总结

1. 后两道题难度相对较大，做的时候可以先仔细思考一下有没有更优解，否则很可能造成超时然后浪费更多的时间
2. 没把握的想好测试用例测完再提交，罚时太难受了

1403. 非递增顺序的最小子序列（简单）

1. 题目描述

   给你一个数组 `nums`，请你从中抽取一个子序列，满足该子序列的元素之和 **严格** 大于未包含在该子序列中的各元素之和。

   如果存在多个解决方案，只需返回 **长度最小** 的子序列。如果仍然有多个解决方案，则返回 **元素之和最大** 的子序列。

   与子数组不同的地方在于，「数组的子序列」不强调元素在原数组中的连续性，也就是说，它可以通过从数组中分离一些（也可能不分离）元素得到。

   注意，题目数据保证满足所有约束条件的解决方案是 **唯一** 的。同时，返回的答案应当按 **非递增顺序** 排列。

   示例 1：

   ```
   输入：nums = [4,3,10,9,8]
   输出：[10,9]
   解释：子序列 [10,9] 和 [10,8] 是最小的、满足元素之和大于其他各元素之和的子序列。但是 [10,9] 的元素之和最大。
   ```

   示例 2：

   ```
   输入：nums = [4,4,7,6,7]
   输出：[7,7,6]
   解释：子序列 [7,7] 的和为 14 ，不严格大于剩下的其他元素之和（14 = 4 + 4 + 6）。因此，[7,6,7] 是满足题意的最小子序列。注意，元素按非递增顺序返回。
   ```

   示例 3：

   ```
   输入：nums = [6]
   输出：[6]
   ```

   提示：

   - `1 <= nums.length <= 500`
   - `1 <= nums[i] <= 100`

2. 比赛实现

   隐含：要求的子序列的和大于整个数组和的一半，且是略大于（因为要求长度最小）

题目中要求的"非递增顺序"给了很大的暗示：可以排序，然后从后往前找，找到最短的、和大于整个数组和一半的、子数组即可

```
class Solution {
public:
    vector<int> minSubsequence(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        int aim = 0;
        for(int i = 0; i < nums.size(); i++)
            aim += nums[i];
        aim = aim / 2;
        int sum = 0;
        int idx = nums.size() - 1;
        vector<int> ans;
        while(idx >= 0){
            sum += nums[idx];
            ans.push_back(nums[idx]);
            if(sum > aim)
                return ans;
            idx--;
        }
        return ans;
    }
};
```

1404. 将二进制表示减到一的步骤数（中等）

1. 题目描述

给你一个以二进制形式表示的数字 `s` 。请你返回按下述规则将其减少到 1 所需要的步骤数：

- 如果当前数字为偶数，则将其除以 2 。
- 如果当前数字为奇数，则将其加上 1 。

题目保证你总是可以按上述规则将测试用例变为 1 。

示例 1：

```
输入：s = "1101"
输出：6
解释："1101" 表示十进制数 13 。
Step 1) 13 是奇数，加 1 得到 14
Step 2) 14 是偶数，除 2 得到 7
Step 3) 7  是奇数，加 1 得到 8
Step 4) 8  是偶数，除 2 得到 4
Step 5) 4  是偶数，除 2 得到 2
Step 6) 2  是偶数，除 2 得到 1
```

示例 2：

```
输入：s = "10"
输出：1
解释："10" 表示十进制数 2 。
Step 1) 2 是偶数，除 2 得到 1
```

示例 3：

```
输入：s = "1"
输出：0
```

提示：

- `1 <= s.length <= 500`
- `s` 由字符 `'0'` 或 `'1'` 组成。
- `s[0] == '1'`

2. 比赛实现

对字符串找规律:

- 如果末位是0（偶数），则直接右移（除2）
- 如果末位是1（奇数），则需要加一，反应在二进制串上相当于不断进位，举几个例子
  - 11001 -> 11010 -> 1101
  - 1011 -> 1100 -> 110 -> 11
  - 从以上例子可以看出，我们可以做的一个阶段性操作为：加1后，将末尾的0都去掉，总共需要的步骤数为：`1(进位) + 当前位起连续的1的个数(相当于进位后末尾新产生多少个0)`

```cpp
class Solution {
public:
    int numSteps(string s) {
        int idx = s.size() - 1;
        int ans = 0;
        while(idx > 0){
            if(s[idx] == '0'){
                ans++;
                idx--;
            }
            else{
                ans++;
                while(idx >= 0 && s[idx] == '1'){
                    ans++;
                    idx--;
                }
                if(idx >= 0)
                    s[idx] = '1';
            }
        }
        return ans;
    }
};
```

1405. 最长快乐字符串（中等）

1. 题目描述

如果字符串中不含有任何 `'aaa'`，`'bbb'` 或 `'ccc'` 这样的字符串作为子串，那么该字符串就是一个「快乐字符串」。

给你三个整数 `a`，`b`，`c`，请你返回 **任意一个** 满足下列全部条件的字符串 `s`：

- `s` 是一个尽可能长的快乐字符串。
- `s` 中 **最多** 有 `a` 个字母 `'a'`、`b` 个字母 `'b'`、`c` 个字母 `'c'`。
- `s` 中只含有 `'a'`、`'b'`、`'c'` 三种字母。

如果不存在这样的字符串 `s`，请返回一个空字符串 `""`。

示例 1：

```
输入：a = 1, b = 1, c = 7
输出："ccaccbcc"
解释："ccbccacc" 也是一种正确答案。
```

示例 2：

```
输入：a = 2, b = 2, c = 1
输出："aabbc"
```

示例 3：

```
输入：a = 7, b = 1, c = 0
输出："aabaa"
解释：这是该测试用例的唯一正确答案。
```

提示：

- `0 <= a, b, c <= 100`
- `a + b + c > 0`

2. 比赛实现

一开始用的dfs，超时，浪费了不少时间再改，实际上，本题只要一个可行解，可以用贪心策略做

- 用ans记录当前字符串，初始化为“”，a/b/c表示剩余可以使用的a/b/c的数量
- 每一步肯定是选取可以加在ans后面的、且剩余最多的字符加在后面，可选的加入数量为1/2
- 如果当前要加的字符是所有字符里数量最多的，且剩余数量大于2，则直接加2个在ans后面，因为它数量多，要尽可能的“浪费“
- 如果当前要加的字符不是所有字符里数量最多的，则只能加1个，因为当前字符起到的作用是”隔板“，用来分隔剩的最多的那个字符，需要省着点用. 例如, a=3, c=11，答案为ccaccaccacc，a作为隔板每次只加一个，才能引入更多的c

```
class Solution {
public:
    string longestDiverseString(int a, int b, int c) {
        map<int, vector<char>, greater<int>> m;//升序记录各字符的剩余数量
        if(a > 0)
            m[a].push_back('a');
        if(b > 0)
```

```
            m[b].push_back('b');
        if(c > 0)
            m[c].push_back('c');
        string ans = "";
        while(1){
            bool f = false;//标识还能不能继续往后面加
            auto it = m.begin();//从剩余数量最多的开始看
            while(!f && it != m.end()){
                int size = it->second.size();
                for(int i = 0; i < size; i++){
                    if(ans == "" || ans.back() != it->second[i]){//可以加在ans后
                        int cnt = it->first;
                        char c = it->second[i];
                        if(cnt >= 2 && it == m.begin()){//加2个
                            ans += c;
                            ans += c;
                            cnt -= 2;
                        }
                        else{//加1个
                            ans += c;
                            cnt -= 1;
                        }
                        if(size == 1)
                            m.erase(it);
                        else
                            it->second.erase(it->second.begin() + i);
                        if(cnt > 0)
                            m[cnt].push_back(c);
                        f = true;
                        break;
                    }
                }
                it++;
            }
            if(!f) break;
        }
        return ans;
    }
};
```

1406. 石子游戏III（困难）

1. 题目描述

Alice 和 Bob 用几堆石子在做游戏。几堆石子排成一行，每堆石子都对应一个得分，由数组 `stoneValue` 给出。

Alice 和 Bob 轮流取石子，**Alice** 总是先开始。在每个玩家的回合中，该玩家可以拿走剩下石子中的的前 **1、2 或 3 堆石子** 。比赛一直持续到所有石头都被拿走。

每个玩家的最终得分为他所拿到的每堆石子的对应得分之和。每个玩家的初始分数都是 **0** 。比赛的目标是决出最高分，得分最高的选手将会赢得比赛，比赛也可能会出现平局。

假设 Alice 和 Bob 都采取 **最优策略** 。如果 Alice 赢了就返回 *"Alice"* ，Bob 赢了就返回 *"Bob"*，平局（分数相同）返回 *"Tie"* 。

示例 1：

```
输入：values = [1,2,3,7]
输出："Bob"
解释：Alice 总是会输，她的最佳选择是拿走前三堆，得分变成 6 。但是 Bob 的得分为 7，Bob 获胜。
```

示例 2：

```
输入：values = [1,2,3,-9]
输出："Alice"
解释：Alice 要想获胜就必须在第一个回合拿走前三堆石子，给 Bob 留下负分。
如果 Alice 只拿走第一堆，那么她的得分为 1，接下来 Bob 拿走第二、三堆，得分为 5 。之后 Alice 只能拿到分数 -9 的石子堆，输掉比赛。
如果 Alice 拿走前两堆，那么她的得分为 3，接下来 Bob 拿走第三堆，得分为 3 。之后 Alice 只能拿到分数 -9 的石子堆，同样会输掉比赛。
注意，他们都应该采取 最优策略 ，所以在这里 Alice 将选择能够使她获胜的方案。
```

示例 3：

```
输入：values = [1,2,3,6]
输出："Tie"
解释：Alice 无法赢得比赛。如果她决定选择前三堆，她可以以平局结束比赛，否则她就会输。
```

示例 4：

```
输入：values = [1,2,3,-1,-2,-3,7]
输出："Alice"
```

示例 5：

```
输入：values = [-1,-2,-3]
输出："Tie"
```

提示：

- `1 <= values.length <= 50000`
- `-1000 <= values[i] <= 1000`

2. 简单实现

比赛时候想到了博弈论、动态规划、sum-a=b、倒着往前规划，几乎所有的要素都具备了，但是是第一次做这类题目，还是没能自己推出来，题解如下：

- 我们用一个数组 dp 来表示“在只剩下第 i 堆到最后一堆石子时，当前要拿石子的玩家最多能拿多少分”。假如算出了这个 dp 数组，那么最终答案就是判断 dp[0] （Alice的分数）和分数总和-dp[0]之间的大小关系即可。
- 我们倒着计算这个 dp 数组，对于dp[i]，我们可以这样思考：当前你的选择有 “取走一、二、三堆”，结果就是给对方留下了 dp[i+1] dp[i+2] dp[i+3] 对应的情况。也就是对方能够得到的最高分就是 dp[i+1]

dp[i+2] dp[i+3] 中的一个，而我们能得到的分数就是剩下的所有分数减去对方能拿到的分数。为了让我们拿到的更多，就得让对方拿到的最少。

- 因此有 dp[i]= sum{i,n} - min{dp[i+1],dp[i+2],dp[i+3]}，分别对应取走一堆、两堆、三堆石子的情况。

博弈论题第一次见肯定懵逼。本题属于信息透明的平等博弈，是博弈论中最基础的一种，思路就是倒着从游戏的最后一步开始反着算，对每个状态计算“玩家从该状态开始能不能获胜/最多能拿多少分”，用类似动态规划的思想一直算到第一步。对博弈论方面有兴趣的同学，推荐cxlove 大佬的两篇博客：

博弈类入门：https://blog.csdn.net/acm_cxlove/article/details/7854530

博弈类进阶：https://blog.csdn.net/acm_cxlove/article/details/7854526

```
class Solution {
public:
    string stoneGameIII(vector<int>& stoneValue) {
        int n = stoneValue.size();
        vector<int> dp(n+1);
        dp[n]=0;
        int sum=0;
        for(int i = n-1; i >= 0; i--){
            dp[i] = INT_MIN;//由于有负值分数，这里注意一下
            sum += stoneValue[i];
            for(int j = i; j < i+3 && j < n; j++)
                dp[i] = max(dp[i], sum-dp[j+1]);
        }
        int alice = dp[0];
        int bob = sum - dp[0];
        if(alice == bob) return "Tie";
        else if(alice > bob) return "Alice";
        else return "Bob";
    }
};
```