

## 面试题 17.01 不用加号的加法（简单）

### 1. 题目描述

设计一个函数把两个数字相加。不得使用 + 或者其他算术运算符。

示例：

输入：a = 1, b = 1

输出：2

提示：

- a, b 均可能是负数或 0
- 结果不会溢出 32 位整数

### 2. 简单实现

位运算

```
class Solution {
public:
    int add(int a, int b) {
        int ans = 0;
        int c = 0;
        for(int idx = 0; idx < 32; idx++){
            int a_cur = a & 1;
            int b_cur = b & 1;
            if(a_cur == 0 && b_cur == 0){
                ans |= c<<idx;
                c = 0;
            }
            else if(a_cur == 1 && b_cur == 1){
                ans |= c<<idx;
                c = 1;
            }
            else
                if(c == 0) ans |= 1<<idx;
            a = a>>1;
            b = b>>1;
        }
        return ans;
    }
};
```

### 3. 最优解法

ps:异或也叫半加运算，其运算法则相当于不带进位的二进制加法：所以异或常被认作不进位加法。

不能用加法，所以只能用二进制进位来算。把相加和进位分开，分成两步。

step1: $a \oplus b$ ，完成不进位加法。step2: $a \& b$ ，完成进位的运算。step3:把step2左移一位，模拟正常加法的向前进一位。一直到进行到进位没有为止，也就是step3=0的时候，说明全部进位完成，加法u全部算完了。

## 代码

```
class Solution {
public:
    int add(int a, int b) {
        if(b==0)
            return a;
        int step1=0, step2=0, step3=0;
        while(b!=0) {
            step1 = a^b;
            step2 = a&b;
            step3 = (unsigned int)step2 << 1;
            a = step1;
            b = step3;
        }
        return step1;
    }
};
```

## 面试题 17.04 消失的数字（简单）

### 1. 题目描述

数组nums包含从0到n的所有整数，但其中缺了一个。请编写代码找出那个缺失的整数。你有办法在 $O(n)$ 时间内完成吗？

注意：本题相对书上原题稍作改动

示例 1:

输入: [3,0,1]

输出: 2

示例 2:

输入: [9,6,4,2,3,5,7,0,1]

输出: 8

### 2. 简单实现

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

## 面试题 17.05 字母与数字 (中等)

### 1. 题目描述

给定一个放有字符和数字的数组，找到最长的子数组，且包含的字符和数字的个数相同。

返回该子数组，若存在多个最长子数组，返回左端点最小的。若不存在这样的数组，返回一个空数组。

示例 1:

输入:

["A","1","B","C","D","2","3","4","E","5","F","G","6","7","H","I","J","K","L","M"]

输出: ["A","1","B","C","D","2","3","4","E","5","F","G","6","7"]

示例 2:

输入: ["A","A"]

输出: []

提示: array.length <= 100000

### 2. 简单实现

a

```
class Solution {
public:
    bool isdigit(string s){//判断是数字还是字符
        if(s[0] >= '0' && s[0] <= '9')
            return true;
        else
            return false;
    }
    vector<string> findLongestSubarray(vector<string>& array) {
        int len = array.size();
        if(len <= 1) return {};
        vector<int> dp(len);//记录array[0...i]中数字减字符的个数
        if(isdigit(array[0])) dp[0] = 1;
        else dp[0] = -1;
        int max_len = 0;
        int ans_l = 0;
        int ans_r = 0;
        for(int i = 1; i < len; i++){
```

```

        if(isdigit(array[i])) dp[i] = dp[i-1] + 1;
        else dp[i] = dp[i-1] - 1;
        if(dp[i] == 0){
            max_len = i+1;
            ans_r = i+1;
        }
    }
    for(int l = 0; l+max_len+1 < len; l++){
        for(int r = len-1; r >= l+max_len+1; r--){//从右向左找
            if(dp[r] == dp[l]){
                ans_l = l+1;
                ans_r = r+1;
                max_len = r-l;
                break;
            }
        }
    }
    if(ans_r > ans_l)
        return vector<string>(array.begin()+ans_l, array.begin()+ans_r);
    else
        return {};
}
};

```

改进：再加一个哈希表记录各个dp值的最左和最右端点，就可以在O(N)时间内完成

## 面试题 17.06 2出现的次数（中等）

### 1. 题目描述

编写一个方法，计算从 0 到 n (含 n) 中数字 2 出现的次数。

示例：

输入：25

输出：9

解释：(2, 12, 20, 21, 22, 23, 24, 25) (注意 22 应该算作两次)

提示：n <= 10<sup>9</sup>

### 2. 正确解法——数位DP

主要思路是数位dp：

以dp[i]表示n的1~i位组成的数字所包含2的个数，关键点在于推导出dp[i]与dp[i-1]的关系

例如：n = 3478

```
dp[1] == numberOf2sInRange(8)
dp[2] == numberOf2sInRange(78)
dp[3] == numberOf2sInRange(478)
dp[4] == numberOf2sInRange(3478)

dp[i] = f(dp[i-1]) ?
```

下面来分析一下dp[i]与dp[i-1]的关系

根据第i位的取值可分为4种情况：

#### 1. 第i位是0

例如：n = 102, 分析dp[2]和dp[1]的关系，即numberOf2sInRange(02)与numberOf2sInRange(2) (02实际是2，写作02便于理解)

第i位是0，该位取值范围只有这一种可能，由此可得

```
dp[2] = dp[1]
numberOf2sInRange(02) = numberOf2sInRange(2)
```

#### 2. 第i位是1

例如：n = 178, 分析dp[3]和dp[2]的关系，即numberOf2sInRange(178)与numberOf2sInRange(78)

第3位是1，该位可能取0,1两种情况：

```
dp[3] = 当第3位是0，1-2位取00~99时2的次数 + 当第3位是1，1-2位取00~78时2的次数
dp[3] = numberOf2sInRange(99) + dp[2]
numberOf2sInRange(178) = numberOf2sInRange(99) + numberOf2sInRange(78)
```

3. 第i位是2

例如:  $n = 233$ , 分析 $dp[3]$ 和 $dp[2]$ 的关系, 即 $numberOf2sInRange(233)$ 与 $numberOf2sInRange(33)$

```
dp[3] = 第3位取0-1, 1-2位取00~99时2的次数 + 第3位是2, 1-2位取00~33时2在1-2位出现的次数 + 第3位是2, 1-2位取00~99时2在第3位出现的次数  
dp[3] = 2 * numberOf2sInRange(99) + dp[2] + 33 + 1  
numberOf2sInRange(233) = 2 * numberOf2sInRange(99) + numberOf2sInRange(33) + 33 + 1
```

4. 第i位大于2

以  $n = 478$  为例，分析  $dp[3]$  和  $dp[2]$  的关系，即  $numberOf2sInRange(478)$  与  $numberOf2sInRange(78)$

$$\text{dp}[3] = \text{第3位取0-3, 1-2位取00-99时2出现在1-2位的次数} + \text{第3位取4, 1-2位取00-78时2的次数} + \text{第3位取2, 1-2位取00-78时2的次数}$$

总结上面4种情况:

dp[i]与dp[i-1]的关系, 假设n的第i位的值为k

$$dp[i] = k * \text{numberOf2sInRange}(99...9 \text{ 共 } i-1 \text{ 个 } 9) + dp[i-1] + \{n \% 10^{(i-1)} + 1\} \{ \text{若 } k == 2 \} + \{ 10^{(i-1)} \}$$

根据递推公式可以发现，若计算 $dp[i]$ ，不仅要知道 $dp[i-1]$ 还要知道 $numberOf2sInRange(99..9)$ ，所以要同时计算 $numberOf2sInRange(99..9)$ 的值

```
dp[i] = k * numberOf2sInRange(99..9){共i-1个9} + dp[i-1] + {n % 10^(i-1) + 1 }{若k
== 2} + { 10^(i-1) } {若k > 2}
```

```

class Solution {
public:
    int numberOf2sInRange(int n) {
        if(n == 0) return 0;
        int digit = int(log10(n)) + 1; //计算n是几位数
        vector<vector<long>> dp(digit+1, vector<long>(2)); //long防止在计算dp[][1]时溢出

        // dp[i][0] = numberOf2sInRange(n % pow(10, i)) 保存0~n的1~i位组成的数包含2的个数
        // dp[i][1] = numberOf2sInRange(99..9) 保存i位均为9包含2的个数
        dp[1][0] = n % 10 >= 2 ? 1:0;
        dp[1][1] = 1;
        for(int i = 2; i <= digit; i++) {
            int k = n / int(pow(10, i-1)) % 10; //取得第i位数字
            dp[i][0] = k * dp[i-1][1] + dp[i-1][0];
            if(k == 2)
                dp[i][0] += n % int(pow(10, i-1)) + 1;
            else if(k > 2)
                dp[i][0] += int(pow(10, i-1));
            dp[i][1] = 10 * dp[i-1][1] + int(pow(10, i-1)); //计算1~i位均为9的值包含2的个数
        }
        return dp[digit][0];
    }
};

```

## 面试题 17.07 婴儿名字 (中等)

### 1. 题目描述

每年，政府都会公布一万个最常见的婴儿名字和它们出现的频率，也就是同名婴儿的数量。有些名字有多种拼法，例如，John 和 Jon 本质上是相同的名字，但被当成了两个名字公布出来。给定两个列表，一个是名字及对应的频率，另一个是本质相同的名字对。设计一个算法打印出每个真实名字的实际频率。注意，如果 John 和 Jon 是相同的，并且 Jon 和 Johnny 相同，则 John 与 Johnny 也相同，即它们有传递和对称性。在结果列表中，选择字典序最小的名字作为真实名字。

示例：

输入：names = ["John(15)","Jon(12)","Chris(13)","Kris(4)","Christopher(19)"],  
synonyms = [("Jon,John)","(John,Johnny)","(Chris,Kris)","(Chris,Christopher)"]  
输出：["John(27)","Chris(36)"]

提示：names.length <= 100000

### 2. 简单实现

```
class Solution {
public:
    vector<string> trulyMostPopular(vector<string>& names, vector<string>&
synonyms) {
        unordered_map<string, int> cnt; // 合并后各名字出现的次数
        unordered_map<string, string> name_as; // 类似于并查集，且以字典序最小的名字为根
        vector<string> ans;
        for(int i = 0; i < names.size(); i++){ // 初始化
            int idx1 = names[i].find('(');
            int idx2 = names[i].find(idx1, ')');
            string name = names[i].substr(0, idx1);
            int times = stoi(names[i].substr(idx1+1, idx2-idx1-1));
            cnt[name] = {times};
            name_as[name] = name;
        }
        for(int i = 0; i < synonyms.size(); i++){ // 并查集操作
            int idx = synonyms[i].find(',');
            string name1 = synonyms[i].substr(1, idx-1);
            string name2 = synonyms[i].substr(idx+1, synonyms[i].size()-idx-2);
            if(name_as.count(name1) <= 0 || name_as.count(name2) <= 0) // 出现多余名字，不用处理
                continue;
            while(name_as[name1] != name1){ // name1 所属集合
                name_as[name1] = name_as[name_as[name1]];
                name1 = name_as[name1];
            }
            while(name_as[name2] != name2){ // name2 所属集合
                name_as[name2] = name_as[name_as[name2]];
                name2 = name_as[name2];
            }
        }
    }
};
```

```

        if(name1 == name2)//name1和name2同属一个集合，已经合并过，不能再算了，否则计
数错误!
            continue;
        else if(name1 < name2){
            name_as[name2] = name1;
            cnt[name1] = cnt[name1] + cnt[name2];
            cnt.erase(name2);
        }
        else{
            name_as[name1] = name2;
            cnt[name2] = cnt[name1] + cnt[name2];
            cnt.erase(name1);
        }
    }
    for(auto it = cnt.begin(); it != cnt.end(); it++)
        ans.push_back(it->first + '(' + to_string(it->second) + ')');
    return ans;
}
};

```

本题写到一半想到了并查集，如果一开始就想到了并查集，可以先用synonyms构建并查集，再依次处理names

## 面试题 17.08 马戏团人塔（中等）

### 1. 题目描述

有个马戏团正在设计叠罗汉的表演节目，一个人要站在另一人的肩膀上。出于实际和美观的考虑，在上面的人要比下面的人矮一点且轻一点。已知马戏团每个人的身高和体重，请编写代码计算叠罗汉最多能叠几个人。

示例：

输入：height = [65,70,56,75,60,68] weight = [100,150,90,190,95,110]

输出：6

解释：从上往下数，叠罗汉最多能叠 6 层：(56,90)，(60,95)，(65,100)，(68,110)，(70,150)，(75,190)

提示：height.length == weight.length <= 10000

### 2. 简单实现——最长上升子序列

与之前做过的面试题08.13堆箱子 类似，但用那个方法会超时

#### 最长上升子序列leetcode300的动态规划思想的应用

- 这里先使用sort根据height从小到大排序，其中关键的是如果height相同则按照weight从大到小排序（注意一个是从从小到大，一个是从大到小）。
  - 这样安排是为了让后续的根据weight求最大上升子序列时不会选到height一样的数据，即让height相同的数据的weight从大到小排列，让它不满足上升序列的“上升”，这样就巧妙避开了重复选取。
- 最后就是根据weight值求最长上升子序列的长度，即leetcode300的方法。
  - 这里使用了STL函数lower\_bound()代替二分查找来简化代码。



```

class Solution {
public:
    int bestSeqAtIndex(vector<int>& height, vector<int>& weight) {
        if (height.size() == 0 || weight.size() == 0) return 0;
        vector<pair<int, int>> data_hw(height.size(), make_pair(0, 0));
        for (int i = 0; i < height.size(); i++) {
            data_hw[i].first = height[i];
            data_hw[i].second = weight[i];
        }
        sort(data_hw.begin(), data_hw.end(), [](pair<int, int> a, pair<int, int> b)
        {
            if (a.first < b.first) return true;
            if (a.first == b.first) return a.second > b.second;
            return false;
        });
        //寻找最长上升子序列
        vector<int> dp;
        dp.push_back(data_hw[0].second);
        for (int i = 1; i < data_hw.size(); i++) {
            if (data_hw[i].second > dp.back()) {
                dp.push_back(data_hw[i].second);
            }
            else {
                auto pos = lower_bound(dp.begin(), dp.end(), data_hw[i].second);
                *pos = data_hw[i].second;
            }
        }
        return dp.size();
    }
};

```

## 面试题 17.09 第k个数 (中等)

### 1. 题目描述

有些数的素因子只有 3, 5, 7, 请设计一个算法找出第 k 个数。注意，不是必须有这些素因子，而是必须不包含其他的素因子。例如，前几个数按顺序应该是 1, 3, 5, 7, 9, 15, 21。

示例 1:

输入: k = 5

输出: 9

### 2. 正确解法——动态规划

```

class Solution {
public:
    int getKthMagicNumber(int k) {
        if (k <= 0) return 0;
        vector<long int> nums(k+1, 1); // 为防止越界, 用long保存
        int p3 = 0, p5 = 0, p7 = 0; // 标记"某个素数"的下标
        for (int i = 1; i < k; ++i) {

```

```

        nums[i] = min(min(3 * nums[p3], 5 * nums[p5]), 7 * nums[p7]);
        if (nums[i] == 3 * nums[p3]) p3++; // p3++是因为由p3所在的素数求得了最小值,
        故不会再由p3所在的素数求得另一个最小值, 下一个最小值可能是3 * nums[p3+1]。下面p5++, p7++同理。
        if (nums[i] == 5 * nums[p5]) p5++; // 注意此处是if,而不是else if,因为可能3
        *nums[p3] == 5 * nums[p5] 或 7 * nums[p7] == 5 * nums[p5]。下面的同理。
        if (nums[i] == 7 * nums[p7]) p7++;
    }
    return nums[k-1];
}
};

```

## 面试题 17.10 主要元素（简单）

### 1. 题目描述

如果数组中多一半的数都是同一个，则称之为主要元素。给定一个整数数组，找到它的主要元素。若没有，返回-1。

示例 1:  
 输入: [1,2,5,9,5,9,5,5,5]  
 输出: 5

示例 2:  
 输入: [3,2]  
 输出: -1

示例 3:  
 输入: [2,2,1,1,1,2,2]  
 输出: 2

### 2. 简单实现

```

class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int ans = nums[0];
        int cnt = 1;
        for(int i = 1; i < nums.size(); i++){
            if(nums[i] == ans) cnt++;
            else{
                if(cnt == 0){
                    ans = nums[i];
                    cnt = 1;
                }
                else
                    cnt--;
            }
        }
        return ans;
    }
};

```

## 面试题 17.11 单词距离 (中等)

### 1. 题目描述

有个内含单词的超大文本文件，给定任意两个单词，找出在这个文件中这两个单词的最短距离(相隔单词数)。如果寻找过程在这个文件中会重复多次，而每次寻找的单词不同，你能对此优化吗？

示例：

输入：words = ["I","am","a","student","from","a","university","in","a","city"],  
word1 = "a", word2 = "student"  
输出：1

提示：words.length <= 100000

### 2. 简单实现

```
//直接暴力遍历，记录最近出现的索引即可
class Solution {
    const int INF=(1LL<<31)-1;
public:
    int findClosest(vector<string>& words, string word1, string word2) {
        int n=words.size();
        int Ans=INF;
        int a=-1,b=-1;
        for (int i=0;i<n;++i) {
            if (words[i]==word1) {
                a=i;
                if (b!=-1) Ans=min(Ans,a-b);
            }
            else if (words[i]==word2) {
                b=i;
                if (a!=-1) Ans=min(Ans,b-a);
            }
        }
        return Ans;
    }
};

//如果多次查找，可以建立map
class Solution {
public:
    int findClosest(vector<string>& words, string word1, string word2) {
        unordered_map<string, vector<int>> m; //记录所有单词出现的索引，升序
        for(int i = 0; i < words.size(); i++)
            m[words[i]].push_back(i);
        int idx1 = 0, idx2 = 0;
        int ans = INT_MAX;
        while(idx1 < m[word1].size() && idx2 < m[word2].size()){
            ans = min(ans, abs(m[word1][idx1]-m[word2][idx2]));
            if(m[word1][idx1] > m[word2][idx2])
                idx2++;
            else
                idx1++;
        }
        return ans;
    }
};
```

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

## 面试题 17.12 BiNode (简单)

### 1. 题目描述

二叉树数据结构TreeNode可用来表示单向链表（其中left置空，right为下一个链表节点）。实现一个方法，把二叉搜索树转换为单向链表，要求值的顺序保持不变，转换操作应是原址的，也就是在原始的二叉搜索树上直接修改。

返回转换后的单向链表的头节点。

注意：本题相对原题稍作改动

示例：

输入： [4,2,5,1,3,null,6,0]

输出： [0,null,1,null,2,null,3,null,4,null,5,null,6]

提示：节点数量不会超过 100000。

### 2. 简单实现

中序遍历

```
class Solution {  
public:  
    TreeNode* convertBiNode(TreeNode* root) {  
        if(!root) return NULL;  
        TreeNode *head;  
        head = convertBiNode(root->left); //转换左子树  
        if(head){  
            TreeNode* tail = head;  
            while(tail->right) //找到转换后左子树的尾巴  
                tail = tail->right;  
            tail->right = root;  
            root->left = NULL; //不要忘  
        }  
        else  
            head = root;  
        root->right = convertBiNode(root->right);  
        return head;  
    }  
};
```

## 面试题 17.13 恢复空格 (中等)

### 1. 题目描述

哦，不！你不小心把一个长篇文章中的空格、标点都删掉了，并且大写也弄成了小写。像句子"I reset the computer. It still didn't boot!"已经变成了"iresetthecomputeritstill didntboot"。在处理标点符号和大小写之前，你得先把它断成词语。当然了，你有一本厚厚的词典dictionary，不过，有些词没在词典里。假设文章用sentence表示，设计一个算法，把文章断开，要求未识别的字符最少，返回未识别的字符数。

注意：本题相对原题稍作改动，只需返回未识别的字符数

示例：

输入：

dictionary = ["looked","just","like","her","brother"]

sentence = "jesslookedjustliketimherbrother"

输出： 7

解释： 断句后为"jess looked just like tim her brother"，共7个未识别字符。

提示：

- $0 \leq \text{len}(\text{sentence}) \leq 1000$
- dictionary中总字符数不超过 150000。
- 你可以认为dictionary和sentence中只包含小写字母。

## 2. 简单实现

动态规划，dp[i]表示sentence的前i个字符所能断出来的最少未识别字符

```
class Solution {
public:
    int respace(vector<string>& dictionary, string sentence) {
        unordered_set<string> s; // 存储所有单词，方便快速查找
        for(int i = 0; i < dictionary.size(); i++)
            s.insert(dictionary[i]);
        int len = sentence.size();
        vector<int> dp(len+1);
        dp[0] = 0; // 空字符
        for(int r = 1; r <= len; r++) {
            dp[r] = r;
            string cur = sentence.substr(0, r); // 取sentence[0...r]
            for(int l = 0; l < r; l++){
                if(s.find(cur) == s.end())
                    dp[r] = min(dp[r], dp[l] + r - l);
                else
                    dp[r] = min(dp[r], dp[l]);
                cur.erase(0,1); // 去掉首字符，对于C++来说时间消耗比较大
            }
        }
        return dp[len];
    }
};
```

可改进点：记录以每个字符结尾的单词的所有可能的长度，对每个字符只遍历对应的长度

## 面试题 17.14 最小K个数（中等）

### 1. 题目描述

设计一个算法，找出数组中最小的k个数。以任意顺序返回这k个数均可。

示例：

输入： arr = [1,3,5,7,2,4,6,8], k = 4

输出： [1,2,3,4]

提示：

- $0 \leq \text{len}(\text{arr}) \leq 100000$
- $0 \leq k \leq \min(100000, \text{len}(\text{arr}))$

## 2. 简单实现

直接排序即可，也可以用线性时间选择找到第K大的数再筛选，O(N)，懒得写了

```
class Solution {
public:
    vector<int> smallestK(vector<int>& arr, int k) {
        if(k == 0) return {};
        sort(arr.begin(), arr.end());
        return vector<int>(arr.begin(), arr.begin()+k);
    }
};
```

## 面试题 17.15 最长单词（中等）

### 1. 题目描述

给定一组单词words，编写一个程序，找出其中的最长单词，且该单词由这组单词中的其他单词组合而成。若有多个长度相同的结果，返回其中字典序最小的一项，若没有符合要求的单词则返回空字符串。

示例：

输入： ["cat", "banana", "dog", "nana", "walk", "walker", "dogwalker"]

输出： "dogwalker"

解释： "dogwalker"可由"dog"和"walker"组成。

提示：

- $0 \leq \text{len}(\text{words}) \leq 100$
- $1 \leq \text{len}(\text{words}[i]) \leq 100$

### 2. 简单实现

考虑到一个单词可能由好多个单词合成，所以用剪枝回溯

```
class Solution {
public:
    bool flag = false;
    static bool cmp(const string& a, const string& b){//从长到短，字典序
        if(a.size() != b.size())
            return a.size() > b.size();
        else
            return a < b;
    }
};
```

```

//回溯法判断, word[idx...]能否由字典里的单词组成
bool judge(string& word, int idx, unordered_map<char, unordered_set<string>>&
m){
    if(flag) return true;
    if(idx == word.size()){
        flag = true;
        return true;
    }
    for(auto it = m[word[idx]].begin(); it != m[word[idx]].end(); it++){
        int len = (*it).size();
        if(len <= word.size()-idx){
            string cur = word.substr(idx, len);
            if(cur == *it && cur != word){
                if(judge(word, idx+len, m))
                    return flag;//提前终止for循环
            }
        }
    }
    return flag;
}

string longestWord(vector<string>& words) {
    int len = words.size();
    if(len <= 1) return "";
    sort(words.begin(), words.end(), cmp);
    unordered_map<char, unordered_set<string>> m;//存储以各个字符开头的单词有哪些
    for(int i = 0; i < len; i++){
        m[words[i][0]].insert(words[i]);
    }
    for(int i = 0; i < len; i++){
        if(judge(words[i], 0, m))
            return words[i];
    }
    return "";
}
};

```

## 面试题 17.16 按模式（简单）

### 1. 题目描述

一个有名的按摩师会收到源源不断的预约请求，每个预约都可以选择接或不接。在每次预约服务之间要有休息时间，因此她不能接受相邻的预约。给定一个预约请求序列，替按摩师找到最优的预约集合（总预约时间最长），返回总的分钟数。

注意：本题相对原题稍作改动

示例 1:

输入: [1,2,3,1]

输出: 4

解释: 选择 1 号预约和 3 号预约，总时长 = 1 + 3 = 4。

示例 2:

输入: [2,7,9,3,1]

输出: 12

解释: 选择 1 号预约、 3 号预约和 5 号预约, 总时长 = 2 + 9 + 1 = 12。

示例 3:

输入: [2,1,4,5,3,1,1,3]

输出: 12

解释: 选择 1 号预约、 3 号预约、 5 号预约和 8 号预约, 总时长 = 2 + 4 + 3 + 3 = 12。

## 2. 简单实现

一开始想的回溯, 想多了, 动态规划就可以

```
class Solution {
public:
    int message(vector<int>& nums) {
        int len = nums.size();
        if(len == 0) return 0;
        else if(len == 1) return nums[0];
        vector<int> dp(len);
        dp[0] = nums[0];
        dp[1] = max(nums[0], nums[1]);
        for(int i = 2; i < len; i++)
            dp[i] = max(dp[i-1], dp[i-2]+nums[i]);
        return dp[len-1];
    }
};
```

## 面试题 17.17 多次搜索 (中等)

### 1. 题目描述

给定一个较长字符串 `big` 和一个包含较短字符串的数组 `smalls`, 设计一个方法, 根据 `smalls` 中的每一个较短字符串, 对 `big` 进行搜索。输出 `smalls` 中的字符串在 `big` 里出现的所有位置 `positions`, 其中 `positions[i]` 为 `smalls[i]` 出现的所有位置。

**示例:**

```
输入:
big = "mississippi"
smalls = ["is","ppi","hi","sis","i","ssippi"]
输出: [[1,4],[8],[], [3],[1,4,7,10],[5]]
```

**提示:**

- `0 <= len(big) <= 1000`
- `0 <= len(smalls[i]) <= 1000`
- `smalls` 的总字符数不会超过 100000。
- 你可以认为 `smalls` 中没有重复字符串。
- 所有出现的字符均为英文小写字母。

### 2. 简单实现



第一反应想到KMP，但考虑到本题涉及多次搜索，因此可以空间换时间，用字典记录big中各个字符出现的位置，再用各个smalls[i]的首字母搜索其可能出现的位置

```
class Solution {
public:
    vector<vector<int>> multiSearch(string big, vector<string>& smalls) {
        vector<vector<int>> idxs(26);
        int len_b = big.size();
        if(len_b == 0) return vector<vector<int>>(smalls.size());
        for(int i = 0; i < big.size(); i++)
            idxs[big[i]-'a'].push_back(i);
        int len = smalls.size();
        vector<vector<int>> ans(len);
        for(int i = 0; i < len; i++){//依次处理smalls[i]
            int cur_len = smalls[i].size();
            if(cur_len == 0) continue;
            for(auto it = idxs[smalls[i][0]-'a'].begin(); it != idxs[smalls[i][0]-'a'].end(); it++){//smalls[i][0]在big中出现的所有位置
                if(*it+cur_len > len_b) break;//big中剩余的长度比当前字符串短，不可能匹配
                int idx1 = *it + 1;
                int idx2 = 1;
                while(idx2 < cur_len && big[idx1]==smalls[i][idx2]){
                    idx1++;
                    idx2++;
                }
                if(idx2 == cur_len)//匹配上了
                    ans[i].push_back(*it);
            }
        }
        return ans;
    }
};
```

### 3. 最优双百解法——前缀树法

处理字符串搜索问题的常用思路

对smalls建trie树，其中每个树节点的sid记录对应的smalls id。

遍历big的所有后缀，并在trie树中查找后缀。对于查找路径上经过的所有有效sid（sid有效值为大于等于0的数），将后缀的起始id加入到sid对应的ans中。

```
struct TrieNode{
    int sid;
    TrieNode *child[26];
    TrieNode(){
        sid=-1;
        for(int i=0;i<26;++i) child[i]=NULL;
    }
};

class Solution {
private:
    TrieNode *root=new TrieNode();
public:
```

```

void insert(string word,int s){
    int n=word.size();
    TrieNode *cur=root;
    for(int i=0;i<n;++i){
        int cid=word.at(i)-'a';
        if(cur->child[cid]==NULL) cur->child[cid]=new TrieNode();
        cur=cur->child[cid];
    }
    cur->sid=s;
}

void search(string word,vector<vector<int>>& ans,int bid){
    int n=word.size();
    TrieNode *cur=root;
    for(int i=0;i<n;++i){
        int cid=word.at(i)-'a';
        if(cur->sid!=-1) ans[cur->sid].push_back(bid);
        if(cur->child[cid]==NULL) return ;
        cur=cur->child[cid];
    }
    if(cur->sid!=-1) ans[cur->sid].push_back(bid);
}

vector<vector<int>> multiSearch(string big, vector<string>& smalls) {
    int n=smalls.size(),m=big.size();
    vector<vector<int>> ans(n,vector<int>{});
    for(int i=0;i<n;++i){
        if(small[s].size()==0) continue;
        insert(small[i],i);
    }
    for(int i=0;i<m;++i){
        string word=big.substr(i,m-i);
        search(word,ans,i);
    }
    return ans;
}
};

```

## 面试题 17.18 最短超串（中等）

### 1. 题目描述

假设你有两个数组，一个长一个短，短的元素均不相同。找到长数组中包含短数组所有的元素的最短子数组，其出现顺序无关紧要。

返回最短子数组的左端点和右端点，如有多个满足条件的子数组，返回左端点最小的一个。若不存在，返回空数组。

#### 示例 1:

输入：  
big = [7,5,9,0,2,1,3,5,7,9,1,1,5,8,8,9,7]  
small = [1,5,9]  
输出: [7,10]

## 示例 2:

输入:  
big = [1,2,3]  
small = [4]  
输出: []

## 提示:

- big.length <= 100000
- 1 <= small.length <= 100000

## 2. 简单实现

可以从之前的某道题——寻找两个数的最短距离扩展得到本题的解法，只要一直保持记录small中各个数字的最新index，并在更新index时更新ans即可

```
class Solution {
public:
    vector<int> shortestSeq(vector<int>& big, vector<int>& small) {
        if(big.size() < small.size()) return {};
        unordered_set<int> s(small.begin(), small.end()); //集合表示small中的数，快速查
找
        unordered_map<int, int> m; //small中各个数字出现的最新位置
        int l = 0; //左端点，即m中记录的最小的索引值
        while(l < big.size() && s.find(big[l]) == s.end()) l++;
        if(l == big.size()) return {};
        else if(small.size() == 1) return {l, l};
        else m[big[l]] = l;
        vector<int> ans;
        int min_len = INT_MAX;
        for(int i = l+1; i < big.size(); i++){ //从l+1找起
            int cur = big[i];
            if(s.find(cur) != s.end()){ //当前数字在small中，需要更新索引了
                if(m.find(cur) == m.end()) //之前没出现过
                    m[cur] = i;
                else{
                    int old = m[cur]; //旧的索引
                    m[cur] = i;
                    if(old == l){ //需要遍历更新一下l值
                        l = i;
                        for(auto it = m.begin(); it != m.end(); it++){
                            l = min(l, it->second);
                        }
                    }
                }
            }
            if(m.size() == small.size() && i-l+1 < min_len){ //出现距离更短的了，更
新答案
                min_len = i-l+1;
                ans = {l, i};
            }
        }
        return ans;
    }
};
```

```
};
```

## 面试题 17.19 消失的两个数字（困难）

### 1. 题目描述

给定一个数组，包含从 1 到 N 所有的整数，但其中缺了两个数字。你能在  $O(N)$  时间内只用  $O(1)$  的空间找到它们吗？

以任意顺序返回这两个数字均可。

#### 示例 1:

输入: [1]  
输出: [2,3]

#### 示例 2:

输入: [2,3]  
输出: [1,4]

#### 提示:

- nums.length <= 30000

### 2. 简单实现

- 类似于鸽巢原理，把num放在索引num-1上
- 由于n-1和n对应索引出界，可以用两个bool值标识是否存在
- 遍历nums，如果位置不对就放在正确的位置上，如果出界则位置不动
- 则最终数组内若 `nums[i] != i+1`，则i+1不存在

```
class Solution {
public:
    vector<int> missingTwo(vector<int>& nums) {
        int flagn_1 = false, flagn = false; // n-1和n对应的索引出界，用flag标识
        int n = nums.size() + 2;
        for(int i = 0; i < nums.size(); i++){
            if(nums[i] != i+1){ // 位置不对
                if(nums[i] > nums.size()){ // 出界
                    if(nums[i] == n-1)
                        flagn_1 = true;
                    else
                        flagn = true;
                }
                else{
                    swap(nums[i], nums[nums[i]-1]); // 放到应在的位置上
                    i--; // 继续看当前位置
                }
            }
        }
        if(!flagn_1 && !flagn) // n-1和n都不存在
            return {n-1, n};
    }
};
```

```

        vector<int> ans;
        if(!flagn_1) ans.push_back(n-1);
        else if(!flagn) ans.push_back(n);
        for(int i = 0; i < nums.size(); i++)
            if(nums[i] != i+1){
                ans.push_back(i+1);
                if(ans.size() == 2)
                    return ans;
            }
        return ans;
    }
};

```

### 3. 两遍求和法——不修改原数组

1. 从1到N数组nums中，缺失了2个元素。那么  $N = \text{nums.size()} + 2$
2. 缺失2个数字的和  $\text{sumOfTwo} = \text{sum}(1..N) - \text{sum}(\text{nums})$
3. 记  $\text{threshold} = \text{sumOfTwo}/2$ 。因为缺失的2个数字不相等，所以一个小于等于threshold,一个大于threshold。
4. 只对小于等于threshold的元素求和，得到第一个缺失的数字：  
 $\text{sum}(1.. \text{threshold}) - \text{sum}(\text{nums中, 小于等于threshold的元素})$
5. 第二个缺失的数字:  $\text{sumOfTwo} - \text{第一个缺失的数字}$

## 面试题 17.20 连续中值（困难）

### 1. 题目描述

随机产生数字并传递给一个方法。你能否完成这个方法，在每次产生新值时，寻找当前所有值的中间值（中位数）并保存。

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

```

### 2. 简单实现

两个优先队列分别存储有序化的输入数据的左右半部分的数字

```

class MedianFinder {
public:

```

```

priority_queue<int, vector<int>, less<int>> q_1; //大根堆, 储存左半部分
priority_queue<int, vector<int>, greater<int>> q_2; //小根堆, 储存右半部分
/** initialize your data structure here. */
MedianFinder() {
}
void addNum(int num) {
    if(q_1.empty()){
        q_1.push(num);
        return;
    }
    if(num <= q_1.top())
        q_1.push(num);
    else
        q_2.push(num);
    //保持两个堆的大小差距不超过1
    while(int(q_1.size() - q_2.size()) > 1){
        q_2.push(q_1.top());
        q_1.pop();
    }
    while(int(q_2.size() - q_1.size()) > 1){
        q_1.push(q_2.top());
        q_2.pop();
    }
}
double findMedian() {
    int n1 = q_1.size();
    int n2 = q_2.size();
    if(n1 == n2)
        return double(q_1.top() + q_2.top()) / 2;
    else if(n1 > n2)
        return q_1.top();
    else
        return q_2.top();
}
};

```

## 面试题 17.21 直方图的水量 (困难)

### 1. 题目描述

给定一个直方图(也称柱状图), 假设有人从上面源源不断地倒水, 最后直方图能存多少水量?直方图的宽度为1。



上面是由数组 [0,1,0,2,1,0,1,3,2,1,2,1] 表示的直方图，在这种情况下，可以接 6 个单位的水（蓝色部分表示水）。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——见42.接雨水

## 3. 单调栈法

1. 两个栈stx和sty，其中stx存储位置， sty存储高度
2. 维护sty栈递减，某个i入栈时,先要pop出sty栈中小于height[i]的所有元素
3. 出栈时计算出栈的高度和height[i]之间可构成的面积 $(i - \text{stx.top}() - 1) * (\text{sty.top}() - \text{dlt})$ ;其中dlt是前一个出栈的高度。
4. 出栈操作结束后，如果当前栈为空，直接将i和height[i]入栈；
5. 如果当前栈不为空则计算 $(i - \text{stx.top}() - 1) * (\text{height}[i] - \text{dlt})$ 加到result中。

实际上栈内存储的是可能构成蓄水槽两端的柱子

```
class Solution {
public:
    int trap(vector<int>& height) {
        if (!height.size()) return 0;
        stack<int> stx;
        stack<int> sty;
        int result = 0;
        for (int i = 0; i < height.size(); ++i) {
            int dlt = 0;
            while (!sty.empty() && height[i] >= sty.top()) {
                //前面所有比当前柱子低的柱子，从低到高一横条一横条地计算蓄水量
                int x = stx.top(), y = sty.top();
                stx.pop();
                sty.pop();
                result += (i - x - 1) * (y - dlt); //宽*高度差
                dlt = y;
            }
            if (!sty.empty()) result += (i - stx.top() - 1) * (height[i] - dlt);
            stx.push(i);
            sty.push(height[i]);
        }
        return result;
    }
};
```

## 4. 双指针法

缓存两端最大值，从最大值较小的一边进行储水结算、移动，并更新最大值。

执行结果：**通过** [显示详情](#) >

执行用时：**1 ms**，在所有 Java 提交中击败了 **100.00%** 的用户

内存消耗：**38.8 MB**，在所有 Java 提交中击败了 **100.00%** 的用户

## 代码

```
class Solution {
    public int trap(int[] height) {
        if(height.length < 3) return 0;

        int left = 0, right = height.length - 1;
        int leftmax = height[left], rightmax = height[right];
        int res = 0;

        while(left < right){
            if(leftmax < rightmax){
                res += leftmax - height[left++];
                leftmax = Math.max(height[left], leftmax);
            }else{
                res += rightmax - height[right--];
                rightmax = Math.max(height[right], rightmax);
            }
        }

        return res;
    }
}
```

## 面试题 17.22 单词转换（中等）

### 1. 题目描述

给定字典中的两个词，长度相等。写一个方法，把一个词转换成另一个词，但是一次只能改变一个字符。每一步得到的新词都必须能在字典中找到。

编写一个程序，返回一个可能的转换序列。如有多个可能的转换序列，你可以返回任何一个。

**示例 1:**



输入：  
beginword = "hit",  
endword = "cog",  
wordList = ["hot","dot","dog","lot","log","cog"]

输出：  
["hit","hot","dot","lot","log","cog"]

## 示例 2:

输入：  
beginword = "hit"  
endword = "cog"  
wordList = ["hot","dot","dog","lot","log"]

输出： []

解释： endword "cog" 不在字典中，所以不存在符合要求的转换序列。

## 2. 简单实现

根据字典构建图，再dfs，注意visited加入后无需删除，因为本题只要找到一条路径即可，因此某个点若可以到达终点则无需删除，无法到达则说明不存在路径因此无需再路过

```
class Solution {
public:
    vector<string> ans;
    bool done = false;
    bool judge_adj(string& a, string& b){//判断a,b是否只差一个字符
        if(a==b) return false;
        int len = a.size();
        bool flag = false;
        for(int i = 0; i < len; i++){
            if(a[i] != b[i]){
                if(!flag)
                    flag = true;
                else
                    return false;
            }
        }
        return true;
    }
    void dfs(const string& cur, unordered_map<string, vector<string>>& g,
            unordered_set<string>& visited, const string& aim){
        if(done) return;
        ans.push_back(cur);
        if(cur == aim){
            done = true;
            return;
        }
        for(auto it = g[cur].begin(); it != g[cur].end(); it++){
            if(visited.find(*it) == visited.end()){
                visited.insert(*it);
```

```

        dfs(*it, g, visited, aim);
        if(done) return;
        // visited.erase(*it);
    }
}
ans.pop_back();
}
vector<string> findLadders(string beginword, string endword, vector<string>&
wordList) {
    unordered_map<string, vector<string>> g;
    int len = beginword.size();
    for(int i = 0; i < wordList.size(); i++){
        if(wordList[i].size() != len) continue; //长度判断
        for(int j = i+1; j < wordList.size(); j++){
            if(wordList[j].size() != len) continue; //长度判断
            if(judge_adj(wordList[i], wordList[j])){
                g[wordList[i]].push_back(wordList[j]);
                g[wordList[j]].push_back(wordList[i]);
            }
        }
    }
    if(g.find(endword) == g.end()) return {};
    unordered_set<string> visited;
    ans = {beginword};
    for(auto it = g.begin(); it != g.end(); it++){
        string cur = it->first;
        if(judge_adj(beginword, cur)){
            visited.insert(cur);
            dfs(cur, g, visited, endword);
            if(done) return ans;
            // visited.erase(cur);
        }
    }
    return {};
}
};

```

## 面试题 17.23 最大黑方阵（中等）

### 1. 题目描述

给定一个方阵，其中每个单元(像素)非黑即白。设计一个算法，找出 4 条边皆为黑色像素的最大子方阵。

返回一个数组 `[r, c, size]`，其中 `r`, `c` 分别代表子方阵左上角的行号和列号，`size` 是子方阵的边长。若有多个满足条件的子方阵，返回 `r` 最小的，若 `r` 相同，返回 `c` 最小的子方阵。若无满足条件的子方阵，返回空数组。

**示例 1:**

输入:

```
[
    [1,0,1],
    [0,0,1],
    [0,0,1]
]
```

输出: [1,0,2]

解释: 输入中 0 代表黑色, 1 代表白色, 标粗的元素即为满足条件的最大子方阵

## 示例 2:

输入:

```
[
    [0,1,1],
    [1,0,1],
    [1,1,0]
]
```

输出: [0,0,1]

## 提示:

- `matrix.length == matrix[0].length <= 200`

## 2. 简单实现

审题: 只需要四个边是黑色即可,一开始以为要填满黑色-\_-|||

动态规划, `cnt[r][c][0/1]` 表示以坐标r,c为起点向左/右最多的连续黑色块的数量

```
class Solution {
public:
    vector<int> findSquare(vector<vector<int>>& matrix) {
        vector<int> ans(3, 0);
        int n = matrix.size();
        if(n == 0) return {};
        if(n == 1){
            if(matrix[0][0] == 0)
                return {0, 0, 1};
            else
                return {};
        }
        //cnt[r][c][0/1], 0右侧, 1下侧
        vector<vector<vector<int>>> cnt(n, vector<vector<int>>(n, vector<int>(2)));
        for(int r = n-1; r >= 0; r--){
            for(int c = n-1; c >= 0; c--){
                if(matrix[r][c] == 1)
                    cnt[r][c][0] = cnt[r][c][1] = 0;
                else{
                    if(r < n-1) cnt[r][c][1] = cnt[r+1][c][1] + 1;
                    else cnt[r][c][1] = 1;

                    if(c < n-1) cnt[r][c][0] = cnt[r][c+1][0] + 1;
                    else cnt[r][c][0] = 1;
                }
                //更新当前最大子方阵
            }
        }
        return ans;
    }
};
```

```

        int len = min(cnt[r][c][0], cnt[r][c][1]); //最大的可能的边长
        while(len >= ans[2]){ //要答案r,c最小, 所以带等号
            if(cnt[r+len-1][c][0] >= len && cnt[r][c+len-1][1] >= len){
                //可以构成长为len的方阵
                ans = {r, c, len};
                break;
            }
            len--;
        }
    }
}
return ans;
};

```

## 面试题 17.24 最大子矩阵（困难）

### 1. 题目描述

给定一个正整数和负整数组成的  $N \times M$  矩阵，编写代码找出元素总和最大的子矩阵。

返回一个数组 `[r1, c1, r2, c2]`，其中 `r1`, `c1` 分别代表子矩阵左上角的行号和列号，`r2`, `c2` 分别代表右下角的行号和列号。若有多个满足条件的子矩阵，返回任意一个均可。

**注意：**本题相对书上原题稍作改动

**示例：**

输入：  
 [
 [-1,0],
 [0,-1]
 ]  
 输出：[0,1,0,1]

**说明：**

- 1 <= matrix.length, matrix[0].length <= 200

### 2. 简单实现

暴力遍历，与之前做过的最大连续子数组和的题类似，可以在矩形的右侧"一条一条"地贴上去

```

class Solution {
public:
    vector<int> getMaxMatrix(vector<vector<int>>& matrix) {
        int m = matrix.size();
        if(m == 0) return {};
        int n = matrix[0].size();
        if(n == 0) return {};
        if(m == 1 && n == 1) return {0, 0, 0, 0};
        vector<vector<int>> sum(m, vector<int>(n)); //sum[i][j]表示matrix[0...i][j]的
        //和, 用于快读
        for(int c = 0; c < n; c++) //第一行初始化

```

```

        sum[0][c] = matrix[0][c];
    for(int r = 1; r < m; r++){
        for(int c = 0; c < n; c++){
            sum[r][c] = sum[r-1][c] + matrix[r][c];
        }
    }
    int max_num = INT_MIN;
    vector<int> ans;
    for(int r1 = 0; r1 < m; r1++){
        for(int c1 = 0; c1 < n; c1++){//遍历每一个位置，作为左上角顶点
            for(int r2 = r1; r2 < m; r2++){//矩形的下边
                int cur = 0;//累计当前最大矩阵和
                for(int c2 = c1; c2 < n; c2++){//矩形的右边
                    cur += sum[r2][c2];
                    if(r1 > 0) cur -= sum[r1-1][c2];
                    if(cur > max_num){
                        max_num = cur;
                        ans = {r1,c1,r2,c2};
                    }
                    if(cur < 0) break;//<0则后续列不用再贴了，可能出现最大了
                }
            }
        }
    }
    return ans;
};

```

## 面试题 17.25 单词矩阵（困难）

### 1. 题目描述

给定一份单词的清单，设计一个算法，创建由字母组成的面积最大的矩形，其中每一行组成一个单词(自左向右)，每一列也组成一个单词(自上而下)。不要求这些单词在清单里连续出现，但要求所有行等长，所有列等高。

如果有多个面积最大的矩形，输出任意一个均可。一个单词可以重复使用。

#### 示例 1:

```

输入: ["this", "real", "hard", "trh", "hea", "iar", "sld"]
输出:
[
  "this",
  "real",
  "hard"
]

```

#### 示例 2:

```

输入: ["aa"]
输出: ["aa","aa"]

```

#### 说明:

- `words.length <= 1000`
- `words[i].length <= 100`
- 数据保证单词足够随机

## 2. 正确解法

1. 以words构建字典树
2. 按递减顺序选定单词宽度，逐行添加单词，同列字母需要位于字典树自顶点向下的路径中（剪枝1）
3. 当 选定单词宽度 \* 最长单词（即最长行数）< area，结束循环（剪枝2）

```
class Trie://前缀树
    def __init__(self):
        self.root = [{}, False]

    def addword(self, word):
        cur = self.root
        for c in word:
            if c not in cur[0]:
                cur[0][c] = [{}, False]
            cur = cur[0][c]
        cur[1] = True

class Solution:
    def maxRectangle(self, words: List[str]) -> List[str]:
        area = 0
        res = []
        trie = Trie()
        for word in words:
            trie.addword(word)

        def dfs(arr, li):
            for word in words:
                if len(word) != len(arr): continue
                for i, c in enumerate(word):
                    if c not in arr[i][0]: break
                else:
                    temp = arr[:]
                    flag = True
                    for i, c in enumerate(word):
                        temp[i] = temp[i][0][c]
                        flag &= temp[i][1]
                    li.append(word)
                    if flag:
                        h, w = len(li), len(word)
                        nonlocal area, res
                        if h * w > area:
                            area = h * w
                            res = li[:]
                    dfs(temp, li)
                    li.pop()

        ll = sorted(set(len(word) for word in words), reverse = True)
        for l in ll:
```

```
        if l * ll[0] < area: break
        dfs([trie.root] * l, [])
    return res
```

## 面试题 17.26 稀疏相似度 (困难)

### 1. 题目描述

两个(具有不同单词的)文档的交集(intersection)中元素的个数除以并集(union)中元素的个数, 就是这两个文档的相似度。例如,  $\{1, 5, 3\}$  和  $\{1, 7, 2, 3\}$  的相似度是 0.4, 其中, 交集的元素有 2 个, 并集的元素有 5 个。给定一系列的长篇文档, 每个文档元素各不相同, 并与一个 ID 相关联。它们的相似度非常“稀疏”, 也就是说任选 2 个文档, 相似度都很接近 0。请设计一个算法返回每对文档的 ID 及其相似度。只需输出相似度大于 0 的组合。请忽略空文档。为简单起见, 可以假定每个文档由一个含有不同整数的数组表示。

输入为一个二维数组 `docs`, `docs[i]` 表示 id 为 `i` 的文档。返回一个数组, 其中每个元素是一个字符串, 代表每对相似度大于 0 的文档, 其格式为 `{id1},{id2}: {similarity}`, 其中 `id1` 为两个文档中较小的 id, `similarity` 为相似度, 精确到小数点后 4 位。以任意顺序返回数组均可。

**示例:**

```
输入:
[
  [14, 15, 100, 9, 3],
  [32, 1, 9, 3, 5],
  [15, 29, 2, 6, 8, 7],
  [7, 10]
]
输出:
[
  "0,1: 0.2500",
  "0,2: 0.1000",
  "2,3: 0.1429"
]
```

**提示:**

- `docs.length <= 500`
- `docs[i].length <= 500`
- 相似度大于 0 的文档对数不会超过 1000

### 2. 简单实现

```
class Solution {
public:
    vector<string> computesimilarities(vector<vector<int>>& docs) {
        unordered_map<int, vector<int>> m; //记录每个数字出现在的所有文档id
        vector<string> ans;
        int size = docs.size();
        ostringstream ss; //用来保证4位小数的
        ss << std::setiosflags(std::ios::fixed | std::ios::showpoint |
std::ios::right)
            << std::setprecision(4);
        for(int i = 0; i < size; i++){
```

```

        unordered_map<int, int> cnt; //记录每个文档里与docs[i]的相同数字的数量
        for(int j = 0; j < docs[i].size(); j++){
            m[docs[i][j]].push_back(i);
            //寻找docs[0...i-1]之间的文档有哪些包含docs[i][j], 并记入cnt
            for(auto it = m[docs[i][j]].begin(); it!=m[docs[i][j]].end() &&
*it<i; it++)
                cnt[*it]++;
        }
        for(auto it = cnt.begin(); it != cnt.end(); it++){
            string temp = to_string(it->first) + "," + to_string(i) + ": ";
            ss << it->second / float(docs[it->first].size() + docs[i].size() -
it->second)+ 1e-9; //+1e-9是因为C++浮点数的精度问题Hhhhhhh
            temp += ss.str();
            ss.str(""); //清空缓冲区
            ans.push_back(temp);
        }
    }
    return ans;
}
};

```