76. 最小覆盖子串（困难）

1. 题目描述

   给你一个字符串 S、一个字符串 T，请在字符串 S 里面找出：包含 T 所有字母的最小子串。

   ```
   示例:
   输入: S = "ADOBECODEBANC", T = "ABC"
   输出: "BANC"
   ```

   说明:

   - 如果 S 中不存这样的子串，则返回空字符串 ""。
   - 如果 S 中存在这样的子串，我们保证它是唯一的答案。

2. 简单实现

   j为左窗口， i为右窗口，统计T里面字母出现的次数，每次出现的字母次数都要-1

   如果s[j]中的字母次数小于0，说明出现的次数超过需求，那么j就要向右移动（这里很神奇的一点在于，把不出现在t中的字母的计数需求设为0，就可以所有字母一视同仁地去做）

   ```
   class Solution {
   public:
       string minWindow(string s, string t) {
           unordered_map<char, int> hash;
           for(auto c : t) hash[c]++;
           int cnt = hash.size();
           int ans_l = -1;
           int ans_len = INT_MAX;
           for(int i = 0 , j = 0, c = 0; i < s.size();i++) {
               if(hash[s[i]] == 1) c++;//该字母包含在t中且出现次数达标
               hash[s[i]]--;
               while(hash[s[j]] < 0)//左窗口可以右移
                   hash[s[j++]]++;
               if(c == cnt && ans_len > i - j + 1){
                   ans_len = i - j + 1;
                   ans_l = j;
               }
           }
           if(ans_l >= 0)
               return s.substr(ans_l, ans_len);
           else
               return "";
       }
   };
   ```

33. 搜索旋转排序数组（中等）

1. 题目描述

假设按照升序排序的数组在预先未知的某个点上进行了旋转。

(例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2])。

搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回 -1 。

你可以假设数组中不存在重复的元素。

你的算法时间复杂度必须是 O(log n) 级别。

```
示例 1:
输入: nums = [4,5,6,7,0,1,2], target = 0
输出: 4

示例 2:
输入: nums = [4,5,6,7,0,1,2], target = 3
输出: -1
```

2. 简单实现

细心分情况讨论

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int l = 0, r = nums.size() - 1;
        while(l <= r){
            int mid = l + (r - l) / 2;
            if(nums[mid] == target) return mid;
            else if(nums[mid] < target){
                if(nums[l] < nums[r])//l-r升序
                    l = mid + 1;
                else{
                    if((target>nums[r] && nums[mid]>=nums[l]) //target和nums[mid]都
在左半区
                       || (target<=nums[r] && nums[mid]<nums[l]))//target和
nums[mid]都在右半区
                        l = mid + 1;
                    else
                        r = mid - 1;
                }
            }
            else{
                if(nums[l] < nums[r])
                    r = mid - 1;
                else{
                    if((target>nums[r] && nums[mid]>=nums[l]) //target和nums[mid]都
在左半区
                       || (target<=nums[r] && nums[mid]<nums[l]))//target和
nums[mid]都在右半区
                        r = mid - 1;
                    else
                        l = mid + 1;
                }
            }
```

```
            }
            return -1;
        }
    };
```

410. 分割数组的最大值（困难）

1. 题目描述

   给定一个非负整数数组和一个整数 m，你需要将这个数组分成 m 个非空的连续子数组。设计一个算法使得这 m 个子数组各自和的最大值最小。

   注意: 数组长度 n 满足以下条件:

   - 1 ≤ n ≤ 1000
   - 1 ≤ m ≤ min(50, n)

```
示例:
输入:
nums = [7,2,5,10,8]
m = 2
输出:
18
解释:
一共有四种方法将nums分割为2个子数组。
其中最好的方式是将其分为[7,2,5] 和 [10,8]，
因为此时这两个子数组各自的和的最大值为18，在所有情况中最小。
```

2. 简单实现

   二分法，突然觉得这种会分割成若干份的有可能就是二分法，之前kick start的一道题也是这种的

```
class Solution {
public:
    int splitArray(vector<int>& nums, int m) {
        int l = nums[0];
        long r = nums[0];
        for(int i = 1; i < nums.size(); i++){
            if(nums[i] > l) l = nums[i];
            r += nums[i];
        }
        while(l < r){
            int mid = l + (r - l) / 2;
            int cnt = 1;//初始化为1
            long temp = 0;
            for(int i = 0; i < nums.size(); i++){
                temp += nums[i];
                if(temp > mid){
                    temp = nums[i];
                    cnt++;
                }
            }
            //cnt==m不代表此时mid是最小的，可能mid-1的cnt也等于m
```

```
            if(cnt > m) l = mid+1;
            else r = mid;
        }
        return l;
    }
};
```

206. 反转链表（简单）

1. 题目描述

   反转一个单链表。

   ```
   示例:
   输入: 1->2->3->4->5->NULL
   输出: 5->4->3->2->1->NULL
   ```

   进阶: 你可以迭代或递归地反转链表。你能否用两种方法解决这道题?

2. 简单实现

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        if(!head || !head->next) return head;
        ListNode* pre = NULL;
        ListNode* cur = head;
        ListNode* next = head->next;
        while(cur){
            cur->next = pre;
            pre = cur;
            cur = next;
            if(next)
                next = next->next;
        }
        return pre;
    }
};
```

297. 二叉树的序列化与反序列化（困难）

1. 题目描述

   序列化是将一个数据结构或者对象转换为连续的比特位的操作，进而可以将转换后的数据存储在一个文件或者内存中，同时也可以通过网络传输到另一个计算机环境，采取相反方式重构得到原数据。

   请设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

   示例:

   你可以将以下二叉树:

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

序列化为 "[1,2,3,null,null,4,5]"
```

提示: 这与 LeetCode 目前使用的方式一致，详情请参阅 LeetCode 序列化二叉树的格式。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。

说明: 不要使用类的成员 / 全局 / 静态变量来存储状态，你的序列化和反序列化算法应该是无状态的。

2. 简单实现

```
    class Codec {
public:
    vector<string> split(string s){
        int l = 0, r = 0;
        vector<string> ans;
        while(r < s.size()){
            if(s[r] == ','){
                ans.push_back(s.substr(l, r-l));
                l = r+1;
                r++;
            }
            else r++;
        }
        ans.push_back(s.substr(l, r-l));
        return ans;
    }

    // Encodes a tree to a single string.
    string serialize(TreeNode* root) {
        if(!root) return "null";
        string ans = "";
        queue<TreeNode*> q;
        q.push(root);
        bool end = false;
        while(!end && !q.empty()){
            int size = q.size();
            end = true;
            for(int i = 0; i < size; i++){
                TreeNode* cur = q.front();
                q.pop();
                if(!cur)
                    ans += ",null";
                else{
                    ans += "," + to_string(cur->val);
                    q.push(cur->left);
                    q.push(cur->right);
                    if(cur->left || cur->right)
                        end = false;
```

```
                }
            }
        }
        return ans.substr(1,ans.size()-1);
    }

    // Decodes your encoded data to tree.
    TreeNode* deserialize(string data) {
        if(data == "null") return NULL;
        vector<string> nodes = split(data);
        TreeNode* root = new TreeNode(stoi(nodes[0]));
        queue<TreeNode*> q;
        q.push(root);
        int i = 1;
        while(i < nodes.size()){
            TreeNode* cur = q.front();
            q.pop();
            if(nodes[i] != "null"){
                cur->left = new TreeNode(stoi(nodes[i]));
                q.push(cur->left);
            }
            i++;
            if(nodes[i] != "null"){
                cur->right = new TreeNode(stoi(nodes[i]));
                q.push(cur->right);
            }
            i++;
        }
        return root;
    }
};
```

289. 生命游戏（中等）

1. 题目描述

   根据 百度百科 ，生命游戏，简称为生命，是英国数学家约翰·何顿·康威在 1970 年发明的细胞自动机。

   给定一个包含 m × n 个格子的面板，每一个格子都可以看成是一个细胞。每个细胞都具有一个初始状态：1 即为活细胞（live），或 0 即为死细胞（dead）。每个细胞与其八个相邻位置（水平，垂直，对角线）的细胞都遵循以下四条生存定律：

   - 如果活细胞周围八个位置的活细胞数少于两个，则该位置活细胞死亡；
   - 如果活细胞周围八个位置有两个或三个活细胞，则该位置活细胞仍然存活；
   - 如果活细胞周围八个位置有超过三个活细胞，则该位置活细胞死亡；
   - 如果死细胞周围正好有三个活细胞，则该位置死细胞复活；

   根据当前状态，写一个函数来计算面板上所有细胞的下一个（一次更新后的）状态。下一个状态是通过将上述规则同时应用于当前状态下的每个细胞所形成的，其中细胞的出生和死亡是同时发生的。

   ```
   示例:
   输入:
   [
   ```

```
  [0,1,0],
  [0,0,1],
  [1,1,1],
  [0,0,0]
]
输出:
[
  [0,0,0],
  [1,0,1],
  [0,1,1],
  [0,1,0]
]
```

进阶:

- 你可以使用原地算法解决本题吗？请注意，面板上所有格子需要同时被更新：你不能先更新某些格子，然后使用它们的更新后的值再更新其他格子。
- 本题中，我们使用二维数组来表示面板。原则上，面板是无限的，但当活细胞侵占了面板边界时会造成问题。你将如何解决这些问题？

2. 简单实现

```
class Solution {
public:
    int m;
    int n;

    void countLive(vector<vector<int>>& board, int x, int y){
        int cnt = 1;
        for(int i = x-1; i <= x+1; i++)
            for(int j = y-1; j <= y+1; j++)
                if(!(i == x && j == y) && (i >= 0 && i < m) && (j >= 0 && j < n) &&
(board[i][j] > 0))
                    cnt++;
        // cnt++;
        if(board[x][y] <= 0)//死细胞用负数计数
            cnt = -cnt;
        board[x][y] = cnt;
    }
    void gameOfLife(vector<vector<int>>& board) {
        m = board.size();
        if(m <= 0) return;
        n = board[0].size();
        if(n <= 0) return;
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                countLive(board, i, j);
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++){
                int cur = board[i][j];
                if(cur > 0){//活细胞(>1)||周围没有活细胞(=1)
                    cur--;
                    if(cur < 2 || cur > 3) board[i][j] = 0;
```

```
                    else board[i][j] = 1;
                }
                else{//死细胞
                    cur++;
                    if(cur == -3) board[i][j] = 1;
                    else board[i][j] = 0;
                }
            }
        }
};
```

299. 猜数字游戏（简单）

1. 题目描述

   你正在和你的朋友玩 猜数字（Bulls and Cows）游戏：你写下一个数字让你的朋友猜。每次他猜测后，你给他一个提示，告诉他有多少位数字和确切位置都猜对了（称为“Bulls”, 公牛），有多少位数字猜对了但是位置不对（称为“Cows”, 奶牛）。你的朋友将会根据提示继续猜，直到猜出秘密数字。

   请写出一个根据秘密数字和朋友的猜测数返回提示的函数，用 A 表示公牛，用 B 表示奶牛。

   请注意秘密数字和朋友的猜测数都可能含有重复数字。

```
示例 1:
输入: secret = "1807", guess = "7810"
输出: "1A3B"
解释: 1 公牛和 3 奶牛。公牛是 8，奶牛是 0，1 和 7。

示例 2:
输入: secret = "1123", guess = "0111"
输出: "1A1B"
解释: 朋友猜测数中的第一个 1 是公牛，第二个或第三个 1 可被视为奶牛。
```

   说明: 你可以假设秘密数字和朋友的猜测数都只包含数字，并且它们的长度永远相等。

2. 简单实现

```
class Solution {
public:
    string getHint(string secret, string guess) {
        unordered_map<char, int> cnt;//secret中各字符数量
        for(int i = 0; i < secret.size(); i++)
            cnt[secret[i]]++;
        int A = 0, B = 0;
        for(int i = 0; i < guess.size(); i++){
            if(guess[i] == secret[i]){//公牛
                A++;
                if(cnt[guess[i]] > 0)
                    cnt[guess[i]]--;
                else//前面出现过奶牛，抢占了当前公牛的位置，例如"123"与"433"遍历到第三位时
                    B--;
            }
            else if(cnt[guess[i]] > 0){//奶牛
```

```
                cnt[guess[i]]--;
                B++;
            }
        }
        return to_string(A) + 'A' + to_string(B) + 'B';
    }
};
```

226. 翻转二叉树（简单）

1. 题目描述

   翻转一棵二叉树。

   示例：

```
输入：
     4
   /   \
  2     7
 / \   / \
1   3 6   9
输出：
     4
   /   \
  7     2
 / \   / \
9   6 3   1
```

   备注: 这个问题是受到 Max Howell 的 原问题 启发的 ：

   - 谷歌：我们90%的工程师使用您编写的软件(Homebrew)，但是您却无法在面试时在白板上写出翻转二叉树这道题，这太糟糕了。

2. 简单实现——递归

```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if(!root) return root;
        TreeNode* l = invertTree(root->right);
        TreeNode* r = invertTree(root->left);
        root->left = l;
        root->right = r;
        return root;
    }
};
```

3. 简单实现——层次遍历

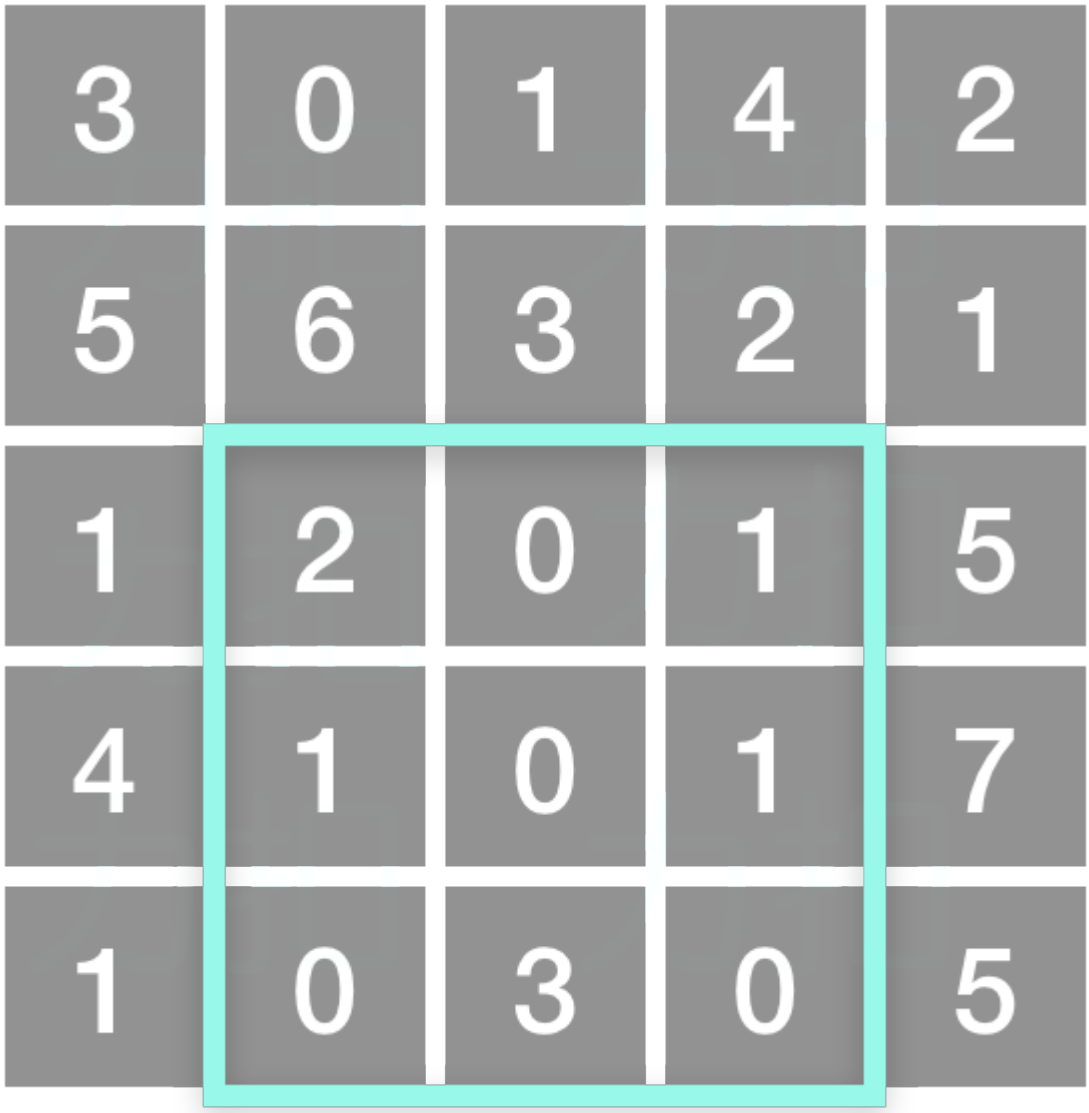
```
class Solution {
public:
```

```
    TreeNode* invertTree(TreeNode* root) {
        if(!root) return root;
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()){
            int size = q.size();
            for(int i = 0; i < size; i++){
                TreeNode* cur = q.front();
                q.pop();
                swap(cur->left, cur->right);//交换左右子树
                if(cur->left)
                    q.push(cur->left);
                if(cur->right)
                    q.push(cur->right);
            }
        }
        return root;
    }
};
```

304. 二位区域和检索 - 矩阵不可变（中等）

1. 题目描述

   给定一个二维矩阵，计算其子矩形范围内元素的总和，该子矩阵的左上角为 (row1, col1) ，右下角为 (row2, col2)。



上图子矩阵左上角 (row1, col1) = (2, 1) ，右下角(row2, col2) = (4, 3)，该子矩形内元素的总和为 8。

```
示例:
给定 matrix = [
  [3, 0, 1, 4, 2],
  [5, 6, 3, 2, 1],
  [1, 2, 0, 1, 5],
  [4, 1, 0, 1, 7],
  [1, 0, 3, 0, 5]
]
sumRegion(2, 1, 4, 3) -> 8
sumRegion(1, 1, 2, 2) -> 11
sumRegion(1, 2, 2, 4) -> 12
```

说明:

- 你可以假设矩阵不可变。
- 会多次调用 sumRegion 方法。
- 你可以假设 row1 ≤ row2 且 col1 ≤ col2。

2. 简单实现

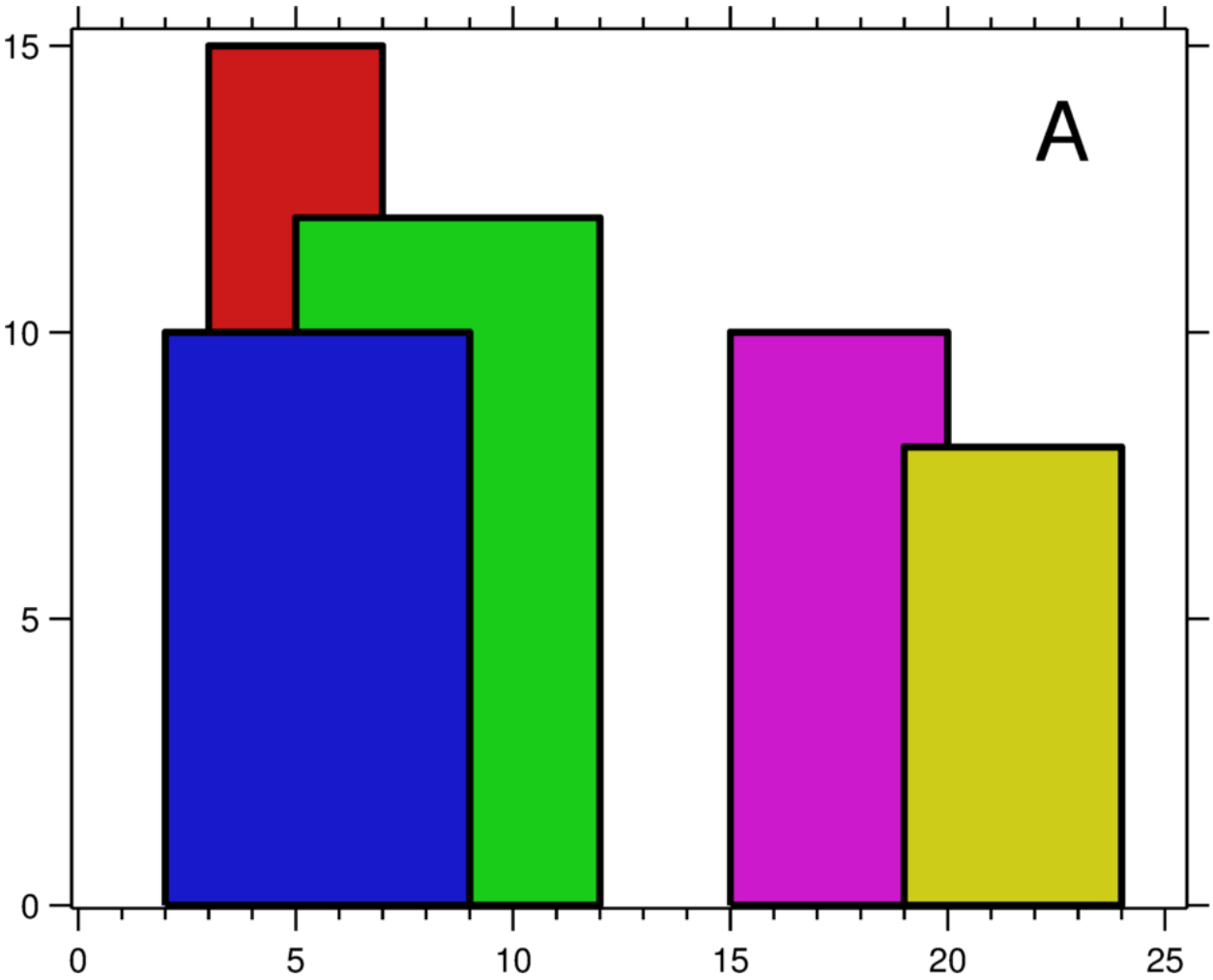
```
class NumMatrix {
public:
    vector<vector<int>> rects;//存储以matrix左上角为起点，[i,j]为右下角点的子矩阵和
    int m;
    int n;
    NumMatrix(vector<vector<int>>& matrix) {
        m = matrix.size();
        if(m <= 0) return;
        n = matrix[0].size();
        rects = vector<vector<int>>(m+1, vector<int>(n+1, 0));
        for(int i = 1; i <= m; i++)
            for(int j = 1; j <= n; j++)//容斥
                 rects[i][j] = rects[i-1][j] + rects[i][j-1]
                               - rects[i-1][j-1] + matrix[i-1][j-1];
    }

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

218. 天际线问题（困难）

1. 题目描述

城市的天际线是从远处观看该城市中所有建筑物形成的轮廓的外部轮廓。现在，假设您获得了城市风光照片（图A）上显示的所有建筑物的位置和高度，请编写一个程序以输出由这些建筑物形成的天际线（图B）。



每个建筑物的几何信息用三元组 `[Li, Ri, Hi]` 表示，其中 `Li` 和 `Ri` 分别是第 i 座建筑物左右边缘的 x 坐标，`Hi` 是其高度。可以保证 `0 ≤ Li, Ri ≤ INT_MAX, 0 < Hi ≤ INT_MAX` 和 `Ri - Li > 0`。您可以假设所有建筑物都是在绝对平坦且高度为 0 的表面上的完美矩形。

例如，图A中所有建筑物的尺寸记录为：`[ [2 9 10], [3 7 15], [5 12 12], [15 20 10], [19 24 8] ]`。

输出是以 `[ [x1,y1], [x2, y2], [x3, y3], ... ]` 格式的“关键点”（图B中的红点）的列表，它们唯一地定义了天际线。关键点是水平线段的左端点。请注意，最右侧建筑物的最后一个关键点仅用于标记天际线的终点，并始终为零高度。此外，任何两个相邻建筑物之间的地面都应被视为天际线轮廓的一部分。

例如，图B中的天际线应该表示为：`[ [2 10], [3 15], [7 12], [12 0], [15 10], [20 8], [24, 0] ]`。

说明:

- 任何输入列表中的建筑物数量保证在 [0, 10000] 范围内。
- 输入列表已经按左 x 坐标 Li 进行升序排列。
- 输出列表必须按 x 位排序。
- 输出天际线中不得有连续的相同高度的水平线。例如 [...[2 3], [4 5], [7 5], [11 5], [12 7]...] 是不正确的答案；三条
- 高度为 5 的线应该在最终输出中合并为一个：[...[2 3], [4 5], [12 7], ...]

2. 简单实现

冷静分析就会发现不难，关键是找出每个阶段的最高的高度是多少

```cpp
class Solution {
public:
    vector<vector<int>> getSkyline(vector<vector<int>>& buildings) {
        multiset<pair<int, int>> all;
        vector<vector<int>> res;

        for (auto& e : buildings) {
            all.insert(make_pair(e[0], -e[2])); // critical point, left corner
            all.insert(make_pair(e[1], e[2])); // critical point, right corner
        }

        multiset<int> heights({0}); // 保存当前位置所有高度。
        vector<int> last = {0, 0}; // 保存上一个位置的横坐标以及高度
        for (auto& p : all) {
            if (p.second < 0) heights.insert(-p.second); // 左端点，高度入堆
            else heights.erase(heights.find(p.second)); // 右端点，移除高度

            // 当前关键点，最大高度
            auto maxHeight = *heights.rbegin();

            // 当前最大高度如果不同于上一个高度，说明这是一个转折点
            if (last[1] != maxHeight) {
                // 更新 last，并加入结果集
                last[0] = p.first;
                last[1] = maxHeight;
                res.push_back(last);
            }
        }

        return res;
    }
};
```

121. 买卖股票的最佳时机（简单）

1. 题目描述

   给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

   如果你最多只允许完成一笔交易（即买入和卖出一支股票一次），设计一个算法来计算你所能获取的最大利润。

   注意：你不能在买入股票前卖出股票。

```
示例 1:
输入: [7,1,5,3,6,4]
输出: 5
解释: 在第 2 天（股票价格 = 1）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，最大利润 = 6-1
= 5 。
     注意利润不能是 7-1 = 6，因为卖出价格需要大于买入价格；同时，你不能在买入前卖出股票。

示例 2:
输入: [7,6,4,3,1]
输出: 0
解释: 在这种情况下，没有交易完成，所以最大利润为 0。
```

2. 简单实现

一次遍历，记录当前最小值（买入）和最大利润

```cpp
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int n = prices.size();
        if(n <= 1) return 0;
        int ans = 0;
        int l = prices[0];
        for(int r = 1; r < n; r++){
            if(prices[r] < l)
                l = prices[r];
            else
                ans = max(ans, prices[r] - l);
        }
        return ans;
    }
};
```

205. 同构字符串（简单）

1. 题目描述

给定两个字符串 s 和 t，判断它们是否是同构的。

如果 s 中的字符可以被替换得到 t ，那么这两个字符串是同构的。

所有出现的字符都必须用另一个字符替换，同时保留字符的顺序。两个字符不能映射到同一个字符上，但字符可以映射自己本身。

```
示例 1:
输入: s = "egg", t = "add"
输出: true

示例 2:
输入: s = "foo", t = "bar"
输出: false

示例 3:
输入: s = "paper", t = "title"
输出: true
```

说明: 你可以假设 s 和 t 具有相同的长度。

2. 简单实现

——映射

```
class Solution {
public:
    bool isIsomorphic(string s, string t) {
        unordered_map<char, char> f1;//s->t
        unordered_map<char, char> f2;//t->s
        for(int i = 0; i < s.size(); i++){
            if(f1.find(s[i]) != f1.end()){
                if (f1[s[i]] != t[i])
                    return false;
            }
            if(f2.find(t[i]) != f2.end()){
                if (f2[t[i]] != s[i])
                    return false;
            }
            f1[s[i]] = t[i];
            f2[t[i]] = s[i];
        }
        return true;
    }
};
```

31. 下一个排列（中等）

1. 题目描述

实现获取下一个排列的函数，算法需要将给定数字序列重新排列成字典序中下一个更大的排列。

如果不存在下一个更大的排列，则将数字重新排列成最小的排列（即升序排列）。

必须原地修改，只允许使用额外常数空间。

以下是一些例子，输入位于左侧列，其相应输出位于右侧列。 1,2,3 → 1,3,2 3,2,1 → 1,2,3 1,1,5 → 1,5,1

2. 简单实现

观察得到替换策略：

- 从右向左遍历nums[idx]，查看nums[idx+1...]中是否有大于nums[idx]的数
  - 如果有，则用大于nums[idx]的数中最小的那个与nums[idx]交换，再将idx[idx+1...]按升序排列，则得到了答案
- 如果遍历完都没有，说明不存在下一个更大的排列，根据题目要求，返回升序排列结果
- 例如4,2,0,2,3,2,0的替换策略为
  - 先将索引为3的2与3替换，即4,2,0,3,2,2,0
  - 再将索引3之后的数字升序，即4,2,0,3,0,2,2，得到答案

```
class Solution {
public:
    void nextPermutation(vector<int>& nums) {
        int n = nums.size();
        int idx = n - 2;
        map<int, int> idxs;//升序记录已经遍历过的数值和它们对应的索引，不需要用multiset，因为如果出现相同数值，和哪个位置的替换都是一样的
        idxs[nums[n-1]] = n-1;
        while(idx >= 0){
            auto it = idxs.upper_bound(nums[idx]);
            if(it != idxs.end()){
                swap(nums[it->second], nums[idx]);
                sort(nums.begin()+idx+1, nums.end());
                return;
            }
            idxs[nums[idx]] = idx;
            idx--;
        }
        sort(nums.begin(), nums.end());
    }
};
```

298. 二叉树最长连续序列（中等）

1. 题目描述

   给你一棵指定的二叉树，请你计算它最长连续序列路径的长度。

   该路径，可以是从某个初始结点到树中任意结点，通过「父 - 子」关系连接而产生的任意路径。

   这个最长连续的路径，必须从父结点到子结点，反过来是不可以的。

```
示例 1:
输入:
   1
    \
     3
    / \
   2   4
        \
         5
输出: 3
解析: 当中，最长连续序列是 3-4-5，所以返回结果为 3
```

```
示例 2:
输入:
   2
    \
     3
    /
   2
  /
 1

输出: 2
解析: 当中，最长连续序列是 2-3。注意，不是 3-2-1，所以返回 2。
```

2. 简单实现——递归

```cpp
class Solution {
public:
    int res = 0;
    int helper(TreeNode* root){//返回以root为根节点的最长连续序列
        if(!root) return 0;
        if(!root->left && !root->right){
            res = max(res, 1);
            return 1;
        }
        int ans = 1;
        int l = helper(root->left);//以左子树为起点
        if(root->left && root->val + 1 == root->left->val)//当前节点可以和左子数连起来
            ans = max(ans, l+1);
        int r = helper(root->right);//右子树同理
        if(root->right && root->val + 1 == root->right->val)
            ans = max(ans, r+1);
        res = max(res, ans);
        return ans;
    }
    int longestConsecutive(TreeNode* root) {
        helper(root);
        return res;
    }
};
```

34. 在排序数组中查找元素的第一个和最后一个位置（中等）

1. 题目描述

给定一个按照升序排列的整数数组 nums，和一个目标值 target。找出给定目标值在数组中的开始位置和结束位置。

你的算法时间复杂度必须是 O(log n) 级别。

如果数组中不存在目标值，返回 [-1, -1]。

```
示例 1:
输入: nums = [5,7,7,8,8,10], target = 8
输出: [3,4]

示例 2:
输入: nums = [5,7,7,8,8,10], target = 6
输出: [-1,-1]
```

2. 简单实现

```
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        int size = nums.size();
        if(size == 0) return {-1, -1};//空数组
        //起始位置
        int l = 0, r = size - 1;
        while(l < r){
            int mid = l + (r - l) / 2;
            if(nums[mid] < target)
                l = mid + 1;
            else
                r = mid;
        }
        if(nums[l] != target)
            return {-1,-1};
        //结束位置
        r = size - 1;
        if(nums[r] == target) return {l, r};//在while里判断不出来的情况，例如[1]找1,
[1,2,2]找2
        vector<int> ans = {l,-1};
        while(l < r){//实际找的是upper_bound
            int mid = l + (r - l) / 2;
            if(nums[mid] <= target)
                l = mid + 1;
            else
                r = mid;
        }
        ans[1] = l-1;//l对应upper_bound
        return ans;
    }
};
```

7. 整数反转（简单）

1. 题目描述

给出一个 32 位的有符号整数，你需要将这个整数中每位上的数字进行反转。

```
示例 1:
输入: 123
输出: 321

示例 2:
输入: -123
输出: -321

示例 3:
输入: 120
输出: 21
```

注意: 假设我们的环境只能存储得下 32 位的有符号整数，则其数值范围为 [−231, 231 − 1]。请根据这个假设，如果反转后整数溢出那么就返回 0。

2. 简单实现

try-catch的妙用可以简化判断代码

```cpp
class Solution {
public:
    int reverse(int x) {
        string s = to_string(abs(x));
        int i = 0, j = s.length()-1;
        while(i < j)
            swap(s[i++], s[j--]);
        try{
            int num = stoi(s);
            return x > 0 ? num : -num;
        }
        catch(exception& e){
            return 0;
        }
    }
};
```

279. 完全平方数（中等）

1. 题目描述

给定正整数 n，找到若干个完全平方数（比如 1, 4, 9, 16, ...）使得它们的和等于 n。你需要让组成和的完全平方数的个数最少。

```
示例 1:
输入: n = 12
输出: 3
解释: 12 = 4 + 4 + 4.

示例 2:
输入: n = 13
输出: 2
解释: 13 = 4 + 9.
```

2. 简单实现动态规划

```
class Solution {
public:
    int numSquares(int n) {
        if(n <= 2) return n;
        vector<int> dp(n+1);//dp[i]保存numSquares(i)
        dp[0] = 0;
        dp[1] = 1;
        for(int i = 2; i <= n; i++){
            dp[i] = i;
            int j = 1;
            int cur = j*j;
            while(cur <= i){
                dp[i] = min(dp[i], dp[i-cur]+1);
                j += 1;
                cur = j * j;
            }
        }
        return dp[n];
    }
};
```

3. 以前的解法——bfs

```
class Solution {
public:
    int numSquares(int n) {
        queue<int> q;
        q.push(n);
        int count = 0;
        while(1){
            count++;
            int size = q.size();
            for(int i = 0; i < size; i ++){
                int cur = q.front();
                q.pop();
                int sqrts = int(sqrt(cur));
                if(sqrts*sqrts == cur)
                    return count;
                for(int k = sqrts; k > 0; k--){
                    q.push(cur-k*k);
```

```
                }
            }
        }
        return -1;
    }
};
```

295. 数据流的中位数（困难）

1. 题目描述

   中位数是有序列表中间的数。如果列表长度是偶数，中位数则是中间两个数的平均值。

   例如，

   [2,3,4] 的中位数是 3

   [2,3] 的中位数是 (2 + 3) / 2 = 2.5

   设计一个支持以下两种操作的数据结构：

   - void addNum(int num) - 从数据流中添加一个整数到数据结构中。
   - double findMedian() - 返回目前所有元素的中位数。

   ```
   示例：
   addNum(1)
   addNum(2)
   findMedian() -> 1.5
   addNum(3)
   findMedian() -> 2
   ```

   进阶:

   - 如果数据流中所有整数都在 0 到 100 范围内，你将如何优化你的算法?
   - 如果数据流中 99% 的整数都在 0 到 100 范围内，你将如何优化你的算法?

2. 简单实现——两个堆

```
class MedianFinder {
    priority_queue<int> lo;                              // max heap
    priority_queue<int, vector<int>, greater<int>> hi;   // min heap

public:
    // Adds a number into the data structure.
    void addNum(int num) {
        lo.push(num);                                    // Add to max heap
        hi.push(lo.top());                               // balancing step
        lo.pop();
        if (lo.size() < hi.size()) {                     // maintain size property
            lo.push(hi.top());
            hi.pop();
        }
    }
    // Returns the median of current data stream
    double findMedian() {
```

```
        return lo.size() > hi.size() ? (double) lo.top() : (lo.top() + hi.top()) *
0.5;
    }
};
```

21. 合并两个有序链表（简单）

1. 题目描述

   将两个升序链表合并为一个新的升序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

   ```
   示例：
   输入：1->2->4, 1->3->4
   输出：1->1->2->3->4->4
   ```

2. 简单实现

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if(!l1) return l2;
        if(!l2) return l1;
        ListNode* head = new ListNode(-1);
        ListNode* temp = head;
        while(l1 && l2){
            if(l1->val <= l2->val){
                temp->next = l1;
                temp = l1;
                l1 = l1->next;
            }
            else{
                temp->next = l2;
                temp = l2;
                l2 = l2->next;
            }
        }
        if(l1) temp->next = l1;
        if(l2) temp->next = l2;
        return head->next;
    }
};
```

215. 数组中的第K个最大元素（中等）

1. 题目描述

   在未排序的数组中找到第 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。

```
示例 1:
输入: [3,2,1,5,6,4] 和 k = 2
输出: 5

示例 2:
输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4
```

说明: 你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

2. 简单实现

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        return findKthLargest(nums, 0, nums.size()-1, k-1);
    }
    int findKthLargest(vector<int>& nums, int m, int n, int k) {
        while(1){
            int re = quick_sort(nums, m, n);
            if(re == k)
                return nums[re];
            else if (re < k)
                return findKthLargest(nums, re+1, n, k);
            else
                return findKthLargest(nums, m, re-1, k);
        }
    }
    int quick_sort(vector<int>& nums, int l, int r){
        int temp = nums[l];
        while(l < r){
            while(l < r && nums[r] < temp)    r--;
            if(l < r) nums[l] = nums[r];
            while(l < r && nums[l] >= temp)  l++;
            if(l < r) nums[r] = nums[l];
        }
        nums[l] = temp;
        return l;
    }
};
```

50. Pow(x,n)（中等）

1. 题目描述

实现 pow(x, n)，即计算 x 的 n 次幂函数。

```
示例 1:
输入: 2.00000, 10
输出: 1024.00000

示例 2:
输入: 2.10000, 3
输出: 9.26100

示例 3:
输入: 2.00000, -2
输出: 0.25000
解释: 2-2 = 1/22 = 1/4 = 0.25
```

说明:

- -100.0 < x < 100.0
- n 是 32 位有符号整数，其数值范围是 [−231, 231 − 1] 。

2. 简单实现

```
class Solution {
public:
    double myPow(double x, int n) {
        if (n == 0) { return 1; }
        if (n == 1) { return x; }
        if (n == -1) { return 1 / x; }
        double half = myPow(x, n / 2);
        double rest = myPow(x, n % 2);
        return rest * half * half;
    }
};
```

271. 字符串的编码与解码（中等）

1. 题目描述

请你设计一个算法，可以将一个 字符串列表 编码成为一个 字符串。这个编码后的字符串是可以通过网络进行高效传送的，并且可以在接收端被解码回原来的字符串列表。

1 号机（发送方）有如下函数：

```
string encode(vector<string> strs) {
  // ... your code
  return encoded_string;
}
```

2 号机（接收方）有如下函数：

```
vector<string> decode(string s) {
  //... your code
  return strs;
}
```

1 号机（发送方）执行：

`string encoded_string = encode(strs);` 2 号机（接收方）执行：

`vector<string> strs2 = decode(encoded_string);` 此时，2 号机（接收方）的 strs2 需要和 1 号机（发送方）的 strs 相同。

请你来实现这个 encode 和 decode 方法。

注意：

- 因为字符串可能会包含 256 个合法 ascii 字符中的任何字符，所以您的算法必须要能够处理任何可能会出现的字符。
- 请勿使用 “类成员”、“全局变量” 或 “静态变量” 来存储这些状态，您的编码和解码算法应该是非状态依赖的。
- 请不要依赖任何方法库，例如 eval 又或者是 serialize 之类的方法。本题的宗旨是需要您自己实现 “编码” 和 “解码” 算法。

2. 简单实现

用`,`做转义字符，`,n`分割字符串，字符串内的`,`编码为`,,`

```
class Codec {
public:
    string encode(vector<string>& strs) {
        string ans = "";
        int size = strs.size();
        for(int i = 0; i < size; i++){
            for(int j = 0; j < strs[i].size(); j++){
                if(strs[i][j] == ',')
                    ans += ",,";
                else
                    ans += strs[i][j];
            }
            ans += ",n";
        }
        return ans;
    }
    vector<string> decode(string s) {
        int len = s.size();
        vector<string> ans;
        int r = 0;
        string cur = "";
        while(r < len){
            if(s[r] == ','){
                r++;
                if(s[r] == 'n'){
                    ans.push_back(cur);
                    cur = "";
                }
                else
```

```
                    cur += s[r];
            }
            else
                cur += s[r];
            r++;
        }
        return ans;
    }
};
```

3. 类比网络协议

这道题的感觉特别像是网络中协议的确定。在tcp编程中，如何从字节流中区分出来字符串。简单情况下，我们都是采取长度的方式来解决问题的。 就是4个字节的字符串长度，字符串这样序列化成一个大的字符串。

```
class Codec {
public:
    const char HEX[16] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A',
'B', 'C', 'D', 'E', 'F'};
    string int2Hex(int n) {
        string res;
        while (n > 0) {
            res += HEX[n % 16];
            n /= 16;
        }
        if (res.size() < 4)
            res.append(4 - res.size(), '0');
        reverse(res.begin(), res.end());
        return res;
    }
    int hex2Int(const string& s) {
        int res = 0;
        for (auto c : s) {
            res <<= 4;
            res |= (c <= '9') ? (c - '0') : (c - 'A' + 10);
        }
        return res;
    }
    // Encodes a list of strings to a single string.
    string encode(vector<string>& strs) {
        string res;
        for (auto& w : strs) {
            res += int2Hex(w.size());
            res += w;
        }
        return res;
    }
    // Decodes a single string to a list of strings.
    vector<string> decode(string s) {
        vector<string> res;
        int i = 0;
        int N = s.size();
        while (i < N) {
```

```
            int len = hex2Int(s.substr(i, 4));
            res.push_back(s.substr(i + 4, len));
            i += 4 + len;
        }
        return res;
    }
};
```

315. 计算右侧小于当前元素的个数（困难）

1. 题目描述

   给定一个整数数组 nums，按要求返回一个新数组 counts。数组 counts 有该性质： counts[i] 的值是 nums[i] 右侧小于 nums[i] 的元素的数量。

```
示例:
输入: [5,2,6,1]
输出: [2,1,1,0]
解释:
5 的右侧有 2 个更小的元素 (2 和 1).
2 的右侧仅有 1 个更小的元素 (1).
6 的右侧有 1 个更小的元素 (1).
1 的右侧有 0 个更小的元素.
```

2. 简单实现

   一下子可以想到的就是从右向左遍历，维护一个排序的数组/二叉树，看当前数字插入的idx是多少

   另一种解法是类似于归并排序求逆序对的方法

```
class Solution {
public:
    vector<int> countSmaller(vector<int>& nums) {
        vector<int> count;//保存结果
        vector<pair<int,int>> num;////保存每个数和它在原数组中的序号，以免在排序过程中打乱顺序
        for(int i =0; i<nums.size(); ++i) {
            count.push_back(0);
            num.push_back(make_pair(nums[i],i));
        }
        merge_sort(num,count);
        return count;
    }
    //归并排序
    void merge_sort(vector<pair<int,int> >& vec, vector<int>& count) {
        if(vec.size() < 2) return;

        int mid = vec.size()/2;
        vector<pair<int,int>> sub_vec1(vec.begin(), vec.begin()+mid);
        vector<pair<int,int>> sub_vec2(vec.begin()+mid, vec.end());
        merge_sort(sub_vec1, count);
```

```
        merge_sort(sub_vec2, count);
        vec.clear();
        merge(sub_vec1, sub_vec2, vec, count);
    }

    //合并两数组
    void merge(vector<pair<int,int> >& sub_vec1, vector<pair<int,int> >& sub_vec2,
               vector<pair<int,int> >& vec, vector<int>& count) {
        int i =0;
        int j =0;
        while(i < sub_vec1.size() && j < sub_vec2.size()) {
            if(sub_vec1[i].first <= sub_vec2[j].first ) {
                vec.push_back(sub_vec1[i]);
                count[sub_vec1[i].second] += j;//记录逆序数
                i++;
            }
            else{
                vec.push_back(sub_vec2[j]);
                j++;
            }
        }
        for(;i<sub_vec1.size();++i) {
            vec.push_back(sub_vec1[i]);
            count[sub_vec1[i].second] += j;// -。-
        }
        for(;j<sub_vec2.size();++j)
            vec.push_back(sub_vec2[j]);
    }
};
```

3. 最佳解法——树状数组

- 找到nums的取值范围，以各个值为树状数组的索引
- 从后向前遍历，每次加入新的数nums[i]相当于update(nums[i], 1)
- 则count[i]就是getsum(nums[i])

```
class Solution {
public:
    vector<int>tree;//树状数组
    vector<int> countSmaller(vector<int>& nums) {
        int size = nums.size();
        if(size == 0) return {};
        if(size == 1) return {0};
        vector<int>res(size);
        int minn=INT_MAX, maxn=INT_MIN;//上下界
        for(int i : nums){
            minn = min(minn, i);
            maxn = max(maxn, i);
        }
        tree.resize(maxn-minn+1);
        for(int i = size-1; i >= 0; i--){
            res[i] = getsum(nums[i]-minn);//不算等于nums[i]的
            updata(nums[i]-minn+1, 1);
```

```
            }
            return res;
        }
        int getsum(int pos){
            int sum=0;
            while(pos > 0){
                sum += tree[pos];
                pos -= lowbit(pos);
            }
            return sum;
        }
        void updata(int pos, int num){
            while(pos < tree.size()){
                tree[pos] += num;
                pos += lowbit(pos);
            }
        }
        int lowbit(int x){
            return x&(-x);
        }
    };
```

124. 二叉树中的最大路径和（困难）

1. 题目描述

   给定一个非空二叉树，返回其最大路径和。

   本题中，路径被定义为一条从树中任意节点出发，达到任意节点的序列。该路径至少包含一个节点，且不一定经过根节点。

```
示例 1:
输入: [1,2,3]
   1
  / \
 2   3
输出: 6

示例 2:
输入: [-10,9,20,null,null,15,7]
   -10
   / \
  9  20
    /  \
   15   7
输出: 42
```

2. 简单实现

```
class Solution {
public:
```

```
    int maxPathSum(TreeNode* root, int &val) {//返回值为root在路径上的最大和，val是全局最大和
        if (root == nullptr) return 0;
        int left = maxPathSum(root->left, val);
        int right = maxPathSum(root->right, val);
        int case1 = root->val + max(0, left) + max(0, right);//左右都连上
        int case2 = root->val + max(0, max(left, right));//连左右中的某一边
        val = max(val, max(case1, case2));
        return case2;
    }
    int maxPathSum(TreeNode* root) {
        int val = INT_MIN;
        maxPathSum(root, val);
        return val;
    }
};
```

57. 插入区间（困难）

1. 题目描述

   给出一个无重叠的 ，按照区间起始端点排序的区间列表。

   在列表中插入一个新的区间，你需要确保列表中的区间仍然有序且不重叠（如果有必要的话，可以合并区间）。

```
示例 1:
输入: intervals = [[1,3],[6,9]], newInterval = [2,5]
输出: [[1,5],[6,9]]

示例 2:
输入: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]
输出: [[1,2],[3,10],[12,16]]
解释: 这是因为新的区间 [4,8] 与 [3,5],[6,7],[8,10] 重叠。
```

2. 简单实现——贪心

```
class Solution {
public:
    vector<vector<int>> insert(vector<vector<int>>& intervals, vector<int>& newInterval) {
        if(intervals.size() == 0) return {newInterval};
        vector<vector<int>> ans;
        for(int i = 0; i < intervals.size(); i++){
            if(intervals[i][1] < newInterval[0])//在newInterval左侧无交集
                ans.push_back(intervals[i]);
            else if(intervals[i][0] > newInterval[1]){
                //在newInterval右侧无交集，之后也不可能再有交集
                ans.push_back(newInterval);//新区间加进去
                while(i < intervals.size())//剩余的区间加进去
                    ans.push_back(intervals[i++]);
                return ans;
```

```
            }
            else{//有交集，更新newInterval为两者合并的结果
                newInterval[0] = min(newInterval[0], intervals[i][0]);
                newInterval[1] = max(newInterval[1], intervals[i][1]);
            }
        }
        ans.push_back(newInterval);//newInterval在最右侧
        return ans;
    }
};
```

二分法肯定更快，但是好麻烦啊懒得写

288. 单词的位移缩写（中等）

1. 题目描述

一个单词的缩写需要遵循 <起始字母><中间字母数><结尾字母> 这样的格式。

以下是一些单词缩写的范例：

a)

```
it                      --> it    (没有缩写)
 1
 ↓
```

b)

```
d|o|g                   --> d1g
                 1    1  1
 1---5----0----5--8
 ↓   ↓    ↓    ↓  ↓
```

c)

```
i|nternationalizatio|n  --> i18n
                 1
 1---5----0
 ↓   ↓    ↓
```

d)

```
l|ocalizatio|n          --> l10n
```

假设你有一个字典和一个单词，请你判断该单词的缩写在这本字典中是否唯一。若单词的缩写在字典中没有任何 其他 单词与其缩写相同，则被称为单词的唯一缩写。

示例：

```
给定 dictionary = [ "deer", "door", "cake", "card" ]
isUnique("dear") -> false
isUnique("cart") -> true
isUnique("cane") -> false
isUnique("make") -> true
```

2. 简单实现

吐了这个题意，题目的关键词：单词缩写是针对不同单词的

上面的描述有一点不好理解，单词缩写的不同，针对的是不同单词；而相同的单词缩写是同一种，一模一样。有点绕，具体按下面几种情况说下：

- 单词如果在字典中重复，这两个单词是同一个单词，依旧属于唯一的缩写。
- 单词如果在字典中没有重复，这就是不同单词，单词缩写如果在哈希表中存在，则其不同单词缩写次数+1。

```
class ValidWordAbbr {
public:
    unordered_map<string, int> hash;
    unordered_map<string, int> wordCnt;

    ValidWordAbbr(vector<string>& dictionary) {
        for (auto word: dictionary) {
            if (!wordCnt[word]) {
                if (word.size() <= 2) {
                    hash[word]++;
                } else {
                    hash[word[0]+to_string(word.size()-2)+word[word.size()-1]]++;
                }
                wordCnt[word]++;
            }
        }
    }

    bool isUnique(string word) {
        if (word.size() <= 2) {
            if (hash.find(word) != hash.end()) {
                return hash[word] == 1;
            }
            return !hash[word];
        } else {
            if (wordCnt[word]) {
                return hash[word[0]+to_string(word.size()-2)+word[word.size()-1]]
== 1;
            }
            return !hash[word[0]+to_string(word.size()-2)+word[word.size()-1]];
        }
    }
};
```

642. 设计搜索自动补全系统（困难）

1. 题目描述

为搜索引擎设计一个搜索自动补全系统。用户会输入一条语句（最少包含一个字母，以特殊字符 '#' 结尾）。除 '#' 以外用户输入的每个字符，返回历史中热度前三并以当前输入部分为前缀的句子。下面是详细规则：

1. 一条句子的热度定义为历史上用户输入这个句子的总次数。
2. 返回前三的句子需要按照热度从高到低排序（第一个是最热门的）。如果有多条热度相同的句子，请按照 ASCII 码的顺序输出（ASCII 码越小排名越前）。
3. 如果满足条件的句子个数少于 3，将它们全部输出。
4. 如果输入了特殊字符，意味着句子结束了，请返回一个空集合。 你的工作是实现以下功能：

构造函数：

`AutocompleteSystem(String[] sentences, int[] times)`: 这是构造函数，输入的是历史数据。Sentences 是之前输入过的所有句子，Times 是每条句子输入的次数，你的系统需要记录这些历史信息。

现在，用户输入一条新的句子，下面的函数会提供用户输入的下一个字符：

`List<String> input(char c)`: 其中 c 是用户输入的下一个字符。字符只会是小写英文字母（'a' 到 'z' ），空格（' '）和特殊字符（'#'）。输出历史热度前三的具有相同前缀的句子。

```
样例 :
操作 :  AutocompleteSystem(["i love you", "island","ironman", "i love leetcode"],
[5,3,2,2])
系统记录下所有的句子和出现的次数:
"i love you" : 5 次
"island" : 3 次
"ironman" : 2 次
"i love leetcode" : 2 次
现在，用户开始新的键入:

输入 :  input('i')
输出 :  ["i love you", "island","i love leetcode"]
解释 :
有四个句子含有前缀 "i"。其中 "ironman" 和 "i love leetcode" 有相同的热度，由于 ' ' 的 ASCII
码是 32 而 'r' 的 ASCII 码是 114，所以 "i love leetcode" 在 "ironman" 前面。同时我们只输出
前三的句子，所以 "ironman" 被舍弃。

输入 :  input(' ')
输出 :  ["i love you","i love leetcode"]
解释:
只有两个句子含有前缀 "i "。

输入 :  input('a')
输出 :  []
解释 :
没有句子有前缀 "i a"。

输入 :  input('#')
输出 :  []
解释 :
用户输入结束，"i a" 被存到系统中，后面的输入被认为是下一次搜索。
```

注释：

1. 输入的句子以字母开头，以 '#' 结尾，两个字母之间最多只会出现一个空格。
2. 即将搜索的句子总数不会超过 100。每条句子的长度（包括已经搜索的和即将搜索的）也不会超过 100。
3. 即使只有一个字母，输出的时候请使用双引号而不是单引号。
4. 请记住清零 AutocompleteSystem 类中的变量，因为静态变量、类变量会在多组测试数据中保存之前结果。详情请看这里。

2. 简单实现

第一反应用前缀树，是可以实现的，但是每次查询都要遍历某个子树的所有节点，个人感觉效率并不算高，因此用了哈希法，见注释

```
class AutocompleteSystem {
public:
    string cur;//当前输入的查询字符串
    int idx;//当前输入查询字符在查询字符串的下标
    vector<string> res;//目前所有可能的补全结果，按题目要求排序（热度递减、字典序递增）
    unordered_map<char, map<string, int>> cnt;//以各个字母开头的所有字符串的历史查询次数
    static bool cmp(pair<string, int>& a, pair<string, int>& b){//排序函数
        if(a.second != b.second)
            return a.second > b.second;
        else
            return a.first < b.first;
    }
    AutocompleteSystem(vector<string>& sentences, vector<int>& times) {
        cur = "";
        idx = 0;
        res.clear();
        cnt.clear();
        for(int i = 0; i < sentences.size(); i++)
            cnt[sentences[i][0]][sentences[i]] = times[i];
    }
    vector<string> input(char c) {
        if(c == '#'){//本此查询完毕，更新cnt，清空缓存量
            cnt[cur[0]][cur]++;
            cur = "";
            idx = 0;
            res.clear();
            return res;
        }
        if(idx == 0){//输入查询字符串的第一个字母
            cur += c;
            idx++;
            //找到所有可能的补全结果
            vector<pair<string, int>> temp;
            for(auto it = cnt[c].begin(); it != cnt[c].end(); it++)
                temp.push_back(*it);
            sort(temp.begin(), temp.end(), cmp);//排序
            for(auto it = temp.begin(); it != temp.end(); it++)
                res.push_back(it->first);
        }
        else{
            auto it = res.begin();
            while(it != res.end()){//去掉res中不可能是补全结果的单词
```

```
                if((*it).size() <= idx || (*it)[idx] != c)
                    res.erase(it);
                else
                    it++;
            }
            cur += c;
            idx++;
        }
        //返回res中的前三个
        if(res.size() < 3)
            return res;
        else
            return vector<string>(res.begin(), res.begin()+3);
    }
};
```

22. 括号生成（中等）

1. 题目描述

数字 n 代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且 有效的 括号组合。

```
示例:
输入: n = 3
输出: [
       "((()))",
       "(()())",
       "(())()",
       "()(())",
       "()()()"
     ]
```

2. 简单实现

```
class Solution {
public:
    vector<string> ans;
    void dfs(int l, int r, string temp){//未添加的左、右括号数量，当前字符串
        if(l > r) return;//无效
        if(l == 0 && r == 1)//只剩一个右括号，结束
            ans.push_back(temp + ')');
        else{
            if(l > 0)
                dfs(l-1, r, temp + '(');
            dfs(l, r-1, temp + ')');//r一定>0
        }
    }
    vector<string> generateParenthesis(int n) {
        if(n == 0) return ans;
        dfs(n, n, "");
        return ans;
```

```
    }
};
```

228. 汇总区间（中等）

1. 题目描述

给定一个无重复元素的有序整数数组，返回数组区间范围的汇总。

```
示例 1:
输入: [0,1,2,4,5,7]
输出: ["0->2","4->5","7"]
解释: 0,1,2 可组成一个连续的区间; 4,5 可组成一个连续的区间。

示例 2:
输入: [0,2,3,4,6,8,9]
输出: ["0","2->4","6","8->9"]
解释: 2,3,4 可组成一个连续的区间; 8,9 可组成一个连续的区间。
```

2. 简单实现

既然是有序的，直接遍历就好了，要注意结尾的细节处理

```
class Solution {
public:
    vector<string> summaryRanges(vector<int>& nums) {
        int size = nums.size();
        if(size == 0) return {};
        if(size == 1) return {to_string(nums[0])};
        int start = nums[0];
        int end = start;
        int idx = 1;
        vector<string> ans;
        while(idx < size){
            if(nums[idx] == end+1){
                end++;
                idx++;
            }
            else{
                if(start == end)
                    ans.push_back(to_string(start));
                else
                    ans.push_back(to_string(start) + "->" + to_string(end));
                start = nums[idx];
                end = start;
                idx++;
                if(idx == size){//注意
                    ans.push_back(to_string(start));
                    return ans;
                }
            }
        }
```

```
            ans.push_back(to_string(start) + "->" + to_string(end));//注意
            return ans;
        }
    };
```

947. 移除最多的同行或同列石头（中等）

1. 题目描述

   我们将石头放置在二维平面中的一些整数坐标点上。每个坐标点上最多只能有一块石头。

   每次 move 操作都会移除一块所在行或者列上有其他石头存在的石头。

   请你设计一个算法，计算最多能执行多少次 move 操作?

   ```
   示例 1:
   输入: stones = [[0,0],[0,1],[1,0],[1,2],[2,1],[2,2]]
   输出: 5

   示例 2:
   输入: stones = [[0,0],[0,2],[1,1],[2,0],[2,2]]
   输出: 3

   示例 3:
   输入: stones = [[0,0]]
   输出: 0
   ```

提示:

- 1 <= stones.length <= 1000
- 0 <= stones[i][j] < 10000

2. 正确解法——dfs

   思路

   首先将处于同一行或同一列的石头两两相连，这样会产生一个图。在这个图里面，互相连通的石子组成一个连通分量。

   显然，总有办法将一个连通分量中的石子依次移除，直到只剩下一个。首先，我们要知道每个石子都属于一个连通分量，同时在一个连通分量中移除石子不会影响到其他的连通分量。在有了这个前提之下，我们可以推断出，如果把连通分量作为一个生成树来看，每次都移除树中的叶子节点，重复这个操作，最后就只会剩下一个根节点。

   算法

   在这里我们用深度优先搜索来计算图中的连通分量的个数，通过深度优先搜索遍历连通分量中的每个节点。在每个连通分量里面，最多能移除石子的数量为 `连通分量中石子数量 - 1`。

   - 时间复杂度： O(N^2)，其中 N 是石头的数量。
   - 空间复杂度： O(N^2)。

3. 正确解法——并查集

把同行/同列的石头合并起来，看存在几个集合，每个集合都可以删除到最后一个石头，所以可删除的石头就是 `总石头数 - 集合数`

```
class DSU{
public:
    int* parent;
    int cnt;
    DSU(int n):parent(new int[n]),cnt(n){
        for(int i=0;i<n;i++) parent[i] = i;
    }
    int find(int a){
        if(a != parent[a]) parent[a] = find(parent[a]);
        return parent[a];
    }
    void union_elem(int a,int b){
        int aroot = find(a);
        int broot = find(b);
        if(aroot!=broot){
            parent[aroot] = broot;
            cnt-=1;
        }
    }
};
class Solution {
public:
    int removeStones(vector<vector<int>>& stones) {
        int n = stones.size();
        if(!n) return 0;
        DSU uf(n);
        for(int i=0; i<n; i++)
            for(int j=i; j<n; j++)
                if(stones[i][0] == stones[j][0] || stones[i][1] == stones[j][1])
                    uf.union_elem(i,j);
        return n-uf.cnt;
    }
};
```

345. 反转字符串中的元音字母（简单）

1. 题目描述

编写一个函数，以字符串作为输入，反转该字符串中的元音字母。

```
示例 1:
输入: "hello"
输出: "holle"

示例 2:
输入: "leetcode"
输出: "leotcede"
```

说明: 元音字母不包含字母"y"。

2. 简单实现

如果是面试，需要问清楚什么是元音字母、以及大小写问题

```
class Solution {
public:
    string reverseVowels(string s) {
        unordered_set<char> dic;
        dic.insert('a');
        dic.insert('e');
        dic.insert('i');
        dic.insert('o');
        dic.insert('u');
        dic.insert('A');
        dic.insert('E');
        dic.insert('I');
        dic.insert('O');
        dic.insert('U');
        int l = 0, r = s.size() - 1;
        while(l < r){
            while(l < r && dic.find(s[l]) == dic.end())
                l++;
            while(l < r && dic.find(s[r]) == dic.end())
                r--;
            if(l < r)
                swap(s[l++], s[r--]);
        }
        return s;
    }
};
```

211. 最大正方形（中等）

1. 题目描述

在一个由 0 和 1 组成的二维矩阵内，找到只包含 1 的最大正方形，并返回其面积。

```
示例:
输入:
1 0 1 0 0
1 0 1 1 1
1 1 1 1 1
1 0 0 1 0
输出: 4
```

2. 简单实现

`dp[i][j][0/1/2]` 分别表示以 `matrix[i][j]` 为结尾点的同行/列的连续1的个数和以 `matrix[i][j]` 为右下角点的最大正方形边长，则状态转移有

- `dp[i][j][0] = matrix[i][j] == '0' ? 0 : dp[i][j-1][0]+1`

- `dp[i][j][1] = matrix[i][j] == '0' ? 0 : dp[i-1][j][1]+1`
- `dp[i][j][2] = matrix[i][j] == '0' ? 0 : min(min(dp[i][j][0], dp[i][j][1]), dp[i-1][j-1][2]+1);`

```
class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
        int m = matrix.size();
        if(m <= 0) return 0;
        int n = matrix[0].size();
        if(n <= 0 ) return 0;
        vector<vector<vector<int>>> dp(m, vector<vector<int>>(n, vector<int>(3)));
        int ans = 0;
        for(int i = 0; i < m; i++){
            for(int j = 0; j < n; j++){
                if(matrix[i][j] == '0')
                    dp[i][j][0] = dp[i][j][1] = dp[i][j][2] = 0;
                else{
                    if(j == 0) dp[i][j][0] = 1;
                    else dp[i][j][0] = dp[i][j-1][0]+1;
                    if(i == 0) dp[i][j][1] = 1;
                    else dp[i][j][1] = dp[i-1][j][1]+1;
                    if(i == 0 || j == 0) dp[i][j][2] = 1;
                    else
                        dp[i][j][2] = min(min(dp[i][j][0], dp[i][j][1]), dp[i-1][j-
1][2]+1);
                }
                ans = max(ans, dp[i][j][2]);
            }
        }
        return ans*ans;
    }
};
```

3. 改进

不保存同行列的也可以的

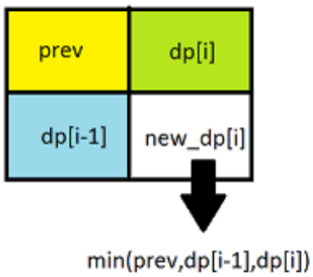
```
class Solution {
public:
    int maximalSquare(vector<vector<char>>& matrix) {
        int row = matrix.size();
        if(row == 0) return 0;
        int col = matrix[0].size();
        if(col == 0) return 0;
        int Max = 0;
        vector<vector<int>> dp(row+1, vector<int>(col+1, 0));
        for(int i=1; i<=row; ++i){
            for(int j=1; j<=col; ++j){
                if(matrix[i-1][j-1] == '1'){
                    dp[i][j] = min(min(dp[i-1][j], dp[i][j-1]), dp[i-1][j-1])+1;
                    Max = max(Max, dp[i][j]);
```

```
                }
            }
        }
        return Max*Max;
    }
};
```

4. 进一步优化

在前面的动态规划解法中，计算 i^{th} 行（row）的 dp 方法中，我们只使用了上一个元素和第 $(i-1)^{th}$ 行，因此我们不需要二维 dp 矩阵，因为一维 dp 足以满足此要求。

我们扫描一行原始矩阵元素时，我们根据公式：$dp[j] = min(dp[j-1], dp[j], prev)$ 更新数组 dp，其中 prev 指的是 $dp[j-1]$，对于每一行，我们重复相同过程并在 dp 矩阵中更新元素。




```
public class Solution {
    public int maximalSquare(char[][] matrix) {
        int rows = matrix.length, cols = rows > 0 ? matrix[0].length : 0;
        int[] dp = new int[cols + 1];
        int maxsqlen = 0, prev = 0;
        for (int i = 1; i <= rows; i++) {
            for (int j = 1; j <= cols; j++) {
                int temp = dp[j];//! !
                if (matrix[i - 1][j - 1] == '1') {
                    dp[j] = Math.min(Math.min(dp[j - 1], prev), dp[j]) + 1;//! !
                    maxsqlen = Math.max(maxsqlen, dp[j]);
                } else {
                    dp[j] = 0;
                }
                prev = temp;//! !
            }
        }
        return maxsqlen * maxsqlen;
    }
}
```

1057. 校园自行车分配（中等）

1. 题目描述

在由 2D 网格表示的校园里有 n 位工人（worker）和 m 辆自行车（bike），n <= m。所有工人和自行车的位置都用网格上的 2D 坐标表示。

我们需要为每位工人分配一辆自行车。在所有可用的自行车和工人中，我们选取彼此之间曼哈顿距离最短的工人自行车对 (worker, bike) ，并将其中的自行车分配給工人。如果有多个 (worker, bike) 对之间的曼哈顿距离相同，那么我们选择工人索引最小的那对。类似地，如果有多种不同的分配方法，则选择自行车索引最小的一对。不断重复这一过程，直到所有工人都分配到自行车为止。

给定两点 p1 和 p2 之间的曼哈顿距离为 Manhattan(p1, p2) = |p1.x - p2.x| + |p1.y - p2.y|。

返回长度为 n 的向量 ans，其中 a[i] 是第 i 位工人分配到的自行车的索引（从 0 开始）。

```
示例 1:
输入: workers = [[0,0],[2,1]], bikes = [[1,2],[3,3]]
输出: [1,0]
解释:
工人 1 分配到自行车 0，因为他们最接近且不存在冲突，工人 0 分配到自行车 1 。所以输出是 [1,0]。

示例 2:
输入: workers = [[0,0],[1,1],[2,0]], bikes = [[1,0],[2,2],[2,1]]
输出: [0,2,1]
解释:
工人 0 首先分配到自行车 0 。工人 1 和工人 2 与自行车 2 距离相同，因此工人 1 分配到自行车 2，工人 2 将分配到自行车 1 。因此输出为 [0,2,1]。
```

提示:

- 0 <= workers[i][j], bikes[i][j] < 1000
- 所有工人和自行车的位置都不相同。
- 1 <= workers.length <= bikes.length <= 1000

2. 简单实现

计算各个对的距离，按照距离、工人id、自行车id的顺序排序，然后一次遍历得到结果

```
struct Pair{
    int worker;
    int bike;
    int dis;
    Pair(int worker, int bike, int dis) : worker(worker), bike(bike), dis{dis} {}
};
bool cmp(Pair& a, Pair& b) {
    if(a.dis != b.dis)
        return a.dis < b.dis;
    else if(a.worker != b.worker)
        return a.worker < b.worker;
    else
        return a.bike < b.bike;
}
class Solution {
public:
    vector<int> assignBikes(vector<vector<int>>& workers, vector<vector<int>>&
bikes) {
        int n = workers.size();
        int m = bikes.size();
        vector<Pair> q;
        for(int i = 0; i < n; i++)
```

```
            for(int j = 0; j < m; j++)
                q.push_back(Pair(i, j, abs(workers[i][0]-bikes[j][0]) +
abs(workers[i][1]-bikes[j][1])));
        sort(q.begin(), q.end(), cmp);//排序
        vector<int> ans(n, -1);//-1代表未分配自行车
        int cnt = n;//待分配自行车的工人数
        int idx = 0;//遍历到q中的索引
        while(cnt){
            if(ans[q[idx].worker] == -1 && bikes[q[idx].bike][0] != -1){//工人和自行
车均未分配
                ans[q[idx].worker] = q[idx].bike;
                bikes[q[idx].bike][0] = -1;
                cnt--;
            }
            idx++;
        }
        return ans;
    }
};
```

3. 最优解法

根据题目限制知道工人和自行车最大的距离为2000，是有界的，因此直接用距离做vector索引就行，免除排序算法的耗时

```
class Solution {
public:
vector<int> assignBikes(vector<vector<int>> &workers, vector<vector<int>> &bikes) {
    vector<vector<pair<int, int>>> buckets(2001);//以距离为索引
    int n = workers.size(), m = bikes.size();
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {//i,j增序可以保证同dis下各piar内序号i和j都递增，满足
题目要求
            int dis = abs(workers[i][0] - bikes[j][0]) + abs(workers[i][1] -
bikes[j][1]);
            buckets[dis].push_back({i, j});
        }
    }
    vector<int> res(n, -1);
    vector<bool> bikeUsed(m, false);
    for (int d = 0; d <= 2000; ++d) {//距离最小的开始
        for (int k = 0; k < buckets[d].size(); ++k) {
            if (res[buckets[d][k].first] == -1 && !bikeUsed[buckets[d][k].second])
{
                bikeUsed[buckets[d][k].second] = true;
                res[buckets[d][k].first] = buckets[d][k].second;
            }
        }
    }
    return res;
}
};
```

162. 寻找峰值（中等）

1. 题目描述

   峰值元素是指其值大于左右相邻值的元素。

   给定一个输入数组 nums，其中 nums[i] ≠ nums[i+1]，找到峰值元素并返回其索引。

   数组可能包含多个峰值，在这种情况下，返回任何一个峰值所在位置即可。

   你可以假设 nums[-1] = nums[n] = -∞。

   ```
   示例 1:
   输入: nums = [1,2,3,1]
   输出: 2
   解释: 3 是峰值元素，你的函数应该返回其索引 2。

   示例 2:
   输入: nums = [1,2,1,3,5,6,4]
   输出: 1 或 5
   解释: 你的函数可以返回索引 1，其峰值元素为 2；或者返回索引 5， 其峰值元素为 6。
   ```

   说明: 你的解法应该是 O(logN) 时间复杂度的。

2. 简单实现

```cpp
class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int l = 0;
        int r = nums.size() - 1;
        //边界处理
        if(nums.size() == 1) return 0;
        if(nums[l+1] < nums[l]) return l;
        if(nums[r-1] < nums[r]) return r;
        while(l < r){
            int mid = l + (r - l) / 2;
            if(nums[mid] < nums[mid+1])//上坡
                l = mid + 1;
            else if(nums[mid] < nums[mid-1])//下坡
                r = mid;
            else
                return mid;

        }
        return l;
    }
};
```

155. 最小栈（简单）

1. 题目描述

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。

- push(x) —— 将元素 x 推入栈中。
- pop() —— 删除栈顶的元素。
- top() —— 获取栈顶元素。
- getMin() —— 检索栈中的最小元素。

```
示例:
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

2. 简单实现

```
class MinStack {
public:
    stack<int> data;
    stack<int> mins;
    /** initialize your data structure here. */
    MinStack() {
    }
    void push(int x) {
        data.push(x);
        if(mins.empty() || x <= mins.top())
            mins.push(x);
    }
    void pop() {
        if(data.top() == mins.top())
            mins.pop();
        data.pop();
    }
    int top() {
        return data.top();
    }
    int getMin() {
        return mins.top();
    }
};
```

3. 最优解法

真的秀啊

```
class MinStack {
    stack<int> s;
    int min;
public:
```

```
    MinStack() {
    }
    void push(int x) {
        if(s.empty())
            min=x;
        else if(min>=x){//更新min前要把旧min保存在栈里
            s.push(min);
            min=x;
        }
        s.push(x);
    }
    void pop() {
        if(s.size()==1)//可以不更新min，但是要防止进else if导致两次pop
            min = INT_MAX;
        else if(s.top()==min){//到了要更新min的时候
            s.pop();
            min=s.top();
        }
        s.pop();
    }
    int top() {
        return s.top();
    }
    int getMin() {
        return min;
    }
};
```

359. 日志速率限制器（简单）

1. 题目描述

请你设计一个日志系统，可以流式接收日志以及它的时间戳。

该日志会被打印出来，需要满足一个条件：当且仅当日志内容 在过去的 10 秒钟内没有被打印过。

给你一条日志的内容和它的时间戳（粒度为秒级），如果这条日志在给定的时间戳应该被打印出来，则返回 true，否则请返回 false。

要注意的是，可能会有多条日志在同一时间被系统接收。

```
示例:
Logger logger = new Logger();
// 日志内容 "foo" 在时刻 1 到达系统
logger.shouldPrintMessage(1, "foo"); returns true;
// 日志内容 "bar" 在时刻 2 到达系统
logger.shouldPrintMessage(2,"bar"); returns true;
// 日志内容 "foo" 在时刻 3 到达系统
logger.shouldPrintMessage(3,"foo"); returns false;
// 日志内容 "bar" 在时刻 8 到达系统
logger.shouldPrintMessage(8,"bar"); returns false;
// 日志内容 "foo" 在时刻 10 到达系统
logger.shouldPrintMessage(10,"foo"); returns false;
```

```
// 日志内容 "foo" 在时刻 11 到达系统
logger.shouldPrintMessage(11,"foo"); returns true;
```

2. 简单实现

```
class Logger {
public:
    unordered_map<string, int> time;
    Logger() {
    }
    bool shouldPrintMessage(int timestamp, string message) {
        if(time.find(message) == time.end() || time[message] + 10 <= timestamp){
            time[message] = timestamp;
            return true;
        }
        return false;
    }
};
```

380. 常数时间插入、删除和获取随机元素（中等）

1. 题目描述

设计一个支持在平均 时间复杂度 O(1) 下，执行以下操作的数据结构。

1. insert(val)：当元素 val 不存在时，向集合中插入该项。
2. remove(val)：元素 val 存在时，从集合中移除该项。
3. getRandom：随机返回现有集合中的一项。每个元素应该有相同的概率被返回。

```
示例 ：
// 初始化一个空的集合。
RandomizedSet randomSet = new RandomizedSet();
// 向集合中插入 1 。返回 true 表示 1 被成功地插入。
randomSet.insert(1);
// 返回 false ，表示集合中不存在 2 。
randomSet.remove(2);
// 向集合中插入 2 。返回 true 。集合现在包含 [1,2] 。
randomSet.insert(2);
// getRandom 应随机返回 1 或 2 。
randomSet.getRandom();
// 从集合中移除 1 ，返回 true 。集合现在包含 [2] 。
randomSet.remove(1);
// 2 已在集合中，所以返回 false 。
randomSet.insert(2);
// 由于 2 是集合中唯一的数字，getRandom 总是返回 2 。
randomSet.getRandom();
```

2. 简单实现

```
class RandomizedSet {
public:
```

```
    unordered_map<int, int> s1;//index-val
    unordered_map<int, int> s2;//val-index
    int cnt;
    /** Initialize your data structure here. */
    RandomizedSet() {
        srand((unsigned)time(NULL));
        cnt = 0;
    }

    /** Inserts a value to the set. Returns true if the set did not already contain
the specified element. */
    bool insert(int val) {
        if(s2.count(val) > 0) return false;
        s2[val] = cnt;
        s1[cnt] = val;
        cnt++;
        return true;
    }

    /** Removes a value from the set. Returns true if the set contained the
specified element. */
    bool removze(int val) {
        if(s2.count(val) <= 0) return false;
        cnt--;
        int idx = s2[val];
        int v = s1[cnt];
        s1[idx] = v;
        s2[v] = idx;
        s1.erase(cnt);
        s2.erase(val);
        return true;
    }

    /** Get a random element from the set. */
    int getRandom() {
        if(cnt == 0)
            return -1;
        int idx = rand() % cnt;
        return s1[idx];
    }
};
```

149. 直线上最多的点数（困难）

1. 题目描述

给定一个二维平面，平面上有 n 个点，求最多有多少个点在同一条直线上。

```
示例 1:
输入: [[1,1],[2,2],[3,3]]
输出: 3
解释:
```

```
^
|
|        o
|     o
|  o
+------------->
0  1  2  3  4

示例 2:
输入: [[1,1],[3,2],[5,3],[4,1],[2,3],[1,4]]
输出: 4
解释:
^
|
|  o
|     o        o
|        o
|  o        o
+------------------->
0  1  2  3  4  5  6
```

2. 简单实现

```
class Solution {
public:
    int maxPoints(vector<vector<int>>& points)
    {
        //两点确定一条直线
        const int size = points.size();
        if(size<3) return size;
        int Max=0;
        for(int i=0;i< size;i++)//i表示数组中的第i+1个点
        {
            //same用来表示和i一样的点
            int same=1;
            for(int j=i+1;j < size;j++)//j表示数组中的第j+1个点
            {
                int count=0;
                // i、j在数组中是重复点，计数
                if(points[i][0]==points[j][0]&&points[i][1]==points[j][1])
                    same++;
                else//i和j不是重复点，则计算和直线ij在一条直线上的点
                {
                    count++;
                    long long xDiff = (long long)(points[i][0] - points[j][0]);//
Δx1
                    long long yDiff = (long long)(points[i][1] - points[j][1]);//
Δy1

//Δy1/Δx1=Δy2/Δx2 => Δx1*Δy2=Δy1*Δx2，计算和直线ji在一条直线上的点
                    for (int k = j + 1; k < size; ++k)
                    {
```

```
                        if (xDiff * (points[i][1] - points[k][1]) == yDiff *
(points[i][0] - points[k][0]))
                            count++;
                    }
                }
                Max=max(Max,same+count);
            }
            if(Max> size /2)
                return Max;//若某次最大个数超过所有点的一半，则不可能存在其他直线通过更多的点
        }
        return Max;
    }
};
```

91. 解码方法（中等）

1. 题目描述

一条包含字母 A-Z 的消息通过以下方式进行了编码：

```
'A' -> 1
'B' -> 2
...
'Z' -> 26
```

给定一个只包含数字的非空字符串，请计算解码方法的总数。

```
示例 1:
输入: "12"
输出: 2
解释: 它可以解码为 "AB" (1 2) 或者 "L" (12) 。

示例 2:
输入: "226"
输出: 3
解释: 它可以解码为 "BZ" (2 26), "VF" (22 6), 或者 "BBF" (2 2 6) 。
```

2. 简单实现

```
class Solution {
public:
    int numDecodings(string s) {
        int n = s.size();
        vector<int> dp(n+1);
        dp[0] = 1;
        if(s[0] == '0')
            return 0;
        dp[1] = 1;
        for(int i = 2; i <= n; i++){
            dp[i] = 0;
            if(s[i-1] != '0')
```

```
                dp[i] += dp[i-1];
            else if(s[i-2] == '0')
                return 0;
            if((s[i-2] == '2' && s[i-1] >= '0' && s[i-1] <= '6') || s[i-2] == '1')
                dp[i] += dp[i-2];
        }
        return dp[n];
    }
};
```

128. 最长连续序列（困难）

1. 题目描述

   给定一个未排序的整数数组，找出最长连续序列的长度。

   要求算法的时间复杂度为 O(n)。

   ```
   示例:
   输入: [100, 4, 200, 1, 3, 2]
   输出: 4
   解释: 最长连续序列是 [1, 2, 3, 4]。它的长度为 4。
   ```

2. 正确解法

   巧用unordered_set达到O(n)

```
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        unordered_set<int> s;
        for(int i = 0; i < nums.size(); i++)
            s.insert(nums[i]);
        int ans = 0;
        for(auto it = s.begin(); it != s.end(); it++){
            int cur = *it;
            if(s.count(cur - 1) <= 0){//某个连续序列的起点
                while(s.count(cur + 1)){
                    cur++;
                    s.erase(cur);//删了、防止之后再遍历了，保证每个数都只遍历到一次（至多两
次）
                }
                ans = max(ans, cur - *it + 1);
            }
        }
        return ans;
    }
};
```