【第三十九课】金融系统中的 RSA 算法 (三)

1、2110. 股票平滑下跌阶段的数目

1. 首先count初始值为prices长度，表示先统计单个数字的阶段
2. 再统计多个数字的阶段，k个连续递减数，总共有(k * (k - 1))/2个阶段
3. 然后恢复k，继续向后找

```
/**
 * @param {number[]} prices
 * @return {number}
 */
const getDescentPeriods = prices => {
    // count初始值为prices长度
    let count = prices.length;
    // 连续递减数的个数
    let k = 1;
    for (let i = 0; i < prices.length; i++) {
        if (prices[i] - prices[i + 1] === 1) {
            k++;
            continue;
        }
        // k个连续递减数，总共有 (k * (k - 1))/2 个阶段
        count += (k * (k - 1)) / 2;
        // 继续向后找，k初始化
        k = 1;
    }
    return count;
};
```

2、2140. 解决智力问题

1. 设有 n 个问题，定义 f[i] 表示解决区间 [i,n - 1] 内的问题可以获得的最高分数。
2. 倒序遍历问题列表，对于第 i 个问题，我们有两种决策：跳过或解决。
3. 若跳过，则有 f[i] = f[i + 1]。
4. 若解决，则需要跳过后续 brainpower[i] 个问题。记 j=i + brainpower[i] + 1，则有
5. $$f[i] = \begin{cases} point[i] + f[j], & j < n \\ point[i], & j \geq n \end{cases}$$

6. 这两种决策取最大值,最后答案为 f[0]。

```
/**
 * @param {number[][]} questions
 * @return {number}
 */
//  倒着刷表
// 动态规划
var mostPoints = function(questions) {
    let len = questions.length;
    let dp = new Array(questions.length).fill(0);
    dp[len - 1] = questions[len - 1][0];
    for (let i = len - 2; i >= 0; i--) {
        dp[i] = Math.max(dp[i + 1], questions[i][0] + (dp[i + questions[i][1] +
1] || 0))
    }
    return dp[0];
};
```

3、2121. 相同元素的间隔之和

1. 用哈希map获取所有相同的数的下标数组；
2. 分别求每个哈希表的value里面的间隔之和；
3. 求出相邻值之间的差值
4. 分别求出值左右的间隔之和

```
/**
 * @param {number[]} arr
 * @return {number[]}
 */
var getDistances = function(arr) {
    const length = arr.length;
    const map = new Map();
    const res = new Array(length).fill(0);
    //第一步：用哈希map获取所有相同的数的下标数组；
    for(let i=0; i<length; i++) {
        const num = arr[i];
        if(map.get(num)) map.get(num).push(i);
        else  map.set(num, [i]);
    }
    //第二步：分别求每个哈希表的value里面的间隔之和；
    for(const indexArr of map.values()) {
        const length = indexArr.length;
        const diff = [];
        //第三步：求出相邻值之间的差值
        for(let i=1; i<length; i++) {
```

```
            diff.push(indexArr[i]-indexArr[i-1])
        }
        //第四步：分别求出值左右的间隔之和 放到答案数组
        let leftSum = 0;
        let rightSum = 0;
        const n = diff.length;
        for(let i=0; i<n; i++){
            rightSum += ((n-i) * diff[i]);
        }
        res[indexArr[0]] = leftSum + rightSum;
        for(let i=0; i<n; i++){
            const distance = diff[i];
            leftSum = leftSum + (i+1) * distance;
            rightSum = rightSum - (n-i) * distance;
            res[indexArr[i+1]] = leftSum + rightSum;
        }
    }
    return res
};
```

4、2115. 从给定原材料中找到所有可以做出的菜

1. hash表记录recipe需要几个ingredients， 当前已经有几个
2. hash表记录每个ingredient，哪些recipe需要它
3. 遍历supplies，给每个需要它的recipe加1，如果该recipe的ingredients已经凑齐了，就把它加入答案，同时加入supplies数组

```
/**
 * @param {string[]} recipes
 * @param {string[][]} ingredients
 * @param {string[]} supplies
 * @return {string[]}
 */
var findAllRecipes = function(recipes, ingredients, supplies) {
    const res = [];
    const hadSupplies = supplies;
    for(let k = 0; k < 100; k ++){
        for(let i = 0; i < recipes.length; i++) {
            let Ok = true;
            if(hadSupplies.includes(recipes[i])) {
                continue;
            }
```

```
            for(let j = 0; j < ingredients[i].length; j++) {
                if(!hadSupplies.includes(ingredients[i][j])) {
                    Ok = false
                    break;
                }
            }
            if(Ok === true) {
                res.push(recipes[i]);
                hadSupplies.push(recipes[i]);
            }
        }
    }
    return res;
};
```

5、2096. 从二叉树一个节点到另一个节点每一步的方向

1. 拼接出根节点到开始节点的路径和根结点到结束节点的路径
2. 裁剪共同路径

```
/**
 * 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
 * @param {number} startValue
 * @param {number} destValue
 * @return {string}
 */

var getDirections = function(root, startValue, destValue) {
    let p1 = '';
    let p2 = '';
    function genPath(node, path) {
        if (!node) return;
        if (node.val === startValue) p1 = path;
```

```
        if (node.val === destValue) p2 = path;
        genPath(node.left, path + 'L');
        genPath(node.right, path + 'R');
    }
    // 拼接路径
    genPath(root, '');
    // 裁剪共同路径
    let i = 0;
    while (p1[i] === p2[i]) i++;
    p1 = p1.slice(i);
    p2 = p2.slice(i);
    return "U".repeat(p1.length) + p2;
};
```

6、1028. 从先序遍历还原二叉树

1. 连字符的个数代表节点的 level（深度）
2. 因为前序遍历 根|左|右根|左|右，字符串开头的节点是根节点，后面的节点可以通过 level 找3、父亲：儿子的 level 要比父亲大 1，不满足就不是父亲
3. 当前节点的父亲，肯定在它的左边，从左往右扫描，儿子的父亲在左边，需要栈去记忆。

```
/**
 * 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 {string} traversal
 * @return {TreeNode}
 */
var recoverFromPreorder = (s) => {
    const stack = []; // 维护一个栈
    for (let i = 0; i < s.length; ) {
        let curLevel = 0; // 当前构建的节点所属的level
        while (i < s.length && s[i] == '-') { // 数数有几个连字符
            curLevel++;       // 统计它的level
            i++;              // 扫描的指针+1
        }
        let start = i;     // 记录下节点值字符串的开始位置
        while (i < s.length && s[i] != '-') { // 扫描节点值字符串
            i++;              // 扫描的指针+1
        }
        const val = s.substring(start, i); // 截取出节点值
```

```
        const curNode = new TreeNode(val); // 创建节点
        if (stack.length == 0) { // 此时栈为空，curNode为根节点
            stack.push(curNode);    // 入栈，成为栈底
            continue;               // 它没有父亲，不用找父亲，continue
        }
        while (stack.length > curLevel) {// 只要栈高>当前节点的level，就栈顶出栈
            stack.pop();
        }
        if (stack[stack.length - 1].left) { // 栈顶是父亲了，但左儿子已经存在
            stack[stack.length - 1].right = curNode; // curNode成为右儿子
        } else {
            stack[stack.length - 1].left = curNode; // 否则，成为左儿子
        }
        stack.push(curNode); // curNode自己也是父亲，入栈，等儿子
    }

    return stack[0]; // 栈底节点肯定是根节点
};
```