1. 两数之和（简单）

1. 题目描述

   给定一个整数数组 nums 和一个目标值 target，请你在该数组中找出和为目标值的那 两个 整数，并返回他们的数组下标。

   你可以假设每种输入只会对应一个答案。但是，你不能重复利用这个数组中同样的元素。

   ```
   示例:
   给定 nums = [2, 7, 11, 15], target = 9
   因为 nums[0] + nums[1] = 2 + 7 = 9
   所以返回 [0, 1]
   ```

2. 简单实现

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

388. 文件的最长绝对路径（中等）

1. 题目描述

   假设我们以下述方式将我们的文件系统抽象成一个字符串:

   字符串 "dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext" 表示:

   ```
   dir
       subdir1
       subdir2
           file.ext
   ```

   目录 dir 包含一个空的子目录 subdir1 和一个包含一个文件 file.ext 的子目录 subdir2 。

   字符串 "dir\n\tsubdir1\n\t\tfile1.ext\n\t\tsubsubdir1\n\tsubdir2\n\t\tsubsubdir2\n\t\t\tfile2.ext" 表示:

```
dir
    subdir1
        file1.ext
        subsubdir1
    subdir2
        subsubdir2
            file2.ext
```

目录 dir 包含两个子目录 subdir1 和 subdir2。 subdir1 包含一个文件 file1.ext 和一个空的二级子目录 subsubdir1。subdir2 包含一个二级子目录 subsubdir2 ， 其中包含一个文件 file2.ext。

我们致力于寻找我们文件系统中文件的最长 (按字符的数量统计) 绝对路径。例如，在上述的第二个例子中，最长路径为 "dir/subdir2/subsubdir2/file2.ext"，其长度为 32 (不包含双引号)。

给定一个以上述格式表示文件系统的字符串，返回文件系统中文件的最长绝对路径的长度。 如果系统中没有文件，返回 0。

说明:

- 文件名至少存在一个 . 和一个扩展名。
- 目录或者子目录的名字不能包含 .。
- 要求时间复杂度为 O(n) ， 其中 n 是输入字符串的大小。

请注意，如果存在路径 aaaaaaaaaaaaaaaaaaaa/sth.png 的话，那么 a/aa/aaa/file1.txt 就不是一个最长的路径。

2. 简单实现

观察可得，目录/文件名字以'\n'为分隔，而名字开头的'\t'的数量为其所处的层级，因此可以一次遍历并解析目录/文件的层级关系，时刻维持一个保存当前遍历到的目录/文件的路径名即可

```
class Solution {
public:
    //从input[idx]开始，获取当前目录/文件的层级、名字和是否为文件，返回值为是否找到文化/目录
    bool getLevelAndName(string& input, int& idx, int& level, string& name, bool&
isfile){
        if(idx >= input.size())
            return false;
        while(input[idx] == '\t'){//开头有几个\t就是第几层的
            level++;
            idx += 1;
        }
        int r = idx;
        while(r < input.size() && input[r] != '\n'){//找名字的结尾
            if(input[r] == '.')
                isfile = true;
            r++;
        }
        name = input.substr(idx, r-idx);
        idx = r+1;
        return true;
    }
    int lengthLongestPath(string input) {
        int ans = 0;
        vector<string> pre_s;//保存当前路径，很多题解里用的栈,从代码也可以看出来，
```

```
        int pre_l = -1;//保存pre_s的最高层级
        int cur_len = 0;//当前路径长度
        int idx = 0;
        string s = "";
        int l = 0;
        bool isfile = false;
        while(getLevelAndName(input, idx, l, s, isfile)){//当有解析出的文件时
            while(pre_l >= l){//不低于新文件层级的路径都要去掉
                cur_len -= pre_s.back().size() + (pre_l>0 ? 1 : 0);
                pre_s.pop_back();
                pre_l--;
            }
            pre_l = l;//当前目录/文件加入路径
            cur_len += s.size() + (l>0 ? 1 : 0);//0级前面没有'/'
            pre_s.push_back(s);
            if(isfile && cur_len > ans)//是文件且路径长度更大
                ans = cur_len;
            s = "";
            l = 0;
            isfile = false;
        }
        return ans;
    }
};
```

929. 独特的电子邮件地址（简单）

1. 题目描述

每封电子邮件都由一个本地名称和一个域名组成，以 @ 符号分隔。

例如，在 alice@leetcode.com中， alice 是本地名称，而 leetcode.com 是域名。

除了小写字母，这些电子邮件还可能包含 '.' 或 '+'。

如果在电子邮件地址的本地名称部分中的某些字符之间添加句点（'.'），则发往那里的邮件将会转发到本地名称中没有点的同一地址。例如，"alice.z@leetcode.com” 和 “alicez@leetcode.com” 会转发到同一电子邮件地址。 （请注意，此规则不适用于域名。）

如果在本地名称中添加加号（'+'），则会忽略第一个加号后面的所有内容。这允许过滤某些电子邮件，例如 m.y+name@email.com 将转发到 my@email.com。 （同样，此规则不适用于域名。）

可以同时使用这两个规则。

给定电子邮件列表 emails，我们会向列表中的每个地址发送一封电子邮件。实际收到邮件的不同地址有多少?

```
示例：
输入：
["test.email+alex@leetcode.com","test.e.mail+bob.cathy@leetcode.com","testemail+david@lee.tcode.com"]
输出：2
解释：实际收到邮件的是 "testemail@leetcode.com" 和 "testemail@lee.tcode.com"。
```

提示：

- 1 <= emails[i].length <= 100
- 1 <= emails.length <= 100
- 每封 emails[i] 都包含有且仅有一个 '@' 字符。

2. 简单实现

```
class Solution {
public:
    int numUniqueEmails(vector<string>& emails) {
        unordered_set<string> sended;
        for(int i = 0; i < emails.size(); i++){
                int idx = 0;
                string cur = "";
                while(emails[i][idx] != '@'){
                    if(emails[i][idx] == '+')
                        break;
                    else if(emails[i][idx] != '.')
                        cur += emails[i][idx];
                    idx++;
                }
                while(emails[i][idx] != '@')
                    idx++;
                cur += emails[i].substr(idx, emails[i].size()-idx);
                sended.insert(cur);
        }
        return sended.size();
    }
};
```

904. 水果成篮（中等）

1. 题目描述

在一排树中，第 i 棵树产生 tree[i] 型的水果。 你可以从你选择的任何树开始，然后重复执行以下步骤：

1. 把这棵树上的水果放进你的篮子里。如果你做不到，就停下来。
2. 移动到当前树右侧的下一棵树。如果右边没有树，就停下来。

请注意，在选择一颗树后，你没有任何选择：你必须执行步骤 1，然后执行步骤 2，然后返回步骤 1，然后执行步骤 2，依此类推，直至停止。

你有两个篮子，每个篮子可以携带任何数量的水果，但你希望每个篮子只携带一种类型的水果。 用这个程序你能收集的水果总量是多少？

```
示例 1：
输入：[1,2,1]
输出：3
解释：我们可以收集 [1,2,1]。

示例 2：
输入：[0,1,2,2]
输出：3
```

```
解释：我们可以收集 [1,2,2].
如果我们从第一棵树开始，我们将只能收集到 [0, 1]。

示例 3：
输入：[1,2,3,2,2]
输出：4
解释：我们可以收集 [2,3,2,2].
如果我们从第一棵树开始，我们将只能收集到 [1, 2]。

示例 4：
输入：[3,3,3,1,2,1,1,2,3,3,4]
输出：5
解释：我们可以收集 [1,2,1,1,2].
如果我们从第一棵树或第八棵树开始，我们将只能收集到 4 个水果。
```

提示：

- 1 <= tree.length <= 40000
- 0 <= tree[i] < tree.length

2. 简单实现

实际上是找包含最多两个不同数字的最长连续子序列，因此用滑动窗口 [l, r) 表示当前连续子序列，type1/2分别指示当前窗口内的两种类型的水果，idx1/idx2表示窗口内两种类型水果最新出现的索引值，r一次遍历更新其他参数即可

```
class Solution {
public:
    int totalFruit(vector<int>& tree) {
        int n = tree.size();
        if(n == 2) return n;
        int l = 0;//左端点
        int type1 = tree[l];
        int idx1 = l;
        while(idx1 < n && tree[idx1] == type1)//找第二个数，同时更新idx1
            idx1++;
        if(idx1 == n) return n;//只有一种数
        int r = idx1;//第二种数
        idx1--;//减回去以维持正确
        int type2 = tree[r];
        int idx2 = r;
        int ans = -1;
        while(r < n){
            if(tree[r] == type1){
                idx1 = r;
                r++;
            }
            else if(tree[r] == type2){
                idx2 = r;
                r++;
            }
            else{//出现第三种数
                ans = max(ans, r-l);//更新答案
                if(tree[r-1] == type1)//前一个出现的是第一种数
```

```
                    l = idx2 + 1;//去掉第二种数，idx2之后的一定都是第一种数
                else{//前一个出现的是第二种数
                    l = idx1 + 1;//去掉第一种数，idx1之后的一定都是第二种数
                    type1 = type2;//把第二种数交换到第一种数的位置去
                    idx1 = idx2;
                }
                type2 = tree[r];//第二种数用来记录新出现的数
                idx2 = r;
            }
        }
        ans = max(ans, r-l);//再次更新最后一个连续子区间值，在while种没能计算
        return ans;
    }
};
```

2. 两数相加（中等）

1. 题目描述

   给出两个 非空 的链表用来表示两个非负的整数。其中，它们各自的位数是按照 逆序 的方式存储的，并且它们的每个节点只能存储 一位 数字。

   如果，我们将这两个数相加起来，则会返回一个新的链表来表示它们的和。

   您可以假设除了数字 0 之外，这两个数都不会以 0 开头。

```
示例：
输入：(2 -> 4 -> 3) + (5 -> 6 -> 4)
输出：7 -> 0 -> 8
原因：342 + 465 = 807
```

2. 简单实现

   细节题

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* head = new ListNode(-1);
        ListNode* cur = head;
        int c = 0;
        while(l1 && l2){
            int sum= l1->val + l2->val + c;
            if(sum >= 10){
                sum %= 10;
                c = 1;
            }
            else
                c = 0;
            cur->next = new ListNode(sum);
            cur = cur->next;
            l1 = l1->next;//前移
            l2 = l2->next;
```

```
        }
        if(!l1 && !l2 && c == 1)
            cur->next = new ListNode(1);
        else{
            if(l2) l1 = l2;
            if(c == 0)
                cur->next = l1;
            else{
                while(c==1 && l1){//进位
                    int sum= l1->val + c;
                    if(sum >= 10){
                        sum %= 10;
                        c = 1;
                    }
                    else
                        c = 0;
                    cur->next = new ListNode(sum);
                    cur = cur->next;
                    l1 = l1->next;
                }
                if(l1)
                    cur->next = l1;
                if(c == 1)//还能进1
                    cur->next = new ListNode(1);
            }
        }
        return head->next;
    }
};
```

200. 岛屿数量（中等）

1. 题目描述

给定一个由 '1'（陆地）和 '0'（水）组成的的二维网格，计算岛屿的数量。一个岛被水包围，并且它是通过水平方向或垂直方向上相邻的陆地连接而成的。你可以假设网格的四个边均被水包围。

```
示例 1:
输入:
11110
11010
11000
00000
输出: 1

示例 2:
输入:
11000
11000
00100
00011
输出: 3
```

2. 简单实现

```
class Solution {
public:
    vector<vector<int>> dirs = {{0,1}, {0,-1}, {-1,0}, {1,0}};
    int numIslands(vector<vector<char>>& grid) {
        int ans = 0;
        int m = grid.size();
        if(m == 0) return 0;
        int n = grid[0].size();
        if(n == 0) return 0;
        for(int i = 0; i < m; i++)
            for(int j = 0; j <n; j++){
                if(grid[i][j] == '1'){
                    ans++;
                    queue<pair<int, int>> q;
                    q.push(make_pair(i,j));
                    grid[i][j] = '0';
                    while(!q.empty()){
                        int x = q.front().first;
                        int y = q.front().second;
                        q.pop();
                        for(int i = 0; i < dirs.size(); i++){
                            int xx = x + dirs[i][0];
                            int yy = y + dirs[i][1];
                            if(xx >= 0 && xx < m && yy >= 0 && yy < n && grid[xx]
[yy] == '1'){
                                grid[xx][yy] = '0';
                                q.push(make_pair(xx, yy));
                            }
                        }
                    }
                }
            }
        return ans;
    }
};
```

3. 无重复字符的最长子串（中等）

1. 题目描述

给定一个字符串，请你找出其中不含有重复字符的 最长子串 的长度。

```
示例 1:
输入: "abcabcbb"
输出: 3
解释: 因为无重复字符的最长子串是 "abc"，所以其长度为 3。

示例 2:
输入: "bbbbb"
```

```
输出: 1
解释: 因为无重复字符的最长子串是 "b"，所以其长度为 1。

示例 3:
输入: "pwwkew"
输出: 3
解释: 因为无重复字符的最长子串是 "wke"，所以其长度为 3。
     请注意，你的答案必须是 子串 的长度，"pwke" 是一个子序列，不是子串。
```

2. 简单实现

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int ans = 0;
        int l = 0, r = 0;
        unordered_set<char> chars;
        while(r < s.size()){
            if(chars.find(s[r]) != chars.end()){
                ans = max(ans, r-l);
                while(s[l] != s[r]){//这里要遍历l去掉重复字符前的所有字符，不能直接跳转l，
否则chars内记录就不对了
                    chars.erase(s[l]);
                    l++;
                }
                chars.erase(s[l]);
                l++;
            }
            chars.insert(s[r]);
            r++;
        }
        ans = max(ans, r-l);
        return ans;
    }
};
```

3. 改进

可以直接跳转啊，始终记录字符出现的最新位置就可以了啊

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        vector<int> m = vector<int>(256, -1);
        int ans = 0;
        int l = -1;
        int r = 0;
        while(r < s.size()){
            l = max(l, m[s[r]]);
            m[s[r]] = r;
            ans = max(ans, r-l);
            r++;
        }
```

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

482. 密钥格式化（简单）

1. 题目描述

   给定一个密钥字符串S，只包含字母，数字以及 '-'（破折号）。N 个 '-' 将字符串分成了 N+1 组。给定一个数字 K，重新格式化字符串，除了第一个分组以外，每个分组要包含 K 个字符，第一个分组至少要包含 1 个字符。两个分组之间用 '-'（破折号）隔开，并且将所有的小写字母转换为大写字母。

   给定非空字符串 S 和数字 K，按照上面描述的规则进行格式化。

```
示例 1:
输入: S = "5F3Z-2e-9-w", K = 4
输出: "5F3Z-2E9W"
解释: 字符串 S 被分成了两个部分，每部分 4 个字符;
     注意，两个额外的破折号需要删掉。

示例 2:
输入: S = "2-5g-3-J", K = 2
输出: "2-5G-3J"
解释: 字符串 S 被分成了 3 个部分，按照前面的规则描述，第一部分的字符可以少于给定的数量，其余部分皆为 2 个字符。
```

提示:

- S 的长度不超过 12,000，K 为正整数
- S 只包含字母数字（a-z，A-Z，0-9）以及破折号'-'
- S 非空

2. 简单实现

   题意不清+千奇百怪的测试用例，我吐了；；题意是K个一组，且第一组的字符数要小于等于K，因此从后往前遍历即可

   如果面试遇到这种题目，应该主要在考察面试者与面试官的沟通，理解题意至关重要！！！

```
class Solution {
public:
    string licenseKeyFormatting(string S, int K) {
        string ans = "";
        int len = S.size();
        int idx = len-1;
        int cnt = 0;
        while(idx >= 0){
            if(S[idx] != '-'){
                if(S[idx] >= 'a' && S[idx] <= 'z')
                    ans += S[idx] - 'a' + 'A';
                else
                    ans += S[idx];
                cnt++;
```

```
                if(cnt == K){//K个一组
                    ans += "-";
                    cnt = 0;
                }
            }
            idx--;
        }
        if(ans.size() > 0 && ans.back() == '-')//ans可能为空
            ans.pop_back();
        reverse(ans.begin(), ans.end());//反转字符串
        return ans;
    }
};
```

56. 合并区间（中等）

1. 题目描述

给出一个区间的集合，请合并所有重叠的区间。

```
示例 1:
输入: [[1,3],[2,6],[8,10],[15,18]]
输出: [[1,6],[8,10],[15,18]]
解释: 区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].

示例 2:
输入: [[1,4],[4,5]]
输出: [[1,5]]
解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

2. 简单实现

```
class Solution {
public:
    static bool cmp(vector<int>& a, vector<int>& b){
        if(a[0] != b[0])
            return a[0] < b[0];
        else
            return a[1] < b[1];
    }
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        int nums = intervals.size();
        if(nums <= 1) return intervals;
        sort(intervals.begin(), intervals.end(), cmp);
        vector<vector<int>> ans = {intervals[0]};
        for(int i = 1; i < nums; i++){
            if(intervals[i][0] > ans.back()[1])
                ans.push_back(intervals[i]);
            else
                ans.back()[1] = max(ans.back()[1], intervals[i][1]);
        }
```

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

66. 加一（简单）

1. 题目描述

   给定一个由整数组成的非空数组所表示的非负整数，在该数的基础上加一。

   最高位数字存放在数组的首位， 数组中每个元素只存储单个数字。

   你可以假设除了整数 0 之外，这个整数不会以零开头。

```
示例 1:
输入: [1,2,3]
输出: [1,2,4]
解释: 输入数组表示数字 123。

示例 2:
输入: [4,3,2,1]
输出: [4,3,2,2]
解释: 输入数组表示数字 4321。
```

2. 简单实现

   又一个细节进位题，小心啊！！！

```
class Solution {
public:
    vector<int> plusOne(vector<int>& digits) {
        int nums = digits.size();
        int idx = nums-1;
        while(idx >= 0 && digits[idx] == 9){
            digits[idx] = 0;
            idx--;
        }
        if(idx < 0)//进一位1
            digits.insert(digits.begin(), 1);
        else//高位加一
            digits[idx]++;
        return digits;
    }
};
```

681. 最近时刻（中等）

1. 题目描述

   给定一个形如 “HH:MM” 表示的时刻，利用当前出现过的数字构造下一个距离当前时间最近的时刻。每个出现数字都可以被无限次使用。

你可以认为给定的字符串一定是合法的。例如，“01:34” 和 “12:09” 是合法的，“1:34” 和 “12:9” 是不合法的。

```
样例 1:
输入: "19:34"
输出: "19:39"
解释: 利用数字 1, 9, 3, 4 构造出来的最近时刻是 19:39，是 5 分钟之后。结果不是 19:33 因为这个时刻是 23 小时 59 分钟之后。

样例 2:
输入: "23:59"
输出: "22:22"
解释: 利用数字 2, 3, 5, 9 构造出来的最近时刻是 22:22。 答案一定是第二天的某一时刻，所以选择可构造的最小时刻。
```

2. 简单实现

将时间用`h1 h2:m1 m2`表示，则可以知道它们的取值范围为：

- h1 = 0,1时，h2 = 0-9；h1 = 2时，h2 = 0-3
- m1 = 0-5
- m2 = 0-9

分情况讨论

- 如果m2不是四个数中最大的数，则直接将m2替换为**大于m2的最小的数即可**
- 否则，需要考虑进位，即看m1
  - 如果存在大于m1且小于等于5的最小的那个数，则把m1替换为该数，m2替换为最小的数即可
  - 否则，需要继续考虑进位，即看h2
    - 如果存在大于h2的最小的那个数
      - 如果h1<2，则把h2替换为该数，m1m2替换为最小的数即可
      - 如果h1=2，
        - 如果这个数小于4，则把h2替换为该数，m1m2替换为最小的数即可
        - 否则，继续考虑进位，即看h1
    - 否则，继续考虑进位，即看h1
      - 如果存在大于h1且小于等于2的最小的那个数，则把h1替换为该数，h2m1m2替换为最小的数即可
      - 否则，相当于进入第二天，把h1h2m1m2都替换成最小的数

```
class Solution {
public:
    string nextClosestTime(string time) {
        char h1 = time[0];
        char h2 = time[1];
        char m1 = time[3];
        char m2 = time[4];
        vector<char> v = {h1, h2, m1, m2};
        sort(v.begin(), v.end());
        auto it = upper_bound(v.begin(), v.end(), m2);
        if(it != v.end())
            m2 = *it;
```

```
        else{
            it = upper_bound(v.begin(), v.end(), m1);
            if(it != v.end() && *it < '6'){
                m1 = *it;
                m2 = v[0];
            }
            else{
                it = upper_bound(v.begin(), v.end(), h2);
                if(it != v.end() && (h1<'2' || (h1=='2' && *it < '4'))){
                    h2 = *it;
                    m1 = m2 = v[0];
                }
                else{
                    it = upper_bound(v.begin(), v.end(), h1);
                    if(it != v.end() && it <= '2'){
                        h1 = *it;
                        h2 = m1 = m2 = v[0
                    }
                    else
                        h1 = h2 = m1 = m2 = v[0];
                }
            }
        }
        string ans = "";
        ans += h1;
        ans += h2;
        ans += ':';
        ans += m1;
        ans += m2;
        return ans;
    }
};
```

42. 接雨水（困难）

1. 题目描述

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。

上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的高度图，在这种情况下，可以接 6 个单位的雨水（蓝色部分表示雨水）。 感谢 Marcos 贡献此图。

```
示例:
输入: [0,1,0,2,1,0,1,3,2,1,2,1]
输出: 6
```

2. 简单实现

```
class Solution {
public:
```

```
    //一层一层地填雨水
    int trap(vector<int>& height) {
        //高度h单调递减的栈，存储<h, idx>
        //相当于保存一个阶梯（每层平面是蓄满水的），每层阶梯高h，且右端点为idx
        stack<pair<int,int>> hs;
        int ans = 0;
        for(int i = 0; i < height.size(); i++){
            while(!hs.empty() && height[i] >= hs.top().first){
                //出现更高的柱子，相等时要更新右端点，写在循环里逻辑是相同的，节省代码
                int h = hs.top().first;//左边第一层平面的高度，以这个平面为底
                hs.pop();
                if(!hs.empty())
                    //以左边第二层平面右端点idx为左界，与新出现的柱子之间围成一个长方形可蓄水区
域
                    //其长度为i-idx-1，高度取决于左右界的最低高度
                    ans += (min(hs.top().first, height[i]) - h) * (i -
hs.top().second - 1);
            }
            hs.push(make_pair(height[i], i));
        }
        return ans;
    }
};
```

146. LRU缓存机制（中等）

1. 题目描述

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。它应该支持以下操作： 获取数据 get 和 写入数据 put 。

获取数据 get(key) - 如果密钥 (key) 存在于缓存中，则获取密钥的值（总是正数），否则返回 -1。 写入数据 put(key, value) - 如果密钥不存在，则写入其数据值。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。

进阶:你是否可以在 O(1) 时间复杂度内完成这两种操作?

```
示例:
LRUCache cache = new LRUCache( 2 /* 缓存容量 */ );
cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // 返回  1
cache.put(3, 3);    // 该操作会使得密钥 2 作废
cache.get(2);       // 返回 -1 (未找到)
cache.put(4, 4);    // 该操作会使得密钥 1 作废
cache.get(1);       // 返回 -1 (未找到)
cache.get(3);       // 返回  3
cache.get(4);       // 返回  4
```

2. 简单实现

```
class LRUCache {
```

```
private:
    int cap;
    // 双链表：装着 (key, value) 元组
    list<pair<int, int>> cache;
    // 哈希表：key 映射到 (key, value) 在 cache 中的位置
    unordered_map<int, list<pair<int, int>>::iterator> map;
public:
    LRUCache(int capacity) {
        this->cap = capacity;
    }

    int get(int key) {
        auto it = map.find(key);
        // 访问的 key 不存在
        if (it == map.end()) return -1;
        // key 存在，把 (k, v) 换到队头
        pair<int, int> kv = *map[key];
        cache.erase(map[key]);
        cache.push_front(kv);
        // 更新 (key, value) 在 cache 中的位置
        map[key] = cache.begin();
        return kv.second; // value
    }

    void put(int key, int value) {

        /* 要先判断 key 是否已经存在 */
        auto it = map.find(key);
        if (it == map.end()) {
            /* key 不存在，判断 cache 是否已满 */
            if (cache.size() == cap) {
                // cache 已满，删除尾部的键值对腾位置
                // cache 和 map 中的数据都要删除
                auto lastPair = cache.back();
                int lastKey = lastPair.first;
                map.erase(lastKey);
                cache.pop_back();
            }
            // cache 没满，可以直接添加
            cache.push_front(make_pair(key, value));
            map[key] = cache.begin();
        } else {
            /* key 存在，更改 value 并换到队头 */
            cache.erase(map[key]);
            cache.push_front(make_pair(key, value));
            map[key] = cache.begin();
        }
    }
};
```

4. 寻找两个有序数组的中位数（困难）

1. 题目描述

给定两个大小为 m 和 n 的有序数组 nums1 和 nums2。

请你找出这两个有序数组的中位数，并且要求算法的时间复杂度为 O(log(m + n))。

你可以假设 nums1 和 nums2 不会同时为空。

```
示例 1:
nums1 = [1, 3]
nums2 = [2]
则中位数是 2.0

示例 2:
nums1 = [1, 2]
nums2 = [3, 4]
则中位数是 (2 + 3)/2 = 2.5
```

2. 正确解法

可以当作在找nums1和nums2中第K大的数，只不过K这里取特殊值，接下来写的实现代码适用于任意K值

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int m = nums1.size();
        int n = nums2.size();
        // if (m > n) { // to ensure m<=n
        //     swap(nums1, nums2);
        //     swap(m,n);
        // }
        int aim = (m + n + 1) / 2;
        //int l = 0, r = m;
        int l = max(0, aim-n), r = min(m, aim);//如果前面不保证m<n，则这里要对lr进行限制
        while (l <= r) {
            //nums1[0..l-1]与nums2[0...j-1]为两个数组的前aim个数
            //需要满足nums1[0..l-1]<=nums2[j] 且 nums2[0...j-1]<=nums1[i]
            int i = l + (r - l) / 2;
            int j = aim - i;
            if (i < r && nums2[j-1] > nums1[i])
                l = i + 1; // i is too small
            else if (i > l && nums1[i-1] > nums2[j])
                r = i - 1; // i is too big
            else { // i is perfect
                int maxLeft = 0;//第aim大的数，就是i/j左侧的最大的数
                if (i == 0) maxLeft = nums2[j-1];
                else if (j == 0) maxLeft = nums1[i-1];
                else maxLeft = max(nums1[i-1], nums2[j-1]);
                if ( (m + n) % 2 == 1 )
                    return maxLeft;

                int minRight = 0;//第aim+1大的数，就是i/j右侧的最小的数
                if (i == m) minRight = nums2[j];
                else if (j == n) minRight = nums1[i];
```

```
                else minRight = min(nums2[j], nums1[i]);

                return (maxLeft + minRight) / 2.0;
            }
        }
        return 0.0;
    }
};
```

1007. 行相等的最少多米诺旋转（中等）

1. 题目描述

   在一排多米诺骨牌中，A[i] 和 B[i] 分别代表第 i 个多米诺骨牌的上半部分和下半部分。（一个多米诺是两个从 1 到 6 的数字同列平铺形成的 —— 该平铺的每一半上都有一个数字。）

   我们可以旋转第 i 张多米诺，使得 A[i] 和 B[i] 的值交换。

   返回能使 A 中所有值或者 B 中所有值都相同的最小旋转次数。

   如果无法做到，返回 -1.

```
示例 1：
输入：A = [2,1,2,4,2,2], B = [5,2,6,2,3,2]
输出：2
解释：
图一表示：在我们旋转之前， A 和 B 给出的多米诺牌。
如果我们旋转第二个和第四个多米诺骨牌，我们可以使上面一行中的每个值都等于 2，如图二所示。

示例 2：
输入：A = [3,5,1,2,3], B = [3,6,3,3,4]
输出：-1
解释：
在这种情况下，不可能旋转多米诺牌使一行的值相等。
```

提示：

- 1 <= A[i], B[i] <= 6
- 2 <= A.length == B.length <= 20000

2. 简单实现

   假设`A[0] = n1, B[0] = n2`，则如果能做到，则后面所有的牌中都必须至少包含n1或n2中的一个

```
class Solution {
public:
    int minDominoRotations(vector<int>& A, vector<int>& B) {
        int n = A.size();
        int n1 = A[0];
        int n2 = B[0];
        int n1_cnt = 0, n2_cnt = 0;//记录这n张牌的n个位置能贡献多少个n1,n2，用于防止同一张牌出现相同数字的重复计数，如A=[2,2,1], B=[3,2,3]
        int a_n1 = 0, a_n2 = 0;//记录A中n1和n2的数量
        int b_n1 = 0, b_n2 = 0;//记录B中n1和n2的数量
```

```
        for(int i = 0; i < n; i++){
            if(A[i] == n1 || B[i] == n1)
                n1_cnt++;
            if(A[i] == n2 || B[i] == n2)
                n2_cnt++;
            if(n1_cnt < i+1 && n2_cnt < i+1)//提前终止
                return -1;
            if(A[i] == n1)
                a_n1++;
            if(A[i] == n2)
                a_n2++;
            if(B[i] == n1)
                b_n1++;
            if(B[i] == n2)
                b_n2++;
        }
        if(n1_cnt < n && n2_cnt < n)
            return -1;
        //能旋转，最少的旋转次数就等于A/B中n1/n2缺失的最少数量
        if(n1_cnt == n)
            return min(min(a_n1, n-a_n1), min(b_n1, n-b_n1));
        else
            return min(min(a_n2, n-a_n2), min(b_n2, n-b_n2));
    }
};
```

683. K个空花盆（困难）

1. 题目描述

花园里有 N 个花盆，每个花盆里都有一朵花。这 N 朵花会在 N 天内依次开放，每天有且仅有一朵花会开放并且会一直盛开下去。

给定一个数组 flowers 包含从 1 到 N 的数字，每个数字表示在那一天开放的花所在的花盆编号。

例如， flowers[i] = x 表示在第 i 天盛开的花在第 x 个花盆中，i 和 x 都在 1 到 N 的范围内。

给你一个整数 k，请你输出在哪一天恰好有两朵盛开的花，他们中间间隔了 k 朵花并且都没有开放。

如果不存在，输出 -1。

```
样例 1:
输入:
flowers: [1,3,2]
k: 1
输出: 2
解释: 在第二天，第一朵和第三朵花都盛开了。

样例 2:
输入:
flowers: [1,2,3]
k: 1
输出: -1
```

注释 :给定的数组范围是 [1, 20000]。

2. 简单实现

看作一个区间划分问题，一个区间代表连续的未开花的花盆区间，每次开花都会将一个连续区间分割成两部分，那么，只要判断这两部分的长度是否有等于K的即可

```
class Solution {
public:
    int kEmptySlots(vector<int>& bulbs, int K) {
        int n = bulbs.size();
        if(K >= n-1) return -1;
        map<int, int> intervals;//<区间左端点， 区间右端点>
        intervals[1] = n;//初始化为[1,n]，两端闭区间
        for(int i = 0; i < n && !intervals.empty(); i++){
            auto it = intervals.upper_bound(bulbs[i]);
            it--;//bulbs[i]应该插入的区间
            int l = it->first;
            int r = it->second;
            if(l > bulbs[i] || r < bulbs[i])//在讨论范围外
                continue;
            intervals.erase(l);
            if(l == bulbs[i]){//正好是左端点
                if(K == 0 && l != 0)//bulbs[i-1]必然是开花的
                    return i+1;
                if(r-l == K && r != n)//右端点不能是n,因为n右侧没有开花的
                    return i+1;
                if(r-l > K)//长度小于K的区间没意义，不讨论
                    intervals[l+1] = r;
            }
            else if(r > bulbs[i]){//分为两端区间
                if((bulbs[i]-l == K && l != 1) || (r - bulbs[i] == K && r != n))
                    return i+1;
                if(bulbs[i]-l > K)//左半段
                    intervals[l] = bulbs[i]-1;
                if(r - bulbs[i] > K)//右半段
                    intervals[bulbs[i]+1] = r;
            }
            else if(r == bulbs[i]){//与左端点是同理的
                if(K == 0)
                    return i+1;
                if(r-l == K && l != 1)
                    return i+1;
                if(r-l > K)
                    intervals[l] = r-1;
            }
        }
        return -1;
    }
};
```

3. 简化版

看题解之后，觉得自己写半天真垃圾，思想明明一样，看看人家看看你、、、

```
class Solution {
public:
    int kEmptySlots(vector<int>& bulbs, int K) {
        set<int> s;
        for (int i = 0; i < bulbs.size(); ++i) {
            auto it = s.insert(bulbs[i]).first;
            if (it != end(s) && *next(it) - *it == K + 1 ||
                it != begin(s) && *it - *prev(it) == K + 1) {
                return i + 1;
            }
        }
        return -1;
    }
};
```

4. 滑动窗口——O(N)

```
class Solution {
public:
    int kEmptySlots(vector<int>& bulbs, int k) {
        int n = bulbs.size();
        int days[n];//存储每盆花开放的日期
        for (int i = 0; i < n; ++i)
            days[bulbs[i] - 1] = i + 1;
        int ans = INT_MAX, st = 0, en = k + 1;
        for (int i = 1; en < n; ++i) {
            if (days[i] > days[st] && days[i] > days[en]) continue;
            if (i == en) // end of window, valid ans
                ans = min(ans, max(days[st], days[en]));
            // next window
            st = i;
            en = st + k + 1;
        }
        return ans < INT_MAX ? ans : -1;
    }
};
```

399. 除法求值（中等）

1. 题目描述

给出方程式 A / B = k, 其中 A 和 B 均为代表字符串的变量， k 是一个浮点型数字。根据已知方程式求解问题，并返回计算结果。如果结果不存在，则返回 -1.0。

```
示例 :
给定 a / b = 2.0, b / c = 3.0
问题: a / c = ?, b / a = ?, a / e = ?, a / a = ?, x / x = ?
返回 [6.0, 0.5, -1.0, 1.0, -1.0 ]
```

输入为: `vector<pair<string, string>> equations, vector<double>& values, vector<pair<string, string>> queries`(方程式，方程式结果，问题方程式)， 其中 `equations.size() == values.size()`，即方程式的长度与方程式结果长度相等（程式与结果一一对应），并且结果值均为正数。以上为方程式的描述。 返回`vector<double>`类型。

基于上述例子，输入如下：

- equations(方程式) = [["a", "b"], ["b", "c"]],
- values(方程式结果) = [2.0, 3.0],
- queries(问题方程式) = [["a", "c"], ["b", "a"], ["a", "e"], ["a", "a"], ["x", "x"]].

输入总是有效的。你可以假设除法运算中不会出现除数为0的情况，且不存在任何矛盾的结果。

2. 简单实现

一开始想要把所有变量都表示为同一个变量的倍数关系，进而想到了用有向图，a->b的权重n表示`a*n = b`，且有`b->a的权重为1/n`，则queries里的`[a,b]`就是从b到a的任意一条路径的权重之积，而寻找路径可以用dfs

```
class Solution {
public:
    bool find = false;
    //寻找从start到end的路径，当前所在节点为temp，ans记录当前路径的权重之积
    void dfs(const string& start, const string& end, string& temp,
 unordered_map<string, vector<pair<string, double>>>& g, double& ans,
unordered_set<string>& visited){
        if (find) return;
        if (temp == end){
            find = true;
            return;
        }
        string back = temp;
        for(auto it = g[temp].begin(); it != g[temp].end(); it++){
            if(visited.find(it->first) == visited.end()){
                visited.insert(it->first);
                ans *= it->second;
                temp = it->first;
                dfs(start, end, temp, g, ans, visited);
                if(find) return;
                temp = back;
                ans /= it->second;
            }
        }
    }
    vector<double> calcEquation(vector<vector<string>>& equations, vector<double>&
values, vector<vector<string>>& queries) {
        unordered_map<string, vector<pair<string, double>>> g;
        for(int i = 0; i < equations.size(); i++){//构造有向图
            g[equations[i][0]].push_back(make_pair(equations[i][1], 1/values[i]));
            g[equations[i][1]].push_back(make_pair(equations[i][0], values[i]));
        }
        vector<double> ans(queries.size());
        for(int i = 0; i < queries.size(); i++){
            string start = queries[i][1];
            string end = queries[i][0];
```

```
            if(g.find(start) == g.end() || g.find(end) == g.end())//起始点有一个不在图
中
                ans[i] = -1.0;
            else if(start == end)
                ans[i] = 1.0;
            else{
                unordered_set<string> visited;
                visited.insert(start);
                ans[i] = 1.0;
                find = false;
                dfs(start, end, start, g, ans[i], visited);
                if(!find)//非连通图可能存在起始点间不存在路径的情况
                    ans[i] = -1.0;
            }
        }
        return ans;
    }
};
```

3. 其他方法——带权的并查集

给我的理解其实就是在图的基础上，对同一连通图内的节点，选择一个节点做根节点，其他所有节点都直接于其带权重相连

```
class Solution {
public:
    unordered_map<int,pair<int,double>>father;
    int Find(int x){
        if(x != father[x].first){
            int t = father[x].first;
            father[x].first = Find(father[x].first);
            father[x].second *= father[t].second;

            return father[x].first;
        }
        return x;
    }
    void Union(int e1,int e2,double result){
        int f1 = Find(e1);
        int f2 = Find(e2);
        if(f1 != f2){
            father[f2].first = f1;
            father[f2].second = father[e1].second * result / father[e2].second;
        }
    }
    vector<double> calcEquation(vector<vector<string>>& equations, vector<double>&
values, vector<vector<string>>& queries) {
        vector<double> ans;
        unordered_map<string,int> now;
        int numString = 1;
        for(int i = 0 ; i < equations.size() ; i++){
            int e1,e2;
            if(now[equations[i][0]] == 0){
```

```
                now[equations[i][0]] = numString++;
                father[numString-1].first = numString-1;
                father[numString-1].second = 1;
            }
            if(now[equations[i][1]] == 0){
                now[equations[i][1]] = numString++;
                father[numString-1].first = numString-1;
                father[numString-1].second = 1;
            }
            e1 = now[equations[i][0]];
            e2 = now[equations[i][1]];
            Union(e1,e2,values[i]);
        }
        for(int i = 0 ; i < queries.size() ; i++){
            if(now[queries[i][0]] == 0 || now[queries[i][1]] == 0){
                ans.push_back(-1.0);
                continue;
            }
            int e1 = now[queries[i][0]];
            int e2 = now[queries[i][1]];
            int f1 = Find(e1);
            int f2 = Find(e2);
            if(f1 != f2)
                ans.push_back(-1.0);
            else{
                double temp = father[e1].second;
                double temp2 = father[e2].second;
                ans.push_back(temp2/temp);
            }
        }
        return ans;
    }
};
```

15. 三数之和（中等）

1. 题目描述

   给你一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c ，使得 a + b + c = 0 ？请你找出所有满足条件且不重复的三元组。

   注意：答案中不可以包含重复的三元组。

   ```
   示例：
   给定数组 nums = [-1, 0, 1, 2, -1, -4]，
   满足要求的三元组集合为：
   [
     [-1, 0, 1],
     [-1, -1, 2]
   ]
   ```

2. 简单实现

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        int n = nums.size();
        if(n < 3) return {};
        sort(nums.begin(), nums.end());//排序
        vector<vector<int>> result;
        for(int i = 0; i < n-2; i++){
            int l = i + 1;
            int r = n-1;
            int aim = -nums[i];
            while(l < r){//滑动窗口
                int num = nums[l]+nums[r];
                if(num == aim){
                    result.push_back({-aim, nums[l++], nums[r--]});
                    while(l < r && nums[l] == nums[l-1])
                        l++;
                    while(r > l && nums[r] == nums[r+1])
                        r--;
                }
                else if(num < aim)
                    l++;
                else
                    r--;
            }
            while(i < n-1 && nums[i] == nums[i+1])
                i++;
        }
        return result;
    }
};
```

5. 最长回文子串（中等）

1. 题目描述

给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

```
示例 1:
输入: "babad"
输出: "bab"
注意: "aba" 也是一个有效答案。

示例 2:
输入: "cbbd"
输出: "bb"
```

2. 简单实现——递归法

```
class Solution {
public:
```

```
    bool judge(string s){
        if(s.size() <= 1) return true;
        int l = 0, r = s.size()-1;
        while(l < r)
            if(s[l++] != s[r--]) return false;
        return true;
    }
    string longestPalindrome(string s) {
        if(s.size() <= 1) return s;
        string case0 = longestPalindrome(s.substr(1, s.size()-1));
        string case1 = s.substr(0, case0.size()+2);
        if(judge(case1)) return case1;
        string case2 = s.substr(0, case0.size()+1);
        if(judge(case2)) return case2;
        return case0;
    }
};
```

3. 中心扩散法

本题最容易想到的一种方法应该就是 **中心扩散法。**
中心扩散法怎么去找回文串?
从每一个位置出发，向两边扩散即可。遇到不是回文的时候结束。举个例子，$str = acdbbdaa$ 我们需要寻找从第一个 `b` （位置为 3）出发最长回文串为多少。怎么寻找?
首先往左寻找与当期位置相同的字符，直到遇到不相等为止。
然后往右寻找与当期位置相同的字符，直到遇到不相等为止。
最后左右双向扩散，直到左和右不相等。如下图所示:

4. 动态规划

```
public String longestPalindrome(String s) {
        if (s == null || s.length() < 2)
            return s;
        int strLen = s.length();
        int maxStart = 0;  //最长回文串的起点
        int maxEnd = 0;    //最长回文串的终点
        int maxLen = 1;  //最长回文串的长度
        boolean[][] dp = new boolean[strLen][strLen];//dp[i][j]表示s[i...j]是否是回文
串
        //注意遍历的顺序，原因是需要dp[i+1][j-1]已知
        for (int r = 1; r < strLen; r++)
            for (int l = 0; l < r; l++) {
                if (s.charAt(l) == s.charAt(r) && (r - l <= 2 || dp[l + 1][r - 1]))
{
                    dp[l][r] = true;
                    if (r - l + 1 > maxLen) {
                        maxLen = r - l + 1;
                        maxStart = l;
                        maxEnd = r;
                    }
                }
            }
        return s.substring(maxStart, maxEnd + 1);
```

```
        }
```

5. manacher算法——O(n)

Manacher（马拉车） 算法

前面解法存在以下缺陷：

1. 由于回文串长度的奇偶性造成了不同性质的对称轴位置，前面解法要对两种情况分别处理。
2. 很多子串被重复多次访问，造成较差的时间效率，例如：

```
字符:   a   b   a   b   a
位置 :  0   1   2   3   4
```

当位置为 1 和 2 时，按中心扩展法，可以看出左边的 `aba` 分别被遍历了一次。

如果我们能改善重复遍历的不足，就很有希望能提高算法的效率。Manacher 正是针对这些问题改进算法。

解决单双两次遍历的问题

首先对字符串做一个预处理，在所有的空隙位置（包括首尾）插入同样的符号，要求这个符号是不会在原串中出现的。这样会使得所有的串都是奇数长度的，并且回文串的中心不会是 `双数`，以插入#号为例：

```
aba  ———>  #a#b#a#
abba ———>  #a#b#b#a#
```

解决重复访问的问题

在前面的基础上，我们认为回文串的中心总是为 `单数`，我们把一个回文串中最左或最右位置的字符与其对称轴的距离称为回文半径，用 `RL` 表示。

用 `RL[i]` 表示以第 `i` 个字符为对称轴的回文串的回文半径。我们一般对字符串从左往右处理，因此这里定义 `RL[i]` 为第 `i` 个字符为对称轴的回文串的最右一个字符与字符 `i` 的距离，如 `aba` 的 `RL[1]=2`，即 `ba`。

对于上面插入分隔符之后的两个串，可以得到 `RL` 数组：

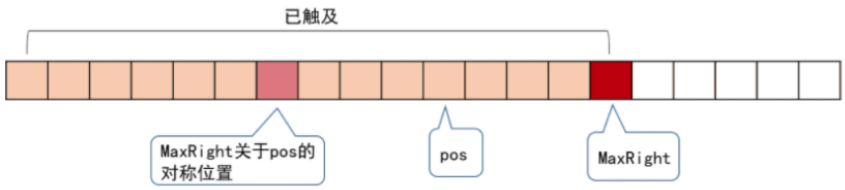
```
字符:   #   a   #   b   #   a   #
RL :    1   2   1   4   1   2   1
RL-1:   0   1   0   3   0   1   0
位置:    0   1   2   3   4   5   6

字符:   #   a   #   b   #   b   #   a   #
RL :    1   2   1   2   5   2   1   2   1
RL-1:   0   1   0   1   4   1   0   1   0
位置:    0   1   2   3   4   5   6   7   8
```

`RL[i]` 的大小总是定义为 `回文串最右的字符位置-回文串的对称轴字符位置+1`，参看上图。

上面我们还求了一下 `RL[i]-1`。通过观察可以发现，`RL[i]-1` 的值，正是在原本那个没有插入过分隔符的串中，以位置 `i` 为对称轴的最长回文串的长度（注意，这里是全串的总长度，不要和 `RL` 半径混在一起了）。

于是问题变成了，怎样 **高效地求的RL数组**。基本思路是利用 **回文串的对称性，扩展回文串**。

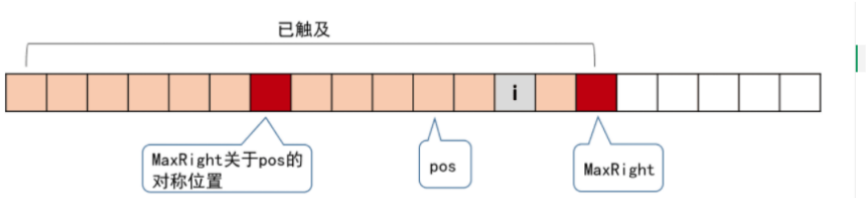
我们再引入一个辅助变量 `MaxRight`，表示当前访问到的所有 `回文子串`，所能触及的最右一个字符的位置。另外还要记录下 `MaxRight` 对应的回文串的对称轴所在的位置，记为 `pos`，它们的位置关系如下。



我们从左往右地访问字符串来求 `RL`，假设当前访问到的位置为 `i`，即要求 `RL[i]`，在对应上图，因为我们是从左到右遍历 `i`，而 `pos` 是遍历到的所有 `回文子串` 中某个对称轴位置（`MaxRight` 最大时），所以必然有 `pos<=i`，所以我们更关注的是，`i` 是在 `MaxRight` 的左边还是右边。我们分情况来讨论。

1）当 `i` 在 `MaxRight` 的左边

可以用下图来刻画：



我们知道，图中两个红色块之间（包括红色块）的串是回文。

并且以 `i` 为对称轴的回文串，是与红色块间的回文串有所重叠的。

我们找到 `i` 关于 `pos` 的对称位置 `j`，这个 `j` 对应的 `RL[j]` 我们是已经算过的。

根据回文串的对称性，以 `i` 为对称轴的回文串和以 `j` 为对称轴的回文串，有一部分是相同的。这里又有两种细分的情况。

1.1）以 `j` 为对称轴的回文串比较短，短到像下图这样

这时我们知道 `RL[i]` 至少不会小于 `RL[j]` ，并且已经知道了部分的以 `i` 为中心的回文串，于是可以令 `RL[i]=RL[j]` 为 `起始半径` 。

又因为 `(j + i) / 2 = pos` ==> `j = 2*pos - i` 得到 `RL[i]=RL2*pos - i]` 。

因此我们以 `RL[i]=RL2*pos - i]` 为起始半径`，继续往左右两边扩展，直到左右两边字符不同，或者到达边界。

1.2）以 `j` 为对称轴的回文串很长，超过了 `MaxRight` 在左侧的对称点

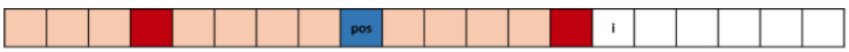
这时，我们只能确定， `MaxRight - i` 的部分是以 `i` 为对称轴的回文半径。

因此我们以 `RL[i] = MaxRight - i` 为 `起始半径` ，继续往左右两边扩展，直到左右两边字符不同，或者到达边界。

综上 `1.1` `1.2` 分析，可以得出：在后面的代码中有体现

```
if (i < MaxRight)
{//  当i在MaxRight的左边
        RL[i] = min(RL[2 * pos - i], MaxRight - i);
}
```

2）当 `i` 在 `MaxRight` 的右边



遇到这种情况，说明以i为对称轴的回文串还没有任何一个部分被访问过，于是只能从i的左右两边开始尝试扩展了，也就是 `RL[i]=1` 。

当左右两边字符不同，或者到达字符串边界时停止。然后更新 `MaxRight` 和 `pos` 。

为了得到字符串，我们还需要一个 `MaxRL` 来记录最大回文串的回文半径。MaxPos 来记录 `MaxRL` 对应的回文串的对称轴所在的位置。

由前面的分析可以知道， `MaxRL- 1` 即为原始最大回文串的长度（注意，这里是全串的总长度，不要和 `RL` 半径混在一起了）。

原始最大回文串的起始位为 `(MaxPos - MaxRL + 1) / 2`

```
class Solution {
public:
    string longestPalindrome(string s) {
        int len = s.length();
```

```
            if (len <= 1) return s;
            // 预处理
            string s1;
            for (int i = 0; i < len; i++) {
                s1 += "#";
                s1 += s[i];
            }
            s1 += "#";

            len = s1.length();
            int MaxRight = 0;                   // 当前访问到的所有回文子串，所能触及的最右一个字符的位置
            int pos = 0;                    // MaxRight对应的回文串的对称轴所在的位置
            int MaxRL = 0;                      // 最大回文串的回文半径
            int MaxPos = 0;                     // MaxRL对应的回文串的对称轴所在的位置
            int* RL = new int[len];             // RL[i]表示以第i个字符为对称轴的回文串的回文半径
            memset(RL, 0, len * sizeof(int));
            for (int i = 0; i < len; i++) {
                if (i < MaxRight)// 1) 当i在MaxRight的左边
                    RL[i] = min(RL[2 * pos - i], MaxRight - i);
                else// 2) 当i在MaxRight的右边
                    RL[i] = 1;
                // 尝试扩展RL[i]，注意处理边界
                while (i - RL[i] >= 0  // 可以把RL[i]理解为左半径,即回文串的起始位不能小于0
                    && i + RL[i] < len // 同上,即回文串的结束位不能大于总长
                    && s1[i - RL[i]] == s1[i + RL[i]]// 回文串特性，左右扩展，判断字符串是否相同

                    )
                    RL[i] += 1;

                // 更新MaxRight, pos
                if (RL[i] + i - 1 > MaxRight){
                    MaxRight = RL[i] + i - 1;
                    pos = i;
                }
                // 更新MaxRL, MaxPos
                if (MaxRL <= RL[i]){
                    MaxRL = RL[i];
                    MaxPos = i;
                }
            }
            return s.substr((MaxPos - MaxRL + 1) / 2, MaxRL - 1);
        }
    };
```

23. 合并K个排序链表（困难）

1. 题目描述

   合并 k 个排序链表，返回合并后的排序链表。请分析和描述算法的复杂度。

```
示例:
输入:
[
  1->4->5,
  1->3->4,
  2->6
]
输出: 1->1->2->3->4->4->5->6
```

2. 简单实现

自定义大顶堆，时间O(nlogk)，空间O(k)

也可以用自底向上的归并排序的方法，可以把空间复杂度变O(1)

```
class Solution {
public:
    struct cmp {
        bool operator () (const ListNode* a, const ListNode* b) {
            return a->val >= b->val;
        }
    };
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        priority_queue<ListNode*, vector<ListNode*>, cmp> q;
        for(int i = 0; i < lists.size(); i++){
            if(lists[i])
                q.push(lists[i]);
        }
        ListNode* ans = new ListNode(-1);
        ListNode* cur = ans;
        while(q.size() > 1){
            cur->next = q.top();
            q.pop();
            cur = cur->next;
            if(cur->next)
                q.push(cur->next);
        }
        if(q.size() == 1)
            cur->next = q.top();
        return ans->next;
    }
};
```

844. 比较含退格的字符串（简单）

1. 题目描述

给定 S 和 T 两个字符串，当它们分别被输入到空白的文本编辑器后，判断二者是否相等，并返回结果。 # 代表退格字符。

```
示例 1:
输入: S = "ab#c", T = "ad#c"
```

```
输出：true
解释：S 和 T 都会变成 “ac”。

示例 2：
输入：S = "ab##", T = "c#d#"
输出：true
解释：S 和 T 都会变成 “”。

示例 3：
输入：S = "a##c", T = "#a#c"
输出：true
解释：S 和 T 都会变成 “c”。

示例 4：
输入：S = "a#c", T = "b"
输出：false
解释：S 会变成 “c”，但 T 仍然是 “b”。
```

提示：

- 1 <= S.length <= 200
- 1 <= T.length <= 200
- S 和 T 只含有小写字母以及字符 '#'。

2. 简单实现

从后往前遍历处理，只要求判断是否相等，因此得到的字符串是倒序的也无妨

```
class Solution {
public:
    void getString(string& S, string& s){
        int idx = S.size() - 1;
        int cnt = 0;
        while(idx >= 0){
            if(S[idx] == '#')
                cnt++;
            else if(cnt)
                cnt--;
            else
                s += S[idx];
            idx--;
        }
    }
    bool backspaceCompare(string S, string T) {
        string s = "", t = "";
        getString(S, s);
        getString(T, t);
        return s == t;
    }
};
```

20. 有效的括号（简单）

1. 题目描述

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，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。
- 注意空字符串可被认为是有效字符串。

```
示例 1:
输入: "()"
输出: true

示例 2:
输入: "()[]{}"
输出: true

示例 3:
输入: "(]"
输出: false

示例 4:
输入: "([)]"
输出: false

示例 5:
输入: "{[]}"
输出: true
```

2. 简单实现

```
class Solution {
public:
    bool isValid(string s) {
        stack<char> cache;
        for(int i = 0; i < s.size(); i++){
            if(s[i] == '(' || s[i] == '[' || s[i] == '{')
                cache.push(s[i]);
            else if(s[i] == ')'){
                if(cache.empty() || cache.top() != '(')
                    return false;
                else
                    cache.pop();
            }
            else if(s[i] == ']'){
                if(cache.empty() || cache.top() != '[')
                    return false;
                else
                    cache.pop();
            }
            else if(s[i] == '}'){
                if(cache.empty() || cache.top() != '{')
                    return false;
```

```
                else
                    cache.pop();
            }
        }
        return cache.empty();
    }
};
```

394. 字符串解码（中等）

1. 题目描述

   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。

```
示例:
s = "3[a]2[bc]", 返回 "aaabcbc".
s = "3[a2[c]]", 返回 "accaccacc".
s = "2[abc]3[cd]ef", 返回 "abcabccdcdcdef".
```

2. 简单实现

```
class Solution {
public:
    string decodeString(string s) {
        stack<string> cache;
        string cur = "";
        int idx = 0;
        while(idx < s.size()){
            if(s[idx] >= '0' && s[idx] <= '9'){//数字
                if(cur !=  "")
                    cache.push(cur);//之前的字符入栈
                cur = "";
                while(s[idx] != '[')//统计数字
                    cur += s[idx++];
                cache.push(cur);//数字入栈
                cur = "";
            }
            else if(s[idx] == ']'){//需要重复了
                int cnt = stoi(cache.top());//重复次数
                cache.pop();
                string temp = "";
                while(cnt--)
                    temp += cur;
                cur = temp;
```

```
                while(!cache.empty() && (cache.top()[0] < '0' || cache.top()[0] >
'9')){
                    //和前面的字符串（相当于同[]级）合并
                    cur = cache.top() + cur;
                    cache.pop();
                }
            }
            else//字母，继续加
                cur += s[idx];
            idx++;
        }
        return cur;
    }
};
```

222. 完全二叉树的节点个数（中等）

1. 题目描述

   给出一个完全二叉树，求出该树的节点个数。

   完全二叉树的定义：在完全二叉树中，除了最底层节点可能没填满外，其余每层节点数都达到最大值，并且最下面一层的节点都集中在该层最左边的若干位置。若最底层为第 h 层，则该层包含 1~ 2^h 个节点。

```
示例:
输入:
    1
   / \
  2   3
 / \  /
4  5 6
输出: 6
```

2. 简单实现

   先求树高，再利用哈夫曼编码，用二分法求最后一层的叶子节点数

```
class Solution {
public:
    bool check(TreeNode* root, int n, int h){//检查高为h的树上，哈夫曼编码为n的叶子节点是
否存在
        TreeNode* cur = root;
        while(--h){
            if(n & (1<<(h-1)))//1表示向右走
                cur = cur->right;
            else//0表示向左走
                cur = cur->left;
        }
        return cur != NULL;
    }
    int countNodes(TreeNode* root) {
        int h = 0;
```

```
        TreeNode* cur = root;
        while(cur){
            ++h;
            cur = cur->left;
        }
        if(h == 0) return 0;//树为空
        int max_leaf = pow(2, h-1);
        int l = 0, r = max_leaf;//高为h的满二叉树的叶子节点从左向右编号为0...2^(h-1)
        while(l < r){//找到第一个为空的叶子节点
            int mid = l + (r - l) / 2;
            if(check(root, mid, h))
                l = mid + 1;
            else
                r = mid;
        }
        return max_leaf - 1 + l;//max_leaf-1正好是树的前h-1层的节点总数，l为第h层的节点数
    }
};
```

340. 最多包含K个不同字符的最长子串（困难）

1. 题目描述

   给定一个字符串 s，找出 至多 包含 k 个不同字符的最长子串 T。

```
示例 1:
输入: s = "eceba", k = 2
输出: 3
解释: 则 T 为 "ece"，所以长度为 3。

示例 2:
输入: s = "aa", k = 1
输出: 2
解释: 则 T 为 "aa"，所以长度为 2。
```

2. 简单实现

   晃滑动窗口

```
class Solution {
public:
    int lengthOfLongestSubstringKDistinct(string s, int k) {
        unordered_map<char, int> contain;//当前窗口内各个字符对应的最大idx
        int len = s.size();
        if(len <= k) return s.size();
        if(k == 0) return 0;
        int ans = 0;
        int l = 0, r = 0;
        while(r < len){
            contain[s[r]] = r;//新字符加入窗口
            if(contain.size() > k){//不符合要求，去掉窗口内最新出现的idx最小的字符
                ans = max(ans, r-l);//更新ans
```

```
                int min_idx = len;
                for(auto it = contain.begin(); it != contain.end(); it++)
                    min_idx = min(min_idx, it->second);
                l = min_idx + 1;
                contain.erase(s[l-1]);//删除
            }
            r++;
        }
        ans = max(ans, r-l);
        return ans;
    }
};
```

686. 重复叠加字符串匹配（简单）

1. 题目描述

   给定两个字符串 A 和 B, 寻找重复叠加字符串A的最小次数，使得字符串B成为叠加后的字符串A的子串，如果不存在则返回 -1。

   举个例子，A = "abcd"，B = "cdabcdab"。

   答案为 3， 因为 A 重复叠加三遍后为 “abcdabcdabcd”，此时 B 是其子串；A 重复叠加两遍后为"abcdabcd"，B 并不是其子串。

   注意: A 与 B 字符串的长度在1和10000区间范围内。

2. 简单实现

   要想B是nA的子串，必须满足nA的长度大于等于B，若nA是第一个长度大于等于B的字符串，且B不是nA的子串，则若B也不是(n+1)A的子串，则答案一定是不存在，证明的话大概是

   - 若存在，则B可以表示成三部分组成 `A[idx1...] + (n-1)A + A[...idx2]`，其中第一、三部分分别表示A的后缀和前缀，且可以为空；n为大于等于0的正整数
   - 因此可以想象一下nA就是只带前缀，(n+1)A就是前后缀都有

```
class Solution {
public:
    int repeatedStringMatch(string A, string B) {
        int lena = A.size();
        int lenb = B.size();
        int cur_len = lena;
        string cur_a = A;
        int ans = 1;
        while(cur_len < lenb){
            cur_a += A;
            cur_len += lena;
            ans++;
        }
        if(cur_a.find(B) != cur_a.npos)
            return ans;
        cur_a += A;
        cur_len += lena;
        if(cur_a.find(B) != cur_a.npos)
```

```
                return ans+1;
            return -1;
        }
    };
```

253. 会议室II（中等）

1. 题目描述

   给定一个会议时间安排的数组，每个会议时间都会包括开始和结束的时间 [[s1,e1],[s2,e2],...] (si < ei)，为避免会议冲突，同时要考虑充分利用会议室资源，请你计算至少需要多少间会议室，才能满足这些会议安排。

```
示例 1:
输入: [[0, 30],[5, 10],[15, 20]]
输出: 2

示例 2:
输入: [[7,10],[2,4]]
输出: 1
```

2. 简单实现
   - 实际上是一个图着色优化问题，把时间冲突的会议作为图中相邻的节点，则需要的最少的会议室数量就是图着色（相邻节点颜色不能相同）需要的最少颜色数
   - 由图着色算法可知，贪心涂色法是可解的
   - 因此，本题无需构造图，只需要也按照贪心的思想来合并区间即可，具体做法为
     - 对intervals按起始时间从小到大、结束时间从小到大的顺序排列
     - 用`vector<vector<int>> rooms`保存目前为止需要的会议室以及各个会议室的已经被占用的起止时间，初始化rooms为intervals[0]
     - 依次遍历intervals[1...]，判断能否安排入rooms的某一个，如果可以，则安排，否则，需要新找一个room
     - 最后返回rooms数组的大小

```
class Solution {
public:
    bool judge(vector<int>& room, const vector<int>& interval){//判断interval能否安排在room里
        //由于已经排序，因此一定有room[0] <= interval[0] < interval[1]
        if(interval[0] >= room[1]){
            room[1] = interval[1];
            return true;
        }
        return false;
    }
    static bool cmp(vector<int>& a, vector<int>& b){//排序
        if(a[0] != b[0])
            return a[0] < b[0];
        return a[1] < b[1];
    }
    int minMeetingRooms(vector<vector<int>>& intervals) {
```

```
        int size = intervals.size();
        if(size <= 1) return size;
        sort(intervals.begin(), intervals.end(), cmp);
        vector<vector<int>> rooms = {intervals[0]};
        for(int i = 1; i < size; i++){
            bool find = false;
            for(auto it = rooms.begin(); it != rooms.end(); it++){
                if(judge(*it, intervals[i])){
                    find = true;
                    break;
                }
            }
            if(!find)
                rooms.push_back(intervals[i]);
        }
        return rooms.size();
    }
};
```

843. 猜猜这个单词（困难）

1. 题目描述

   这个问题是 LeetCode 平台新增的交互式问题 。

   我们给出了一个由一些独特的单词组成的单词列表，每个单词都是 6 个字母长，并且这个列表中的一个单词将被选作秘密。

   你可以调用 master.guess(word) 来猜单词。你所猜的单词应当是存在于原列表并且由 6 个小写字母组成的类型字符串。

   此函数将会返回一个整型数字，表示你的猜测与秘密单词的准确匹配（值和位置同时匹配）的数目。此外，如果你的猜测不在给定的单词列表中，它将返回 -1。

   对于每个测试用例，你有 10 次机会来猜出这个单词。当所有调用都结束时，如果您对 master.guess 的调用不超过 10 次，并且至少有一次猜到秘密，那么您将通过该测试用例。

   除了下面示例给出的测试用例外，还会有 5 个额外的测试用例，每个单词列表中将会有 100 个单词。这些测试用例中的每个单词的字母都是从 'a' 到 'z' 中随机选取的，并且保证给定单词列表中的每个单词都是唯一的。

```
示例 1:
输入: secret = "acckzz", wordlist = ["acckzz","ccbazz","eiowzz","abcczz"]
解释:
master.guess("aaaaaa") 返回 -1, 因为 "aaaaaa" 不在 wordlist 中.
master.guess("acckzz") 返回 6, 因为 "acckzz" 就是秘密，6个字母完全匹配。
master.guess("ccbazz") 返回 3, 因为 "ccbazz" 有 3 个匹配项。
master.guess("eiowzz") 返回 2, 因为 "eiowzz" 有 2 个匹配项。
master.guess("abcczz") 返回 4, 因为 "abcczz" 有 4 个匹配项。
我们调用了 5 次master.guess，其中一次猜到了秘密，所以我们通过了这个测试用例。
```

2. 简单实现

   - 首先，根据题意可以发现一些规律：
     - 如果`master.guess(s)=0`，则任何与s在某个位置字母相同的字符串都不可能是答案

    - 如果`master.guess(s)=x, x>0`，则只有与s恰好有x个位置字母相同的字符串才有可能是答案
- 因此，可以定义dis(a,b)为字符串a,b在相同位置字母相同的个数
- 构造`unordered_map<int, unordered_map<int, vector<int>>> g; //<idx1, <dis, vector<idx2>>>`，表示对idx1所对应的字符而言，与其dis为dis的那些字符串的下标构成`vector<idx2>`
- 用`set<int> valid`记录当前的可行解下标，初始化为整个wrodlist
- 如果`master.guess(s)=x, x>=0`，则可行解变为`valid`与`g[idx(s)][x]`的交集
- 现在的问题在于，如何选择s？是随机选，还是用一些优化策略？经给实验，直接随机选是有可能达不到题目要求的，因此需要采用贪心优化策略`getBestChoice(g, valid, match)`，其中match为上一次guess得到的结果
    - 对当前可行解valid中的每一个元素s，计算它下一步可能构成的各个可行解(guess(s) = [match...6]，因为按现在的策略，match不可能变小)，然后取所有可能下的最大值cur_max(s)
    - 然后选取最小的cur_max(s)，即选取下一步所有可能的可行解的最大元素个数最小的那个s，对这个s进行guess，能最大程度缩小下一步的可行解

```
class Solution {
public:
    int getDis(string& a, string& b){//获取dis(a, b)
        int ans = 0;
        for(int i = 0; i < 6; i++)
            if(a[i] == b[i])
                ans++;
        return ans;
    }
    int getBestChoice(unordered_map<int, unordered_map<int, vector<int>>> g,
set<int>& valid, int& match){
        int ans = -1;
        int min_num = 101;
        for(auto it = valid.begin(); it != valid.end(); it++){
            int cur_max = 0;
            for(int dis = match; dis < 6; dis++){
                int cur = 0;
                for(int i = 0; i < g[*it][dis].size(); i++){
                    if(valid.find(g[*it][dis][i]) != valid.end())//只有在valid里的才
计数
                        ++cur;
                }
                cur_max = max(cur_max, cur);
            }
            if(cur_max < min_num){
                ans = *it;
                min_num = cur_max;
            }
        }
        return ans;
    }
    void findSecretWord(vector<string>& wordlist, Master& master) {
        unordered_map<int, unordered_map<int, vector<int>>> g;//<idx1, <dis, idx2>>
        int size = wordlist.size();
```

```
        set<int> valid;
        for(int i = 0; i < size; i++){//初始化
            valid.insert(i);
            for(int j = i+1; j < size; j++){
                int dis = getDis(wordlist[i], wordlist[j]);
                g[i][dis].push_back(j);
                g[j][dis].push_back(i);
            }
        }
        int cnt = 10;
        int match = 0;
        while(cnt--){
            int idx = getBestChoice(g, valid, match);
            match = master.guess(wordlist[idx]);
            if(match == 6) break;//猜对了
            set<int> temp(g[idx][match].begin(), g[idx][match].end());
            set<int> intersection;
            set_intersection(valid.begin(), valid.end(),
                             temp.begin(), temp.end(),
                             inserter(intersection,intersection.begin()));
            valid = intersection;
        }
        while(cnt--)
            master.guess(wordlist[0]);
    }
};
```

17. 电话号码的字母组合（中等）

1. 题目描述

   给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。

   给出数字到字母的映射如下（与电话按键相同）。注意 1 不对应任何字母。


```
示例:
输入: "23"
输出: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
```

说明:尽管上面的答案是按字典序排列的，但是你可以任意选择答案输出的顺序。

2. 简单实现

```
class Solution {
public:
    vector<string> dictionary;
    Solution()
    {
        dictionary.resize(2);
        dictionary.push_back("abc");
        dictionary.push_back("def");
        dictionary.push_back("ghi");
        dictionary.push_back("jkl");
        dictionary.push_back("mno");
        dictionary.push_back("pqrs");
        dictionary.push_back("tuv");
        dictionary.push_back("wxyz");
    }
    vector<string> ans;
    void dfs(string& s, int idx, string temp){
        if(idx == s.size()-1)
            for(int i = 0; i < dictionary[s[idx]-'0'].size(); i++)
```

```
                ans.push_back(temp + dictionary[s[idx]-'0'][i]);
        else
            for(int i = 0; i < dictionary[s[idx]-'0'].size(); i++)
                dfs(s, idx+1, temp + dictionary[s[idx]-'0'][i]);
    }
    vector<string> letterCombinations(string digits) {
        int len = digits.length();
        if(len <= 0) return ans;
        dfs(digits, 0, "");
        return ans;
    }
};
```

10. 正则表达式匹配（困难）

1. 题目描述

给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 '.' 和 '*' 的正则表达式匹配。

- '.' 匹配任意单个字符
- '*' 匹配零个或多个前面的那一个元素

所谓匹配，是要涵盖 整个 字符串 s的，而不是部分字符串。

说明:

- s 可能为空，且只包含从 a-z 的小写字母。
- p 可能为空，且只包含从 a-z 的小写字母，以及字符 . 和 *。

```
示例 1:
输入:
s = "aa"
p = "a"
输出: false
解释: "a" 无法匹配 "aa" 整个字符串。

示例 2:
输入:
s = "aa"
p = "a"
输出: true
解释: 因为 '' 代表可以匹配零个或多个前面的那一个元素，在这里前面的元素就是 'a'。因此，字符串 "aa" 可被视为 'a' 重复了一次。

示例 3:
输入:
s = "ab"
p = "."
输出: true
解释: "." 表示可匹配零个或多个（'*'）任意字符（'.'）。

示例 4:
输入:
s = "aab"
```

```
p = "cab"
输出: true
解释: 因为 '*' 表示零个或多个，这里 'c' 为 0 个, 'a' 被重复一次。因此可以匹配字符串 "aab"。

示例 5:
输入:
s = "mississippi"
p = "misisp*."
输出: false
```

2. 简单实现

记得以前做的时候是各种复杂判断做的，做了很久，各种边界条件，很恶心，这次尝试用NFA做了，代码上简单了写，但是细节上还是有很多值得注意的地方，一不小心就会出错（主要还是之前学的形式语言与自动机的课的一些细节问题有点忘了）

- 用`unordered_map<int, multimap<char, int>> g`保存NFA的状态转移，因为同一个输入可以对应不同的转换状态，因此用multimap，int保存各个状态id
- 对给定模式串p构造NFA的方法：
  - 初始化状态id=0，从左向右遍历P
    - 当前字符char不为'.'，且下一个字符不是'*'：直接构造当前`id[char]=id+1`，即状态id通过char转移为状态id+1
    - 当前字符char不为'.'，且下一个字符是'*'：构造`id[char]=id`和`id['#']=id+1`，表示既可以通过char扔保留在当前状态，又可以不通过char（用'#'表示空字符）直接进入下一个状态（在自动机课程里学过，这样的模块代表匹配任意多个字符char）
    - 对当前字符char为'.'的情况，处理逻辑和前面是类似的，只不过要对a-z都构造相应的状态转移
  - NFA的终止状态为最后一个状态id
- 利用NFA判断s的方法：
  - 用`unordered_set<int> curStates`记录当前状态，初始化为从0开始不通过任何字符就可以到达的所有状态（注意这里不能直接初始化为0）
  - 遍历s，在每一步中，对每个curStates里的状态都根据NFA做状态转移，记录在nextStates里，记得不要忘记那些不通过任何字符就可以直接转移到的状态，然后更新curStates
  - 遍历完s后，查看curStates可以到达的状态，如果里有终止状态则匹配，否则不匹配

```
class Solution {
public:
    bool isMatch(string s, string p) {
        unordered_map<int, multimap<char, int>> g;//NFA
        //构造NFA
        int id = 0;
        int len_p = p.size();
        for(int i = 0; i < len_p; ){
            if(i == len_p - 1 || p[i+1] != '*'){
                if(p[i] != '.')
                    g[id].insert(make_pair(p[i], id+1));
                else
                    for(int j = 0; j < 26; j++)
                        g[id].insert(make_pair('a'+j, id+1));
                id++;
```

```
                i++;
            }
            else{
                if(p[i] != '.')
                    g[id].insert(make_pair(p[i], id));
                else
                    for(int j = 0; j < 26; j++)
                        g[id].insert(make_pair('a'+j, id));
                g[id].insert(make_pair('#', id+1));
                id++;
                i += 2;
            }
        }
        int end = id;//终止状态
        //利用NFA判断s
        unordered_set<int> curStates;
        curStates.insert(0);
        int tmp = 0;
        auto tmp_it = g[tmp].find('#');
        while(tmp_it != g[tmp].end()){//从tmp开始能够不通过任何字符转移到的所有状态
            curStates.insert(tmp_it->second);
            tmp = tmp_it->second;
            tmp_it = g[tmp].find('#');
        }
        for(int i = 0; i < s.size() && !curStates.empty(); i++){//遍历s
            unordered_set<int> nextStates;
            for(auto it = curStates.begin(); it != curStates.end(); it++){//每一个可
能的当前状态
                int cnt = g[*it].count(s[i]);//当前状态通过s[i]走一步后可以到达的状态数
                if(cnt > 0){
                    auto cur = g[*it].find(s[i]);
                    while(cnt--){//遍历所有可以一步到达的状态
                        int tmp = cur->second;
                        nextStates.insert(tmp);//加入nextStates
                        //把从tmp开始不通过任何字符能转移到的所有状态也加入nextStates
                        auto tmp_it = g[tmp].find('#');
                        while(tmp_it != g[tmp].end()){
                            tmp = tmp_it->second;
                            nextStates.insert(tmp);//通过'#'只能转移到一个状态
                            tmp_it = g[tmp].find('#');
                        }
                        cur++;
                    }
                }
            }
            curStates = nextStates;
        }
        //判断遍历完s后所有可以到达的状态，里面有没有终止状态
        for(auto it = curStates.begin(); it != curStates.end(); it++)
            if(*it == end)
                return true;
        return false;
    }
```

```
};
```

3. C++递归

```
bool isMatch(char * s, char * p){
    if (!*p) return !*s;
    bool first_match = *s && (*s == *p || *p == '.');
    if (*(p+1) == '*')
        return isMatch(s, p+2) || (first_match && isMatch(++s, p));
    else
        return first_match && isMatch(++s, ++p);
}
```

4. 动态规划

懒得截图了，https://leetcode-cn.com/problems/regular-expression-matching/solution/dong-tai-gui-hua-zui-xiang-xi-jie-da-you-jian-zhi-/

489. 扫地机器人（困难）

1. 题目描述

房间（用格栅表示）中有一个扫地机器人。格栅中的每一个格子有空和障碍物两种可能。

扫地机器人提供4个API，可以向前进，向左转或者向右转。每次转弯90度。

当扫地机器人试图进入障碍物格子时，它的碰撞传感器会探测出障碍物，使它停留在原地。

请利用提供的4个API编写让机器人清理整个房间的算法。

```
interface Robot {
  // 若下一个方格为空，则返回true，并移动至该方格
  // 若下一个方格为障碍物，则返回false，并停留在原地
  boolean move();

  // 在调用turnLeft/turnRight后机器人会停留在原位置
  // 每次转弯90度
  void turnLeft();
  void turnRight();

  // 清理所在方格
  void clean();
}
```

```
示例:
输入:
room = [
  [1,1,1,1,1,0,1,1],
  [1,1,1,1,1,0,1,1],
  [1,0,1,1,1,1,1,1],
  [0,0,0,1,0,0,0,0],
  [1,1,1,1,1,1,1,1]
],
```

```
row = 1,
col = 3
解析:
房间格栅用0或1填充。0表示障碍物，1表示可以通过。
机器人从row=1，col=3的初始位置出发。在左上角的一行以下，三列以右。
```

注意:

- 输入只用于初始化房间和机器人的位置。你需要“盲解”这个问题。换而言之，你必须在对房间和机器人位置一无所知的情况下，只使用4个给出的API解决问题。
- 扫地机器人的初始位置一定是空地。
- 扫地机器人的初始方向向上。
- 所有可抵达的格子都是相连的，亦即所有标记为1的格子机器人都可以抵达。
- 可以假定格栅的四周都被墙包围。

2. 简单实现

DFS，由于不知道当前位置，可以用当前相对于起点的相对位置表示当前坐标

注意机器人只能连续走，到死角时要原路返回——即每一步dfs后都要恢复状态

```
class Solution {
public:
    vector<vector<int>> dirs = {{-1,0}, {0,1}, {1,0}, {0,-1}};//顺时针方向的改变
    void dfs(Robot& robot, unordered_set<string>& visited, int x, int y, int dir){
        robot.clean();//清扫
        for(int i = 1; i <= 4; i++){//转四次后返回进入函数的初始状态
            robot.turnRight();//右转
            int new_dir = (dir + i) % 4;//下一步的方向
            int xx = x + dirs[new_dir][0];//下一步位置x
            int yy = y + dirs[new_dir][1];//下一步位置y
            string next = to_string(xx) + ',' + to_string(yy);
            if(visited.find(next) != visited.end())
                continue;
            visited.insert(next);//记录探查过
            if(robot.move()){//是空地
                dfs(robot, visited, xx, yy, new_dir);//继续扫
                //恢复原状
                robot.turnRight();
                robot.turnRight();//向后转
                robot.move();//前进
                robot.turnRight();
                robot.turnRight();//向后转
            }
        }
    }
    void cleanRoom(Robot& robot) {
        unordered_set<string> visited;
        visited.insert("0,0");
        dfs(robot, visited, 0, 0, 0);
    }
};
```

975. 奇偶跳（困难）

1. 题目描述

给定一个整数数组 A，你可以从某一起始索引出发，跳跃一定次数。在你跳跃的过程中，第 1、3、5... 次跳跃称为奇数跳跃，而第 2、4、6... 次跳跃称为偶数跳跃。

你可以按以下方式从索引 i 向后跳转到索引 j（其中 i < j）：

- 在进行奇数跳跃时（如，第 1，3，5... 次跳跃），你将会跳到索引 j，使得 A[i] <= A[j]，A[j] 是可能的最小值。如果存在多个这样的索引 j，你只能跳到满足要求的最小索引 j 上。
- 在进行偶数跳跃时（如，第 2，4，6... 次跳跃），你将会跳到索引 j，使得 A[i] >= A[j]，A[j] 是可能的最大值。如果存在多个这样的索引 j，你只能跳到满足要求的最小索引 j 上。
- （对于某些索引 i，可能无法进行合乎要求的跳跃。）

如果从某一索引开始跳跃一定次数（可能是 0 次或多次），就可以到达数组的末尾（索引 A.length - 1），那么该索引就会被认为是好的起始索引。

返回好的起始索引的数量。

```
示例 1:
输入: [10,13,12,14,15]
输出: 2
解释:
从起始索引 i = 0 出发，我们可以跳到 i = 2，（因为 A[2] 是 A[1], A[2], A[3], A[4] 中大于或等于 A[0] 的最小值），然后我们就无法继续跳下去了。
从起始索引 i = 1 和 i = 2 出发，我们可以跳到 i = 3，然后我们就无法继续跳下去了。
从起始索引 i = 3 出发，我们可以跳到 i = 4，到达数组末尾。
从起始索引 i = 4 出发，我们已经到达数组末尾。
总之，我们可以从 2 个不同的起始索引（i = 3, i = 4）出发，通过一定数量的跳跃到达数组末尾。

示例 2:
输入: [2,3,1,1,4]
输出: 3
解释:
从起始索引 i=0 出发，我们依次可以跳到 i = 1, i = 2, i = 3:

在我们的第一次跳跃（奇数）中，我们先跳到 i = 1，因为 A[1] 是（A[1], A[2], A[3], A[4]）中大于或等于 A[0] 的最小值。

在我们的第二次跳跃（偶数）中，我们从 i = 1 跳到 i = 2，因为 A[2] 是（A[2], A[3], A[4]）中小于或等于 A[1] 的最大值。A[3] 也是最大的值，但 2 是一个较小的索引，所以我们只能跳到 i = 2，而不能跳到 i = 3。

在我们的第三次跳跃（奇数）中，我们从 i = 2 跳到 i = 3，因为 A[3] 是（A[3], A[4]）中大于或等于 A[2] 的最小值。

我们不能从 i = 3 跳到 i = 4，所以起始索引 i = 0 不是好的起始索引。

类似地，我们可以推断:
从起始索引 i = 1 出发， 我们跳到 i = 4，这样我们就到达数组末尾。
从起始索引 i = 2 出发， 我们跳到 i = 3，然后我们就不能再跳了。
从起始索引 i = 3 出发， 我们跳到 i = 4，这样我们就到达数组末尾。
从起始索引 i = 4 出发，我们已经到达数组末尾。
总之，我们可以从 3 个不同的起始索引（i = 1, i = 3, i = 4）出发，通过一定数量的跳跃到达数组末尾。
```

示例 3：
输入：[5,1,3,4,2]
输出：3
解释：
我们可以从起始索引 1，2，4 出发到达数组末尾。

提示：

- 1 <= A.length <= 20000
- 0 <= A[i] < 100000

2. 简单实现

假设`dp[i][0/1]`表示从A[i]开始，先进行 偶跳跃/奇跳跃 ，能否到达终点，则状态转移方程为：

- `dp[i][0] = dp[A[i]通过偶跳跃到达的下一个索引j(j>i)][1]`
- `dp[i][1] = dp[A[i]通过奇跳跃到达的下一个索引j(j>i)][0]`

则最后要求的答案为：`dp[i][1]=1` 的起始点i的个数

因此，现在要解决的问题就是：如何求A[i]通过 偶跳跃/奇跳跃 后到达的下一个索引j

观察发现，从后往前遍历A[i]，A[i]下一步会跳到的地方只与已经遍历过的A[i+1,...]的值以及当前是奇跳跃还是偶跳跃有关：

- 奇跳跃：跳到略大于A[i]的(大于A[i]的数里最小的)、索引最小的值
- 偶跳跃：跳到略小于A[i]的(小于A[i]的数里最大的)、索引最小的值

为了达到O(nlogn)的时间复杂度，需要上述的每一次跳跃步骤在O(logn)内完成，因而想到了用可排序的set，插入和查找都是O(logn)

```
class Solution {
public:
    int oddEvenJumps(vector<int>& A) {
        int size = A.size();
        if(size == 1) return 1;
        vector<vector<bool>> dp(size, vector<bool>(2, false));
        dp[size-1][0] = true;//从终点开始无奇偶跳跃都为true
        dp[size-1][1] = true;
        set<pair<int, int>> idxs;//升序存储已经遍历过的<A[i], i>数据对
        idxs.insert(make_pair(A[size-1], size-1));
        int ans = 1;
        for(int i = size-2; i >= 0; i--){//从后向前遍历
            auto it = idxs.upper_bound(make_pair(A[i], i));//返回的指针严格大于搜索值
            if(it == idxs.end()){//无法进行奇跳跃
                //偶跳跃
                it--;
                int next = it->first;//遍历过的略小于A[i]的数，即小于A[i]的数里最大的那个
                //要一直找到值为该数的索引最小的那个数
                while(it != idxs.begin() && it->first == next)
                    it--;
                if(it != idxs.begin() || (it == idxs.begin() && it->first != next))
                    it++;
                dp[i][0] = dp[it->second][1];
            }
            else if(it->first > A[i]){//可以奇跳跃到更大的数
```

```
                dp[i][1] = dp[it->second][0];//当前所指就是遍历过的略大于A[i]的数里索引最小的
                if(it != idxs.begin()){//可以进行偶跳跃到更小的数
                    it--;
                    int next = it->first;
                    while(it != idxs.begin() && it->first == next)
                        it--;
                    if(it != idxs.begin() || (it == idxs.begin() && it->first != next)){
                            it++;
                    }
                    dp[i][0] = dp[it->second][1];
                }
            }
            else{//遍历过的数里有和A[i]相等的，奇偶跳跃都会到该数上
                dp[i][0] = dp[it->second][1];
                dp[i][1] = dp[it->second][0];
            }
            if(dp[i][1]) ans++;
            idxs.insert(make_pair(A[i], i));
        }
        return ans;
    }
};
```

3. 也可以从前向后遍历，用单调栈做，见网上题解

139. 单词拆分（中等）

1. 题目描述

   给定一个非空字符串 s 和一个包含非空单词列表的字典 wordDict，判定 s 是否可以被空格拆分为一个或多个在字典中出现的单词。

   说明：

   - 拆分时可以重复使用字典中的单词。
   - 你可以假设字典中没有重复的单词。

```
示例 1:
输入: s = "leetcode", wordDict = ["leet", "code"]
输出: true
解释: 返回 true 因为 "leetcode" 可以被拆分成 "leet code"。

示例 2:
输入: s = "applepenapple", wordDict = ["apple", "pen"]
输出: true
解释: 返回 true 因为 "applepenapple" 可以被拆分成 "apple pen apple"。
     注意你可以重复使用字典中的单词。

示例 3:
输入: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
输出: false
```

2. 简单实现

Backtrack会超时，感觉backtrack经常超时，因为一旦剪枝不好就是O(2^n)、、、、

之前做的时候用的dp，鬼知道我之前怎么想的，当时的我真聪明啊~

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        int size = wordDict.size();
        if(size <= 0) return false;
        unordered_set<string> dict;
        int max_len = -1;
        for(int i = 0; i < size; i++){
            dict.insert(wordDict[i]);
            max_len = max(max_len, int(wordDict[i].size()));
        }
        vector<bool> dp = vector<bool>(s.size(), false);
        for(int i = 0; i < s.size(); i++){
            if(dict.count(s.substr(0, i+1)) > 0){
                dp[i] = true;
                continue;
            }
            for(int j = i-1; j>=0 && i-j<=max_len; j--){
                if(dp[j] == true && dict.count(s.substr(j+1,i-j)) > 0){
                        dp[i] = true;
                        break;
                }
            }
        }
        return dp[s.size()-1];
    }
};
```

159. 最多包含两个不同字符的最长子串（中等）

1. 题目描述

给定一个字符串 s ，找出 至多 包含两个不同字符的最长子串 t 。

```
示例 1:
输入: "eceba"
输出: 3
解释: t 是 "ece"，长度为3。

示例 2:
输入: "ccaabbb"
输出: 5
解释: t 是 "aabbb"，长度为5。
```

2. 简单实现——滑动窗口，更新idx

```
class Solution {
public:
    int lengthOfLongestSubstringTwoDistinct(string s) {
        int len = s.size();
        if(len <= 2) return len;
        char c1 = s[0];
        int idx1 = 0;
        char c2 = '#';
        int idx2 = -1;
        int l = 0;
        int ans = 0;
        for(int i = 1; i < len; i++){
            if(s[i] == c1)
                idx1 = i;
            else if(s[i] == c2)
                idx2 = i;
            else if(c2 == '#'){
                c2 = s[i];
                idx2 = i;
            }
            else{
                ans = max(ans, i-l);
                if(idx1 < idx2){
                    l = idx1+1;
                    c1 = s[i];
                    idx1 = i;
                }
                else{
                    l = idx2+1;
                    c2 = s[i];
                    idx2 = i;
                }
            }
        }
        ans = max(ans, len-l);
        return ans;
    }
};
```

163. 缺失的区间（中等）

1. 题目描述

   给定一个排序的整数数组 nums ， 其中元素的范围在 闭区间 [lower, upper] 当中，返回不包含在数组中的缺失区间。

   ```
   示例:
   输入: nums = [0, 1, 3, 50, 75], lower = 0 和 upper = 99,
   输出: ["2", "4->49", "51->74", "76->99"]
   ```

2. 简单实现

用map按序保存各区间即可，注意安全处理

```
class Solution {
public:
    vector<string> findMissingRanges(vector<int>& nums, int lower, int upper) {
        int n = nums.size();
        map<int, int> intervals;//当前缺失的各个区间
        intervals[lower] = upper;
        for(int i = 0; i < n; i++){
            auto it = intervals.upper_bound(nums[i]);//it--为nums[i]可以插入的区间
            if(it == intervals.begin())//[2->5],[7->9]中插入1的情况
                continue;
            it--;
            if(it->second < nums[i])//[2->5],[7->9]中插入6的情况
                continue;
            int l = it->first;
            int r = it->second;
            intervals.erase(it);
            if(l == r)//l=r=nums[i]
                continue;
            else if(nums[i] == l)
                intervals[l+1] = r;
            else if(nums[i] == r)
                intervals[l] = r-1;
            else{
                intervals[l] = nums[i] - 1;
                intervals[nums[i]+1] = r;
            }
        }
        vector<string> ans;
        for(auto it = intervals.begin(); it != intervals.end(); it++){
            if(it->first != it->second)
                ans.push_back(to_string(it->first) + "->" + to_string(it->second));
            else
                ans.push_back(to_string(it->first));
        }
        return ans;
    }
};
```

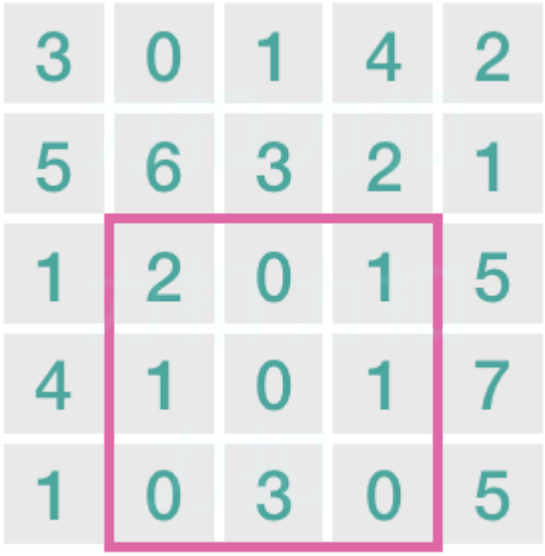
3. 其他解法

看网上题解，因为nums是有序的，所以以两个数为上下边界，对nums数组直接遍历构造缺失部分，时间复杂度是O(N)，要注意复杂的边界情况

308. 二维区域和检索-可变（困难）

1. 题目描述

给你一个 2D 矩阵 matrix，请计算出从左上角 (row1, col1) 到右下角 (row2, col2) 组成的矩形中所有元素的和。



上述粉色矩形框内的，该矩形由左上角 (row1, col1) = (2, 1) 和右下角 (row2, col2) = (4, 3) 确定。其中，所包括的元素总和 sum = 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
update(3, 2, 2)
sumRegion(2, 1, 4, 3) -> 10
```

注意:

- 矩阵 matrix 的值只能通过 update 函数来进行修改
- 你可以默认 update 函数和 sumRegion 函数的调用次数是均匀分布的
- 你可以默认 row1 ≤ row2，col1 ≤ col2

2. 简单实现

只实现了用多个一维树状数组的代码，二维的没写成功

为每行构造树状数组，求面积时一行一行累加

```
class TreeArray{//树状数组
public:
    TreeArray(vector<int>& nums){
        this->n = nums.size();
        data = vector<int>(n+1, 0);
        for(int i = 0; i < n; i++)
            update(i+1, nums[i]);
    }
    int lowbit(int x) {
        return x&(-x);
    }
```

```
    void update(int idx, int num) {
        while(idx <= n){
            data[idx] += num;
            idx += lowbit(idx);
        }
    }
    int getsum(int idx) {
        int sum = 0;
        while(idx > 0) {
            sum += data[idx];
            idx -= lowbit(idx);
        }
        return sum;
    }
private:
    vector<int> data;
    int n;
};

class NumMatrix {
public:
    NumMatrix(vector<vector<int>>& matrix) {
        this->matrix = matrix;
        for(int i = 0; i < matrix.size(); i++)
            rows_sum.push_back(new TreeArray(matrix[i]));
    }

    void update(int row, int col, int val) {
        rows_sum[row]->update(col+1, val-matrix[row][col]);
        matrix[row][col] = val;
    }

    int sumRegion(int row1, int col1, int row2, int col2) {
        int ans = 0;
        for(int i = row1; i <= row2; i++)
            ans += rows_sum[i]->getsum(col2+1) - rows_sum[i]->getsum(col1);
        return ans;
    }
private:
    vector<vector<int>> matrix;
    vector<TreeArray*> rows_sum;
};
```

3. 二维树状数组解法

面积为四个矩阵的容斥

```
#define lowbit(x) ((x) & (-x))
class NumMatrix {
public:
    int n, m;
    vector<vector<int>> M;//树状数组
    vector<vector<int>> matrix;
```

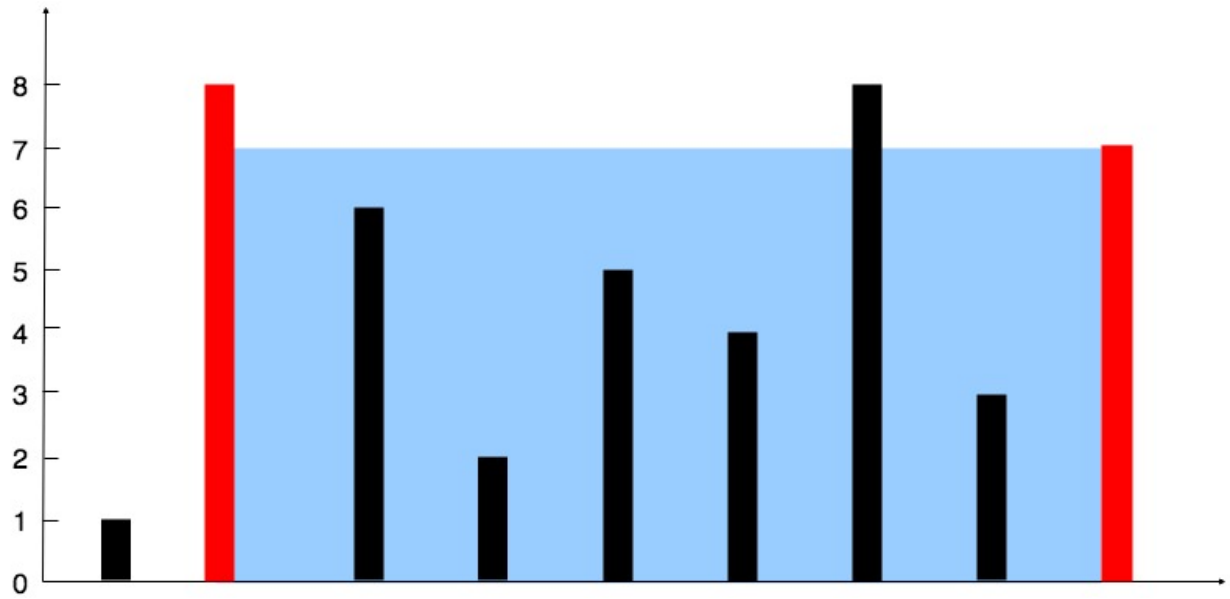
```
    NumMatrix(vector<vector<int>> &matrix) : matrix(matrix) {
        n = matrix.size();
        if (n) m = matrix[0].size();
        else m = 0;
        M = vector<vector<int>>(n + 1, vector<int>(m + 1));
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < m; ++j)
                modify(i + 1, j + 1, matrix[i][j]);
    }
    void modify(int i, int j, int delta) {
        for (int x = i; x <= n; x += lowbit(x))
            for (int y = j; y <= m; y += lowbit(y))
                M[x][y] += delta;
    }
    int getsum(int i, int j) {
        int ret = 0;
        for (int x = i; x; x -= lowbit(x))
            for (int y = j; y; y -= lowbit(y))
                ret += M[x][y];
        return ret;
    }
    void update(int x, int y, int v) {
        int delta = v - matrix[x][y];
        matrix[x][y] = v;
        modify(x + 1, y + 1, delta);
    }
    int sumRegion(int x1, int y1, int x2, int y2) {
        return getsum(x2 + 1, y2 + 1) - getsum(x1, y2 + 1) - getsum(x2 + 1, y1) +
getsum(x1, y1);
    }
};
```

11. 盛水最多的容器（中等）

1. 题目描述

   给你 n 个非负整数 a1，a2，...，an，每个数代表坐标中的一个点 (i, ai) 。在坐标内画 n 条垂直线，垂直线 i 的两个端点分别为 (i, ai) 和 (i, 0)。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。

   说明：你不能倾斜容器，且 n 的值至少为 2。



图中垂直线代表输入数组 [1,8,6,2,5,4,8,3,7]。在此情况下，容器能够容纳水（表示为蓝色部分）的最大值为 49。

```
示例:
输入: [1,8,6,2,5,4,8,3,7]
输出: 49
```

2. 简单实现

神奇！！！双指针从两端向中间即可！！

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int ans = -1;
        int l = 0;
        int r = height.size()-1;
        while(l < r){
            int temp = (r - l) * min(height[l], height[r]);
            if(temp > ans) ans = temp;
            if(height[l] <= height[r])
                l++;
            else
                r--;
        }
        return ans;
    }
};
```

53. 最大子序和（简单）

1. 题目描述

给定一个整数数组 nums ， 找到一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

```
示例:
输入: [-2,1,-3,4,-1,2,1,-5,4],
输出: 6
解释: 连续子数组 [4,-1,2,1] 的和最大，为 6。
```

进阶:如果你已经实现复杂度为 O(n) 的解法，尝试使用更为精妙的分治法求解。

2. 简单实

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int ans = INT_MIN;//有可能只有负数，必须初始化为最小值
        int cur = 0;
        for(int i = 0; i < nums.size(); i++){
            cur += nums[i];
            ans = max(ans, cur);
            if(cur < 0)
                cur = 0;
        }
        return ans;
    }
};
```

346. 数据流中的移动平均值（简单）

1. 题目描述

给定一个整数数据流和一个窗口大小，根据该滑动窗口的大小，计算其所有整数的移动平均值。

```
示例:
MovingAverage m = new MovingAverage(3);
m.next(1) = 1
m.next(10) = (1 + 10) / 2
m.next(3) = (1 + 10 + 3) / 3
m.next(5) = (10 + 3 + 5) / 3
```

2. 简单实现

双端队列，记得用double保存sum

```
class MovingAverage {
public:
    double sum;
    int size;
    deque<int> q;
    MovingAverage(int size) {
        sum = 0;
        this->size = size;
    }
    double next(int val) {
```

```
        if(q.size() == size){
            sum -= q.front();
            q.pop_front();
        }
        q.push_back(val);
        sum += val;
        return sum / q.size();
    }
};
```

54. 螺旋矩阵（中等）

1. 题目描述

给定一个包含 m x n 个元素的矩阵（m 行, n 列），请按照顺时针螺旋顺序，返回矩阵中的所有元素。

```
示例 1:
输入:
[
 [ 1, 2, 3 ],
 [ 4, 5, 6 ],
 [ 7, 8, 9 ]
]
输出: [1,2,3,6,9,8,7,4,5]

示例 2:
输入:
[
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9,10,11,12]
]
输出: [1,2,3,4,8,12,11,10,9,5,6,7]
```

2. 简单实现

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        int m = matrix.size();
        if(m <= 0) return {};
        int n = matrix[0].size();
        int full = m * n;
        vector<int> ans(full);
        int idx = 0;
        int pad = 0;
        while(idx < full){
            for(int j = pad; j < n - pad; j++)//l->r
                ans[idx++] = matrix[pad][j];
            if(idx >= full) break;
```

```
            for(int i = pad + 1; i < m-pad; i++)//u->d
                ans[idx++] = matrix[i][n - 1 - pad];
            if(idx >= full) break;

            for(int j = n - 2 - pad; j >= pad; j--)//r->l
                ans[idx++] = matrix[m-1-pad][j];
            if(idx >= full) break;

            for(int i = m- 2 - pad; i > pad; i--)//d->u
                ans[idx++] = matrix[i][pad];

            pad++;
        }
        return ans;
    }
};
```