【第四十八课】本月度测试题

1、1302. 层数最深叶子节点的和

1. 遍历二叉树，将每一层值以数组的形式存入数组，将获取的数组末尾元素叠加求和输出结果

```
/**
 * Definition for a binary tree node.
/**
 * Definition for a binary tree node.
 * function TreeNode(val) {
 *     this.val = val;
 *     this.left = this.right = null;
 * }
 */
/**
 * @param {TreeNode} root
 * @return {number}
 */
var deepestLeavesSum = function(root) {
    var arr = [];
    dfs(root, 0);
    var res = 0;
    if(arr.length === 0) return res;
    for(var i=0;i<arr[arr.length-1].length;i++){
        res += arr[arr.length-1][i];
    }
    return res;
    function dfs(root, n){
        if(!root) return;
        if(!Array.isArray(arr[n])) arr[n] = [];
        arr[n].push(root.val);
        dfs(root.left, n+1);
        dfs(root.right, n+1);
    }
};
```

2、2145. 统计隐藏数组数目

1. 利用diff关系可以算出数组的最大值和最小值（值相对于第零个数），就可以求解出数组的数值区

间大小，和给定的区间大小进行比较即可

```
/**
 * @param {number[]} differences
 * @param {number} lower
 * @param {number} upper
 * @return {number}
 */
var numberOfArrays = function(differences, lower, upper) {
    let minNum = 0, maxNum = 0;
    let num = 0;
    for(let i = 0; i < differences.length; ++i){
        num += differences[i];
        // 计算数组的最大值和最小值（值都相对于第0个数，最后我们算区间，所以这不影响）
        minNum = Math.min(minNum, num);
        maxNum = Math.max(maxNum, num);
    }
    // 算两个区间
    let diff1 = maxNum - minNum;
    let diff2 = upper - lower;
    return diff1 > diff2 ? 0 : diff2 - diff1 + 1;
};
```

3、2202. K 次操作后最大化顶端元素

1. 重点是第二个条件，可以从删除的元素中找任意一个，返回栈顶，求的也是最终栈顶的最大值。
2. 很容易想到我们尽可能删除多点元素（有放回的时候最多删除 k - 1），这样可挑选的数字更多，然后从这些数中挑选最大的放回去就可以。注意处理一下边缘条件。

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */
var maximumTop = function(nums, k) {
    if (k === 0) {
        return nums[0]
    }
    if (k === 1) {
        return nums[1] ?? -1
    }
    if (nums.length === 1 && (k % 2 === 1)) {
        return -1
    }

    const slice = nums.slice(0, k - 1)
```

```
    return Math.max(...slice, nums[k] || 0)
};
```

4、2210. 统计数组中峰和谷的数量

1. 我们可以遍历数组 nums 判断并统计峰和谷的数量。
2. 具体地，我们判断除了数组首尾元素（首尾元素一定不可能是峰或谷）以外的每个下标 i 是否为峰与谷的一部分。
3. 我们用 res 来维护这一数量。
4. 为了防止重复遍历，我们只判断每个可能的峰或谷的第一个下标，即当 nums[i] = nums[i − 1] 时，我们跳过该下标。
5. 对于每个需要判断的下标 i，我们用整数 left 与 right 来表示它左右是否存在不相等邻居以及最近的不相等邻居与该元素的大小关系：其中 1 代表邻居大于该元素，-1 代表邻居小于该元素，0 代表未找到或不存在该方向的不相等邻居。left 与 right 的初值均为 0。
6. 为了计算 left 的状态值，我们从下标 i - 1 开始向左遍历，直至找到第一个不等于 $$\textit{nums}[i]$ 的元素（此时还需要按要求更新状态值）或到达数组开头。类似地，我们从下标 i + 1 开始向右遍历计算 right 的值。
7. 最终，下标 i 为峰或谷的一部分当且仅当 left = right 且 left 不等于0。如果满足这两个条件，则我们将 res 加上 1。当遍历完成数组后，res 即为数组峰与谷的数量，我们返回它作为答案。

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var countHillValley = function(nums) {
    let res = 0;    // 峰与谷的数量
    let n = nums.length;
    for (let i = 1; i < n - 1; ++i) {
        if (nums[i] == nums[i-1]) {
            // 去重
            continue;
        }
        let left = 0;    // 左边可能的不相等邻居对应状态
        for (let j = i - 1; j >= 0; --j) {
            if (nums[j] > nums[i]) {
                left = 1;
                break;
            } else if (nums[j] < nums[i]) {
                left = -1;
                break;
            }
        }
        let right = 0;    // 右边可能的不相等邻居对应状态
        for (let j = i + 1; j < n; ++j) {
            if (nums[j] > nums[i]) {
```

```
                right = 1;
                break;
            } else if (nums[j] < nums[i]) {
                right = -1;
                break;
            }
        }
        if (left == right && left != 0) {
            // 此时下标 i 为峰或谷的一部分
            ++res;
        }
    }
    return res;
};
```

5、2211. 统计道路上的碰撞次数

1. 用栈模拟，从左至右遍历，directions
2. 遇到 L，判断栈顶是否为 R 两份 或者 S 一份，如果是，分别积分，然后将栈顶置为 S。
3. 如果遇到 R，如果栈顶不是 R 需要清栈，再 push R。
4. 遇到 S，需要把栈中的 R 全部取出来计分。
5. 在这个栈中 R 和 S 不能同时存在。

```
/**
 * @param {string} directions
 * @return {number}
 */
var countCollisions = function(directions) {
    const stack = []
    let ans = 0
    for (let i = 0; i < directions.length; i ++) {
        const dir = directions[i]
        if (dir === 'L') {
            if (stack.length && stack[stack.length - 1] === 'R') {
                ans += 2
                stack.pop()
                for (let j = 0; j < stack.length; j ++) {
                    if (stack[j] === 'R') {
                        ans ++
                    }
                }
                stack.length = 0
                stack.push('S')
            } else if (stack.length && stack[stack.length - 1] === 'S') {
                ans += 1
```

```
            }
        } else if (dir === 'R') {
            if (stack.length && stack[stack.length - 1] === 'S') {
                stack.pop()
            }
            stack.push('R')
        } else {
            for (let j = 0; j < stack.length; j ++) {
                if (stack[j] === 'R') {
                    ans ++
                }
            }
            stack.length = 0
            stack.push('S')
        }
    }
    return ans
};
```

6、1560. 圆形赛道上经过次数最多的扇区

1. 【模拟】由于马拉松全程只会按照同一个方向跑，中间不论跑了多少圈，对所有扇区的经过次数的贡献都是相同的。
2. 因此此题的答案仅与起点和终点相关。从起点沿着逆时针方向走到终点的这部分扇区，就是经过次数最多的扇区。

```
/**
 * @param {number} n
 * @param {number[]} rounds
 * @return {number[]}
 */
var mostVisited = function(n, rounds) {
    const ret = [];
    const size = rounds.length;
    const start = rounds[0], end = rounds[size - 1];
    if (start <= end) {
        for (let i = start; i <= end; i++) {
            ret.push(i);
        }
    } else { // 由于题目要求按扇区大小排序，因此我们要将区间分成两部分
        for (let i = 1; i <= end; i++) {
            ret.push(i);
        }
        for (let i = start; i <= n; i++) {
```

```
            ret.push(i);
        }
    }
    return ret;
};
```

7、1567. 乘积为正数的最长子数组长度

1. 每一个数只有三种可能 大于小于0和等于0
2. 根据无后效性，当前状态只于前一个状态有关
3. 我们开俩个数组 一个 l 和 r 记录到目前为止乘积为正数的子数组最大长度和乘积为负数的子数组最大长度。
4. 我们为什么要记录乘积为负数的子数组的最大长度呢?
5. 因为计算乘积为正数的最大子数组长度转移状态时需要用到乘积为负数的最大长度子数组。
6. 如nums[i]>0时候 l [i]直接为 l [i-1]+1
7. nums[i]<0的时候, l [i]则需要 r [i-1]的长度， l [i]= r [i-1]+1;
8. 当nums[i]=0的时候，我们一切都要归零重新计算。
9. 返回计算过程中最大的 l [i]

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var getMaxLen = function(nums) {
    let l = 0
    let r = 0
    let max = 0
    let len = nums.length
    for(let i=0;i<len;i++){
        if(nums[i]>0){
            if(r!=0) r++
            l++
        }else if(nums[i]<0){
            [l,r] = [r,l]
            if(l!=0) l++
            r++
        }else{
            l = 0
            r = 0
        }
        max = Math.max(max,l)
    }
    return max
};
```

8、1573. 分割字符串的方案数

1. 首先统计字符串中 1 的个数，如果不是 3 的倍数，则直接返回 0，如果 cnt = 0, 则返回 (n-1)*(n-2)/2 % (10**9+7) 相当于在所有的位置中选择两个位置
2. 因为分为 3 组，所以可以计算出每一组应该有的 1 的数量，遍历字符串，找到所有 1 的位置，得到每一个分组中的 0 的个数，即为可以划分的方法（因为 0 的加入不会影响前边的分组），最后的结果为 cnt1 * cnt2

```
var numWays = function(s) {
    let n = s.length,
        dp = new Array(n).fill(0)
    // 统计字符中出现的次数
    // 如 10101 dp = [1,1,2,2,3]     然后根据排列组合方法解答
    if(s[0] == '1') dp[0] = 1
    for(let i=1;i<n;i++){
        if(s[i] == '1') dp[i] = dp[i-1] + 1
        else    dp[i] = dp[i-1]
    }
    if(dp[n-1] == 0)    return (n-1)*(n-2)/2 % (10**9+7)
    if(dp[n-1] %3 !== 0)    return 0
    let q1 = dp[n-1] / 3,
        q2 = q1 * 2
    let numAtArrayCount = (arr, num) => {
        return arr.lastIndexOf(num) - arr.indexOf(num) + 1;
    }
    return (numAtArrayCount(dp,q1) * numAtArrayCount(dp,q2)) % (10**9+7)
};
```

9、1546. 和为目标值且不重叠的非空子数组的最大数目

1. 采用Set存放前缀和，由于期望得到最大数目不重叠子数组，因此可以贪心选择，遍历到符合条件的区间就进行计数，随后清空Set重新开始贪心选择过程。

```
/**
 * @param {number[]} nums
 * @param {number} target
 * @return {number}
 */
var maxNonOverlapping = function(nums, target) {
    let s = new Set([0]);
    let sum = 0, res = 0;
```

```
    for(let n of nums) {
        sum += n;
        if(s.has(sum - target)) {
            res ++;
            s.clear();
        }
        s.add(sum);
    }
    return res;
};
```

10、1696. 跳跃游戏 VI

1. 从dp到dp+优先队列。
2. dp会超时
3. 所以优化的点在于把求窗口Max的过程缓存 减少重复计算
4. dp+单调队列的时候，主要要做的是维护好区间 start【max(0, i - k)】，单调队列中需要移除 dp[start - 1] 的值

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */
var maxResult = function(nums, k) {
    let n = nums.length;
    let dp = new Array(n);
    dp[0] = nums[0];
    let que = [0];
    for(let i  = 1;i<n;i++){
        while(que.length && que[0] < i - k){
            que.shift();
        }
        dp[i] = dp[que[0]] + nums[i];
        while(que.length && dp[que[que.length - 1]] < dp[i]){
            que.pop()
        }
        que.push(i)
    }
    return dp.pop();
};
```