为精简算法，忽略所有输入输出函数，所有输入内容已在程序中给定

算法实现题 4-1 会场安排问题：

```
const activityArrange = (k, acts) => {
    // res 记录需要的会场数目，valid 记录当前会场最后一个活动的结束时间
    let res = 0, valid = 0
    while (k) {
        // 遍历所有活动
        acts.forEach(act => {
            // 当前活动开始时间晚于最后一个活动结束时间
            // 且当前活动未被安排
            if (act[0] > valid && !act[2]) {
                // 将当前活动加入当前会场,更新最后一个活动的结束时间
                valid = act[1]
                // 将当前活动从之后的遍历中排除
                act[2] = true
                k--
            }
        })
        // 重置特征量
        valid = 0
        // 结束当前会场的安排,所需会场数+1
        res++
    }
    // 返回所需的会场数量
    return res
}

// 使用一个二维数组记录活动情况
let activities = [
    [1, 23, false],
    [12, 28, false],
    [25, 35, false],
    [27, 80, false],
    [36, 50, false]
]
// 使用测试数据进行测试
activityArrange(activities.length, activities) // 输出结果为 3
```

本题贪心的点在于，先按开始时间将所有活动排序，每次就可以选当前活动结束后最早开始的一个活动，再更新当前会场的活动结束时间 valid，如果输入未按开始时间排序，需花费 NlogN 时间进行排序，如果在一次遍历后仍有活动未被安排，则需要新开一个会场，重复安排的步骤，共需遍历 N 次，每次需遍历所有未进行的活动，最坏情况下的时间复杂度为 O(N ^ 2)

课后作业

算法实现题 硬币找钱问题

```
const values = [5, 10, 20, 50, 100, 200]
const coinChange = (coins, target) => {
    // 所需硬币总数
    let res = 0
    // 转换为以分为单位便于处理
    target *= 100
    // 从大到小遍历所有硬币
    for (let i = 5; i >= 0; i++) {
        // 如果当前面值的硬币还未找零
        if (coins[i]) {
            // 从小到大遍历所有可用于找零的硬币
            for (let j = 0; j <= i; j++) {
                // 本次找零金额
                let money = values[i] = values[j]
                // 当前支付的金额小于目标金额，开始选取硬币
                if (target >= money) {
                    // 当前硬币面值可以支付找零
                    if (coins[i] * money >= target) {
                        // 本次消耗的硬币数量
                        let amount = Math.floor(target - money)
                        res += amount * 2
                        // 查询顾客是否已有足够硬币
                        for (let k = 0; k < coins.length; k++) {
                            // 能找零且顾客手中仍有该面值硬币
                            if (coins.findIndex(money / values[k]) !== -1
&& coins[money - values[k]]) {
                                amount = Math.min(amount, coins[money /
values[k]])
                                // 扣除顾客手中硬币
                                coins[money / values[k]]--
                                // 不用找零，修正消耗数
                                res--
                            } else {
                                // 顾客手中硬币不够找零，直接跳过
                                break
                            }
                        }
                        // 减去本次找零金额，更新目标值
                        target -= money
                    }
                } else {
                    // 当前硬币面值不能完成找零，用掉所有当前面值硬币
                    res += coins[i]
```

```
                    coins[i] = 0
                    // 更新目标值
                    target -= coins[i] * values[i]
                }
            }
        }
    }
    /*
        如果最终没有完成找钱(target 不为 0)或没有可用方案(res === 0)
        输出-1 表示找零失败
        否则找零成功，输出结果
    */
    return (target || !res) ? -1 : res
}
```

使用给定数据进行测试：

```
coinChange([2, 4, 2, 2, 1, 0], 0.95) // 2
coinChange([2, 4, 2, 0, 1, 0], 0.55) // 3
coinChange([2, 4, 2, 0, 0, 0], 0.95) // -1 // 不足以支付
coinChange([0, 0, 0, 0, 1, 0], 0.55) // -1 // 不足以找零
```

本题贪心的思路在于，每次选择最大的硬币尝试找钱，并由此计算出一个单次的实际付款金额(分解为子问题)，并对子问题进行类似背包问题的求解，同时通过已有硬币和找零硬币的组合，等效增加了可用于找零的硬币面值，减少了找零所需的硬币数目，要注意每次需判断顾客是否有足够的该面额硬币，没有所需硬币时要打断尝试避免出现非法的找零操作。设硬币种类数有 N 种，每种硬币数量有 M 个，本题时间复杂度近似为 O(N ^ 3 * M)