【第四十三课】傅里叶变换与信息隐写术（三）

1、2181. 合并零之间的节点

1. 初始化双指针：cur=head表示当前遍历指针，pre=null表示上一个零节点指针
2. 遍历链表，判断当前节点是否为零节点：
3. cur.val==0，表示是零节点，更新pre指针的next指向：
4. （1）如果cur不是最后一个零节点，则pre.next=cur；
5. （2）cur为最后一个零节点，则pre.next=null。更新pre指针指向新的零节点，pre=cur。
6. （3）cur.val!=0，表示为非零节点，将数值在零节点上进行求和，pre.val+=cur.val。
7. 返回链表头节点head

```
/**
 * Definition for singly-linked list.
 * function ListNode(val, next) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 * }
 */
/**
 * @param {ListNode} head
 * @return {ListNode}
 */
var mergeNodes = function(head) {
    let cur = head;//遍历指针
    let pre = null;//上一个零点
    while(cur){
        if(cur.val === 0){//是零节点，更新零节点指针pre
        // 就是指向当前节点值，更新next 去指向下一个零节点值
            if(pre){
                pre.next = cur.next ? cur : null;// 是否为最后一个零节点值，是的话置为空
            }
            pre = cur;
        }else{
            pre.val += cur.val;//否则，将数值项在零节点处累加
        }
        cur = cur.next;
    }
    return head;
};
```

2、2187. 完成旅途的最少时间

1. 二分查找
2. 找到完成 至少 totalTrips 趟旅途需要花费的 最少时间，可以使用二分查找降低时间复杂度
3. 首先可以找到完成一趟旅途所需要花费的时间的最少时间 min 作为二分查找的左边界 left
4. 然后可以通过计算 totalTrips * min 获得旅途花费的最多时间 作为二分查找的右边界 right
5. 计算中间时间 middle 以及 在 middle 内能完成得旅途数目 trips
6. 如果 trips 大于 totalTrips，表明仍然存在更小的时间来完成旅途，所以继续向左查找
7. 如果 trips 小于 totalTrips，表明 middle 内不足以完成约定的旅途，继续向右查找
8. 最终找到复合条件的时间

```
/**
 * @param {number[]} time
 * @param {number} totalTrips
 * @return {number}
 */
var minimumTime = function(time, totalTrips) {
    let left = Math.min(...time);//找到左边界 最小值
    let right = totalTrips * left;//找到右边界 最大值
    while(left < right){//证明二分查找是一个正确的区间
        let mid = left + Math.floor((right - left) / 2);//计算中间数
        let trips = 0;
        for(const t of time){//统计每辆车要完成旅途的时间
            trips += Math.floor(mid / t);
            if(trips >= totalTrips) break;
        }
        // 超过 totalTrips，缩短边界，往左查找；否则，往右查找
        if(trips >= totalTrips){
            right = mid;
        }else{
            left = mid + 1;
        }
    }
    return left;
};
```

3、2182. 构造限制重复的字符串

1. 本题的思路就是为了使字典序最大，满足条件下应该先放"大"的字母进去，于是我们先遍历一遍串统计一下字母出现的次数，然后从末尾开始放，每次放的时候先获得当前字母次数cnt[i]c 和r repeatLimit中的较小者，以num 储存，然后先放num 当前字母进去，然后cnt[i] -= num ；
2. 若当前字母都摆放完了，即 cnt[i]==0了，那么就可以去处理下一个字母了,i--即可;
3. 若当前字母还没摆完，即cnt[i] != 0，那么先放一个次大的字母进去，用下标jj从i - 1开始往0找第一

个 cnt[j]!=0的元素，

4. 如果能找到这个次大的字母，说明现在还能在遵守规则的前提放字母，把j指向的字母放进去，然后--cnt[j]表示用了一个j指向的字母，此时i不减，下轮循环继续处理i指向的字母；
5. 若找不到这个次大的字母，即j < 0，那么说明现在已经没办法放字母了，因为唯一能放的字母就是

```
/**
 * @param {string} s
 * @param {number} repeatLimit
 * @return {string}
 */
var repeatLimitedString = function(s, repeatLimit) {
        let arr = new Array(26).fill(0);
        for(let i = 0;i < s.length;++i){
            ++arr[s.charCodeAt(i) - 97];//记录字母出现的次数
        }
        let res = "",i = 25;
        while(i >= 0){
            let num  = arr[i];
            if(num == 0){
                --i;
                continue;//字母的数量是0，继续检查下一个
            }
            if(num <= repeatLimit){
                res += String.fromCharCode(i + 97).repeat(num);
                --i;//字母的数量小于等于限制可以直接加入结果
            }else{
               res += String.fromCharCode(i + 97).repeat(repeatLimit);
               arr[i] -= repeatLimit;
               let j = i - 1;
               while(arr[j] === 0){
                   --j;
               }
               if(j >= 0){
                    res += String.fromCharCode(j + 97);
                    --arr[j]
               }else{
                   break;
               }
            }
        }
        return res;
};
```

4、2178. 拆分成最多数目的正偶数之和

1. 因为是希望拆分的数目最多，那肯定是数字越小越好，所以定一个一个变量，
2. 从 2 开始，每次自增 2，当加到加不动了，把剩余还没有加上的偶数，直接加到最后一个数字上。
3. 因为前面的数字都是最小不重复的偶数，这样拆分的结果就符合最多数目。

```
/**
 * @param {number} finalSum
 * @return {number[]}
 */
var maximumEvenSplit = function(finalSum) {
    if(finalSum % 2 !== 0){
        return [ ];
    }
    let cur = 2;
    let sum = 0;
    let ans = [ ];
    while(true){
        sum += cur;
        ans.push(cur);
        if(sum > finalSum){
            ans.pop();
            ans[ans.length - 1] += (finalSum - sum + cur);
            break;
        }
        if(sum === finalSum){
            break;
        }
         cur += 2;
    }
    return ans;
};
```

5、2170. 使数组变成交替数组的最少操作数

1. 根据题目要求，我们最后要将数组变化成如下形式
    - 所有奇数下标的元素均相同
    - 所有奇数下标的元素均相同
    - 奇数下标的元素和偶数下标的元素不能相同

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var minimumOperations = function(nums) {
```

```
    const len = nums.length;
    if(len < 2) return 0;
    // 建立两个hash表，分别统计奇偶下标下，各个元素出现的频率
    let map1 = new Map(),map2 = new Map();
    for(let i = 0;i < len;i++){
        // 判断奇偶
        if( i & 1){
            // 统计奇数下标下面，所有元素出现的频率
            map1.has(nums[i]) ? map1.set(nums[i],map1.get(nums[i]) + 1) :
map1.set(nums[i],1);
        }else{
            map2.has(nums[i]) ? map2.set(nums[i],map2.get(nums[i]) + 1) :
map2.set(nums[i],1);
        }
    }
    // 将hash表制作成二维数组，
    let arr1 = [...map1.entries()],arr2 = [...map2.entries()];
    // 根据各个元素出现的频率，降序排序
    arr1.sort((a,b) => b[1] - a[1]);
    arr2.sort((a,b) => b[1] - a[1]);
    // 最大值不相等
    if(arr1[0][0] !== arr2[0][0]){
        return len - arr1[0][1] - arr2[0][1];
    }else{
        // 最大值相等的
        let sum1,sum2;
        // 分类讨论，arr2中次最大值 是否存在
        arr2[1] ? sum1 = arr1[0][1] + arr2[1][1] : sum1 = arr1[0][1];
        arr1[1] ? sum2 = arr2[0][1] + arr1[1][1] : sum2 = arr2[0][1];
        return Math.min(len - sum1,len - sum2);
    }

};
```