数组和字符串

15.三数之和（中等）

1. 题目描述

   给定一个包含 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 $a + b + c = 0$？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。

```
例如，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) {
        if(nums.size() < 3) return vector<vector<int>>();
        sort(nums.begin(), nums.end());//排序
        vector<vector<int>> result;
        for(int i = 0; i < nums.size()-2; i++){
            int l = i + 1;
            int r = nums.size()-1;
            int aim = -nums[i];
            while(l < r){
                int num = nums[l]+nums[r];
                if(num == aim){
                    vector<int> temp;
                    temp.push_back(-aim);
                    temp.push_back(nums[l++]);
                    temp.push_back(nums[r--]);
                    result.push_back(temp);
                    while(l < r && nums[l] == nums[l-1])//去重
                        l++;
                    while(r > l && nums[r] == nums[r+1])//去重
                        r--;
                }
                else if(num < aim){
                    l++;
                }
                else{
```

```
                    r--;
                }
            }
            while(i < nums.size()-1 && nums[i] == nums[i+1])//去重
                i++;
        }
        return result;
    }
};
```

73.矩阵置零（中等）

1. 题目描述

   给定一个 *m* x *n* 的矩阵，如果一个元素为 0，则将其所在行和列的所有元素都设为 0。请使用**原地**算法。

   示例 1:

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

   示例 2:

```
输入:
[
  [0,1,2,0],
  [3,4,5,2],
  [1,3,1,5]
]
输出:
[
  [0,0,0,0],
  [0,4,5,0],
  [0,3,1,0]
]
```

   进阶:

   - 一个直接的解决方案是使用 O(*m***n*) 的额外空间，但这并不是一个好的解决方案。
   - 一个简单的改进方案是使用 O(*m* + *n*) 的额外空间，但这仍然不是最好的解决方案。
   - 你能想出一个常数空间的解决方案吗?

2. 简单实现

O(m+n)算法，即用集合保存每一个0所在行号列号，再统一置零

```
class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        int m = matrix.size();
        if(m == 0) return;
        int n = matrix[0].size();
        if(n == 0) return;
        unordered_set<int> rows;
        unordered_set<int> cols;
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                if(matrix[i][j] == 0){
                    rows.insert(i);
                    cols.insert(j);
                }
        for(auto it = rows.begin(); it != rows.end(); it++)
            for(int j = 0; j < n; j++)
                matrix[*it][j] = 0;
        for(auto it = cols.begin(); it != cols.end(); it++)
            for(int i = 0; i < m; i++)
                matrix[i][*it] = 0;
    }
};
```

3. O(1)空间的暴力

遍历原始矩阵，如果发现如果某个元素 $cell_{ij}$ 为 0，我们将第 i 行和第 j 列的所有非零元素设成很大的负虚拟值（比如说 -1000000）。注意，正确的虚拟值取值依赖于问题的约束，任何允许值范围外的数字都可以作为虚拟之。 最后，我们便利整个矩阵将所有等于虚拟值（常量在代码中初始化为 MODIFIED）的元素设为 0。

49.字母异位词分组（中等）

1. 题目描述

给定一个字符串数组，将字母异位词组合在一起。字母异位词指字母相同，但排列不同的字符串。

示例:

```
输入: ["eat", "tea", "tan", "ate", "nat", "bat"],
输出:
[
  ["ate","eat","tea"],
  ["nat","tan"],
  ["bat"]
]
```

说明:

- 所有输入均为小写字母。
- 不考虑答案输出的顺序。

2. 简单实现

```
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        unordered_map<string, vector<string>> m;
        for(int i = 0; i < strs.size(); i++){
            string temp = strs[i];
            sort(temp.begin(), temp.end());
            if(m.count(temp) <= 0)
                m[temp] = {strs[i]};
            else
                m[temp].push_back(strs[i]);
        }
        vector<vector<string>> ans;
        for(unordered_map<string, vector<string>>::iterator it = m.begin(); it !=
m.end(); it++)
            ans.push_back(it->second);
        return ans;
    }
};
```

3.无重复字符的最子串（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) {
        vector<int> m = vector<int>(256, -1);//记录每个字符最近一次出现的位置
        int ans = 0;
        int l = -1;
        int r = 0;
        while(r < s.size()){
            l = max(l, m[s[r]]);//出现重复字符时滑窗左边界右移
            m[s[r]] = r;
            ans = max(ans, r-l);
            r++;
        }
        return ans;
    }
};
```

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

   暴力解法会超时，采用递归思想：假设除首字符外，其余字符串最长回文串已经找到，则加入首字母后整个字符串的最长回文串有三种可能：

   - 最长回文串出现在开头，加入首字母后长度加2，如 `"d" + "cbaabcdofp"`
   - 最长回文串出现在开头，加入首字母后长度加1，如 `"a" + "aaaaawhsd"`
   - 其他情况下，最长回文串长度不可能超过之前找到的，所以直接返回

```
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);//看前case0+2个字符是否构成回文串
        if(judge(case1)) return case1;
        string case2 = s.substr(0, case0.size()+1);//看前case0+1个字符是否构成回文串
        if(judge(case2)) return case2;
        return case0;
    }
};
```

3. 中心扩展法

   以某一个或两个字符为中心向外扩展回文串

```
    string longestPalindrome(string s)
    {
        if (s.length() < 1) return "";
        int start = 0, end = 0;
        for (int i = 0; i < s.length(); i++) {
            int len1 = expandAroundCenter(s, i, i);//一个元素为中心
            int len2 = expandAroundCenter(s, i, i + 1);//两个元素为中心
            int len = max(len1, len2);
            if (len > end - start) {
                start = i - (len - 1) / 2;
                end = i + len / 2;
            }
        }
        return s.substr(start, end - start + 1);
    }
    int expandAroundCenter(string s, int left, int right) {
        int L = left, R = right;
        while (L >= 0 && R < s.length() && s[L] == s[R]) {// 计算以left和right为中心
的回文串长度
            L--;
            R++;
        }
        return R - L - 1;
    }
```

4. ## Manacher（马拉车） 算法（最优）

   前面解法存在以下缺陷：

   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。 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：

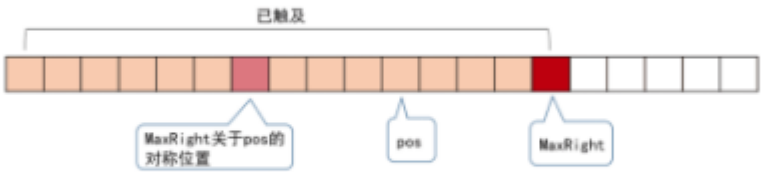
字符: # a # b # a # RL : 1 2 1 4 1 2 1 RL-1: 0 1 0 3 0 1 0 位置: 0 1 2 3 4 5 6

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。

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。**

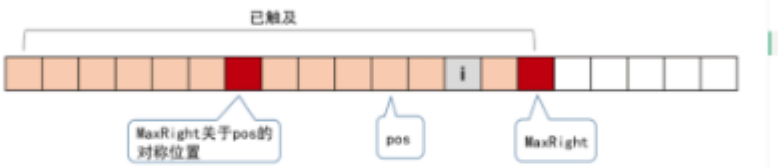
我们再引入一个辅助变量 `MaxRight`，表示当前访问到的所有 回文子串，所能触及的最右一个字符的位置。另外还要记录下 `MaxRight` 对应的回文串的对称轴所在的位置，记为 `pos`，它们的位置关系如下。



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` 。

1) 2) 分析结合的代码为：

```
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;
}
```

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`

代码如下：

```
string longestPalindrome(string s) {
    int len = s.length();
    if (len < 1)    {
        return "";
    }
    // 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);
}
```

334.递增的三元子序列（中等）

1. 题目描述

给定一个未排序的数组，判断这个数组中是否存在长度为 3 的递增子序列。

数学表达式如下:

> 如果存在这样的 *i, j, k,* 且满足 0 ≤ *i* < *j* < *k* ≤ *n*-1 ， 使得 *arr[i]* < *arr[j]* < *arr[k]* ， 返回 true ; 否则返回 false 。

说明: 要求算法的时间复杂度为 O(*n*)，空间复杂度为 O(*1*) 。

示例 1:

```
输入: [1,2,3,4,5]
输出: true
```

示例 2:

```
输入：[5,4,3,2,1]
输出：false
```

2. 最优实现（神奇！！！）

   先说下这道题的思路： 首先找到一个相对小的值，然后找到比这个小一点的值大的值(中间值)，然后看能够在最后找到比中间值大的值。

   下面来说下为什么这种思路能保证覆盖所有的情况：

   - 首先，如果只有一个最小值，然后找不到中间值，那么这个数组必然不包含递增的三个数（因为连递增的两个数都找不到）。
   - 然后假设我们找到了两个递增的值，那么
     - 如果下一个值小于最小值，我们就应该将最小值的指针定位到这个值上。我们尽可能的使用最小值，防止后面出现了更小的一对递增值，而即使不出现，也不妨碍我们找到解**（因为最终是看能否找到大于中间值的值）**
     - 如果下一个值大于最小值，且小于中间值，则我们使用该值作为中间值(**因为这之前的中间值没找到解，而这之后如果最小的中间值都得不到解，那么一定就是false，**这样也保证了覆盖所有的情况)。
     - 最后，如果找到了大于中间值的值，则为true.

   思维误区：要找的只是有没有递增子序列，所以不一定非得找到这个递增子序列是什么

```cpp
class Solution {
public:
    const long INF = 1e10;
    bool increasingTriplet(vector<int>& nums) {
        long n1 = INF;
        long n2 = INF;
        long n3 = INF;
        for (auto x : nums) {
            if (x < n1)
                n1 = x;
            else if (x != n1 && x < n2)
                n2 = x;
            else if (x != n1 && x != n2 && x < n3)
                n3 = x;
            if (n1 < n2 && n2 < n3 && n3 < INF) {
                return true;
            }
        }
        return false;
    }
}
```

树和图

103.二叉树的锯齿形层次遍历（中等）

1. 题目描述

给定一个二叉树，返回其节点值的锯齿形层次遍历。（即先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行）。

例如：给定二叉树 `[3,9,20,null,null,15,7]`，

```
    3
   / \
  9  20
    /  \
   15   7
```

返回锯齿形层次遍历如下：

```
[
  [3],
  [20,9],
  [15,7]
]
```

2. 简单实现

- 观察发现，上一层种先访问的节点，其孩子节点在下一层中会后访问，满足这种性质就要用栈
- 因此使用栈代替队列进行层序遍历，每一次循环后栈内存储一层的数据
- 但由于栈的push操作会影响pop操作的结果，因此需要先进先出的队列进行辅助

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        if(!root) return ans;
        bool l2r = true;//从左到右push子节点
        stack<TreeNode*> s;
        s.push(root);
        while(!s.empty()){
            queue<TreeNode*> q;
            while(!s.empty()){//将本层将访问的栈内元素全部移入队列
                q.push(s.top());
                s.pop();
            }
            vector<int> temp;
            int size = q.size();
            for(int i = 0; i < size; i++){
                TreeNode* cur = q.front();
                q.pop();
                temp.push_back(cur->val);
                if(l2r){//先左后右
                    if(cur->left) s.push(cur->left);
                    if(cur->right) s.push(cur->right);
                }
                else{//先右后左
                    if(cur->right) s.push(cur->right);
```

```
                    if(cur->left) s.push(cur->left);
                }
            }
            ans.push_back(temp);
            l2r = !l2r;//反转方向
        }
        return ans;
    }
};
```

3. 自我改进

   其实无需那么复杂，只要记录当前行数奇偶，在层序遍历过程中对偶数行的结果反转即可

```
class Solution {
public:
    vector<vector<int>> zigzagLevelOrder(TreeNode* root) {
        vector<vector<int>> ans;
        if(!root) return ans;
        bool l2r = true;
        queue<TreeNode*> q;
        q.push(root);
        while(!q.empty()){
            vector<int> temp;
            int size = q.size();
            for(int i = 0; i < size; i++){
                TreeNode* cur = q.front();
                q.pop();
                temp.push_back(cur->val);
                if(cur->left) q.push(cur->left);
                if(cur->right) q.push(cur->right);
            }
            if(!l2r) reverse(temp.begin(), temp.end());//反转偶数行结果
            ans.push_back(temp);
            l2r = !l2r;
        }
        return ans;
    }
};
```

230.二叉搜索树中第K小的元素（中等）

1. 题目描述

   给定一个二叉搜索树，编写一个函数 `kthSmallest` 来查找其中第 **k** 个最小的元素。

   说明： 你可以假设 k 总是有效的，1 ≤ k ≤ 二叉搜索树元素个数。

   示例 1:

```
输入: root = [3,1,4,null,2], k = 1
   3
  / \
 1   4
  \
   2
输出: 1
```

示例 2:

```
输入: root = [5,3,6,2,4,null,null,1], k = 3
       5
      / \
     3   6
    / \
   2   4
  /
 1
输出: 3
```

进阶：如果二叉搜索树经常被修改（插入/删除操作）并且你需要频繁地查找第 k 小的值，你将如何优化 `kthSmallest` 函数?

2. 简单实现

在中序遍历的过程中找到第k个

```
class Solution {
public:
    int track(TreeNode* root, int &k){
        if(root->left){
            int val = track(root->left, k);
            if(k == 0) return val;
        }
        k--;
        if(k == 0) return root->val;
        if(root->right) return track(root->right, k);
        return -1;
    }
    int kthSmallest(TreeNode* root, int k) {
        return track(root, k);
    }
};
```

回溯算法

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;
    }
};
```

22.生成括号（中等）

1. 题目描述

   给出 *n* 代表生成括号的对数，请你写出一个函数，使其能够生成所有可能的并且**有效的**括号组合。

   例如，给出 *n* = 3，生成结果为：

```
[
  "((()))",
  "(()())",
  "(())()",
  "()(())",
  "()()()"
]
```

2. 简单实现

   一个认知：只要当前左括号的数量大于等于右括号的数量，那么就是有效的

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

```
        dfs(n, n, "");
        return ans;
    }
};
```

46.全排列（中等）

1. 题目描述

   给定一个**没有重复**数字的序列，返回其所有可能的全排列。

   示例:

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

2. 简单实现

```
class Solution {
public:
    vector<vector<int>> ans;
    vector<int> temp;

    void dfs(vector<int>& nums) {
        int len = nums.size();
        if(len <= 0) ans.push_back(temp);
        else {
            for(int i = 0; i < len; i++) {
                int buf = nums[i];
                temp.push_back(buf);
                nums.erase(nums.begin()+i);

                dfs(nums);

                temp.pop_back();
                nums.insert(nums.begin()+i,buf);
            }
        }
    }
    vector<vector<int>> permute(vector<int>& nums) {
        if(nums.size()<=0) return ans;
        else dfs(nums);
        return ans;
    }
```

```
};
```

78.子集（中等）

1. 题目描述

   给定一组**不含重复元素**的整数数组 *nums*，返回该数组所有可能的子集（幂集）。

   说明：解集不能包含重复的子集。

   示例:

```
输入: nums = [1,2,3]
输出:
[
  [3],
  [1],
  [2],
  [1,2,3],
  [1,3],
  [2,3],
  [1,2],
  []
]
```

2. 简单实现

```
class Solution {
public:
    vector<vector<int>> ans;
    vector<int> temp;
    void dfs(vector<int>& nums, int i) {
        if(i == nums.size())
            ans.push_back(temp);
        else {
            temp.push_back(nums[i]);
            dfs(nums,i+1);//放进去
            temp.pop_back();

            dfs(nums,i+1);//不放
        }
    }
    vector<vector<int>> subsets(vector<int>& nums) {
        if(nums.size() <= 0)
            return ans;
        else
            dfs(nums,0);
        return ans;
    }
};
```

79.单词搜索（中等）

1. 题目描述

给定一个二维网格和一个单词，找出该单词是否存在于网格中。

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

示例:

```
board =
[
  ['A','B','C','E'],
  ['S','F','C','S'],
  ['A','D','E','E']
]

给定 word = "ABCCED", 返回 true.
给定 word = "SEE", 返回 true.
给定 word = "ABCB", 返回 false.
```

2. 简单实现

深度优先搜索，不能用广度，没法应付visited重复的情况！

```
class Solution {
public:
    vector<vector<bool>> visited;
    int m,n;
    bool dfs(vector<vector<char>>& board, int x, int y, string& word, int idx){
        if(idx == word.size()) return true;
        if(x-1 >= 0 && !visited[x-1][y] && board[x-1][y] == word[idx]){
            visited[x-1][y] = true;
            if(dfs(board, x-1, y, word, idx+1)) return true;
            visited[x-1][y] = false;
        }
        if(x+1 < m && !visited[x+1][y] && board[x+1][y] == word[idx]){
            visited[x+1][y] = true;
            if(dfs(board, x+1, y, word, idx+1)) return true;
            visited[x+1][y] = false;
        }
        if(y-1 >= 0 && !visited[x][y-1] && board[x][y-1] == word[idx]){
             visited[x][y-1] = true;
             if(dfs(board, x, y-1, word, idx+1)) return true;
             visited[x][y-1] = false;
        }
        if(y+1 < n && !visited[x][y+1] && board[x][y+1] == word[idx]){
            visited[x][y+1] = true;
            if(dfs(board, x, y+1, word, idx+1)) return true;
            visited[x][y+1] = false;
        }
        return false;
    }
    bool exist(vector<vector<char>>& board, string word) {
```

```
        m = board.size();
        if(m == 0) return false;
        n = board[0].size();
        if(n == 0) return false;
        if(word.size() <= 0 || word.size() > m*n) return false;
        visited = vector<vector<bool>>(m, vector<bool>(n, false));
        for(int i = 0; i < m; i++)
            for(int j = 0; j < n; j++)
                if(board[i][j] == word[0]){//找到一个可能的开头，以此为起点BFS
                    visited[i][j] = true;
                    if(dfs(board, i, j, word, 1))
                        return true;
                    visited[i][j] = false;
                }
        return false;
    }
};
```

排序和搜索

75.颜色分类（中等）

1. 题目描述

   给定一个包含红色、白色和蓝色，一共 *n* 个元素的数组，**原地**对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

   此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

   注意: 不能使用代码库中的排序函数来解决这道题。

   示例:

   ```
   输入: [2,0,2,1,1,0]
   输出: [0,0,1,1,2,2]
   ```

   进阶:

   - 一个直观的解决方案是使用计数排序的两趟扫描算法。 首先，迭代计算出0、1 和 2 元素的个数，然后按照0、1、2的排序，重写当前数组。
   - 你能想出一个仅使用常数空间的一趟扫描算法吗?

2. 简单实现

   计数排序，两趟扫描

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        int len = nums.size();
        if(len <= 0) return;
        int count[3] = {0};
        for(int i = 0; i < len; i++)
```

```
            count[nums[i]]++;
        int j = 0;
        while(count[0]) {
            nums[j++] = 0;
            count[0]--;
        }
        while(count[1]) {
            nums[j++] = 1;
            count[1]--;
        }
        while(count[2]) {
            nums[j++] = 2;
            count[2]--;
        }
    }
};
```

3. 进阶解法

从两头向中间计算并填好0和2的位置，剩下的部分补1即可

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        int len = nums.size();
        if(len <= 0) return;
        int l = 0, l_cur = 0;//0填充到的位置，从左起遍历到的位置
        int r = len-1, r_cur = len-1;//2填充到的位置，从右起遍历到的位置
        while(l_cur <= r_cur){
            while(l_cur <= r_cur && nums[l_cur] == 0){//是0的时候左边填充
                nums[l] = 0;
                l++;
                l_cur++;
            }
            while(r_cur >= l_cur && nums[r_cur] == 2){//是2的时候右边填充
                nums[r] = 2;
                r--;
                r_cur--;
            }
            if(l_cur < r_cur)
                swap(nums[l_cur], nums[r_cur]);//交换
            while(l_cur <= r_cur && nums[l_cur] == 1) l_cur++;//左边跳过为1的那些，继续找0
            while(l_cur <= r_cur && nums[r_cur] == 1) r_cur--;//左边跳过为1的那些，继续找2
        }
        while(l <= r) nums[l++] = 1;//两边所有的0和2都已经填充好，中间全都是1
    }
};
```

4. 最优解法

思想差不多，更简洁，真正的一遍遍历（我上面的是一遍半），这貌似就是某个版本的快排代码？？。。

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        if(nums.size()<=1)
            return;
        int left = 0;
        int right = nums.size()-1;
        int i = 0;
        while(i<=right){
            if(nums[i]==0){
                swap(nums[i], nums[left]);
                ++i;//换过来的肯定是1，i直接移到下一位
                ++left;
            }
            else if(nums[i]==2){
                swap(nums[i], nums[right]);
                --right;
            }
            else{
                ++i;
            }
        }
    }
};
```

347.前 K 个高频元素（中等）

1. 题目描述

   给定一个非空的整数数组，返回其中出现频率前 **k** 高的元素。

   示例 1:

   ```
   输入: nums = [1,1,1,2,2,3], k = 2
   输出: [1,2]
   ```

   示例 2:

   ```
   输入: nums = [1], k = 1
   输出: [1]
   ```

   说明:

   - 你可以假设给定的 *k* 总是合理的，且 1 ≤ k ≤ 数组中不相同的元素的个数。
   - 你的算法的时间复杂度**必须**优于 O(*n* log *n*) , *n* 是数组的大小。

2. 简单实现

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        vector<int> ans(k);
```

```
        //计数
        unordered_map<int, int> cnt;//记录每个数字出现的频次
        for(int i = 0; i < nums.size(); i++)
            if(cnt.count(nums[i]) == 0)
                cnt[nums[i]] = 1;
            else
                cnt[nums[i]]++;
        //排序
        map<int, vector<int>, greater<int>> inv;//记录每个频次对应的数字，注意是有序映射
        for(auto it = cnt.begin(); it != cnt.end(); it++)
            if(inv.count(it->second) == 0)
                inv[it->second] = {it->first};
            else
                inv[it->second].push_back(it->first);
        //输出
        auto it = inv.begin();
        int idx = 0;
        while(idx < k){
            vector<int> cur = it->second;
            for(int i = 0; i < cur.size() && idx < k; i++){
                ans[idx] = cur[i];
                idx++;
            }
            it++;
        }
        return ans;
    }
};
```

3. 最优解法

排序时用大小为k的最小堆，这样能更快，O(nlogk)

```
class Solution {
public:
    class cmp {
        public:
            //通过重载操作符来对优先队列定义排序规则
            bool operator()(pair<int, int>& a, pair<int, int>& b)       {
                return a.second > b.second;
            }
    };
    vector<int> topKFrequent(vector<int>& nums, int k) {
        //map<int, int> myMap;
        //不使用map的原因是map底层是红黑树，每次插入会进行排序，增加了时间复杂度
        unordered_map<int, int> myMap;
        //统计该值出现的次数
        for (auto num : nums)
            myMap[num]++;
        //遍历map，用最小堆保存频率最大的k个元素
        //优先队列，把最小的元素放在队首
        priority_queue<pair<int, int>, vector<pair<int, int>>, cmp> pq;
```

```
        for(unordered_map<int, int>::iterator beg = myMap.begin(); beg !=
myMap.end(); beg++){
            pq.push(make_pair(beg->first, beg->second));//压入该键值对
            if (pq.size() > k)//堆中元素多于k个
                pq.pop();
        }
        //取出最小堆中的元素
        vector<int> res;
        while (!pq.empty()) {
            res.push_back(pq.top().first);
            pq.pop();
        }
        //结果要逆序输出
        vector<int> tempRes(res.rbegin(), res.rend());
        return tempRes;
    }
};
```

215.数组中的第K个最大元素（中等）

1. 题目描述

   在未排序的数组中找到第 **k** 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

   示例 1:

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

   示例 2:

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

   说明:

   你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

2. 简单实现

   线性时间选择

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

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

162.寻找峰值（中等）

1. 题目描述

   峰值元素是指其值大于左右相邻值的元素。

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

   数组可能包含多个峰值，在这种情况下，返回任何一个峰值所在位置即可。

   你可以假设 `nums[-1] = nums[n] = -∞`。

   示例 1:

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

   示例 2:

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

   说明:

   你的解法应该是 *O*(*logN*) 时间复杂度的。

2. 简单实现

   利用 `nums[i] ≠ nums[i+1]` 和 `nums[-1] = nums[n] = -∞`，可以使用二分法，在上坡时右边一定有峰值，在下坡时左边一定有峰值

```
class Solution {
```

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

        }
        return l;
    }
};
```

34.在排序数组中查找元素的第一个和最后一个位置（中等）

1. 题目描述

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

   你的算法时间复杂度必须是 *O*(log *n*) 级别。

   如果数组中不存在目标值，返回 `[-1, -1]`。

   示例 1:

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

   示例 2:

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

2. 简单实现

```
class Solution {
public:
    int findleftbound(vector<int>& nums, int target){
        int l = 0;
        int r = nums.size() - 1;
        while(l < r){
            int mid = l + (r - l) / 2;
```

```
            if(nums[mid] == target){
                if(mid == 0) return mid;
                if(nums[mid-1] == target)
                    r = mid;
                else
                    return mid;
            }
            else if(nums[mid] < target)
                l = mid + 1;
            else
                r = mid;
        }
        if(nums[l] == target) return l;
        else return -1;
    }
    int findrightbound(vector<int>& nums, int target){
        int l = 0;
        int r = nums.size() - 1;
        while(l < r){
            int mid = l + (r - l) / 2;
            if(nums[mid] == target){
                if(mid == nums.size() - 1) return mid;
                if(nums[mid+1] != target)
                    return mid;
                else
                    l = mid + 1;
            }
            else if(nums[mid] < target)
                l = mid + 1;
            else
                r = mid;
        }
        if(nums[l] == target) return l;
        else return -1;
    }
    vector<int> searchRange(vector<int>& nums, int target) {
        if(nums.size() == 0) return vector<int>(2, -1);
        return {findleftbound(nums, target), findrightbound(nums, target)};
    }
};
```

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

先按每个interval的起始位置从小到大排序，然后依次合并

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        vector<vector<int>> ans;
        if(intervals.size() == 0) return ans;
        map<int, int> m;//利用map存储起始和终止坐标，并实现了排序
        for(int i = 0; i < intervals.size(); i++)
            if(m.count(intervals[i][0]) <= 0)
                m[intervals[i][0]] = intervals[i][1];
            else if(m[intervals[i][0]] < intervals[i][1])//有相同起始坐标的区间只保留最
大的
                m[intervals[i][0]] = intervals[i][1];
        map<int, int>::iterator it = m.begin();
        vector<int> cur = {it->first, it->second};
        for(it++; it != m.end(); it++){
            if(it->first <= cur[1]){//有重叠可合并
                if(it->second > cur[1])//向右扩展
                    cur[1] = it->second;
            }
            else{//无重叠，合并完现在的区间了，开始合并新区间
                ans.push_back(cur);
                cur = {it->first, it->second};
            }
        }
        ans.push_back(cur);
        return ans;
    }
};
```

240.搜索二维矩阵 II（中等）

1. 题目描述

编写一个高效的算法来搜索 *m* x *n* 矩阵 matrix 中的一个目标值 target。该矩阵具有以下特性：

- 每行的元素从左到右升序排列。
- 每列的元素从上到下升序排列。

示例:

现有矩阵 matrix 如下：

```
[
  [1,   4,  7, 11, 15],
  [2,   5,  8, 12, 19],
  [3,   6,  9, 16, 22],
  [10, 13, 14, 17, 24],
  [18, 21, 23, 26, 30]
]
```

给定 target = `5`，返回 `true`。

给定 target = `20`，返回 `false`。

2. 简单实现

剑指offer的题，从右上角开始找

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int m = matrix.size();
        if(m == 0) return false;
        int n = matrix[0].size();
        if(n == 0) return false;
        int x = 0, y = n - 1;
        while(x < m && y >= 0){
            if(matrix[x][y] == target)
                return true;
            else if(matrix[x][y] > target)
                y--;
            else
                x++;
        }
        return false;
    }
};
```

动态规划

55.跳跃游戏（中等）

1. 题目描述

给定一个非负整数数组，你最初位于数组的第一个位置。数组中的每个元素代表你在该位置可以跳跃的最大长度。判断你是否能够到达最后一个位置。

示例 1:

```
输入: [2,3,1,1,4]
输出: true
解释: 我们可以先跳 1 步，从位置 0 到达 位置 1，然后再从位置 1 跳 3 步到达最后一个位置。
```

示例 2:

```
输入: [3,2,1,0,4]
输出: false
解释: 无论怎样，你总会到达索引为 3 的位置。但该位置的最大跳跃长度是 0 ， 所以你永远不可能到达最后一个位置。
```

2. 简单实现

dp[i]代表能否到达位置i，状态转移为：

`dp[i] = 是否存在0<=j<i，使得dp[j]=true且j+nums[j]>=i`

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int len = nums.size();
        if(len <= 1) return true;
        vector<bool> dp = vector<bool>(len, false);
        dp[0] = true;
        for(int i = 1; i < len; i++)
            for(int j = i-1; j >= 0; j--)
                if(dp[j] && j + nums[j] >= i){
                    dp[i] = true;
                    break;
                }
        return dp[len-1];
    }
};
```

3. 最优解法

观察到，如果能到i，就一定能到i之前的任何位置，所以可以从右往左推，省去dp的空间和一层for循环

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int size=nums.size();
        int LastPosition=size-1;
        for(int i=size-2;i>=0;--i)          {
            if(i+nums[i]>=LastPosition)//从后往前推
                LastPosition=i;
        }
        return LastPosition==0;
    }
};
```

4. 其他解法

- 如果某一个作为 起跳点 的格子可以跳跃的距离是 3，那么表示后面 3 个格子都可以作为 起跳点。
- 可以对每一个能作为 起跳点 的格子都尝试跳一次，把 能跳到最远的距离 不断更新。
- 如果可以一直跳到最后，就成功了。

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int k = 0;
        for (int i = 0; i < nums.size(); i++)  {
            if (i > k)
                return false;
            k = max(k, i + nums[i]);
        }
        return true;
    }

};
```

62.不同路径（中等）

1. 题目描述

一个机器人位于一个 *m x n* 网格的左上角 （起始点在下图中标记为“Start” ）。机器人每次只能向下或者向右移动一步。机器人试图达到网格的右下角（在下图中标记为“Finish”）。问总共有多少条不同的路径?

例如，上图是一个7 x 3 的网格。有多少可能的路径?

说明：m和 n 的值均不超过 100。

示例 1:

```
输入: m = 3, n = 2
输出: 3
解释:
从左上角开始，总共有 3 条路径可以到达右下角。
1. 向右 -> 向右 -> 向下
2. 向右 -> 向下 -> 向右
3. 向下 -> 向右 -> 向右
```

示例 2:

```
输入: m = 7, n = 3
输出: 28
```

2. 简单实现

`dp[i][j] = dp[i-1][j] + dp[i][j-1];`

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        if(m ==1 || n==1) return 1;
        vector<vector<int>> dp = vector<vector<int>>(m+1, vector<int>(n+1));
        for(int i = 2; i <= n; i++)
            dp[1][i] = 1;
        for(int i = 2; i <= m; i++)
            dp[i][1] = 1;
        for(int i = 2; i <= m; i++)
            for(int j = 2; j <= n; j++)
                dp[i][j] = dp[i-1][j]+dp[i][j-1];
        return dp[m][n];
    }
};
```

322.零钱兑换（中等）

1. 题目描述

   给定不同面额的硬币 coins 和一个总金额 amount。编写一个函数来计算可以凑成总金额所需的最少的硬币个数。如果没有任何一种硬币组合能组成总金额，返回 `-1`。

   示例 1:

   ```
   输入: coins = [1, 2, 5], amount = 11
   输出: 3
   解释: 11 = 5 + 5 + 1
   ```

   示例 2:

   ```
   输入: coins = [2], amount = 3
   输出: -1
   ```

   说明: 你可以认为每种硬币的数量是无限的。

2. 简单实现

   `dp[i][j]`表示用前i个硬币凑j所需要的最少的硬币数，状态转移为

   `dp[i][j] = min(dp[i-1], dp[i-1][j-k*coins[i]] + k)`

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        if(amount == 0) return 0;
        sort(coins.begin(), coins.end());
        vector<vector<long>> dp = vector<vector<long>>(coins.size(), vector<long>
(amount+1, long(INT_MAX)+1));
        for(int i = 0; i < coins.size(); i++)
```

```
            dp[i][0] = 0;//必不可少的初始化
        for(int i = 1; i*coins[0] <= amount; i++)
            dp[0][i*coins[0]] = i;//必不可少的初始化

        for(int i = 1; i < coins.size(); i++)
            for(int j = 1; j <= amount; j++){
                if(coins[i] > j)
                    dp[i][j] = dp[i-1][j];
                else if(coins[i] == j)
                    dp[i][j] = 1;
                else{
                    dp[i][j] = dp[i-1][j];
                    for(int k = 1; k*coins[i] <= j; k++)
                        dp[i][j] = min(dp[i][j], dp[i-1][j-k*coins[i]] + k);
                }
            }
        if(dp[coins.size()-1][amount] > INT_MAX)
            return -1;
        else
            return dp[coins.size()-1][amount];
    }
};
```

但是时间太久了嘤嘤嘤

3. 最优解法

f[i]表示凑够i需要的最少硬币数，状态转移为

`f[i] = min(f[i-某个硬币值]) + 1`

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> f = vector<int>(amount+1, -1);
        f[0] = 0;
        for(int i = 1; i <= amount; i++)
            for(int j = 0; j < coins.size(); j++)
                if(coins[j] <= i && f[i-coins[j]] != -1){
                        if(f[i] == -1)
                            f[i] = f[i-coins[j]] + 1;
                        else
                            f[i] = min(f[i],  f[i-coins[j]] + 1);
                }
        return f[amount];
    }
};
```

300最长上升子序列（中等）

1. 题目描述

给定一个无序的整数数组，找到其中最长上升子序列的长度。

示例:

```
输入: [10,9,2,5,3,7,101,18]
输出: 4
解释: 最长的上升子序列是 [2,3,7,101]，它的长度是 4。
```

说明:

- 可能会有多种最长上升子序列的组合，你只需要输出对应的长度即可。
- 你算法的时间复杂度应该为 O(*n2*) 。

进阶: 你能将算法的时间复杂度降低到 O(*n* log *n*) 吗?

2. 简单实现

dp[i]表示以nums[0...i]的最长上升子序列长度，状态转移为：

`dp[i] = 所有满足nums[j] < nums[i]的那些dp[j]的最大值+1`

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

3. 最优解法——动态规划+二分法实现O(nlogn)

解题思路：

- **降低复杂度切入点：** 解法一中，遍历计算 dp 列表需 $O(N)$，计算每个 $dp[k]$ 需 $O(N)$。

  1. 动态规划中，通过线性遍历来计算 dp 的复杂度无法降低；
  2. 每轮计算中，需要通过线性遍历 $[0,k)$ 区间元素来得到 $dp[k]$ 。我们考虑：是否可以通过重新设计**状态定义**，使整个 dp 为一个**排序列表**；这样在计算每个 $dp[k]$ 时，就可以通过二分法遍历 $[0,k)$ 区间元素，将此部分复杂度由 $O(N)$ 降至 $O(logN)$。

- **设计思路：**

  - **新的状态定义：**
    - 我们考虑维护一个列表 $tails$，其中每个元素 $tails[k]$ 的值代表 **长度为 $k+1$ 的子序列尾部元素的值。**
    - 如 $[1,4,6]$ 序列，长度为 $1,2,3$ 的子序列尾部元素值分别为 $tails=[1,4,6]$。
  - **状态转移设计：**
    - 设常量数字 N，和随机数字 x，我们可以容易推出：当 N 越小时，$N<x$ 的几率越大。例如：$N=0$ 肯定比 $N=1000$ 更可能满足 $N<x$。
    - 在遍历计算每个 $tails[k]$，不断更新长度为 $[1,k]$ 的子序列尾部元素值，**始终保持每个尾部元素值最小**（例如 $[1,5,3]$，遍历到元素 5 时，长度为 2 的子序列尾部元素值为 5；当遍历到元素 3 时，尾部元素值应更新至 3，因为 3 遇到比它大的数字的几率更大）。
  - $tails$ **列表一定是严格递增的：** 即当尽可能使每个子序列尾部元素值最小的前提下，子序列越长，其序列尾部元素值一定更大。
    - **反证法证明：** 当 $k<i$，若 $tails[k]>=tails[i]$，代表较短子序列的尾部元素的值 $>$ 较长子序列的尾部元素的值。这是不可能的，因为从长度为 i 的子序列尾部倒序删除 $i-1$ 个元素，剩下的为长度为 k 的子序列，设此序列尾部元素值为 v，则一定有 $v<tails[i]$（即长度为 k 的子序列尾部元素值一定更小），这和 $tails[k]>=tails[i]$ 矛盾。
    - 既然严格递增，每轮计算 $tails[k]$ 时就可以使用二分法查找需要更新的尾部元素值的对应索引 i。

- **算法流程：**

  - **状态定义：**
    - $tails[k]$ 的值代表 长度为 $k+1$ 子序列 的尾部元素值。
  - **转移方程：** 设 res 为 $tails$ 当前长度，代表直到当前的最长上升子序列长度。设 $j\in[0,res)$，考虑每轮遍历 $nums[k]$ 时，通过二分法遍历 $[0,res)$ 列表区间，找出 $nums[k]$ 的大小分界点，会出现两种情况：
    - **区间中存在** $tails[i]>nums[k]$：将第一个满足 $tails[i]>nums[k]$ 执行 $tails[i]=nums[k]$；因为更小的 $nums[k]$ 后更可能接一个比它大的数字（前面分析过）。
    - **区间中不存在** $tails[i]>nums[k]$：意味着 $nums[k]$ 可以接在前面所有长度的子序列之后，因此肯定是接到最长的后面（长度为 res），新子序列长度为 $res+1$。
  - **初始状态：**
    - 令 $tails$ 列表所有值 $=0$。
  - **返回值：**
    - 返回 res，即最长上升子子序列长度。

复杂度分析：

- **时间复杂度** $O(NlogN)$：遍历 $nums$ 列表需 $O(N)$，在每个 $nums[i]$ 二分法需 $O(logN)$。
- **空间复杂度** $O(N)$：$tails$ 列表占用线性大小额外空间。

```
class Solution {
```

```
public:
    int lengthOfLIS(vector<int>& nums) {
        if(nums.size() == 0) return 0;
        if(nums.size() == 1) return 1;
        vector<int> tail;
        tail.push_back(nums[0]);
        for(int i = 1; i < nums.size(); i++){
            if(tail[tail.size() - 1] < nums[i])
                tail.push_back(nums[i]);
            else{
                int l = 0;
                int r = tail.size()-1;
                while(l < r){
                    int mid = l + (r - l) / 2;
                    if(tail[mid] == nums[i]) break;
                    else if(tail[mid] < nums[i])
                        l = mid + 1;
                    else
                        r = mid;
                }
                if(l >= r)
                    tail[l] = nums[i];
            }
        }
        return tail.size();
    }
};
```

数学

172.阶乘后的零（简单）

1. 题目描述

给定一个整数 *n*，返回 *n*! 结果尾数中零的数量。

示例 1:

```
输入: 3
输出: 0
解释: 3! = 6, 尾数中没有零。
```

示例 2:

```
输入: 5
输出: 1
解释: 5! = 120, 尾数中有 1 个零.
```

说明: 你算法的时间复杂度应为 *O*(log *n*) 。

2. 简单实现

能构成10的就是10和2 * 5，而10 = 2 * 5，其实就是数n的因数里有几个5（因为2的个数比5多）

```
class Solution {
public:
    int trailingZeroes(int n) {
        int cnt = 0;
        while(n){
            n /= 5;
            cnt += n;
        }
        return cnt;
    }
};
```

171.Excel表列序号（简单）

1. 题目描述

给定一个Excel表格中的列名称，返回其相应的列序号。

例如，

```
    A -> 1
    B -> 2
    C -> 3
    ...
    Z -> 26
    AA -> 27
    AB -> 28
    ...
```

示例 1:

```
输入: "A"
输出: 1
```

示例 2:

```
输入: "AB"
输出: 28
```

示例 3:

```
输入: "ZY"
输出: 701
```

2. 简单实现

```
class Solution {
public:
    int titleToNumber(string s) {
        if(s.size() == 0) return 0;
        int ans = s[0] - 'A' + 1;
        for(int i = 1; i < s.size(); i++){
            ans = ans * 26 + (s[i] - 'A' + 1);//26进制
        }
        return ans;
    }
};
```

50.Pow(x, n)（中等）

1. 题目描述

   实现 pow(*x*, *n*)，即计算 x 的 n 次幂函数。

   示例 1:

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

   示例 2:

   ```
   输入: 2.10000, 3
   输出: 9.26100
   ```

   示例 3:

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

   说明:

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

2. 简单实现

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

69. x 的平方根（简单）

1. 题目描述

   实现 `int sqrt(int x)` 函数。

   计算并返回 *x* 的平方根，其中 *x* 是非负整数。

   由于返回类型是整数，结果只保留整数的部分，小数部分将被舍去。

   示例 1:

```
输入: 4
输出: 2
```

   示例 2:

```
输入: 8
输出: 2
说明: 8 的平方根是 2.82842...,
     由于返回类型是整数，小数部分将被舍去。
```

2. 简单实现

```
class Solution {
public:
    int mySqrt(int x) {
        int l = 0;
        int r = x;
        while(l <= r){
            long long mid = l + (r - l) / 2;
            long long temp = mid*mid;
            if(temp <= x && temp+2*mid+1 > x)
                return mid;
            else if(temp < x)
                l = mid + 1;
            else
                r = mid - 1;
        }
        return -1;
```

```
    }
};
```

29.两数相除（中等）

1. 题目描述

   给定两个整数，被除数 `dividend` 和除数 `divisor`。将两数相除，要求不使用乘法、除法和 mod 运算符。

   返回被除数 `dividend` 除以除数 `divisor` 得到的商。

   示例 1:

   ```
   输入: dividend = 10, divisor = 3
   输出: 3
   ```

   示例 2:

   ```
   输入: dividend = 7, divisor = -3
   输出: -2
   ```

   说明:

   - 被除数和除数均为 32 位有符号整数。
   - 除数不为 0。
   - 假设我们的环境只能存储 32 位有符号整数，其数值范围是 [−2^31, 2^31 − 1]。本题中，如果除法结果溢出，则返回 $2^{31} - 1$

2. 简单实现

   - 难点一：边界处理，直接用Long简化了
   - 难点二：除法每次对除数翻倍比较，不翻倍会超时

```
class Solution {
public:
    int divide(int dividend, int divisor) {
        if(dividend == 0) return 0;
        if(divisor == 1) return dividend;
        if(divisor == -1){
            if(dividend>INT_MIN) return -dividend;// 只要不是最小的那个整数，都是直接返回相反数就好啦
            return INT_MAX;// 是最小的那个，那就返回最大的整数啦
        }
        long a = dividend;
        long b = divisor;
        int sign = 1;
        if((a>0&&b<0) || (a<0&&b>0)){
            sign = -1;
        }
        a = a>0?a:-a;
        b = b>0?b:-b;
        long res = div(a,b);
```

```
        if(sign>0)return res>INT_MAX?INT_MAX:res;
        return -res;
    }
    int div(long a, long b){  // 似乎精髓和难点就在于下面这几句
        if(a<b) return 0;
        long count = 1;
        long tb = b; // 在后面的代码中不更新b
        while((tb+tb)<=a){
            count = count + count; // 最小解翻倍
            tb = tb+tb; // 当前测试的值也翻倍
        }
        return count + div(a-tb,b);
    }
};
```

166. 分数到小数（中等）

1. 题目描述

   给定两个整数，分别表示分数的分子 numerator 和分母 denominator，以字符串形式返回小数。

   如果小数部分为循环小数，则将循环的部分括在括号内。

   示例 1:

```
输入: numerator = 1, denominator = 2
输出: "0.5"
```

   示例 2:

```
输入: numerator = 2, denominator = 1
输出: "2"
```

   示例 3:

```
输入: numerator = 2, denominator = 3
输出: "0.(6)"
```

2. 简单实现

   整数部分小数部分分开算

```
class Solution {
public:
    void getDecimal(long& a, long& b, string& ans){//a/b的小数部分，保证0 < a < b
        unordered_map<int, int> m;
        int idx = ans.size();
        while(1){
            a *= 10;
            ans += to_string(a/b);
            a %= b;
            if(a == 0)
```

```
                break;
            if(m.count(a) <= 0)
                m[a] = idx++;
            else{
                if(ans[m[a]] == ans[ans.size()-1])//以1/3为例
                    ans = ans.substr(0, m[a]) +
                        '(' + ans.substr(m[a], ans.size()-m[a]-1) + ')';
                else//以1/6为例
                    ans = ans.substr(0, m[a]+1) +
                        '(' + ans.substr(m[a]+1, ans.size()-m[a]-1) + ')';
                break;
            }
        }
    }
    string fractionToDecimal(int numerator, int denominator) {
        if(numerator == 0) return "0";
        if(denominator == 0) return "inf";//实际上题目测试用例无次情况，去掉就能打败
100%...
        string ans = "";
        if((numerator>0 && denominator<0) || (numerator<0 && denominator>0))
            ans += "-";
        long a = numerator;//防止溢出
        long b = denominator;//防止溢出
        a = abs(a);
        b = abs(b);
        long integer = a / b;//防止溢出
        ans += to_string(integer);//整数部分
        a %= b;
        if(a == 0) return ans;
        ans += ".";
        getDecimal(a, b, ans);//获取小数部分
        return ans;
    }
};
```

其他

371. 两整数之和（简单）

1. 题目描述

   不使用运算符 `+` 和 `-` ，计算两整数 `a` 、 `b` 之和。

   示例 1:

```
输入: a = 1, b = 2
输出: 3
```

   示例 2:

```
输入: a = -2, b = 3
输出: 1
```

2. 最优实现

用异或做加法，用与操作做进位检验，例如可以用三步走的方式计算二进制值相加： 5---101，7---111

- 第一步：相加各位的值，不算进位，得到010（二进制每位相加就相当于各位做异或操作，101^111）
- 第二步：计算进位值，得到1010（相当于各位进行与操作得到101，再向左移一位得到1010，(101&111)<<1）
- 第三步：上两步结果相加仍是答案，因此重复上述两步，各位相加 010^1010=1000，进位值为100=(010 & 1010)<<1。
- 继续重复上述步骤：1000^100 = 1100，进位值为0，跳出循环，1100为最终结果。
- 结束条件：进位为0，即a为最终的求和结果。

```java
class Solution {
    public int getSum(int a, int b) {
        while(b != 0){
            int temp = a ^ b;//相加值
            b = (a & b) << 1;//进位值
            a = temp;
        }
        return a;
    }
}
```

150.逆波兰表达式求值（中等）

1. 题目描述

根据逆波兰表示法，求表达式的值。

有效的运算符包括 `+`, `-`, `*`, `/` 。每个运算对象可以是整数，也可以是另一个逆波兰表达式。

说明：

- 整数除法只保留整数部分。
- 给定逆波兰表达式总是有效的。换句话说，表达式总会得出有效数值且不存在除数为 0 的情况。

示例 1：

```
输入: ["2", "1", "+", "3", "*"]
输出: 9
解释: ((2 + 1) * 3) = 9
```

示例 2：

```
输入: ["4", "13", "5", "/", "+"]
输出: 6
解释: (4 + (13 / 5)) = 6
```

示例 3：

```
输入: ["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"]
输出: 22
解释:
  ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

2. 简单实现

用栈就可以了

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

```
    }
};
```

169. 多数元素（简单）

1. 题目描述

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

   你可以假设数组是非空的，并且给定的数组总是存在多数元素。

   示例 1:

   ```
   输入: [3,2,3]
   输出: 3
   ```

   示例 2:

   ```
   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(ans == nums[i]) cnt++;
            else if(cnt > 0) cnt--;
            else{
                ans = nums[i];
                cnt = 1;
            }
        }
        return ans;
    }
};
```

621.任务调度器（中等）

1. 题目描述

   给定一个用字符数组表示的 CPU 需要执行的任务列表。其中包含使用大写的 A - Z 字母表示的26 种不同种类的任务。任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。CPU 在任何一个单位时间内都可以执行一个任务，或者在待命状态。

   然而，两个**相同种类**的任务之间必须有长度为 **n** 的冷却时间，因此至少有连续 n 个单位时间内 CPU 在执行不同的任务，或者在待命状态。

你需要计算完成所有任务所需要的**最短时间**。

示例 1：

```
输入：tasks = ["A","A","A","B","B","B"], n = 2
输出：8
执行顺序：A -> B -> （待命） -> A -> B -> （待命） -> A -> B.
```

注：

1. 任务的总个数为 [1, 10000]。
2. n 的取值范围为 [0, 100]。

2. 解法设计

在前两种方法中，我们了解到应当尽早安排出现次数较多的任务。我们假设 `A` 为出现次数最多的任务，假设其出现了 `p` 次，考虑到冷却时间，那么执行完所有任务的时间至少为 `(p - 1) * (n + 1) + 1`。我们把这个过程形象化地用图 `1` 表现出，可以发现，CPU 产生了 `(p - 1) * n` 个空闲时间，只有 `p` 个时间是在工作的。

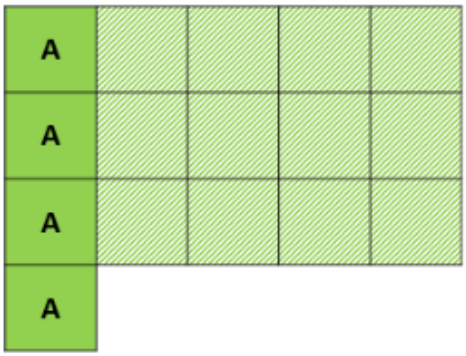


Figure 1

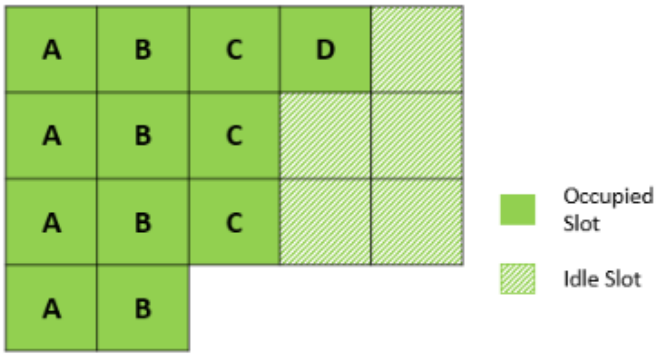


Figure 2

因此我们应当考虑把剩余的任务安排到这些空闲时间里，我们仍然按照这些任务的出现次序，从大到小进行安排，会有下面三种情况：

- 某个任务和 `A` 出现的次数相同，例如图 `2` 中的任务 `B`。此时我们只能让 `B` 占据 `p - 1` 个空闲时间，而在非空闲时间里额外安排一个时间给 `B` 执行；
- 某个任务比 `A` 出现的次数少 `1`，例如图 `2` 中的任务 `C`。此时我们可以让 `C` 占据 `p - 1` 个空闲时间，就可以全部执行完；
- 某个任务比 `A` 出现的次数少 `2` 或更多，例如图 `2` 中的任务 `D`。此时我们可以按照列优先的顺序，将 `D` 填入空闲时间中。因为 `D` 出现的次数少于 `p - 1`，因此无论从哪个位置开始按照列优先的顺序放置，都可以保证相邻的两个 `D` 之间满足冷却时间的要求。

在将所有的任务安排完成后，如果仍然有剩余的空闲时间，那么答案即为（任务的总数 + 剩余的空闲时间）；如果在安排某一个任务时，遇到了剩余的空闲时间不够的情况，那么答案一定就等于任务的总数。这是因为我们可以将空闲时间增加虚拟的一列，继续安排任务。如果不考虑这些虚拟的列，在原本空闲时间对应的那些列中的任务是可以按照要求顺序执行的，而增加了这些虚拟的列后，两个相邻的相同任务的间隔不可能减少，并且虚拟的列中的任务也满足冷却时间的要求，因此仍然顺序执行并跳过虚拟的列中剩余的“空闲时间”一定是可行的。

```
class Solution {
public:
```

```cpp
    int leastInterval(vector<char>& tasks, int n) {
        if(n == 0) return tasks.size();
        vector<int> cnt = vector<int>(26, 0);
        for(int i = 0; i < tasks.size(); i++)
            cnt[tasks[i]-'A']++;
        sort(cnt.begin(), cnt.end());
        int max_val = cnt[25] - 1, idle_slots = max_val * n;
        for (int i = 24; i >= 0 && cnt[i] > 0; i--)
            idle_slots -= min(cnt[i], max_val);
        cout << max_val << ' ' << idle_slots << ' ' << tasks.size();
        return idle_slots > 0 ? idle_slots + tasks.size() : tasks.size();
    }
};
```