【第三十七周】金融系统中的 RSA 算法 (一)

1、914. 卡牌分组

1. 题目要求， 将整副牌分组，组内每张牌的数值相同，各组牌的张数相同。
2. 统计牌面数值的出现次数。使用哈希Map:次数统计非常简单，只需要遍历 deck 数组，依次统计出现的次数就行。
3. 分析合适的每组张数: 使用 辗转相除法 就可以解决张数的问题。
4. 每一组都有 X 张牌，那么 X 和卡牌总数 N 是什么关系？从数学的角度上说，X 是 N 的约数。
5. 辗转相除法是以除数和余数反复做除法运算，当余数为 0 时，取当前算式除数为最大公约数的计算公式。查看gcd函数
6. 为了符合题意，最后还需要 判断该公约数是否大于等于 2。

```
/**
 * @param {number[]} deck
 * @return {boolean}
 */
var hasGroupsSizeX = function(deck) {
    // 最大公约数计算公式
    function gcd(num1, num2) {
        // 利用辗转相除法来计算最大公约数
        return num2 === 0 ? num1 : gcd(num2, num1 % num2);
    }

    // 相同牌出现次数Map
    let timeMap = new Map();

    // 遍历牌
    deck.forEach(num => {
        // 统计每张牌出现的次数
        timeMap.set(num, timeMap.has(num) ? timeMap.get(num) + 1 : 1);
```

```
    });

    // Map.protype.values()返回的是一个新的Iterator对象，所以可以使用扩展运算符(...)来构造成数组
    let timeAry = [...timeMap.values()];

    /*
    最大公约数
    因为该数组是出现次数数组，最小值至少为1（至少出现1次），所以默认赋值为数组首位对公约数计算无干扰
    */
    let g = timeAry[0];

    // 遍历出现次数，计算最大公约数
    timeAry.forEach(time => {
        // 因为需要比较所有牌出现次数的最大公约数，故需要一个中间值
        g = gcd(g, time);
    });

    // 判断是否满足题意
    return g >= 2;
};
```

2、457. 环形数组是否存在循环

1. 在每次移动中，快指针需要走 2 次，而慢指针需要走 1 次；
2. 每次移动的步数等于数组中每个位置存储的元素；
3. 当快慢指针相遇的时候，说明有环。
4. 起始时，让快指针先比慢指针多走一步，当两者在满足题目的两个限制条件的情况下，快满指针能够相遇，则说明有环。
5. 关键在于 题目的两个限制条件：
6. （1）在每次循环的过程中，必须保证所经历过的所有数字都是 同号 的。所以，在快指针经历过的每个位置都要判断一下和出发点的数字是不是相同的符号。
7. （2）当快慢指针相遇的时候，还要判断环的大小不是 1。所以，找到相遇点的位置后，如果再走 1 步，判断是不是自己。

```
/**
 * @param {number[]} nums
 * @return {boolean}
 */
var circularArrayLoop = function(nums) {
    const n = nums.length;
    for (let i = 0; i < n; i++) {
        if (nums[i] === 0) {
            continue;
```

```
        }
        let slow = i, fast = next(nums, i);
        // 判断非零且方向相同
        while (nums[slow] * nums[fast] > 0 && nums[slow] * nums[next(nums,
fast)] > 0) {
            if (slow === fast) {
                if (slow !== next(nums, slow)) {
                    return true;
                } else {
                    break;
                }
            }
            slow = next(nums, slow);
            fast = next(nums, next(nums, fast));
        }
        let add = i;
        while (nums[add] * nums[next(nums, add)] > 0) {
            const tmp = add;
            add = next(nums, add);
            nums[tmp] = 0;
        }
    }
    return false;
}

const next = (nums, cur) => {
    const n = nums.length;
    return ((cur + nums[cur]) % n + n) % n; // 保证返回值在 [0,n) 中
}
```

3、926. 将字符串翻转到单调递增

1. 写动态规划看状态转移方程，写状态转移方程看定义状态。
2. 定义dp[i][0], dp[i][0]表示前i个元素递增且第i个元素为0的最小翻转次数，
3. 定义dp[i][1]， dp[i][1]表示前i个元素递增且第i个元素为1的最小翻转次数。
4. 由定义可知，如果前i个元素最后以0结尾且满足单调递增，那么前i个元素必须全部为0，由此可得dp[i][0]的状态转移如下：
   dp[i][0] = dp[i-1][0] + (s.charAt(i)=='0'?0:1);
5. 由定义可知， dp[i][1]只要满足最后一个元素为1就行，那么前i-1个元素既可以为0，也可以为1，因此dp[i][1]的状态转移如下：
   dp[i][1] = min(dp[i-1][1], dp[i-1][0]) + (s.charAt(i)=='1'?0:1);
6. 最后取dp[i][0],dp[i][1]中的较小的即可。

```
/**
 * @param {string} s
```

```
 * @return {number}
 */
var minFlipsMonoIncr = function(s) {
    //dp[i][0]表示前i个元素，最后一个元素为0的最小翻转次数；
    //dp[i][1]表示前i个元素，最后一个元素为1的最小翻转次数
        let dp = Array.from({ length: s.length + 1 }).map(item => [0, 0]);
        //初始化
        dp[0][0]=s.charAt(0)=='0'?0:1;
        dp[0][1]=s.charAt(0)=='1'?0:1;
        //状态转移
        for (let  i = 1; i <s.length ; i++) {
            dp[i][0]=dp[i-1][0]+(s.charAt(i)=='0'?0:1);
            dp[i][1]=Math.min(dp[i-1][0],dp[i-1][1])+(s.charAt(i)=='1'?0:1);
        }
        return Math.min(dp[s.length-1][0],dp[s.length-1][1]);
};
```

4、1201. 丑数 III

1. 先找到a,b,c里最小的那个数，比如是a，那么第n个丑数肯定是小于等于 n * a
2. 因为 0 到 n*a范围内是有可能出现数字，可以被b或c整除的
3. 开始二分法的做法了，将 n * a 置为上限 ceil，0 置为下限 0。
4. mid = (ceil+floor) / 2 这个数里包含了多少丑数
5. 如果上一步的数字等于 n，判断当前的 mid 是否是丑数，如果是，直接返回 mid，如果不是，将 ceil 置为 mid - 1；
   如果上一步的数字大于 n,将ceil置 为mid - 1；
   如果上一步的数字小于 n,将floor置为 mid + 1；
6. 指定数字num范围内的丑数数量为：num/a + num/b + num/c - num/lcm(ab) - num/lcm(ac) - num/lcm(bc) + num/lcm(abc)

```
/**
 * @param {number} n
 * @param {number} a
 * @param {number} b
 * @param {number} c
 * @return {number}
 */
var nthUglyNumber = function(n, a, b, c) {
    // 先将数值转换为 BigInt 类型
    a = BigInt(a), b = BigInt(b), c = BigInt(c), n = BigInt(n);

    // BigInt 不能使用 Math 函数判断，所以自己写一个
    const min = (a, b, c) => {
        let m = a;
        if (m > b) {
```

```
            m = b;
        }
        if (m > c) {
            m = c;
        }

        return m;
    };
    // 求最大公约数
    const gcd = (a, b) => {
        if (b === 0n) {
            return a;
        } else {
            return gcd(b, a % b);
        }
    };
    // 求最小公倍数
    const lcm = (a, b) => {
        return a * b / gcd(a, b);
    };
    // 检查是否是丑数
    const check = (val) => {
        return val % a === 0n || val % b === 0n || val % c === 0n;
    };

    let r = n * min(a, b, c);
    let l = 0n;
    let a_b = lcm(a, b);
    let a_c = lcm(a, c);
    let b_c = lcm(b, c);
    let a_b_c = lcm(a_b, c);

    // 二分查找丑数
    while (l < r) {
        let mid = l + (r - l) / 2n;
        let count = mid / a + mid / b + mid / c - mid / a_b - mid / b_c - mid /
a_c + mid / a_b_c;

        if (count === n) {
        // 当 count 等于 n 时还需要再判断是否为丑数，因为对于BigInt的除法来说， 4 / 2 和
5 / 2 的结果是相等的
            if (check(mid)) {
                return mid;
            } else {
                r = mid - 1n;
            }
        } if (count < n) {
            l = mid + 1n;
        } else {
```

```
            r = mid - 1n;
        }
    }

    return check(l) ? l : -1;
};
```