【第三十五周】月度测试题

1、114. 二叉树展开为链表

1. 二叉树进行前序遍历，递归地获得各节点被访问到的顺序。
2. 遍历结束之后更新每个节点的左右子节点的信息，将二叉树展开为单链表。

```
var flatten = function(root) {
    const list = [];
    preorderTraversal(root, list);
    const size = list.length;
    for (let i = 1; i < size; i++) {
        const prev = list[i - 1], curr = list[i];
        prev.left = null;
        prev.right = curr;
    }
};

const preorderTraversal = (root, list) => {
    if (root != null) {
        list.push(root);
        preorderTraversal(root.left, list);
        preorderTraversal(root.right, list);
    }
}
```

2、1829. 每个查询的最大异或值

1. 注意条件 k < 2maximumBit以及 0 <= nums[i] < 2maximumBit，根据条件可知，最大的异或结果是 (1 << maximumBit) - 1。是不是每次都可以异或得到这个值呢？当然可以了，k 是我们自己挑的。
2. 比如现在 maximumBit = 3 时，(1 << maximumBit) - 1 的结果是 7。如果前面的异或结果是 7，那 k 取 0；如果前面异或结果是 0，k 取 7。无论如何最大异或结果都是 (1 << maximumBit) - 1。

   只需要使用变量保存前缀异或结果，拿它和最大异或结果再异或一次就可以得到 k 的值了

```
/**
 * @param {number[]} nums
 * @param {number} maximumBit
```

```
 * @return {number[]}
 */
var getMaximumXor = function(nums, maximumBit) {
    const maxXor = (1 << maximumBit) - 1;
    let prev = 0;
    const ans = [];
    for(let i = 0;i < nums.length;i++){
        prev ^= nums[i];
        ans.push(prev ^ maxXor) ;
    }
    return ans.reverse();
};
```

3、724. 寻找数组的中心下标

1. 首先在原数组上计算每个位置的前缀和
2. nums[i]代表：nums[0]+nums[1]+nums[2]+...+nums[i]
3. 遍历前缀和数组nums，判断左边和与右边和是否相等
4. 左边和 = nums[i - 1] || 0
5. 右边和 = sum - nums[i]

```
/**
 * @param {number[]} nums
 * @return {number}
 */
var pivotIndex = nums => {
    const len = nums.length;
    for (let i = 1; i < len; i++) {
        // 计算前缀和
        nums[i] = nums[i - 1] + nums[i];
    }
    // 数组总和
    const sum = nums[len - 1];
    for (let i = 0; i < len; i++) {
        // 左边和
        const left = nums[i - 1] || 0;
        // 右边和
        const right = sum - nums[i];
        // 若相等，返回下标
        if (left === right) return i;
    }
    // 遍历一遍，没找到，返回-1
    return -1;
};
```

4、1649. 通过指令创建有序数组

1. 线段树区间查询用循环实现;
2. 对于每一个要插入的数x，只查询大于x的数。小于x的数用总数减出来。

```
 /**
 * @param {number[]} instructions
 * @return {number}
 */
var Tree = function (maxNum) {
    var treeSize = maxNum & (maxNum - 1) == 0 ? 2 * maxNum : 4 * maxNum;
    this.treeArr = new Array(treeSize).fill(0);
};

Tree.prototype.getRangeCount = function (targetStart, targetEnd, currStart,
currEnd, n = 0) {
    if (targetEnd < currStart || targetStart > currEnd) {
        return 0;
    }
    if (targetStart <= currStart && targetEnd >= currEnd) {
        return this.treeArr[n];
    }
    // 开始二分
    var temp = (currStart + currEnd) >> 1;
    var a = this.getRangeCount(targetStart, targetEnd, currStart, temp, n * 2 +
1);
    var b = this.getRangeCount(targetStart, targetEnd, temp + 1, currEnd, n * 2
+ 2);
    return a + b;
}

Tree.prototype.addNum = function (num, currStart, currEnd, n = 0) {
    if (currStart <= num && currEnd >= num) {
        this.treeArr[n]++;
        if (currStart != currEnd) {
            // 二分赋值
            var temp = (currStart + currEnd) >> 1;
            this.addNum(num, currStart, temp, n * 2 + 1);
            this.addNum(num, temp + 1, currEnd, n * 2 + 2);
        }
    }
}

var createSortedArray = function (instructions) {
    // 直接使用搜索树无法获取相对数量!
    // 使用线段树统计!
    size = instructions.length;
    if (size < 3) {
```

```
        return 0;
    }
    var maxNum = Number.MIN_VALUE;
    for (i = 0; i < size; i++) {
        maxNum = Math.max(instructions[i], maxNum);
    }
    var tree = new Tree(maxNum);
    tree.addNum(instructions[0], 0, maxNum)
    tree.addNum(instructions[1], 0, maxNum)
    var ret = 0;
    var pivot = 1000000007;
    for (i = 2; i < size; i++) {
        // 比新插入值小的数字的数量
        var temp1 = tree.getRangeCount(0, instructions[i] - 1, 0, maxNum);
        temp1 = Math.min(temp1, i - temp1);
        // 比新插入值大的数字的数量
        var temp2 = tree.getRangeCount(instructions[i] + 1, maxNum, 0, maxNum);
        temp2 = Math.min(temp2, i - temp2);
        var cost = Math.min(temp1, temp2);
        ret += cost;
        ret = ret % pivot;
        tree.addNum(instructions[i], 0, maxNum);
    }
    return ret;
};
```

5、1838. 最高频元素的频数

1. 首先排序
2. 定义左指针，初始指向0
3. res代表频次，最低是1
4. 右指针每次遍历，更新sum
5. 如果sum比k大，需要降低sum，通过左移左指针来降低
6. 在sum允许的范围内，更新最高的频次，即两指针的最大距离

```
/**
 * @param {number[]} nums
 * @param {number} k
 * @return {number}
 */
var maxFrequency = (nums, k) => {
    // 排序
    nums.sort((a, b) => a - b);
    const len = nums.length;
    // 定义左指针，初始指向0
    // res最低是1
```

```
    let [sum, left, res] = [0, 0, 1];
    for (let right = 1; right < len; right++) {
        // 右指针每次遍历，更新sum
        sum += (nums[right] - nums[right - 1]) * (right - left);
        // 如果sum比k大，需要降低sum
        // 通过左移左指针来降低
        while (sum > k) {
            sum -= nums[right] - nums[left];
            left++;
        }
        // 在sum允许的范围内，更新最高的频次
        // 即两指针的最大距离
        res = Math.max(res, right - left + 1);
    }
    return res;
};
```

6、525. 连续数组

1. 问题是找0和1数量相同的连续子数组，如果把0看作-1，那么问题转化为和为0的连续子数组
2. 用map记录第一次出现某前缀和的下标
3. 设空数组的和为0，下标为-1
4. 遍历数组，用一个变量pre记录当前下标时的前缀和，用res记录当前最长子数组长度
5. 若元素为0，则pre - 1;若元素为1，则pre + 1
6. 判断Map中是否已有此前缀和
7. 若有，则res = Math.max(res, i - map.get(pre))（i为当前下标）
8. 若无，则map.set(pre, i)（i为当前下标）
9. 最终得到最长子数组长度为res

```
var findMaxLength = function(nums) {
    const n = nums.length
    const map = new Map()
    map.set(0, -1)
    let pre = 0
    let res = 0
    for (let i = 0; i < n; i++) {
        pre += nums[i] == 0 ? -1 : 1
        if (map.has(pre)) {
            res = Math.max(res, i - map.get(pre))
        } else {
            map.set(pre, i)
        }
    }
    return res
};
```

7、670. 最大交换

1. 核心就一句话：就是把第一个小数和它后面最大的大数进行交换
2. 要让一个数变大，要尽可能的让其高位变大 ,让最高位的小数和后面的大数交换

```
/**
 * @param {number} num
 * @return {number}
 */
var maximumSwap = function(num) {
    let last = new Array(10).fill(-1);
    num = Array.from(num.toString());
    for(let i = 0;i < num.length;i++){
        last[num[i] - '0'] = i;
    }
    for(let i = 0;i < num.length;i++){
        for(let d = 9;d > (num[i] - '0');d--){
            if(last[d] > i){
                let temp = num[last[d]];
                num[last[d]] = num[i];
                num[i] = temp;
                return Number(num.join(''));
            }
        }
    }
    return Number(num.join(''));
};
```

8、637. 二叉树的层平均值

1. 用一个二维数组来存放每层节点的值，用数组下标来标识当前的层级
2. 举例将每层的值处理成 [[3], [9, 20], [15, 7]] 这样一个二维数组，然后再计算每层（子数组）的平均值即可

```
/**
* Definition for a binary tree node.
* function TreeNode(val, left, right) {
*     this.val = (val===undefined ? 0 : val)
*     this.left = (left===undefined ? null : left)
*     this.right = (right===undefined ? null : right)
* }
```

```
 */
/**
 * @param {TreeNode} root
 * @return {number[]}
 */
var averageOfLevels = function(root) {
    const stack = [];
    const loop = (node,h) =>{
        if(!node) return;
        if(!stack[h]) stack[h] = [];
        stack[h].push(node.val);
        loop(node.left,h + 1);
        loop(node.right,h + 1);
    }
    loop(root,0);
    return stack.map(i => i.reduce((sum,cur) => sum + cur,0) / i.length);
};
```

9、304. 二维区域和检索 - 矩阵不可变

1. 初始化时对矩阵的每一行计算前缀和，检索时对二维区域中的每一行计算子数组和，然后对每一行的子数组和计算总和。
2. 具体实现方面，创建 m 行 n+1 列的二维数组 sums，其中 m 和 n 分别是矩阵 matrix 的行数和列数，nums[i] 为 matrix[i] 的前缀和数组。将 sums 的列数设为 n+1 的目的是为了方便计算每一行的子数组和，不需要对 col 1=0 的情况特殊处理。

```
 /**
 * @param {number[][]} matrix
 */
var NumMatrix = function(matrix) {
    const m = matrix.length;
    if(m > 0){
        const n = matrix[0].length;
        this.sums = new Array(m).fill(0).map(() => new Array(n + 1).fill(0));
        for(let i = 0;i < m;i++){
            for(let j = 0;j < n;j++){
                this.sums[i][j + 1] = this.sums[i][j] + matrix[i][j];
            }
        }
    }
};

/**
 * @param {number} row1
 * @param {number} col1
```

```
 * @param {number} row2
 * @param {number} col2
 * @return {number}
 */
NumMatrix.prototype.sumRegion = function(row1, col1, row2, col2) {
    let sum = 0;
    for(let i = row1; i <= row2;i++){
        sum += this.sums[i][col2 + 1] - this.sums[i][col1];
    }
    return sum;
};

/**
 * Your NumMatrix object will be instantiated and called as such:
 * var obj = new NumMatrix(matrix)
 * var param_1 = obj.sumRegion(row1,col1,row2,col2)
 */
```