    }

    vector<string> removeInvalidParentheses(string s) {
        // 回溯？ 回溯得到所有的括号组合？
        // 复用:用来判断当前字符串是否合法
    int leftExtra = 0, rightExtra = 0;
    int balance = 0;
    for (char c : s) {
        if (c == '(') {
            balance++;
        } else if (c == ')') {
            if (balance > 0) {
                balance--; // 有匹配的左括号
            } else {
                rightExtra++; // 右括号超前，必须删除
            }
        }
    }
    leftExtra = balance; // 剩余未匹配的左括号，必须删除

    // 正确的目标长度 = 原长 - 必须删除的括号数
    int last_lenth = s.size() - leftExtra - rightExtra;
        traversal(s, 0, last_lenth);
        if(result.empty() == true) return vector<string>{""};
        else return vector<string>(result.begin(), result.end());
    }
};
```

406. 根据身高重建队列

假设有打乱顺序的一群人站成一个队列，数组 people 表示队列中一些人的属性（不一定按顺序）。每个 people[i] = [hi, ki] 表示第 i 个人的身高为 hi ，前面 正好 有 ki 个身高大于或等于 hi 的人。

请你重新构造并返回输入数组 people 所表示的队列。返回的队列应该格式化为数组 queue ，其中 queue[j] = [hj, kj] 是队列中第 j 个人的属性（queue[0] 是排在队列前面的人）

按照身高降序 位置升序排序/先安排高的 矮的随便插队/golang

```
class Solution {
public:
    static bool cmp(vector<int>& a, vector<int>& b) {
        if(a[0] == b[0]) { //第二位降序
            if(a[1] < b[1]) return true;
            else return false;
        }
        else { // 第一位升序
            if(a[0] > b[0]) return true;
            else return false;
        }
    }
    vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
        // 理解这道题目的意思就是 一个人的的位置和比他矮的人的位置无关
        // 所以可以先安排高的 矮的随便插队
        sort(people.begin(), people.end(), cmp);
        vector<vector<int>> que; //这里只能是insert 不能初始化que的容量的 需要动态扩容
        for(int i{0};i<people.size();i++) {
            int k = people[i][1];
            que.insert(que.begin() + k, people[i]);
        }
        return que;
    }
};
```

51. N 皇后

按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

n 皇后问题 研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。

给你一个整数 n ，返回所有不同的 n 皇后问题 的解决方案。

每一种解法包含一个不同的 n 皇后问题 的棋子放置方案，该方案中 'Q' 和 '.' 分别代表了皇后和空位。

模板函数声明和定义不能分开/IsValid函数还要注重细节

```
class Solution {
private:
    vector<string> chessboard;
    vector<vector<string>> result;
public:
    // 回溯的一个经典例子
    // n就是树的深度和广度
    // 这个相当于是多个集合 每次是从不同的集合开始 所以for的起始都是列的开始
    // 每次搜索都是从每一行的第一列开始搜索到最后一列 找到满足条件的位置防止皇后
    bool isVaild(int row, int col, int n) {
        //检查当前行的所有列
        for(int i{0};i<row;i++) {
            if(chessboard[i][col] == 'Q') {  //一定不可以等于 等于就会出问题 检查的是上面行不包括当
                return false;
            }
        }
        //检查左上
        for(int i{row - 1},j{col - 1};i>=0 && j>=0;i--,j--) { //条件一定要加 &&
            if(chessboard[i][j] == 'Q') {
                return false;
            }
        }
        //检查右上
        for(int i{row - 1},j{col + 1};i >=0 && j < n;i--,j++) { //条件一定要加 &&
            if(chessboard[i][j] == 'Q') {
                return false;
            }
        }
        return true;
    }
    void backtracking(int n, int row) {
        if(row == n) {
            result.push_back(chessboard);
            return ;
        }
        for(int j{0}; j<n; j++) {
            // 每次从每一行的第一列开始搜索
            if(isVaild(row, j, n) == true) {
                chessboard[row][j] = 'Q';
                backtracking(n, row + 1);
                chessboard[row][j] = '.';
            }
        }
```

```
    }
    vector<vector<string>> solveNQueens(int n) {
        chessboard = vector<string>{static_cast<size_t>(n), string(n, '.')};
        backtracking(n,0);
        return result;
    }
};
```

287. 寻找重复数

给定一个包含 n + 1 个整数的数组 nums ，其数字都在 [1, n] 范围内（包括 1 和 n），可知至少存在一个重复的整数。

假设 nums 只有 一个重复的整数 ，返回 这个重复的数 。

你设计的解决方案必须 不修改 数组 nums 且只用常量级 O(1) 的额外空间

正负数标记法无敌/golang

```
func findDuplicate(nums []int) int {
    var result int = 0;
         for _, value := range nums {
                  fmt.Println("value is:", value);
                  var index int = 0;
                  if value < 0 {
                           index = -value;
                  } else {
                           index = value;
                  }
                  var i int = index - 1;
                  if nums[i] < 0 {
                           if value < 0 {
                                    result = -value;
                           } else {
                                    result = value;
                           }
                  } else {
                           nums[i] = -nums[i];
                  }
         }
         return result;
}
```

283. 移动零

给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。

请注意 ，必须在不复制数组的情况下原地对数组进行操作。

最朴素的就是，非零元素直接左移填充原数组，最后数组后面补零就完事了

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        // 来点最朴素的想法，把非零元素左移，直接覆盖原数组，后面直接补0就行了
        int cur{0};
        for(int i{0}; i < nums.size(); i++) {
            if(nums[i] != 0) {
                nums[cur] = nums[i];
                cur++;
            }
        }
        for(int i{cur}; i < nums.size(); i++) {
            nums[i] = 0;
        }
    }
};
```

207. 课程表

你这个学期必须选修 numCourses 门课程，记为 0 到 numCourses - 1 。

在选修某些课程之前需要一些先修课程。 先修课程按数组 prerequisites 给出，其中 prerequisites[i] = [ai, bi] ，表示如果要学习课程 ai 则 必须 先学习课程 bi 。

例如，先修课程对 [0, 1] 表示：想要学习课程 0 ，你需要先完成课程 1 。
请你判断是否可能完成所有课程的学习？如果可以，返回 true ；否则，返回 false

三色标记法+visited要&！！/1标记为已经访问，如果后面一条路径中再一次遇到了1，那就是说明有环。如果一条路径没有环，那么再最后设置所有路径的节点为2说明没有环。

```
class Solution {
private:
public:
    bool dfs(vector<vector<int>>& graph, vector<int>& visited, int x) { // 用于检测一个节点是否存
        if(visited[x] == 1) return false; // 遇到自己了 说明有环
        if(visited[x] == 2) return true; // 被设置为2就说明可以正常遍历完 无环 也避免了重复检测
        visited[x] = 1;// 标记为正在访问
        //下面对可达路径进行深搜
        for(int i{0};i<graph[x].size();i++) {
            int next = graph[x][i];
            if(dfs(graph,visited,next) == false) return false;
        }
        visited[x] = 2;
        return true;
    }
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        // 先建立邻接表 然后在进行所有可达路径的检测是否存在环
        vector<vector<int>> graph(numCourses);
        vector<int> visited(numCourses,0);
        for(const auto& it : prerequisites) {
            int first = it[0];
            int second = it[1];
            graph[first].push_back(second);
        } // 建立邻接表
        for(int i{0};i<graph.size();i++) {
            if(dfs(graph,visited,i) == false) return false; // 对每一个课程节点进行一个检测 所有课
        }
        return true;
    }
};
```

802. 找到最终的安全状态

有一个有 n 个节点的有向图，节点按 0 到 n - 1 编号。图由一个 索引从 0 开始 的 2D 整数数组 graph表示， graph[i]是与节点 i 相邻的节点的整数数组，这意味着从节点 i 到 graph[i]中的每个节点都有一条边。

如果一个节点没有连出的有向边，则该节点是 终端节点 。如果从该节点开始的所有可能路径都通向 终端节点（或另一个安全节点），则该节点为 安全节点。

返回一个由图中所有 安全节点 组成的数组作为答案。答案数组中的元素应当按 升序 排列。

一条路径遍历完了并且没有环,把当前节点设置为2。但是由于递归的存在,栈上面的所有节点,就是路径上面的所有节点都会被设置为了2

```
class Solution {
public:
    bool dfs(vector<vector<int>>& graph, int x, vector<int>& visited) {
        if(visited[x] == 1) return false; // 遇到自己 有环
        if(visited[x] == 2) return true; // 遍历完成 没有环
        visited[x] = 1;
        // 进行所有可达路径的搜索
        for(int i{0};i<graph[x].size();i++) {
            int next = graph[x][i];
            bool result = dfs(graph,next,visited);
            if(result == false) return false;
        }
        //一条路径遍历完了并且没有环,把当前节点设置为2
        // 但是由于递归的存在,栈上面的所有节点,就是路径上面的所有节点都会被设置为了2
        visited[x] = 2;
        return true;
    }
    vector<int> eventualSafeNodes(vector<vector<int>>& graph) {
        // 就是判断有没有环 没有环的节点就是终端节点
        // 对于邻接图的判断节点有没有环使用三色法
        vector<int> visited(graph.size(), 0);
        vector<int> ans;
        for(int i{0};i<graph.size();i++) {
            bool result = dfs(graph,i,visited);
            if(result == true) ans.push_back(i);
        }
        return ans;
    }
};
```

25. K 个一组翻转链表

给你链表的头节点 head ，每 k 个节点一组进行翻转，请你返回修改后的链表。

k 是一个正整数，它的值小于或等于链表的长度。如果节点总数不是 k 的整数倍，那么请将最后剩余的节点保持原有顺序。

你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

1.pre设为nullptr，pre是为了保持head的一致性，cur设为head，同时启用dummy和移动用的p
2.这道题目仍是很抽象，第一个for循环不足k个的条件应该是j + k <= length，还有最后的移动p的时候是p移动到p_next的位置，因为反转之后这个p_next下一个位置。难死了我靠。

```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode() : val(0), next(nullptr) {}
 *     ListNode(int x) : val(x), next(nullptr) {}
 *     ListNode(int x, ListNode *next) : val(x), next(next) {}
 * };
 */
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        if(head == nullptr) return head;
        int length{0};
        ListNode* p0 = head;
        while(p0 != nullptr) {
            length++;
            p0 = p0->next;
        }
        if(length == 1) return head;
        ListNode* pre = nullptr;
        ListNode* cur = head;
        ListNode* p = new ListNode();
        p->next = head;
        ListNode* dummy = p;
        for(int j{0};j + k <= length; j = j + k) {
            // 进入反转k的时候 要先保存当前指针的下一个
            ListNode* p_next = p->next;
            for(int i{0}; i < k; i++) {
                ListNode* cur_next = cur->next;
                cur->next = pre;
                pre = cur;
                cur = cur_next;
            }
            p->next = pre;
            p_next->next = cur;
            p = p_next;
        }
        return dummy->next;
    }
};
```

128. 最长连续序列

给定一个未排序的整数数组 nums ，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。

请你设计并实现时间复杂度为 O(n) 的算法解决此问题

这道题目只能使用哈希，遍历中寻找x+1是否存在表中，其中x-1是实现on的关键

```
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        unordered_set<int> uset(nums.begin(), nums.end());
        int max_len{0};
        for(int i{0}; i < nums.size(); i++) {// 这里遍历nums会超时,遍历uset就不会超时.......
            int x = nums[i];
            if(uset.find(x - 1) != uset.end()) continue; //实现on的关键
            int next_x = x + 1;
            while(uset.find(next_x) != uset.end()) {
                next_x++;
            }
            // 最后的数字是next-x - 1,因为最后找到next_x还会自增的
            max_len = max(max_len, next_x - x);
            if (max_len * 2 >= uset.size()) break;
        }
        return max_len;
    }
};
```

114. 二叉树展开为链表

给你二叉树的根结点 root ，请你将它展开为一个单链表：

展开后的单链表应该同样使用 TreeNode ，其中 right 子指针指向链表中下一个结点，而左子指针始终为 null 。
展开后的单链表应该与二叉树 先序遍历 顺序相同。

不用记住什么奇怪的难饶的算法，直接记住前序遍历获得数组，再重建这颗树就行了/go写起来真难受呀/go的切片一旦扩容就会新建一个新的的了

```go
/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func traversal(root *TreeNode, pre_nums *[]int) {
    if root == nil {
        return ;
    }
    *pre_nums = append(*pre_nums, root.Val);
    traversal(root.Left, pre_nums);
    traversal(root.Right, pre_nums);
}
func flatten(root *TreeNode)  {
    // 前序遍历获得顺序 然后重建这棵树
    var pre_nums []int = make([]int, 0);
    traversal(root, &pre_nums);
    fmt.Println(pre_nums);
    for i,n := 0,len(pre_nums); i<n; i++ {
        root.Val = pre_nums[i];
        if i == len(pre_nums) - 1 {
            break;
        }
        root.Left = nil;
        if root.Right == nil {
            node := &TreeNode{};
            root.Right = node;
            // ???
            root = root.Right;
        } else {
            root = root.Right;
        }
    }
}
```

904. 水果成篮

你正在探访一家农场，农场从左到右种植了一排果树。这些树用一个整数数组 fruits 表示，其中 fruits[i] 是第 i 棵树上的水果 种类 。

你想要尽可能多地收集水果。然而，农场的主人设定了一些严格的规矩，你必须按照要求采摘水果：

你只有 两个 篮子，并且每个篮子只能装 单一类型 的水果。每个篮子能够装的水果总量没有限制。
你可以选择任意一棵树开始采摘，你必须从 每棵 树（包括开始采摘的树）上 恰好摘一个水果 。采摘的水果应当符合篮子中的水果类型。每采摘一次，你将会向右移动到下一棵树，并继续采摘。
一旦你走到某棵树前，但水果不符合篮子的水果类型，那么就必须停止采摘。
给你一个整数数组 fruits ，返回你可以收集的水果的 最大 数目。

典型不定长滑动窗口的例题/go的delete第二个参数是key/写题的时候先写思路不然很容易乱的

```
func totalFruit(fruits []int) int {
    // 不定长的滑动窗口
    var left int = 0
    var right int = 0
    var max_fruit int = 0;
    var fruits_map map[int]int = make(map[int]int)
    for right < len(fruits) {
        // 当新水果存在篮子就加入 右加加
        // 新水果不属于篮子 判断篮子现在的种类是否==2 等于2那只能移除左边的
        // 如果不等于2 那么可以加入
        var cur_fruir = fruits[right]
        _, exit := fruits_map[cur_fruir]
        for exit == false && len(fruits_map) == 2 {
            var left_fruit int = fruits[left];
            fruits_map[left_fruit]--;
            left++;
            if fruits_map[left_fruit]== 0 {
                delete(fruits_map, left_fruit)
            }
        }
        fruits_map[cur_fruir]++;
        max_fruit = max(max_fruit, right - left + 1)
        right++;
    }
    return max_fruit;
}
```

209. 长度最小的子数组

给定一个含有 n 个正整数的数组和一个正整数 target 。

找出该数组中满足其总和大于等于 target 的长度最小的 子数组 [numsl, numsl+1, ..., numsr-1, numsr] ，并返回其长度。如果不存在符合条件的子数组，返回 0 。

不定长滑动窗口，更新的机制还自己熟悉

```
class Solution {
public:
    int minSubArrayLen(int target, vector<int>& nums) {
        int result = INT32_MAX;
        int slow=0;
        int len = nums.size();
        int sum = 0;
        int sublen = 0;
        for(int fast=0;fast<len;fast++){
            // int sublen = fast - slow + 1;
            sum+=nums[fast];
            while(sum>=target){
                // slow++;
                sublen = fast - slow + 1; //sublen必须要在这里改变 在前面写会出现slow前移了但是sub
                result = result<sublen ?result :sublen;
                sum-=nums[slow++];
            }
        }
        return result==INT32_MAX ?0:result;

    }
};
```

643. 子数组最大平均数 I

给你一个由 n 个元素组成的整数数组 nums 和一个整数 k 。

请你找出平均数最大且 长度为 k 的连续子数组，并输出该最大平均数。

任何误差小于 10-5 的答案都将被视为正确答案。

定长的滑动窗口和不定长的模板还是很相似的，记住模板就可以ac很多题目了

```
class Solution {
public:
    double findMaxAverage(vector<int>& nums, int k) {
        // 这个是定长滑动窗口吧
        int left{0};
        int right{0};
        double sum{0};
        double result{0};
        while(right < nums.size()) {
            sum = sum + nums[right];
            while(right - left + 1 > k) {
                sum = sum - nums[left];
                left++;
            }
            if(right - left + 1 == k) result = max(result, double(sum / k));
            right++;
        }
        return result;
    }
};
```

2090. 半径为 k 的子数组平均值

给你一个下标从 0 开始的数组 nums ，数组中有 n 个整数，另给你一个整数 k 。

半径为 k 的子数组平均值 是指：nums 中一个以下标 i 为 中心 且 半径 为 k 的子数组中所有元素的平均值，即下标在 i - k 和 i + k 范围（含 i - k 和 i + k）内所有元素的平均值。如果在下标 i 前或后不足 k 个元素，那么 半径为 k 的子数组平均值 是 -1 。

构建并返回一个长度为 n 的数组 avgs ，其中 avgs[i] 是以下标 i 为中心的子数组的 半径为 k 的子数组平均值 。

x 个元素的 平均值 是 x 个元素相加之和除以 x ，此时使用截断式 整数除法 ，即需要去掉结果的小数部分。

例如，四个元素 2、3、1 和 5 的平均值是 (2 + 3 + 1 + 5) / 4 = 11 / 4 = 2.75，截断后得到 2 。

强化定长和不定长的滑动窗口题目 滑动窗口只能使用与非负数的情况 因为这个方法依赖单调性

```cpp
class Solution {
public:
    vector<int> getAverages(vector<int>& nums, int k) {
        // 依旧不定长滑动窗口
        int left{0}, right{0};
        vector<int> result(nums.size(), -1);
        long long sum{0};
        while(right < nums.size()) {
            // 1.进入窗口
            sum = sum + nums[right];
            // 2.维护窗口
            while(right - left + 1 > 2 * k + 1) {
                sum = sum - nums[left];
                left++;
            }
            // 3.更新答案
            if(right - left + 1 == 2 * k + 1) {
                int index{0};
                index = (left + right) / 2; //中心节点是(left + right) / 2
                result[index] = sum / (2 * k + 1);
            }
            right++;
        }
        return result;
    }
};
```

200. 岛屿数量

给你一个由 '1'（陆地）和 '0'（水）组成的的二维网格，请你计算网格中岛屿的数量。

岛屿总是被水包围，并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外，你可以假设该网格的四条边均被水包围。

使用一个visited就行了标记是否看过，图论dfs没有见到回溯visited的，岛屿数量就是dfs的次数/go

```
class Solution {
private:
    int four_dir[4][2] = {1,0, 0,1, -1,0, 0,-1};
    int count = 0;
public:
    void dfs(vector<vector<char>>& grid,vector<vector<bool>>& visited, int x, int y) {
        if(x < 0 || x >= grid.size() || y <0 || y >=grid[0].size() || grid[x][y] == '0' || visit
        visited[x][y] = true;
        for(int i{0};i<4;i++) {
            int nextx = x + four_dir[i][0];
            int nexty = y + four_dir[i][1];
            dfs(grid,visited,nextx,nexty);
        }
    }
    int numIslands(vector<vector<char>>& grid) {
        int row = grid.size();
        int col = grid[0].size();
        vector<vector<bool>> visited(row, vector<bool>(col, false));
        for(int i{0};i<row;i++) {
            for(int j{0};j<col;j++) {
                if(grid[i][j] == '1' && visited[i][j] == false) {
                    dfs(grid,visited,i,j);
                    count++;
                }
            }
        }
        return count;
    }
};
```

994. 腐烂的橘子

在给定的 m x n 网格 grid 中，每个单元格可以有以下三个值之一：

值 0 代表空单元格；
值 1 代表新鲜橘子；
值 2 代表腐烂的橘子。
每分钟，腐烂的橘子 周围 4 个方向上相邻 的新鲜橘子都会腐烂。

返回 直到单元格中没有新鲜橘子为止所必须经过的最小分钟数。如果不可能，返回 -1 。

dfs拼尽全力无法战胜/dfs在元宝的帮助下通过了，但是及其复杂

```
class Solution {
private:
    vector<vector<int>> dir{{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
public:
    void dfs(vector<vector<int>> &graph, vector<vector<int>>& visited, int x, int y, int time) {
        //  if(visited[x][y] == 1) return ;
        if(visited[x][y] != -1 && time > visited[x][y]) return ; //说明已经被污染了，而且再一次被
        visited[x][y] = time; //保留污染时间
        for(int i{0}; i < dir.size(); i++) {
            int next_x = x + dir[i][0];
            int next_y = y + dir[i][1];
            if(next_x >= 0 && next_x < graph.size() && next_y >= 0 && next_y < graph[0].size() {
                dfs(graph, visited, next_x, next_y, time + 1); // 污染的时候 把时间也带到所有被污
            }
        }
    }
    int orangesRotting(vector<vector<int>>& grid) {
        int time{0};
        vector<vector<int>> visited(grid.size(), vector<int>(grid[0].size(), -1));
        for(int i{0};i<grid.size();i++) {
            for(int j{0};j<grid[0].size();j++) {
                if(grid[i][j] == 2) {
                    dfs(grid,visited,i,j,0);
                }
            }
        }
        for(int i{0};i<visited.size();i++) {
            for(int j{0};j<visited[0].size();j++) {
                time = max(time, visited[i][j]);
                if(grid[i][j] == 1 && visited[i][j] == -1) {
                    return -1;
                }
            }
        }
        return time;
        }
};
```

647. 回文子串

给你一个字符串 s ，请你统计并返回这个字符串中 回文子串 的数目。

回文字符串 是正着读和倒过来读一样的字符串。

子字符串 是字符串中的由连续字符组成的一个序列。

中心扩展法不是很好做嘛？ 中心扩展法的思路是，选定一个中心，然后不断扩展两边位置，如果两边相等，那么这就是一个满足的子串！！！（也选定一个中心可以扩展出大于1的回文子串！！！这是你第二次做的误区）

```
class Solution {
public:
    // 中心扩展法
    // 选定一个中心然后往外扩展 如果两者相同就继续 不然就结束
    // 这个函数的作用是 每次选定一个中心向外扩展 如果两边相同 那么就是一个满足的回文子串了
    int SubStr(int left, int right, int left_bound, int right_bound, string s) {
        int res{0};
        while(left >= left_bound && right <= right_bound && s[left] == s[right]) {
            left--;
            right++;
            res++;
        }
        return res;
    }
    int countSubstrings(string s) {
        int result{0};
        for(int i{0}; i < s.size(); i++) {
            result = result + SubStr(i, i, 0, s.size() - 1, s);
            result = result + SubStr(i, i + 1, 0, s.size() - 1, s);
        }
        return result;
    }
};
```

5. 最长回文子串

给你一个字符串 s，找到 s 中最长的 回文 子串。

中心扩展法+答案数组秒了

```
class Solution {
public:
    vector<int> extend(string& s, int left, int right) {
        while(left >= 0 && right < s.size() && s[left] == s[right]) {
            left--;
            right++;
        }
        // 不满足的前一个就是最后保留的下标
        return vector<int>{left + 1, right - 1, right - 1 - (left + 1)}; // left right length
    }
    string longestPalindrome(string s) {
        // 保留距离最大的那两个下标
        vector<int> result(3, 0);
        for(int i{0};i < s.size(); i++) {
            // 寻找奇数长度的回文子串
            vector<int>res1 = extend(s, i, i);
            // 比较结果 如果更长就更新
            if(res1[2] > result[2]) {
                result[0] = res1[0];
                result[1] = res1[1];
                result[2] = res1[2];
            }
            // 寻找偶数长度的回文子串
            vector<int>res2 = extend(s, i, i + 1);
            if(res2[2] > result[2]) {
                result[0] = res2[0];
                result[1] = res2[1];
                result[2] = res2[2];
            }
        }
        return s.substr(result[0], result[2] + 1);
    }
};
```

1143. 最长公共子序列

给定两个字符串 text1 和 text2，返回这两个字符串的最长 公共子序列 的长度。如果不存在 公共子序列 ，返回 0 。

一个字符串的 子序列 是指这样一个新的字符串：它是由原字符串在不改变字符的相对顺序的情况下删除某些字符（也可以不删除任何字符）后组成的新字符串。

例如，"ace" 是 "abcde" 的子序列，但 "aec" 不是 "abcde" 的子序列。
两个字符串的 公共子序列 是这两个字符串所共同拥有的子序列。

子序列的长度dp很好做/golang 最长回文子序列就反过来做dp就行了

```
func longestCommonSubsequence(text1 string, text2 string) int {
    var dp [][]int = make([][]int, len(text1) + 1);
        for i,n := 0, len(dp); i < n; i++ {
                dp[i] = make([]int, len(text2) + 1)
        }

        // 初始化dp[0][j] = 0
        // dp[i][0] = 0;
        for i,n :=1, len(dp); i < n; i++ {
                for j,m := 1, len(dp[0]); j < m; 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[len(text1)][len(text2)]
}
```

560. 和为 K 的子数组

给你一个整数数组 nums 和一个整数 k ，请你统计并返回 该数组中和为 k 的子数组的个数 。

子数组是数组中元素的连续非空序列。

前缀和+哈希表 好恶心的题目 滑动窗口失败了 umap[0]=1一定要初始化 为了包含从开头开始的子数组

因为存在了负数 所以不可以用滑动窗口了 因为滑动窗口依赖单调性的

```
func subarraySum(nums []int, k int) int {
        var umap map[int]int = make(map[int]int, len(nums))
    umap[0] = 1 //这个是神来之笔 也是最臭的一笔
        var result int = 0;
        var pre_sum []int = make([]int, len(nums))
        pre_sum[0] = nums[0]
        for i,n := 1, len(nums); i < n; i++ {
                pre_sum[i] = pre_sum[i - 1] + nums[i]
        }

        for i,n := 0, len(nums); i < n; i++ {
                var target int = pre_sum[i] - k
                value, exit := umap[target] //如果exit 那么 _ 就是对应的value
                if exit {
                        result = result + value;
                }
                umap[pre_sum[i]]++;
        }
        return result
}
```

437. 路径总和 III

给定一个二叉树的根节点 root ，和一个整数 targetSum ，求该二叉树里节点值之和等于 targetSum 的 路径 的数目。

路径 不需要从根节点开始，也不需要在叶子节点结束，但是路径方向必须是向下的（只能从父节点到子节点）。

前缀和+哈希表 对于回溯的理解又加深了，只是umap[0]=1一定要记住

```
class Solution {
private:
    int result;
public:
    void traversal(TreeNode* root, int targetSum, long long& presum, unordered_map<long long,lo
        if(root == nullptr) return ;
        presum = presum + root->val; // 当前节点的前缀和
        long long target  = presum - targetSum; // 需要寻找的目标数字
        if(umap.find(target) != umap.end()) {
            result = result + umap[target];
        }
        umap[presum]++;
        traversal(root->left,targetSum,presum,umap);
        traversal(root->right,targetSum,presum,umap);
        // 回溯
        umap[presum]--;
        presum = presum - root->val;
    }
    int pathSum(TreeNode* root, int targetSum) {
        // 前缀和 计算一条路径上的前缀和 然后使用一个哈希表去统计前面出现过的前缀和
        long long presum{0};
        unordered_map<long long,long long> umap;
        umap[0] = 1;
        traversal(root,targetSum,presum,umap);
        return result;
    }
};
```

416. 分割等和子集

给你一个 只包含正整数 的 非空 数组 nums 。请你判断是否可以将这个数组分割成两个子集，使得两个子集的元素和相等。

01背包且价值问题，问题特征是问你是否可以填满/go

```
func canPartition(nums []int) bool {
    var sum int = 0
        for _, value := range nums {
                sum = sum + value
        }
        if sum % 2 != 0 {
                return false
        }
        var target int = sum / 2
        var dp [][]int  = make([][]int, len(nums))
        for i, n := 0, len(nums); i < n; i++ {
                dp[i] = make([]int, target + 1)
        }
        // dp[0][j] 在j>=nums[0]的时候就是 nums[0]
        // dp[i][0]就是0 不用再初始化了
        for j, m := 0, len(dp[0]); j < m; j++ {
                if j < nums[0] {
                        continue
                } else {
                        dp[0][j] = nums[0]
                }
        }
        for i, n := 1, len(nums); i < n; i++ {
                for j, m := 0, len(dp[0]); j < m; j++ {
                        if j < nums[i] {
                                dp[i][j] = dp[i - 1][j]
                        } else {
                                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - nums[i]] + nums
                        }
                }
        }
        var ret int = dp[len(nums) - 1][target]
        if ret == target {
                return true
        } else {
                return false
        }
}
```

494. 目标和

给你一个非负整数数组 nums 和一个整数 target 。

向数组中的每个整数前添加 '+' 或 '-' ，然后串联起所有整数，可以构造一个 表达式：

例如，nums = [2, 1] ，可以在 2 之前添加 '+' ，在 1 之前添加 '-' ，然后串联起来得到表达式 "+2-1" 。
返回可以通过上述方法构造的、运算结果等于 target 的不同 表达式 的数目。

01背包且数目问题，问题特征是问你返回最后的数目/go 使用去零的方案去做很好处理边界情况，最后乘法分步原理两种情况相乘得到最终答案，终于解决这个恶心的题目了

```go
func findTargetSumWays(nums []int, target int) int {
    var sum int = 0
        var num_no_zero []int = make([]int, 0)
        var zero_cnt int = 0
        for _, value := range nums {
                if value != 0 {
                        num_no_zero = append(num_no_zero, value)
                } else {
                        zero_cnt++
                }
        }
    if zero_cnt == len(nums) {
        return (1 << zero_cnt)
    }
        for _, value := range num_no_zero {
                sum = sum  + value;
        }
    if target > sum {
        return 0
    } else if (target + sum) % 2 != 0 {
        return 0
    }
        var targetsum int = (target + sum) / 2 ;
    if targetsum < 0 {
        return 0
    }
        var dp [][]int = make([][]int, len(num_no_zero));
        for index, _ := range dp {
                dp[index] = make([]int, targetsum + 1);
        }
        // 转化为非0的情况 非0情况下
        // dp[i][0] 只能是0 因为没有任何办法可以刚好装满0容量的背包 有一种 那就是不装
    for i, n := 0, len(dp); i < n; i++ {
                dp[i][0] = 1;
        }
        // dp[0][j] 在j==nums[0]的时候 就装它 等于1
        for j, m := 0, len(dp[0]); j < m; j++ {
                if j == num_no_zero[0] {
                        dp[0][j] = 1;
                }
        }
        for i, n := 1, len(num_no_zero); i < n; i++ {
                for j, m := 1, len(dp[0]); j < m; j++ {
```

```
                        if j < num_no_zero[i] {
                                dp[i][j] = dp[i - 1][j];
                        } else {
                                dp[i][j] = dp[i - 1][j] + dp[i - 1][j - num_no_zero[i]];
                        }
                }
        }
        return dp[len(num_no_zero) - 1][targetsum] * (1 << zero_cnt);
}
```

75. 颜色分类

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。

必须在不使用库内置的 sort 函数的情况下解决这个问题。

借助这道题目又复习了一遍快速排序，注意base和 base+1 递归分治完成全部。但是这道题目正确解法是荷兰国旗问题，左右一个栈，0入左栈，2入右栈，1不变

```
class Solution {
public:
    int QuickSort(vector<int>& nums, int left, int right) {
        // 快排的思路是找到一个base 然后一轮就是比他小的在左边 比他大的在右边
        // 「一次划分定一值，递归分治定全局」
        int base = nums[left];
        while(left < right) {
            while(left < right && nums[right] >= base) right--;
            swap(nums[left], nums[right]);
            while(left < right && nums[left] <= base) left++;
            swap(nums[left], nums[right]);
        }
        nums[left] = base;
        return left;
    }
    // 递归-分治
    void Sort(vector<int>& nums, int left, int right) {
        if(left >= right) return ;
        int base = QuickSort(nums,left,right);
        Sort(nums,left,base);
        Sort(nums,base + 1, right);
    }
    void sortColors(vector<int>& nums) {
        Sort(nums,0,nums.size() - 1);
    }
};
```

4. 寻找两个正序数组的中位数

给定两个大小分别为 m 和 n 的正序（从小到大）数组 nums1 和 nums2。请你找出并返回这两个正序数组的 中位数 。

算法的时间复杂度应该为 O(log (m+n)) 。

重新理解了一遍快选，但是这道题目使用排序链表的方法去做不就可以了嘛（重新理解排序链表，但是不是真正的排序链表，需要用到排序链表的思想，谁小就记录谁，然后指针移动，边界条件有点恶心），不要管那个时间复杂度了，不会做/golang

```
func findMedianSortedArrays(nums1 []int, nums2 []int) float64 {
    // 思路就是找中间那个数字 就是排序两个有序链表的思路
        // 两个指针比较 更新指针,谁小谁移动, 一共移动的步数就是要找的那个数字的下标
        var index1 int = 0
        var index2 int = 0
        var length int = len(nums1) + len(nums2)
        // 如果是奇数 那么length / 2 就刚好是哪个下标 如果是偶数 length / 2 就是后一个数字
        var k1 int = length / 2
        var pre float64 = 0;
        var cur float64 = 0;
        for k1 >= 0 {
                pre = cur
                if index1 < len(nums1) && index2 < len(nums2) && nums1[index1] < nums2[index2
          // 条件是 两个都灭有遍历完或者2遍历完了 这个时候对1操作 不然就对2操作
                        cur = float64(nums1[index1]) //记录当前的数值-插入新链表
                        index1++;
                } else {
                        cur = float64(nums2[index2]) //记录当前的数值
                        index2++;
                }
                k1--
        }
        if length % 2 != 0 {
                return cur
        } else {
                return (cur + pre) / 2
        }
}
```

23. 合并 K 个升序链表

给你一个链表数组，每个链表都已经按升序排列。

请你将所有链表合并到一个升序链表中，返回合并后的链表

归并排序，分治递归，我的建议是left包含那个中间节点，右边不包含ok？后序遍历，这道题目比排序链表简单

```
class Solution {
public:
    ListNode* Merge(ListNode* node1, ListNode* node2) {
        ListNode* dummy = new ListNode();
        ListNode* p = dummy;
        while(node1 != nullptr && node2 != nullptr) {
            if(node1->val <= node2->val) {
                p->next = node1;
                node1 = node1->next;
                p = p->next;
            }
            else {
                p->next = node2;
                node2 = node2->next;
                p = p->next;
            }
        }
        p->next = node1 == nullptr ? node2 : node1;
        return dummy->next;
    }
    // 递归-分治 需要一个递归来实现最后的逻辑
    ListNode* traversal(vector<ListNode*>& lists, int left, int right) {
        if(left > right) return nullptr; //区间违法 那就是递归结束
        if(left == right) return lists[left]; //区间相等 那就是只有一个链表 这个链表是有序的
        // 后序遍历
        int k = left + (right - left) / 2;
        ListNode* left_node = traversal(lists, left, k);
        ListNode* right_node = traversal(lists, k + 1, right);
        return Merge(left_node, right_node);
    }
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        return traversal(lists, 0, lists.size() - 1);
    }
};
```

148. 排序链表

给你链表的头结点 head ，请将其按 升序 排列并返回 排序后的链表 。

归并排序，分治递归，比上面的难，我建议这两道题目过两周再复习一下。12.18复习一下吧

```cpp
class Solution {
public:
    ListNode* divice(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head;
        ListNode* pre = nullptr;
        // slow当成是第二个链表的第一个节点 要断开slow的前一个结点
        while(fast != nullptr && fast->next != nullptr && slow != nullptr) {
            pre = slow;
            fast = fast->next->next;
            slow = slow->next;
        }
        pre->next = nullptr;
        return slow;
    }
    ListNode* merge(ListNode* node1, ListNode* node2) {
        ListNode* dummy = new ListNode(0);
        ListNode *p = dummy;
        while(node1 != nullptr && node2 != nullptr) {
            if(node1->val <= node2->val) {
                p->next = node1;
                node1 = node1->next;
                p = p->next;
            } else {
                p->next = node2;
                node2 = node2->next;
                p = p->next;
            }
        }
        p->next = node1 != nullptr ? node1 : node2;
        return dummy->next;
    }
    ListNode* traversal(ListNode* head) {
        if(head == nullptr) return nullptr;
        if(head != nullptr && head->next == nullptr) return head; //一个节点天然有序
        ListNode* head2 = divice(head);
        ListNode* left = traversal(head);
        ListNode* right = traversal(head2);
        return merge(left, right);
    }
    ListNode* sortList(ListNode* head) {
        return traversal(head);
```

```
    }
};
```

Q1：三个线程交替打印abc

```
var is_ok chan bool = make(chan bool) //实现顺序的核心 不过没有这个的话 main会不断的丢数据进chan 这
var map_thread map[byte]chan byte = make(map[byte]chan byte, 3)

func PrintChar(c byte) {
        for {
                var char byte = <- map_thread[c]
                fmt.Printf("thread %c : %c \n", c, char)
                is_ok <- true
        }
}

func main() {
        map_thread['a'] = make(chan byte)
        map_thread['b'] = make(chan byte)
        map_thread['c'] = make(chan byte)
        go PrintChar('a')
        go PrintChar('b')
        go PrintChar('c')

        var str string = "abc"
        for i,n := 0,10; i < n; i++ {
                fmt.Printf("This is %d cout \n", i + 1)
                for j,m := 0,len(str); j < m; j++ {
                        var c byte  = str[j]
                        map_thread[c] <- c
                        <- is_ok
                }

        }
}
```

Q2：100个协程顺序打印1-1000数字，并且尾号要对应

```
// 5.开启100个协程，顺序打印1-1000，且保证协程号1的，打印尾数为1的数字
// 思路是使用一个对应关系的容器实现当前数字交给哪一个协程去打印 这个实现最好的就是map了
// 最好采用匿名函数去做 这里不采用匿名函数的话就需要全局变量了
var is_ok chan bool = make(chan bool) // 同步变量 检查是否打印完了
var map_thread map[int]chan int = make(map[int]chan int, 100) //100个哈希表，对应100和协程
func PrintNums(id int) {
        for {
                var num int = <-map_thread[id]
                fmt.Printf("goruntinue%d is printing %d \n",id, num)
                is_ok <- true //写入同步变量 表示已经完成 这个信号量告诉主协程main已经完成一个操
        }
}

func main() {
    // ------------------5.100 goroutinue----------------------
        // 一定要先创建这100个chan 不然就是nil chan 会出现一堆错误
        for i,n := 1,100; i <= n; i++ {
                map_thread[i] = make(chan int)
        }
        //启动100个协程
        for i,n := 1,100; i <= n; i++ {
                go PrintNums(i)
        } //启动100个协程
        // 下面开始分配数字
        for i,n := 1,1000; i <= n; i++ {
                var id int = i % 100
                if id == 0 {
                        id = 100
                }
                map_thread[id] <- i
                <-is_ok
        }
}
```

1114. 按序打印

给你一个类：

```
public class Foo {
public void first() { print("first"); }
public void second() { print("second"); }
public void third() { print("third"); }
```

}
三个不同的线程 A、B、C 将会共用一个 Foo 实例。

线程 A 将会调用 first() 方法
线程 B 将会调用 second() 方法
线程 C 将会调用 third() 方法
请设计修改程序，以确保 second() 方法在 first() 方法之后被执行，third() 方法在 second() 方法之后被执行。

使用mutex和条件变量的方式实现顺序打印，沿用go的设计方式，main函数做一个击鼓传花的作用。c++的并发比golang复杂多了。c++里面的cv.wait()有提前释放锁的功能，不然就等{}代码块执行完毕了。

没有具体代码 参考下一个题目吧

1115. 交替打印 FooBar
给你一个类：

```
class FooBar {
public void foo() {
for (int i = 0; i < n; i++) {
print("foo");
}
}

public void bar() {
for (int i = 0; i < n; i++) {
print("bar");
}
}
}
```

两个不同的线程将会共用一个 FooBar 实例：

线程 A 将会调用 foo() 方法，而
线程 B 将会调用 bar() 方法
请设计修改程序，以确保 "foobar" 被输出 n 次

和上一题一样的道理。复习了lambda表达式的作用，[]接受外部参数，主打一个可见性，() 是传参，主打一个自己赋予变量。

```cpp
#include <iostream>
#include <algorithm>
#include <condition_variable>
#include <mutex>
#include <vector>
#include <thread>

using namespace std;

// 先不用类看看 先直接写两个线程
mutex mtx;
condition_variable cv;
int flag{0};

void Print_Foo(int count) {
    while(count--) {
        unique_lock<mutex> lock(mtx); //尝试获得锁
        cv.wait(lock, [](){ return flag == 1; });
        // cout<<"Print_Foo is running!" <<endl;
        cout<<"foo";
        flag = 2;
        cv.notify_all();
    }
}

void Print_Bar(int count) {
    while(count--) {
        unique_lock<mutex> lock(mtx);
        cv.wait(lock, [](){ return flag == 2; });
        // cout<<"Print_Bar is running!" <<endl;
        cout<<"bar ";
        flag = 0;
        cv.notify_all();
    }
}

int main() {
    int count{5};

    thread t1(&Print_Foo, count);
    thread t2(&Print_Bar, count);

    {
```

```
        while(count --) {
            unique_lock<mutex> lock(mtx);
            flag = 1;
            // cout<<"First function is main."<<endl;
            cv.notify_all();
            cv.wait(lock, [](){ return flag == 0; });
        }
    }

    t1.join(); //想当于打了一个阻塞点在这里 告诉main说 在这里等t1执行完 可以理解成同步原语。
    t2.join();

}
```

代码
测试用例
测试用例
测试结果

1116. 打印零与奇偶数

现有函数 printNumber 可以用一个整数参数调用，并输出该整数到控制台。

例如，调用 printNumber(7) 将会输出 7 到控制台。
给你类 ZeroEvenOdd 的一个实例，该类中有三个函数：zero、even 和 odd 。ZeroEvenOdd 的相同实例将会传递给三个不同线程：

线程 A：调用 zero() ，只输出 0
线程 B：调用 even() ，只输出偶数
线程 C：调用 odd() ，只输出奇数
修改给出的类，以输出序列 "010203040506..." ，其中序列的长度必须为 2n 。

实现 ZeroEvenOdd 类：

ZeroEvenOdd(int n) 用数字 n 初始化对象，表示需要输出的数。
void zero(printNumber) 调用 printNumber 以输出一个 0 。
void even(printNumber) 调用printNumber 以输出偶数。
void odd(printNumber) 调用 printNumber 以输出奇数。

一样的道理，每次执行完一个线程就回到主线程，只是这里的条件变量多一点。/golang go的实现更加符合直觉

```cpp
#include <iostream>
#include <mutex>
#include <condition_variable>
#include <thread>

using namespace std;


// 启动三个线程 和 一个主线程 主线程负责分配哪一个线程工作 每次子线程工作完回到主线程

mutex mtx;
condition_variable cv;
int flag_zero{-1};
int print_number{1};

void PrintZero(int count) {
    for(int i{1}; i <= count; i++) {
        unique_lock<mutex> lock(mtx);
        cv.wait(lock, [](){ return flag_zero == 1; });
        cout<<"0";
        flag_zero = 0;
        cv.notify_all();
        // 出去unique_lock作用域自动释放锁
    }
}
void PrintEven(int count) {
    // while(1) {
    //while(1)会出现线程卡死 因为有一个线程永远不会退出 因为负责击鼓传花的主线程那边已经结束了
    for(int i{1}; i <= count; i = i + 2) {
        unique_lock<mutex> lock(mtx);
        cv.wait(lock, [](){
            if(print_number % 2 != 0 && flag_zero == 0) {
                //奇数
                return true;
            }
            else {
                return false;
            }
        });
        cout<<print_number;
        print_number++;
        flag_zero = -1;
        cv.notify_all();
```

```
        if(print_number > count) break;
    }
}

void PrintOdd(int count) {
    // while(1) {
    for(int i{2}; i <= count; i = i + 2) { //使用这个明确确定打印次数
        unique_lock<mutex> lock(mtx);
        cv.wait(lock, [](){
            if(print_number % 2 == 0 && flag_zero == 0) {
                //偶数
                return true;
            }
            else {
                return false;
            }
        });
        cout<<print_number;
        print_number++;
        flag_zero = -1;
        cv.notify_all();
        if(print_number > count) break;
    }
}

int main() {
    // main作击鼓传花的作用
    int count{5};

    thread t1(&PrintZero,count);
    thread t2(&PrintEven,count);
    thread t3(&PrintOdd,count);

    for(int i{0}; i < count; i++) {
        unique_lock<mutex> lock(mtx);
        cv.wait(lock,[](){ return flag_zero == -1; });
        flag_zero = 1;
        cv.notify_all();
    }

    t1.join();
    t2.join();
    t3.join();
```

```
    cout<<endl;
}
```

42. 接雨水

给定 n 个非负整数表示每个宽度为 1 的柱子的高度图，计算按此排列的柱子，下雨之后能接多少雨水。

双dp，好做。不要使用单调栈的做法，不会。

```
class Solution {
public:
    int trap(vector<int>& height) {
        // 当前节点格子能够接到的最大雨水和其左右的柱子的最大高度和自身的高度有关
        // 那么维护两个数组 分别获得当前位置左右两边最大柱子的高度
        // 最后统计所有的雨水总量就是答案
        vector<int> leftheight(height.size(), 0);
        vector<int> rightheight(height.size(), 0);
        int maxwater{0};
        leftheight[0] = height[0];
        rightheight[height.size() - 1] = height[height.size() - 1];
        for(int i{1}; i<height.size(); i++) {
            leftheight[i] = max(leftheight[i - 1], height[i]);
        }
        for(int i = height.size() - 2; i>=0; i--) {
            rightheight[i] = max(rightheight[i + 1], height[i]);
        }
        for(int i{0}; i<height.size(); i++) {
            int curwater = min(leftheight[i], rightheight[i]) * 1 - height[i];
            maxwater = maxwater + curwater;
        }
        return maxwater;
    }
};
```

438. 找到字符串中所有字母异位词

给定两个字符串 s 和 p，找到 s 中所有 p 的 异位词 的子串，返回这些子串的起始索引。不考虑答案输出的顺序。

这个umap[c] > pmap[c]包含了两层意思 就是一个是c存在两者而且c的数量比右边的多；第二种情况就是只有左边有c，这个时候查询右边会是end，但是会写入一个0，因为有这个pmap[c]赋值的操作。

```cpp
class Solution {
public:
    vector<int> findAnagrams(string s, string p) {
        unordered_map<char,int> umap;
        unordered_map<char,int> pmap;
        vector<int> result;
        for(char& c : p) {
            pmap[c]++;
        }
        int left{0};
        int right{0};
        while(right < s.size()) {
            // 1.进入窗口
            char c = s[right];
            umap[c]++;
            // 2.维护窗口
            while(umap[c] > pmap[c]) {
                char left_char = s[left];
                umap[left_char]--;
                left++;
            }
            // 3.更新答案 移动
            if(right - left + 1 == p.size()) result.push_back(left);
            right++;
        }
        return result;
    }
};
```

155. 最小栈

设计一个支持 push ， pop ， top 操作，并能在常数时间内检索到最小元素的栈。

实现 MinStack 类:

MinStack() 初始化堆栈对象。
void push(int val) 将元素val推入堆栈。
void pop() 删除堆栈顶部的元素。
int top() 获取堆栈顶部的元素。
int getMin() 获取堆栈中的最小元素。

双栈实现最小栈st和st_min。类似一个数组，我们要得到当前下标i为止的最小值，就像一个动态规划一个一个移动，我们需要另外一个数组，从头开始一个一个递增下标的去比较。这个另外的数组就是st_min。每次压入一个元素，我们把这个元素和以往的最小值比较（以往的最小值就是st_min栈顶的元素了，就是另外一个数组的最新下标的值）。然后一起压入两个栈顶。pop的时候也一起pop。那么一开始st_min是空的这个时候无法和栈顶元素比较，因为空栈，所以这个时候需要压入一个最大值，怎么比第一个元素都是第一个元素。

```
// 双栈小子会出答案
class MinStack {
private:
    stack<int> st;
    stack<int> st_min;
public:
    MinStack() {
        st_min.push(INT_MAX);
    }

    void push(int val) {
        int num = st_min.top();
        int min_num = min(val, num);
        st.push(val);
        st_min.push(min_num);
    }

    void pop() {
        st.pop();
        st_min.pop();
    }

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

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

347. 前 K 个高频元素

给你一个整数数组 nums 和一个整数 k ，请你返回其中出现频率前 k 高的元素。你可以按 任意顺序 返回答案

1.map统计频率；2.基于频率建桶，每个桶收集一个频率的数值；3.vector直接拷贝一个vector就使用insert；注意最后是i >0 && result.size() < k.因为可能会很稀疏。

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int,int> umap;//数值-频率
        int max_fre{0};
        for(int& i : nums) {
            umap[i]++;
            max_fre = max(max_fre, umap[i]);
        }
        //建桶
        vector<vector<int>> bucket(max_fre + 1);
        for(auto it = umap.begin(); it != umap.end(); it++) {
            int fre = it->second;
            int num = it->first;
            bucket[fre].push_back(num);
        }
        vector<int> result;
        for(int i = bucket.size() - 1; i > 0 && result.size() < k; i--) {
            result.insert(result.end(), bucket[i].begin(), bucket[i].end());
        }
        return result;
    }
};
```

461. 汉明距离

两个整数之间的 汉明距离 指的是这两个数字对应二进制位不同的位置的数目。

给你两个整数 x 和 y，计算并返回它们之间的汉明距离。

1.异或得到答案的十进制数 2.使用BK算法计算得到这个答案里面1的个数，就是汉明距离。
ps：BK算法就是res = res & (res - 1) 每次运算都会去掉最右边的一个1.

```
class Solution {
public:
    int hammingDistance(int x, int y) {
        // 异或的运算关系
        int res =  x ^ y;
        int result{0};
        while(res > 0) {
            res = res & (res - 1);
            result++;
        }
        return result;
    }
};
```

169. 多数元素

给定一个大小为 n 的数组 nums ，返回其中的多数元素。多数元素是指在数组中出现次数 大于 ⌊ n/2 ⌋ 的元素。

你可以假设数组是非空的，并且给定的数组总是存在多数元素。

1.快排找处于数组中间的位置的数字

```go
func Sort(nums []int, left int, right int) int {
    var base int = nums[left];
    for left < right {
        for left < right && nums[right] >= base {
            right--;
        }
        nums[left], nums[right] = nums[right], nums[left];
        for left < right && nums[left] <= base {
            left++;
        }
        nums[left], nums[right] = nums[right], nums[left];
    }
    nums[left] = base;
    return left;
}

func QuickSort(nums []int, left int, right int, y int) {
    // 递归-分治
    if left >= right {
        return ; //递归结束
    }
    k := Sort(nums,left,right);
    if k < y {
        QuickSort(nums, k + 1, right, y);
    } else if k > y {
        QuickSort(nums, left, k - 1, y);
    } else {
        return;
    }
}
func majorityElement(nums []int) int {
    // 快选 快排的进化
    var y int = len(nums) / 2;
    QuickSort(nums, 0, len(nums) - 1, y);
    return nums[y];
}
```

150. 逆波兰表达式求值

给你一个字符串数组 tokens ，表示一个根据 逆波兰表示法 表示的算术表达式。

请你计算该表达式。返回一个表示表达式值的整数。

注意：

有效的算符为 '+'、'-'、'*' 和 '/' 。
每个操作数（运算对象）都可以是一个整数或者另一个表达式。
两个整数之间的除法总是 向零截断 。
表达式中不含除零运算。
输入是一个根据逆波兰表示法表示的算术表达式。
答案及所有中间计算结果可以用 32 位 整数表示。

遇到数字就入栈就行了，然后遇到运算符就取出两个数字运算，最后把结果压入栈。没啥好说的。

```
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        stack<int> st;
        for(int i{0};i < tokens.size(); i++) {
            if(tokens[i] == "+") {
                int second = st.top();
                st.pop();
                int first = st.top();
                st.pop();
                st.push(first + second);
            }
            else if(tokens[i] == "-") {
                int second = st.top();
                st.pop();
                int first = st.top();
                st.pop();
                st.push(first - second);
            }
            else if(tokens[i] == "*") {
                int second = st.top();
                st.pop();
                int first = st.top();
                st.pop();
                st.push(first * second);
            }
            else if(tokens[i] == "/") {
                int second = st.top();
                st.pop();
                int first = st.top();
                st.pop();
                st.push(first / second);
            }
            else st.push(stoi(tokens[i]));
        }
        return st.top();
    }
};
```

394. 字符串解码

给定一个经过编码的字符串，返回它解码后的字符串。

编码规则为: k[encoded_string]，表示其中方括号内部的 encoded_string 正好重复 k 次。注意 k 保证为正整数。

你可以认为输入字符串总是有效的；输入字符串中没有额外的空格，且输入的方括号总是符合格式要求的。

此外，你可以认为原始数据不包含数字，所有的数字只表示重复的次数 k ，例如不会出现像 3a 或 2[4] 的输入。

1.这道题目是使用栈模拟递归 2.需要明确的两个点是遇到左括号就是说明进到下一层递归了，这个时候需要保存上一层的变量，然后进入下一层 3.遇到右括号说明要退出当前层了，这个时候需要重复当前层的字符串然后返回给上一层拼接起来，指的注意的是上一层的字符串保存在了st_str的栈顶了（这是你犯错误的地方）。重复的次数就是上一层栈顶的数字了。这到题目思路一定要清洗。

```cpp
class Solution {
public:
    string decodeString(string s) {
    // 就是用栈去模拟递归
    // 在进入[的时候就是进入了一层新的递归 这个时候需要保存一个当前的一个变量
    // 然后去处理这个新的一层的变量
    // 当遇到 ]的时候就是退出了这一层递归 这个时候就需要把当前层的结果拼接到上一层的结果
    // 双栈 遇到数字进数字栈 字母进去字母栈
    // 上一层的数字就是当前层需要重复的一个次数
    stack<string> st_str;
    stack<int> st_num;
    int num{0};
    string result;
    for(char& c : s) {
        if(c >= '0' && c <= '9') {
            // 数字
            int temp = c - '0';
            num = num * 10 + temp;
        }
        else if(c >= 'a' && c <= 'z') {
            // 字母
            result =  result + c; //拼在一起
        } else if(c == '[') {
            // 左括号说明进入下一层递归 需要保存当前变量
            st_str.push(result);
            result = "";
            st_num.push(num);
            num = 0;
        } else {
            // 右括号 说明当前递归结束 开始重复和返回当前层结果
            int count = st_num.top();
            st_num.pop();
            string this_result{""};
            while(count--) {
                this_result += result;
            }
            // 此处的this_result就是当前层结果 上层结果last_result就在栈中
            string last_result = st_str.top();
            st_str.pop();
            result = last_result + this_result;
        }
    }
    return result;
```

```
    }
};
```

239. 滑动窗口最大值

给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。

返回 滑动窗口中的最大值 。

1.单调递减的双端队列仅仅做一个存储候选最大值的作用，不能用它的大小来判断滑动窗口的大小，事实是滑动窗口只要在i>k的时候就会一直形成，要一直从deque的队首取出答案。

2.具体做法就是遍历到当前下标i的时候和deque队尾的元素判断大小，维护队列的单调递减，小于直接压入队尾，不然一直弹出队尾元素直到满足条件。之后压入一个元素之后就要判断队首是否还在窗口内了，判断依据就是看当前下标i和deque.front()之间的距离是否超过了k。超过就弹出队首元素。最后从队首取出极大值压入答案。

3.注意存储的是下标，而且窗口形成的条件是i - 0 + 1 >= k。
/golang 加深了对[]int的使用

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    // 这是一个降本增效的故事
    // 维护一个单调减双端队列 这个队列用来存储元素 这个队列的作用是存储候选极大值 是一个很有意思的作用
    // 遍历到当前元素的时候 和队列尾部元素比较 如果小于那就直接压入 不然的话一直弹出直到队列满足
    // 还要检查队首的下标是否已经离开了窗口 如果离开了那么也要弹出队首
    // 当这个容器长度为k的时候 队首就是答案？错误 当窗口开始形成的时候 就需要从这个队列中取出元素填入
    deque<int> de;
    vector<int> result;
    for(int i{0}; i < nums.size(); i++) {
        while(de.empty() == false && nums[i] >= nums[de.back()]) {
            // 需要弹出队尾元素
            de.pop_back();
        }
        // 压入当前的下标
        de.push_back(i);
        // 判断队首是否已经离开了窗口
        while(i - de.front() + 1 > k) de.pop_front();
        // 这里有一个很有意思的问题 就是deque仅仅只是一个候选最大值的作用 他并不能表示当前窗口的大小
        // 也就是说deque里面存储的下标不一定合法 仅仅是候选的最大值
        if(i - 0 + 1 >= k) result.push_back(nums[de.front()]); // 窗口开始形成 i > k 就一定形成了
    }
    return result;
    }
};
```

102. 二叉树的层序遍历

给你二叉树的根节点 root ，返回其节点值的 层序遍历 。 （即逐层地，从左到右访问所有节点）。

1.使用队列存储当前根节点，然后在这个队列中一边出队获得当前节点的数值然后一边入队当前节点的左右孩子。/golang

```
func levelOrder(root *TreeNode) [][]int {
    if root == nil {
        return [][]int{};
    }
    var result [][]int = make([][]int, 0);
         var queue []*TreeNode = make([]*TreeNode, 0);
         // 1.头节点入队
         queue = append(queue, root);
         for len(queue) > 0 {
                  var size int = len(queue);
        var ans []int = make([]int, 0);
                  for i,n := 0, size; i < n; i++ {
                           var node *TreeNode = queue[0];
                           queue = queue[1:]; //出队
                           ans = append(ans, node.Val);
                           // 入队
                           if node.Left != nil {
                                    queue = append(queue, node.Left);
                           }
                           if node.Right != nil {
                                    queue = append(queue, node.Right);
                           }
                  }
        result = append(result, ans);
          }
    return result;
}
```

11. 盛最多水的容器

给定一个长度为 n 的整数数组 height 。有 n 条垂线，第 i 条线的两个端点是 (i, 0) 和 (i, height[i]) 。

找出其中的两条线，使得它们与 x 轴共同构成的容器可以容纳最多的水。

返回容器可以储存的最大水量。

说明：你不能倾斜容器。

居然秒了bro

1.双指针遍历头尾

2.谁小移动谁 同时使用一个数值计算当前的最大蓄水量 直到两个指针相遇 因为短板不移动 永远都得不到更加好的一个数值。

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        // 双指针遍历头尾
        // 谁小移动谁 同时使用一个数值计算当前的最大蓄水量 直到两个指针相遇
        int left{0};
        int right = height.size() - 1;
        int result{0};
        while(left <= right) {
            int gap = right - left;
            int now_water = min(height[left], height[right]) * gap;
            result = max(result, now_water);
            if(height[left] < height[right]) left++;
            else right--;
        }
        return result;
    }
};
```

62. 不同路径

一个机器人位于一个 m x n 网格的左上角 （起始点在下图中标记为 “Start” ）。

机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为 “Finish” ）。

问总共有多少条不同的路径?

1.简单二维dp，分类加法原理。

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        // 就是一个分类加法 因为有上和左的来路
        vector<vector<int>> dp(m, vector<int>(n, 0));
        // dp[0][j] = 1 因为第一行只能从左边来
        for(int j{0}; j < n; j++) {
            dp[0][j] = 1;
        }
        // dp[i][0] = 1 因为第一列只能从上边来
        for(int i{0}; i < m; i++) {
            dp[i][0] = 1;
        }
        for(int i{1}; i < m; i++) {
            for(int j{1}; j < n; j++) {
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }
        return dp[m - 1][n - 1];
    }
};
```

LCR 056. 两数之和 IV - 输入二叉搜索树

给定一个二叉搜索树的 根节点 root 和一个整数 k , 请判断该二叉搜索树中是否存在两个节点它们的值之和等于 k 。假设二叉搜索树中节点的值均唯一。

1.简单前序遍历+哈希表的一个两数之和。

```
class Solution {
private:
    unordered_map<int,int> tree_map;
public:
    void traversal(TreeNode* root, int k, bool& fage) {
        // 前序遍历
        if(root == nullptr) return ;
        // 当前处理逻辑
        int find_num = k - root->val;
        if(tree_map.find(find_num) != tree_map.end()) {
            fage = true;
            return ;
        }
        tree_map[root->val]++; //存入
        traversal(root->left, k, fage);
        traversal(root->right, k, fage);
    }
    bool findTarget(TreeNode* root, int k) {
        // 递归 使用一个map进行存储当前遍历过的节点的数值 但是回溯的时候需要去掉这个嘛？ 不需要遍历过
        bool fage = false;
        traversal(root,k,fage);
        return fage;
    }
};
```

48. 旋转图像

给定一个 n × n 的二维矩阵 matrix 表示一个图像。请你将图像顺时针旋转 90 度。

你必须在 原地 旋转图像，这意味着你需要直接修改输入的二维矩阵。请不要 使用另一个矩阵来旋转图像。

1.难以理解的一道题目，采用的方法类似螺旋矩阵的遍历，一圈一圈的交换顺序，必须用swap，不然爆炸了

2.上右顺序对应交换；

3.下左顺序对应交换；

4.上下逆序交叉交换（从right开始到left，而且位置是交叉对应的）；

5.up++；down--；left++；right--；更新到下一圈，巨逆天的一道题目。

```
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        // 一个思路就是上右交换
        // 下左交换
        // 之后上下交换
        // 就完成了一圈的旋转了
        // 就是使用swap 不然我不会做了
        int left{0};
        int right = matrix[0].size() - 1;
        int up{0};
        int down = matrix.size() - 1;
        while(left <= right && up <= down) {
            // 上右交换 顺序对应位置
            for(int i = left;i<right;i++) swap(matrix[up][i],matrix[i][right]);
            // 下左交换 顺序对应位置
            for(int i{right};i>left;i--) swap(matrix[down][i],matrix[i][left]);
            // 上下交换 交叉反序位置
            for(int i = right;i>left;i--) swap(matrix[up][matrix[0].size() - 1 - i],matrix[down
            left++;
            right--;
            up++;
            down--;
        }
    }
};
```

593. 有效的正方形

给定2D空间中四个点的坐标 p1, p2, p3 和 p4，如果这四个点构成一个正方形，则返回 true 。

点的坐标 pi 表示为 [xi, yi] 。 输入没有任何顺序 。

一个 有效的正方形 有四条等边和四个等角(90度角)。

1.计算六条边+排序这六条边+勾股定理。简单的思路很丑陋的代码。

```
class Solution {
public:
    bool validSquare(vector<int>& p1, vector<int>& p2, vector<int>& p3, vector<int>& p4) {
        // 计算六条边 四条边 加上 两个对角线 四条边相等的情况下进行勾股定理判断 满足的就是正方形
        // p1-p2
        // |  |
        // p3-p4
        vector<double>length;
        double p1Top2 = ((p1[0] - p2[0]) * (p1[0] - p2[0]) + (p1[1] - p2[1]) * (p1[1] - p2[1]))
        length.push_back(p1Top2);
        double p1Top3 = ((p1[0] - p3[0]) * (p1[0] - p3[0]) + (p1[1] - p3[1]) * (p1[1] - p3[1]))
        length.push_back(p1Top3);
        double p2Top4 = ((p2[0] - p4[0]) * (p2[0] - p4[0]) + (p2[1] - p4[1]) * (p2[1] - p4[1]))
        length.push_back(p2Top4);
        double p3Top4 = ((p3[0] - p4[0]) * (p3[0] - p4[0]) + (p3[1] - p4[1]) * (p3[1] - p4[1]))
        length.push_back(p3Top4);
        double p1Top4 = ((p1[0] - p4[0]) * (p1[0] - p4[0]) + (p1[1] - p4[1]) * (p1[1] - p4[1]))
        length.push_back(p1Top4);
        double p2Top3 = ((p2[0] - p3[0]) * (p2[0] - p3[0]) + (p2[1] - p3[1]) * (p2[1] - p3[1]))
        length.push_back(p2Top3);
        // 排序获得四条边和后两条边之后勾股定理
        sort(length.begin(),length.end());
        double d1 = length[0];
        double d2 = length[1];
        double d3 = length[2];
        double d4 = length[3];
        double dd1 = length[4]; //对角线
        double dd2 = length[5]; //对角线
        if(d1 == 0) return false; //若是最小的边都是0的话那么必然不可能组成正方形。
        if(d1 == d2 && d1 == d3 && d1 == d4 && (d1 + d2 == dd1 && d3 + d4 == dd2)) return true;
        return false;
    }
};
```

54. 螺旋矩阵

给你一个 m 行 n 列的矩阵 matrix ，请按照 顺时针螺旋顺序 ，返回矩阵中的所有元素。

1.起猛了，直接秒了。直接使用螺旋矩阵的思路就行了。

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        int row = matrix.size(); //行数
        int col{0};
        if(matrix.empty() == false) {
            col = matrix[0].size(); //列数
        }
        int left_col{0};
        int right_col = col - 1;
        int up_row{0};
        int down_row = row - 1;
        vector<int> ans;
        while(1) {
            for(int j{left_col};j<=right_col;j++) ans.push_back(matrix[up_row][j]);
            up_row++;
            if(up_row > down_row) break;
            for(int i{up_row};i<=down_row;i++) ans.push_back(matrix[i][right_col]);
            right_col--;
            if(right_col < left_col) break;
            for(int j{right_col};j>=left_col;j--) ans.push_back(matrix[down_row][j]);
            down_row--;
            if(down_row < up_row) break;
            for(int i{down_row};i>=up_row;i--) ans.push_back(matrix[i][left_col]);
            left_col++;
            if(left_col > right_col) break;
        }
        return ans;
    }
};
```

59. 螺旋矩阵 II

给你一个正整数 n ，生成一个包含 1 到 n2 所有元素，且元素按顺时针顺序螺旋排列的 n x n 正方形矩阵 matrix 。

1.复用上面思路依旧秒了。

```cpp
class Solution {
public:
    vector<vector<int>> generateMatrix(int n) {
        // 依旧螺旋矩阵
        vector<vector<int>> result(n, vector<int>(n, 0));
        int row = n;
        int col = n;
        int left = 0;
        int right = col - 1;
        int up = 0;
        int down = row - 1;
        int num = 1;
        while(left <= right && up <= down) {
            for(int j{left}; j <= right; j++) result[up][j] = num++;
            up++;
            if(up > down) break;
            for(int i{up}; i <= down; i++) result[i][right] = num++;
            right--;
            if(right < left) break;
            for(int j{right}; j >= left; j--) result[down][j] = num++;
            down--;
            if(down < up) break;
            for(int i{down}; i >= up; i--) result[i][left] = num++;
            left++;
            if(left > right) break;
        }
        return result;
    }
};
```