贪心算法刷题路线

- 贪心算法是一种在每一步选择中都力图使局部最优解能够逐步达到全局最优解的算法。与其他算法不同的是，贪心算法在选择过程中不进行回溯，选择的过程往往基于某种局部最优策略。在一些特定的问题中，贪心算法可以通过逐步构建最优解来实现全局最优。
- 贪心算法的核心思想是通过一系列局部最优选择来构建全局最优解。

力扣

分发饼干

题目描述：

假设你是一位很棒的家长，想要给你的孩子们一些小饼干。但是，每个孩子最多只能给一块饼干。

对每个孩子 `i`，都有一个胃口值 `g[i]`，这是能让孩子们满足胃口的饼干的最小尺寸；并且每块饼干 `j`，都有一个尺寸 `s[j]`。如果 `s[j] >= g[i]`，我们可以将这个饼干 `j` 分配给孩子 `i`，这个孩子会得到满足。你的目标是**满足尽可能多**的孩子，并输出这个最大数值。

示例 1:

- **输入:** `g = [1,2,3]`, `s = [1,1]`
- **输出:** `1`
- **解释:** 你有三个孩子和两块小饼干，3 个孩子的胃口值分别是：1, 2, 3。虽然你有两块小饼干，由于它们的尺寸都是 1，你只能让胃口值是 1 的孩子满足。所以你应该输出 1。

示例 2:

- **输入:** `g = [1,2]`, `s = [1,2,3]`
- **输出:** `2`
- **解释:** 你有两个孩子和三块小饼干，2 个孩子的胃口值分别是 1, 2。你拥有的饼干数量和尺寸都足以让所有孩子满足。所以你应该输出 2。

提示：

- $1 \leq g.length \leq 3 * 10^4$
- $0 \leq s.length \leq 3 * 10^4$
- $1 \leq g[i], s[j] \leq 2^{31} - 1$

```cpp
#include <vector>
#include <algorithm>

class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        // 排序：孩子胃口从小到大，饼干尺寸从小到大
        sort(g.begin(), g.end());
        sort(s.begin(), s.end());

        int child = 0;    // 已满足的孩子数
        int cookie = 0;   // 当前尝试的饼干索引

        // 贪心匹配：用最小尺寸饼干满足最小胃口孩子
        while (child < g.size() && cookie < s.size()) {
```

```
            if (s[cookie] >= g[child]) {  // 当前饼干能满足当前孩子
                child++;  // 孩子被满足，转向下一个孩子
            }
            cookie++;  // 无论是否满足，当前饼干已使用（或太小弃用）
        }
        return child;  // 返回满足的孩子总数
    }
};
```

排序后从小到大匹配，避免大饼干被浪费在小胃口上

贪心策略：每次用最小可用饼干满足最小未满足孩子

时间复杂度：O(nlogn)

K 次取反后最大化的数组和

题目描述：

给你一个整数数组 `nums` 和一个整数 `k` ，按以下方法修改该数组：

- 选择某个下标 `i` 并将 `nums[i]` 替换为 `-nums[i]` 。

重复这个过程恰好 `k` 次。可以多次选择同一个下标 `i` 。

以这种方式修改数组后，返回数组**可能的最大和**。

示例 1:

- **输入:** `nums = [4,2,3], k = 1`
- **输出:** `5`
- **解释:** 选择下标 1， `nums` 变为 `[4,-2,3]` 。 (注：此处示例为原题描述，实际最大和应为选择最小正数取反，但示例 1 仅展示一次操作后的结果)

示例 2:

- **输入:** `nums = [3,-1,0,2], k = 3`
- **输出:** `6`
- **解释:** 选择下标 (1, 2, 2)， `nums` 变为 `[3,1,0,2]` 。

示例 3:

- **输入:** `nums = [2,-3,-1,5,-4], k = 2`
- **输出:** `13`
- **解释:** 选择下标 (1, 4)， `nums` 变为 `[2,3,-1,5,4]` 。

提示：

- $1 \leq nums.length \leq 10^4$
- $-100 \leq nums[i] \leq 100$
- $1 \leq k \leq 10^4$

```
#include <algorithm>

class Solution {
public:
```

```
    int largestSumAfterKNegations(vector<int>& nums, int k) {
        // 排序：负数在前，正数在后，便于处理
        sort(nums.begin(), nums.end());

        // 贪心：优先取反负数（将负数变正数，总和增加最多）
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] < 0 && k > 0) {  // 还有负数和操作次数
                nums[i] = -nums[i];  // 取反
                k--;  // 消耗一次操作
            }
        }

        // 剩余奇数次操作：取反当前最小数（损失最小）
        if (k % 2 == 1) {
            sort(nums.begin(), nums.end());  // 重新排序找最小值
            nums[0] = -nums[0];  // 取反最小数
        }

        // 计算最终总和
        int sum = 0;
        for (int i = 0; i < nums.size(); i++) {
            sum += nums[i];
        }

        return sum;
    }
};
```

贪心核心：优先取反最小的负数，收益最大

第一次排序：让负数集中在前面，便于优先处理

第二次排序：处理完负数后，重新找到当前数组的最小值

k 是偶数，结果不变

k 是奇数，结果反转

第一步：将所有负数变成正数

第二步：把最小的正数变成负数

时间复杂度：O(nlogn)，空间复杂度：O(1)

柠檬水找零

在柠檬水摊上，每一杯柠檬水的售价为 `5` 美元。顾客排队购买你的产品，（按账单 `bills` 支付的顺序）一次购买一杯。

每位顾客只买一杯柠檬水，然后向你付 `5` 美元、`10` 美元或 `20` 美元。你必须给每个顾客正确找零，也就是说净交易是每位顾客向你支付 `5` 美元。

注意，一开始你手头没有任何零钱。

给你一个整数数组 `bills` ，其中 `bills[i]` 是第 `i` 位顾客付的账。如果你能给每位顾客正确找零，返回 `true` ，否则返回 `false` 。

示例 1：

输入： `bills = [5,5,5,10,20]`
输出： `true`
解释： 前 3 位顾客那里，我们按顺序收取 3 张 5 美元的钞票。 第 4 位顾客那里，我们收取一张 10 美元的钞票，并返还 5 美元。 第 5 位顾客那里，我们找还一张 10 美元的钞票和一张 5 美元的钞票。 由于所有客户都得到了正确的找零，所以我们输出 `true`。

示例 2：

输入： `bills = [5,5,10,10,20]`
输出： `false`
解释： 前 2 位顾客那里，我们按顺序收取 2 张 5 美元的钞票。 对于接下来的 2 位顾客，我们收取一张 10 美元的钞票，然后返还 5 美元。 对于最后一位顾客，我们无法退回 15 美元，因为我们现在只有两张 10 美元的钞票。 由于不是每位顾客都得到了正确的找零，所以答案是 `false`。

提示：

- `1 <= bills.length <= 10^5`
- `bills[i]` 不是 `5` 就是 `10` 或是 `20`

```cpp
class Solution {
public:
    bool lemonadeChange(vector<int>& bills) {
        // 记录当前拥有的5美元和10美元数量（20美元不用于找零）
        int five = 0;
        int ten = 0;

        // 遍历每位顾客的支付
        for(int i = 0; i < bills.size(); i++){
            int bill = bills[i];

            if(bill == 5){
                // 收到5美元：无需找零，直接收下
                five++;
            } else if(bill == 10){
                // 收到10美元：需要找零5美元
                if(five > 0){
                    five--;  // 给出一张5美元
                    ten++;   // 收下10美元
                } else {
                    return false;  // 没有5美元零钱，无法找零
                }
            } else if(bill == 20){
                // 收到20美元：需要找零15美元
                // 贪心策略：优先使用10+5的组合（保留更多5美元）
                if(five > 0 && ten > 0){
                    five--;  // 给出一张5美元
                    ten--;   // 给出一张10美元
                } else if(five >= 3){
                    // 没有10美元，使用三张5美元
                    five -= 3;
                } else {
                    return false;  // 无法找零15美元
                }
                // 注意：20美元不参与找零，只收下不统计
            }
        }
        return true;  // 所有顾客都成功找零
    }
};
```

贪心策略：找零时优先使用10+5的组合，保留更多5美元

5美元更灵活（可找零10和20），10美元只能找零20

保留更多5美元可以提高后续找零成功率

顾客不会用20美元给其他顾客找零

时间复杂度：O(n)，空间复杂度：O(1)

买股票的最佳时机 II

你一个整数数组 `prices` ，其中 `prices[i]` 表示某支股票第 `i` 天的价格。

在每一天，你可以决定是否购买和/或出售股票。你在任何时候 **最多** 只能持有 **一股** 股票。然而，你可以在 **同一天** 多次买卖该股票，但要确保你持有的股票不超过一股。

返回 你能获得的 **最大** 利润 。

示例 1：

> **输入：** `prices = [7,1,5,3,6,4]`
> **输出：** `7`
> **解释：** 在第 2 天（股票价格 = 1）的时候买入，在第 3 天（股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5 - 1 = 4。 随后，在第 4 天（股票价格 = 3）的时候买入，在第 5 天（股票价格 = 6）的时候卖出，这笔交易所能获得利润 = 6 - 3 = 3。 最大总利润为 4 + 3 = 7 。

示例 2：

> **输入：** `prices = [1,2,3,4,5]`
> **输出：** `4`
> **解释：** 在第 1 天（股票价格 = 1）的时候买入，在第 5 天 （股票价格 = 5）的时候卖出，这笔交易所能获得利润 = 5 - 1 = 4。 最大总利润为 4 。

示例 3：

> **输入：** `prices = [7,6,4,3,1]`
> **输出：** `0`
> **解释：** 在这种情况下，交易无法获得正利润，所以不参与交易可以获得最大利润，最大利润为 0。

提示：

- `1 <= prices.length <= 3 * 10^4`
- `0 <= prices[i] <= 10^4`

```cpp
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int sum = 0;  // 累计利润

        // 遍历价格数组，从第2天开始（第1天无法卖出）
        for(int i = 1; i < prices.size(); i++){
            // 如果今天价格高于昨天，就完成一次交易
            if(prices[i] > prices[i-1]){
                sum += (prices[i] - prices[i-1]);  // 累加利润
```

```
            }
        }
        return sum;  // 返回最大利润
    }
};
```

如果想买，手里必须没股票

如果想卖，手里必须有股票。

可以无限次买卖。

假设股票价格是：`[1, 5, 9]`
长线做法：第一天 1 块买，第三天 9 块卖。利润就是9-1=8
短线做法：第一天 1 块买，第二天 5 块卖。赚了 $5-1=4$。第二天 5 块又买回来，第三天 9 块卖。赚了 $9-5=4$。总利润就是4+4=8
将长线变成短线，公式如下：

第1天买第3天卖 = (第2天卖+买) + (第3天卖)

$$(Price_3 - Price_1) = (Price_2 - Price_1) + (Price_3 - Price_2)$$

贪心策略：只要第二天价格高于前一天，就进行一次买卖

捕捉每一天的上涨波段

忽略下跌和横盘，只赚取正向波动

时间复杂度：O(n)，空间复杂度：O(1)

最大子数组和

给你一个整数数组 `nums` ，请你找出一个具有最大和的连续子数组（子数组最少包含一个元素），返回其最大和。

子数组 是数组中的一个连续部分。

示例 1：

输入： `nums = [-2,1,-3,4,-1,2,1,-5,4]`
输出： `6`
解释： 连续子数组 `[4,-1,2,1]` 的和最大，为 `6` 。

示例 2：

输入： `nums = [1]`
输出： `1`

示例 3：

输入： `nums = [5,4,-1,7,8]`
输出： `23`

提示：

- `1 <= nums.length <= 10^5`
- `-10^4 <= nums[i] <= 10^4`

```cpp
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        // 记录最大子数组和，初始化为第一个元素（处理全负数情况）
        int maxSum = nums[0];
        // 记录当前子数组和
        int currentSum = 0;

        for (int i = 0; i < nums.size(); i++) {
            int num = nums[i];
            // 将当前元素加入子数组
            currentSum += num;

            // 更新最大子数组和
            if (currentSum > maxSum) {
                maxSum = currentSum;
            }

            // 贪心策略：如果当前子数组和为负，重置为0
            // 因为负数会拖累后续子数组的和
            if (currentSum < 0) {
                currentSum = 0;
            }
        }

        return maxSum;
    }
};
```

> 卡丹算法

> 贪心策略：
> 当前子数组和为负时，果断放弃（重置为0）
> 因为负的子数组和只会减少后续子数组的和
> 从下一个元素重新开始构建子数组

> 遍历所有可能的子数组结尾位置

> 对于每个结尾位置，维护能达到的最大子数组和

> 通过舍弃负前缀，确保每个位置都是最优的

> 时间复杂度：O(n)，空间复杂度：O(1)

加油站

在一条环路上有 `n` 个加油站，其中第 `i` 个加油站有汽油 `gas[i]` 升。

你有一辆油箱容量无限的汽车，从第 `i` 个加油站开往第 `i+1` 个加油站需要消耗汽油 `cost[i]` 升。你从其中的一个加油站出发，开始时油箱为空。

给定两个整数数组 `gas` 和 `cost` ，如果你可以按顺序绕环路行驶一周，则返回出发时加油站的编号，否则返回 `-1` 。如果存在解，则 **保证** 它是 **唯一** 的。

示例 1:

> **输入:** `gas = [1,2,3,4,5], cost = [3,4,5,1,2]`
> **输出:** `3`
> **解释:** 从 3 号加油站(索引为 3 处)出发，可获得 4 升汽油。此时油箱有 = 0 + 4 = 4 升汽油 开往 4 号加油站，此时油箱有 4 - 1 + 5 = 8 升汽油 开往 0 号加油站，此时油箱有 8 - 2 + 1 = 7 升汽油 开往 1 号加油站，此时油箱有 7 - 3 + 2 = 6 升汽油 开往 2 号加油站，此时油箱有 6 - 4 + 3 = 5 升汽油 开往 3 号加油站，你需要消耗 5 升汽油，正好足够你返回到 3 号加油站。 因此，3 可为起始索引。

示例 2:

> **输入:** `gas = [2,3,4], cost = [3,4,3]`
> **输出:** `-1`
> **解释:** 你不能从 0 号或 1 号加油站出发，因为没有足够的汽油可以让你行驶到下一个加油站。 我们从 2 号加油站出发，可以获得 4 升汽油。 此时油箱有 = 0 + 4 = 4 升汽油 开往 0 号加油站，此时油箱有 4 - 3 + 2 = 3 升汽油 开往 1 号加油站，此时油箱有 3 - 3 + 3 = 3 升汽油 你无法返回 2 号加油站，因为返程需要消耗 4 升汽油，但是你的油箱只有 3 升汽油。 因此，无论怎样，你都不可能绕环路行驶一周。

提示:

- `gas.length == n`
- `cost.length == n`
- `1 <= n <= 10^5`
- `0 <= gas[i], cost[i] <= 10^4`

```cpp
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int currentSum = 0;    // 当前剩余油量（从start站开始的累计）
        int totalSum = 0;      // 总剩余油量（全程累计）
        int start = 0;         // 可能的起点加油站

        for (int i = 0; i < gas.size(); i++) {
            // 计算在i站的净收益（加油量 - 消耗量）
            int net = gas[i] - cost[i];

            // 更新当前剩余油量和总剩余油量
            currentSum += net;
            totalSum += net;

            // 贪心策略：如果当前剩余油量<0，说明从start出发无法到达i+1
            // 因此起点不可能是[start, i]之间的任何站，将起点设为i+1
            if (currentSum < 0) {
                start = i + 1;  // 从下一站重新尝试
                currentSum = 0;  // 重置当前剩余油量
            }
        }

        // 如果总剩余油量<0，说明所有站的油量总和小于消耗总和，不可能完成全程
        if (totalSum < 0) {
            return -1;
        }
        return start;  // 返回找到的起点
    }
};
```

贪心策略：当从start出发到i站时剩余油量<0
说明[start, i]区间内任何站作为起点都会失败
因为从start到i的累计油量已经为负，起点在中间只会更差
所以下一个可能的起点是i+1站

totalSum = 所有gas[i]-cost[i]的总和

如果totalSum<0，说明总油量不够，无论如何都无法完成全程

如果totalSum>=0，一定存在解（且找到的start就是正确起点）

时间复杂度：O(n)，空间复杂度：O(1)

跳跃游戏

给你一个非负整数数组 `nums` ，你最初位于数组的 **第一个下标** 。数组中的每个元素代表你在该位置可以跳跃的最大长度。

判断你是否能够到达最后一个下标，如果可以，返回 `true` ；否则，返回 `false` 。

示例 1：

输入： `nums = [2,3,1,1,4]`
输出： `true`
解释： 可以先跳 1 步，从下标 0 到达下标 1, 然后再从下标 1 跳 3 步到达最后一个下标。

示例 2：

输入： `nums = [3,2,1,0,4]`
输出： `false`
解释： 无论怎样，总会到达下标为 3 的位置。但该下标的最大跳跃长度是 0 ， 所以永远不可能到达最后一个下标。

提示：

- `1 <= nums.length <= 10^4`
- `0 <= nums[i] <= 10^5`

```cpp
class Solution {
public:
    bool canJump(vector<int>& nums) {
        // position 表示当前能到达的最远位置
        int position = 0;

        // 只遍历当前能到达的范围
        for (int i = 0; i <= position; i++) {
            // 更新能到达的最远位置
            // i + nums[i] 表示从位置i出发能到达的最远位置
            position = max(position, i + nums[i]);

            // 如果已经能到达最后一个位置，返回true
            if (position >= nums.size() - 1) {
                return true;
            }
        }

        // 遍历完所有可达位置后仍无法到达终点
        return false;
```

```
    }
};
```

核心思想：
不断更新当前能够到达的最远位置
只关注能否到达，不关注具体路径

如果某个位置不可达（i > position），则后续位置也都不可达

边界情况：
数组长度为1时，已经在终点，返回true
数组中有0：如果position>i，可以跳过0；如果position==i，则无法前进

当前位置i能否到达终点只取决于i+nums[i]的最大值

时间复杂度：O(n)，空间复杂度：O(1)

用最少数量的箭引爆气球

有一些球形气球贴在一堵用 XY 平面表示的墙面上。墙面上的气球记录在整数数组 `points` ，其中 `points[i] = [xstart, xend]` 表示水平直径在 `xstart` 和 `xend` 之间的气球。你不知道气球的确切 y 坐标。

一支弓箭可以沿着 x 轴从不同点 **完全垂直** 地射出。在坐标 `x` 处射出一支箭，若有一个气球的直径的开始和结束坐标为 `xstart` ，`xend` ，且满足 `xstart <= x <= xend` ，则该气球会被 **引爆** 。可以射出的弓箭的数量 **没有限制** 。 弓箭一旦被射出之后，可以无限地前进。

给你一个数组 `points` ，返回引爆所有气球所必须射出的 **最小** 弓箭数。

示例 1：

输入： points = [[10,16],[2,8],[1,6],[7,12]]
输出： 2
解释： 气球可以用2支箭来爆破： -在x = 6处射出箭，击破气球[2,8]和[1,6]。 -在x = 11处发射箭，击破气球[10,16]和[7,12]。

示例 2：

输入： points = [[1,2],[3,4],[5,6],[7,8]]
输出： 4
解释： 每个气球需要射出一支箭，总共需要4支箭。

示例 3：

输入： points = [[1,2],[2,3],[3,4],[4,5]]
输出： 2
解释： 气球可以用2支箭来爆破：
- 在x = 2处发射箭，击破气球[1,2]和[2,3]。
- 在x = 4处射出箭，击破气球[3,4]和[4,5]。

提示：

- `1 <= points.length <= 10^5`
- `points[i].length == 2`
- `-2^31 <= xstart < xend <= 2^31 - 1`

```cpp
class Solution {
public:
    // 自定义比较函数：按区间右端点（结束位置）升序排序
    static bool myCompare(const vector<int>& a, const vector<int>& b) {
        return a[1] < b[1];  // 比较区间的结束位置
    }

    int findMinArrowShots(vector<vector<int>>& points) {
        // 处理空数组的情况
        if (points.empty()) return 0;

        // 按区间结束位置排序，贪心策略：优先考虑结束早的区间
        sort(points.begin(), points.end(), myCompare);

        int arrows = 1;  // 至少需要一支箭
        int arrowPos = points[0][1];  // 第一支箭射在第一个区间的结束位置

        for (int i = 1; i < points.size(); i++) {
            // 如果当前区间的开始位置 > 箭的位置，说明需要新的箭
            if (points[i][0] > arrowPos) {
                arrows++;  // 增加一支箭
                arrowPos = points[i][1];  // 新箭射在当前区间的结束位置
            }
            // 如果当前区间的开始位置 <= 箭的位置，说明当前箭可以射中该区间
            // 不需要操作，继续使用当前箭
        }

        return arrows;  // 返回最少需要的箭数
    }
};
```

一箭射中尽可能多的重叠区间

按区间结束位置排序：优先处理结束早的区间

将箭射在第一个区间的结束位置，这样可以覆盖所有与该区间重叠的区间

示例：points = [[1,6],[2,8],[7,12],[10,16]]

- 排序后：[[1,6],[2,8],[7,12],[10,16]]
- 第一支箭在6，可以射中[1,6],[2,8]
- 第二支箭在12，可以射中[7,12],[10,16]
- 总共需要2支箭

边界情况：空数组：返回0

时间复杂度：O(nlogn)（排序主导），空间复杂度：O(1)

排序：把气球按 **右边界** 从小到大排序。
第一枪：瞄准第一个气球的右边界。
遍历：看后续的气球。
如果某个气球的 **左边界** 小于等于当前箭的位置，说明它和前一个气球重叠，这支箭能顺便把它带走（不用加箭）。
如果某个气球的 **左边界** 大于当前箭的位置，说明它完全在右边，之前的箭够不着。这时候必须 **加一支新箭**，并把新箭的位置更新为当前这个气球的右边界。

无重叠区间

给定一个区间的集合 `intervals` ，其中 `intervals[i] = [starti, endi]` 。返回 需要移除区间的最小数量，使剩余区间互不重叠。

注意 只在一点上接触的区间是 **不重叠** 的。例如 `[1, 2]` 和 `[2, 3]` 是不重叠的。

示例 1：

> **输入：** intervals = [[1,2],[2,3],[3,4],[1,3]]
> **输出：** 1
> **解释：** 移除 [1,3] 后，剩下的区间没有重叠。

示例 2：

> **输入：** intervals = [[1,2], [1,2], [1,2]]
> **输出：** 2
> **解释：** 你需要移除两个 [1,2] 来使剩下的区间没有重叠。

示例 3：

> **输入：** intervals = [[1,2], [2,3]]
> **输出：** 0
> **解释：** 你不需要移除任何区间，因为它们已经是无重叠的了。

提示：

- `1 <= intervals.length <= 10^5`
- `intervals[i].length == 2`
- `-5 * 10^4 <= starti < endi <= 5 * 10^4`

```cpp
class Solution {
public:
    // 自定义比较函数：按区间右端点（结束位置）升序排序
    static bool cmp(const vector<int>& a, const vector<int>& b) {
        return a[1] < b[1];  // 比较区间的结束位置
    }

    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        // 处理空数组情况
        if (intervals.empty()) return 0;

        // 按区间结束位置排序，贪心策略：优先保留结束早的区间
        sort(intervals.begin(), intervals.end(), cmp);

        int keep = 1;  // 至少保留一个区间（第一个区间）
        int current_end = intervals[0][1];  // 当前保留区间的结束位置

        for (int i = 1; i < intervals.size(); i++) {
            // 如果当前区间的开始位置 >= 当前保留区间的结束位置，则保留该区间
            if (intervals[i][0] >= current_end) {
                keep++;  // 增加保留区间数
                current_end = intervals[i][1];  // 更新当前保留区间的结束位置
            }
            // 否则：当前区间与保留区间重叠，跳过（相当于删除）
        }

        // 需要删除的区间数 = 总区间数 - 保留的区间数
```

```
        return intervals.size() - keep;
    }
};
```

核心思想：保留尽可能多的不重叠区间，等价于删除最少区间
按区间结束位置排序：结束早的区间为后续留下更多空间
总是选择结束最早的区间保留，然后跳过所有与它重叠的区间
这样可以为后面留下更多的可用空间

与上一题类似

示例：intervals = [[1,2],[2,3],[3,4],[1,3]]

- 排序后：[[1,2],[2,3],[1,3],[3,4]]
- 保留[1,2] → 当前结束时间=2
- [2,3]开始时间2>=2 → 保留，更新结束时间=3
- [1,3]开始时间1<3 → 跳过（删除）
- [3,4]开始时间3>=3 → 保留
- 保留3个区间，删除1个区间

时间复杂度：O(nlogn)（排序主导），空间复杂度：O(1)

根据身高重建队列

假设有打乱顺序的一群人站成一个队列，数组 `people` 表示队列中一些人的属性（不一定按顺序）。每个 `people[i] = [hi, ki]` 表示第 `i` 个人的身高为 `hi` ，前面 **正好** 有 `ki` 个身高大于或等于 `hi` 的人。

请你重新构造并返回输入数组 `people` 所表示的队列。返回的队列应该格式化为数组 `queue` ，其中 `queue[j] = [hj, kj]` 是队列中第 `j` 个人的属性（`queue[0]` 是排在队列前面的人）。

示例 1：

输入： people = [[7,0],[4,4],[7,1],[5,0],[6,1],[5,2]]
输出： [[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]]
解释： 编号为 0 的人身高为 5 ，没有身高更高或者相同的人排在他前面。
编号为 1 的人身高为 7 ，没有身高更高或者相同的人排在他前面。
编号为 2 的人身高为 5 ，有 2 个身高更高或者相同的人排在他前面，即编号为 0 和 1 的人。
编号为 3 的人身高为 6 ，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。
编号为 4 的人身高为 4 ，有 4 个身高更高或者相同的人排在他前面，即编号为 0、1、2、3 的人。
编号为 5 的人身高为 7 ，有 1 个身高更高或者相同的人排在他前面，即编号为 1 的人。
因此 [[5,0],[7,0],[5,2],[6,1],[4,4],[7,1]] 是重新构造后的队列。

示例 2：

输入： people = [[6,0],[5,0],[4,0],[3,2],[2,2],[1,4]]
输出： [[4,0],[5,0],[2,2],[3,2],[1,4],[6,0]]

提示：

- `1 <= people.length <= 2000`
- `0 <= hi <= 10^6`
- `0 <= ki < people.length`
- 题目数据确保队列可以被重建

```cpp
class Solution {
public:
    // 自定义排序规则：身高降序，k值升序
    static bool cpm(const vector<int>& a, const vector<int>& b) {
        if (a[0] == b[0]) {  // 身高相同
            return a[1] < b[1];  // k值小的排前面（前面人少）
        }
        return a[0] > b[0];  // 身高高的排前面
    }

    vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
        // 按身高降序排序，相同身高按k值升序排序
        sort(people.begin(), people.end(), cpm);

        vector<vector<int>> res;  // 重建后的队列

        // 按排序顺序插入队列
        for (int i = 0; i < people.size(); i++) {
            vector<int> person = people[i];
            int position = person[1];  // 当前人的k值

            // 将当前人插入到结果队列的position位置
            // 前面已有position个身高 >= 当前身高的人
            res.insert(res.begin() + position, person);
        }
        return res;
    }
};
```

题目理解：
假设这是一个班级在排队。 每个人有两个属性 [h, k]：

- h：这个人的身高。
- k：在这个人前面，必须有 `k` 个人的身高是 ≥ 他的。

需求：给这群人重新排队，使得每个人都能满足自己的 k 要求。

难点：

- 如果你把个子矮的放在前面，可能会影响个子高的人的 k 值（虽然矮的不算在高的 `k` 里，但矮的占了位置）。
- 如果你随便插队，后面的人一动，前面的人的 k 可能就不对了。

假设个子高的人根本看不见个子矮的人。
如果我们先把高个子的人排好，后面再插入矮个子的人，是不会影响已经排好的高个子的人的属性的

解题步骤
第一步：排序 既然矮个子不影响高个子，那我们先安排高个子。

- 身高 (h)：从高到低排。
- 如果身高相同：`k` 小的站前面（因为 `k` 小意味着前面人少，得排在更前面）。

第二步：插入按照排好的顺序，一个一个把人领出来，根据他的 k 值，把他插到队伍的第 k 个位置。
因为现在在队伍里的人，全都是身高 ≥ 他的人
既然大家都比他高，如果他要求前面有 `k` 个人，那他就老老实实站在索引 `k` 的位置（前面正好有 0 到 k-1 共 k 个人）。

总结：先给高个子排位置，再把矮个子插进去。矮个子插进去的时候，高个子虽然被挤到了后面，但因为矮个子不计入高个子的统计，所以高个子没有任何感觉

示例：people = [[7,0],[4,4],[7,1],[5,0],[6,1],[5,2]]

- 排序后：[[7,0],[7,1],[6,1],[5,0],[5,2],[4,4]]
- 插入过程：
- [7,0]

- [7,0],[7,1]
- 7,0],[6,1],[7,1]
- [5,0],[7,0],[6,1],[7,1]
- [5,0],[7,0],[5,2],[6,1],[7,1]
- 5,0],[7,0],[5,2],[6,1],[4,4],[7,1]

划分字母区间

给你一个字符串 `s` 。我们要把这个字符串划分为尽可能多的片段，同一字母最多出现在一个片段中。例如，字符串 "ababcc" 能够被分为 ["abab", "cc"]，但类似 ["aba", "bcc"] 或 ["ab", "ab", "cc"] 的划分是非法的。

注意，划分结果需要满足：将所有划分结果按顺序连接，得到的字符串仍然是 `s` 。

返回一个表示每个字符串片段的长度的列表。

示例 1：

> **输入：** s = "ababcbacadefegdehijhklij"
> **输出：** [9,7,8]
> **解释：** 划分结果为 "ababcbaca"、"defegde"、"hijhklij" 。 每个字母最多出现在一个片段中。 像 "ababcbacadefegde", "hijhklij" 这样的划分是错误的，因为划分的片段数较少。

示例 2：

> **输入：** s = "eccbbbbdec" **输出：** [10]

提示：

- `1 <= s.length <= 500`
- `s` 仅由小写英文字母组成

```cpp
class Solution {
public:
    vector<int> partitionLabels(string s) {
        // 记录每个字母最后出现的位置
        int lastOccur[26] = {0};

        // 第一次遍历：记录每个字符在字符串中最后出现的位置
        for (int i = 0; i < s.size(); i++) {
            lastOccur[s[i] - 'a'] = i;
        }

        vector<int> result;  // 存储每个片段的长度
        int left = 0;        // 当前片段的起始位置
        int right = 0;       // 当前片段的结束位置

        // 第二次遍历：根据最后出现位置划分片段
        for (int i = 0; i < s.size(); i++) {
            // 更新当前片段的结束位置：取当前字符最后出现位置的最大值
            right = max(right, lastOccur[s[i] - 'a']);

            // 如果当前位置等于当前片段的结束位置
            // 说明当前片段中的所有字符都不会出现在后面
            if (right == i) {
                // 计算当前片段的长度并加入结果
                result.push_back(right - left + 1);
                // 更新下一个片段的起始位置
                left = i + 1;
            }
```

```
        }

        return result;
    }
};
```

算法解析（贪心策略）：
核心思想：每个片段尽可能短，但要保证同一字母都在同一片段中

- 记录每个字母最后出现的位置
- 遍历时维护当前片段的结束位置，确保当前片段中所有字母的最后出现位置都在片段内
- 当遍历到当前片段结束位置时，就可以切割一个片段

每次切割的位置尽可能靠前，但又要保证同一字母不跨片段

通过维护当前片段的结束位置，确保了当前片段中所有字母都被包含

示例：s = "ababcbacadefegdehijhklij"

- 字母最后出现位置：a:8, b:5, c:7, d:14, e:15, f:11, g:13, h:19, i:22, j:23, k:20, l:21
- 遍历过程：
- i=0(a): right = max(0,8)=8
- i=1(b): right = max(8,5)=8
- i=2(a): right = max(8,8)=8
- ...
- i=8(a): right=8, i=8 -> 切割片段[0,8]，长度9
- 继续处理剩余部分...
- 最终结果：[9,7,8] 表示三个片段的长度