线程池与任务队列

最近的请求次数

由于只需要最近的3000以内的数据，每次加入后，然后判断序列最先边的数据是否在这个范围内，如何不在就把它移出，最重要的是，这个输入的时间序列是一个有序的，
程序的时间复杂度为:一个平均的时间节点插入数据的个数。

```
// 933. 最近的请求次数
var RecentCounter = function() {
    this.pingArray = [];
};

/**
 * @param {number} t
 * @return {number}
 */
RecentCounter.prototype.ping = function(t) {
    this.pingArray.push(t);
    while(this.pingArray[0] < t - 3000){
        this.pingArray.shift();
    }
    return this.pingArray.length;
};
// return this.q.length;
/**
 * Your RecentCounter object will be instantiated and called as
such:
 * var obj = new RecentCounter()
 * var param_1 = obj.ping(t)
 */
```

第K个数

1.由题设可知，起始的几个素数是1、3、5、7，其中基础因子是3、5、7;

2.后续的素数由3、5、7这三个数互相乘法结合(也就是因式分解后只有3、5、7这3个因子);

3.设num3、num5、num7代表3、5、7要取答案队列中第几位的数来进行相乘(如3、5、7就是与队列中第一位的1分别相乘的结果; 9、15、21则为第二位3分别相乘的结果);

4.后续数规律: 3中各自在答案队列中取得的数乘以自身(3、5、7), 取三者间最小的数为下一个入队的数, 并且要将入答案队列的对应数加

```
// 面试题 17.09. 第 k 个数
/**
 * @param {number} k
 * @return {number}
 */
 var getKthMagicNumber = function(k) {
    var p3 = 0;
    var p5 = 0;
    var p7 = 0;
    var dp = new Array(k);
    dp[0] = 1;
    for (let i = 1; i < k; i++) {
        dp[i] = Math.min(dp[p3] * 3,  Math.min(dp[p5] * 5, dp[p7] *
7));
        if (dp[i] == dp[p3] * 3) p3++;
        if (dp[i] == dp[p5] * 5) p5++;
        if (dp[i] == dp[p7] * 7) p7++;
    }
    return dp[k-1];
};
```

亲密字符串

两种情况返回true：

（1）只有2处不同，且两次不同是可交换，如ab、ba。

（2）没有不同，但是所组成的字母有重复

```
// 859. 亲密字符串
/**
 * @param {string} A
 * @param {string} B
 * @return {boolean}
 */
 var buddyStrings = function(A, B) {
```

```
  if(A.length !== B.length) return false
  if(A === B) {
    return A.length > new Set(A).size
  }
  let a = ''
  let b = ''
  for(let i = 0; i < A.length; i++){
    if(A[i] !== B[i]){
      a = A[i] + a
      b += B[i]
    }
  }
  return a.length === 2 && a === b
};
```

柠檬水找零

由于顾客只可能给你三个面值的钞票，而且我们一开始没有任何钞票，因此我们拥有的钞票面值只可能是 5 美元，10 美元和 20 美元三种。基于此，我们可以进行如下的分类讨论。

- 5 美元，由于柠檬水的价格也为 5 美元，因此我们直接收下即可。
- 10 美元，我们需要找回 5 美元，如果没有 5 美元面值的钞票，则无法正确找零。
- 20美元，我们需要找回 15 美元，此时有两种组合方式，
    - 一种是一张 10 美元和 5 美元的钞票，一种是 3 张 5 美元的钞票，如果两种组合方式都没有，则无法正确找零。
    - 当可以正确找零时，两种找零的方式中我们更倾向于第一种，即如果存在 5 美元和 10 美元，我们就按第一种方式找零，否则按第二种方式找零，因为需要使用5 美元的找零场景会比需要使用 10 美元的找零场景多，我们需要尽可能保留 5 美元的钞票。

基于此，我们维护两个变量five 和ten 表示当前手中拥有的 5 美元和 10 美元钞票的张数，从前往后遍历数组分类讨论即可。

```
// 860. 柠檬水找零
/**
 * @param {number[]} bills
 * @return {boolean}
 */
 var lemonadeChange = function(bills) {
    let five = 0, ten = 0
```

```
    for (const bill of bills) {
        if (bill === 5) five++
        if (bill === 10) {
            five--
            ten++
        }
        if (bill === 20) {
           if(ten && five){
                ten--
                five--
           }else if (five >=3){
               five  -=3;
           }else{
               return false;
           }
        }

        if (ten < 0 || five < 0) return false;
    }
    return true
};
```

煎饼排序

我们可以将最大的元素（在位置 i，下标从 1 开始）进行煎饼翻转 i 操作将它移动到序列的最前面，然后再使用煎饼翻转 A.length 操作将它移动到序列的最后面。 此时，最大的元素已经移动到正确的位置上了，所以之后我们就不需要再使用 k 值大于等于 A.length 的煎饼翻转操作了。

我们可以重复这个过程直到序列被排好序为止。 每一步，我们只需要花费两次煎饼翻转操作。

```
// 969. 煎饼排序
/**
 * @param {number[]} arr
 * @return {number[]}
 */
//求得最大数的下标
function getMaxIndex(nums) {
    let max = 0;
    for (let i = 0; i < nums.length; i++) {
        if (nums[i] > nums[max]) {
```

```
                max = i;
            }
        }
        return max;
}
// 反转前K个元素
var reverse = function (arr, k) {
    if (k < 1) {
        return;
    }
    let i = 0; j = k;
    while (i < j) {
        [arr[i], arr[j]] = [arr[j], arr[i]];
        i++;
        j--;
    }
}
var pancakeSort = function (arr) {
    let ans = [], max;
    while (arr.length > 1) {
        max = getMaxIndex(arr);
        max > 0 && (ans.push(max + 1));
        reverse(arr, max);
        reverse(arr, arr.length - 1);
        ans.push(arr.length)
        arr.pop();
    }
    return ans;
};
```

任务调度器

设置一个矩阵，矩阵列数为n+1，因为同一行不能出现两次相同任务才能保证相隔n个时间

执行最多次数的任务，次数为maxExec

假设共需要执行3次A，3次B，2次C，则，行数maxExec=3，前maxExec-1行，都是满的，共(maxExec-1)(n+1)个时间
最后一行的个数，为等于重复最多次数的任务种类数maxCount。

如上面例子，重复最多次数为3，执行3次的任务有两个，分别为A和B，所以maxCount=2。总时间=(maxExec-1)(n+1)+maxCount，有一种情况是，(maxExec-1)(n+1)+maxCount比任务总次数少，那总时间=tasks.length。

这种情况时，任务间就不需要空闲间隔了，可以一个挨一个的执行

```
// 621. 任务调度器
var leastInterval = function(tasks, n) {
    const freq = _.countBy(tasks);

    const maxExec = Math.max(...Object.values(freq));

    let maxCount = 0;
    Object.values(freq).forEach(v => {
        if (v === maxExec) {
            maxCount++;
        }
    })

    return Math.max((maxExec - 1) * (n + 1) + maxCount,
tasks.length);
};
```