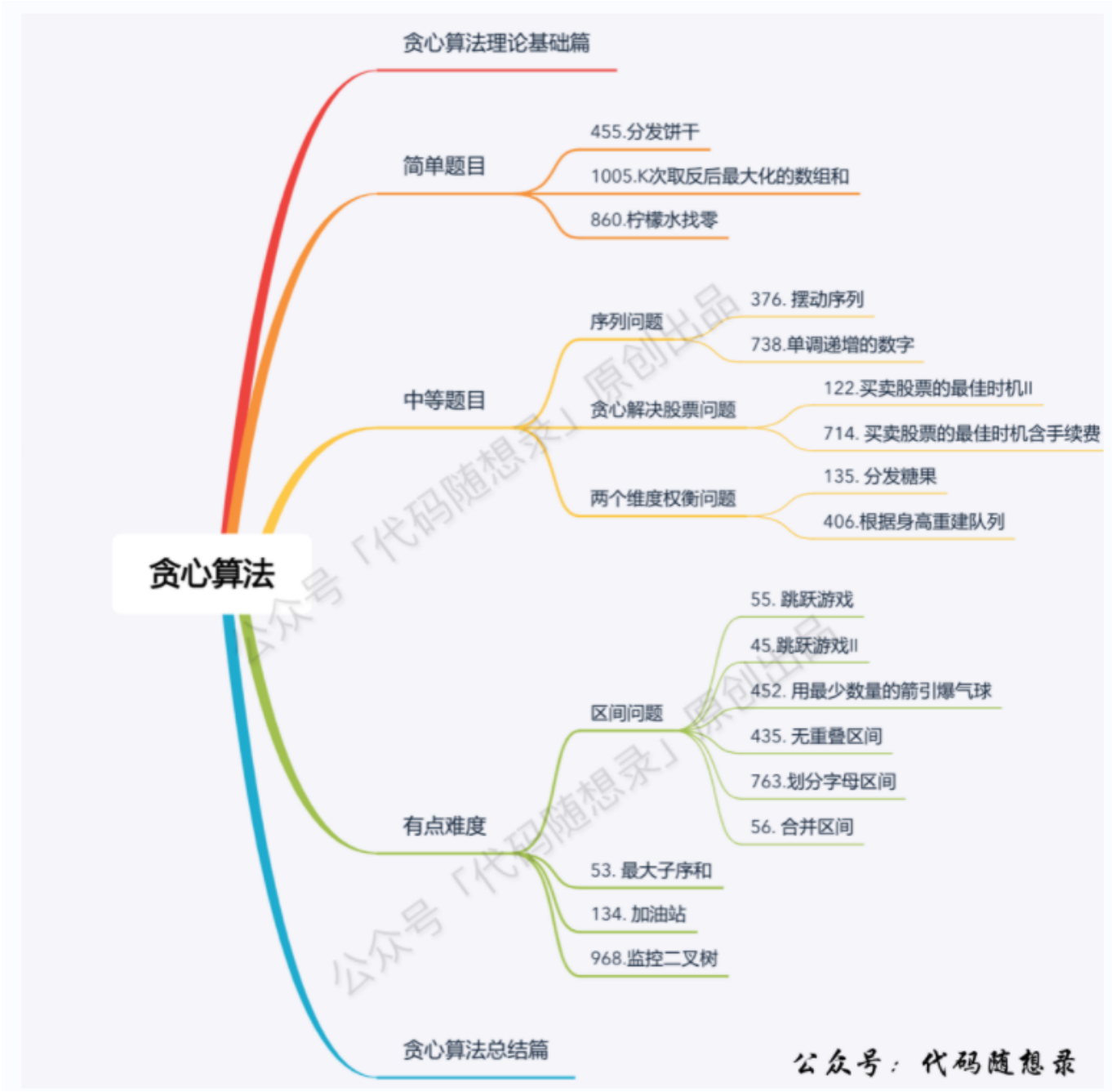


贪心算法解题大致过程

- 将问题分解为若干个子问题
- 找出适合的贪心策略
- 求解每一个子问题的最优解
- 将局部最优解堆叠成全局最优解

简单题目

455.分发饼干

贪心思想：局部最优是用大饼干先喂大胃口的孩子，充分利用饼干尺寸喂饱一个，全局最优就是喂饱尽可能多的小孩。

我当初做的时候是用小饼干喂小胃口孩子的思想。

1005.K次取反后最大化的数组和

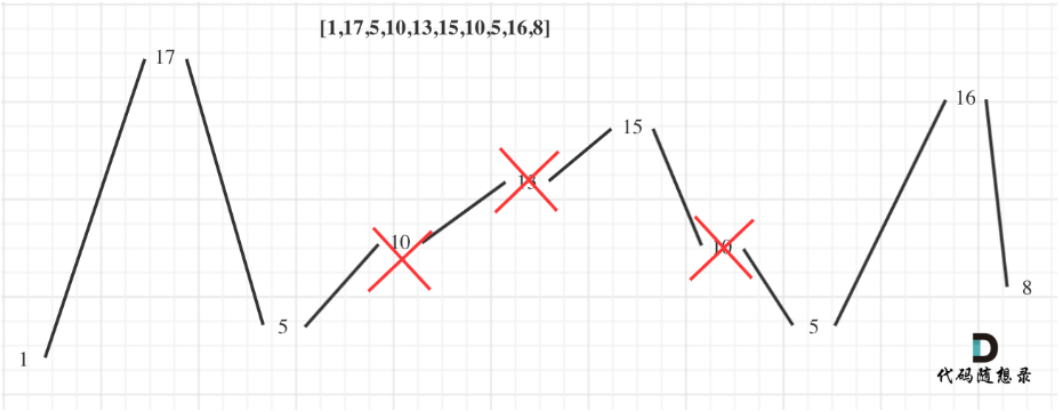
1. 将数组按照绝对值大小，从大到小排序
2. 从前向后遍历，遇到负数将其转变为正数，同时K--
3. 如果k还大于0，那么就重复转变数组中最小的元素，将k用完
4. 求和

中等题目

序列问题

376.摆动序列

用示例二来举例，如图所示：



局部最优：删除单调坡度上的节点（不包括单调坡度两端的节点），那么这个坡度就可以有两个局部峰值。

整体最优：整个序列有最多的局部峰值，从而达到最长摆动序列。

这是我们思考本题的一个大题思路，但本题要考虑三种情况：
情况一：上下坡中有平坡
情况二：数组首尾两端
情况三：单调坡中有平坡

738.单调递增的数字

例如：本题只要想清楚个例，例如**98**，一旦出现strNum[i - 1] > strNum[i]的情况（非单调递增），首先想让strNum[i - 1]减一，strNum[i]赋值9，这样这个整数就是**89**。就可以很自然想到对应的贪心解法了。想到了贪心，还要考虑遍历顺序，只有从后向前遍历才能重复利用上次比较的结果。最后代码实现的时候，也需要一些技巧，例如用一个flag来标记从哪里开始赋值9。

贪心解决股票问题

122.买卖股票的最佳时机-ii

局部最优：收集每天的正利润。 全局最优：求得最大利润

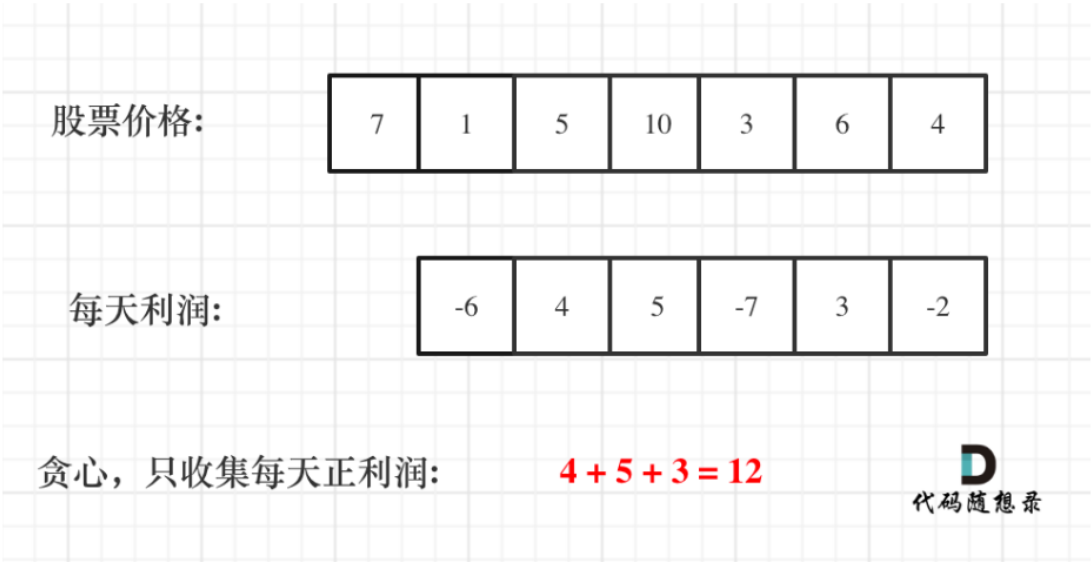
假如第 0 天买入，第 3 天卖出，那么利润为：prices[3] - prices[0]。

相当于(prices[3] - prices[2]) + (prices[2] - prices[1]) + (prices[1] - prices[0])。

此时就是把利润分解为每天为单位的维度，而不是从 0 天到第 3 天整体去考虑！

那么根据 prices 可以得到每天的利润序列：(prices[i] - prices[i - 1]).....(prices[1] - prices[0])。

如图：



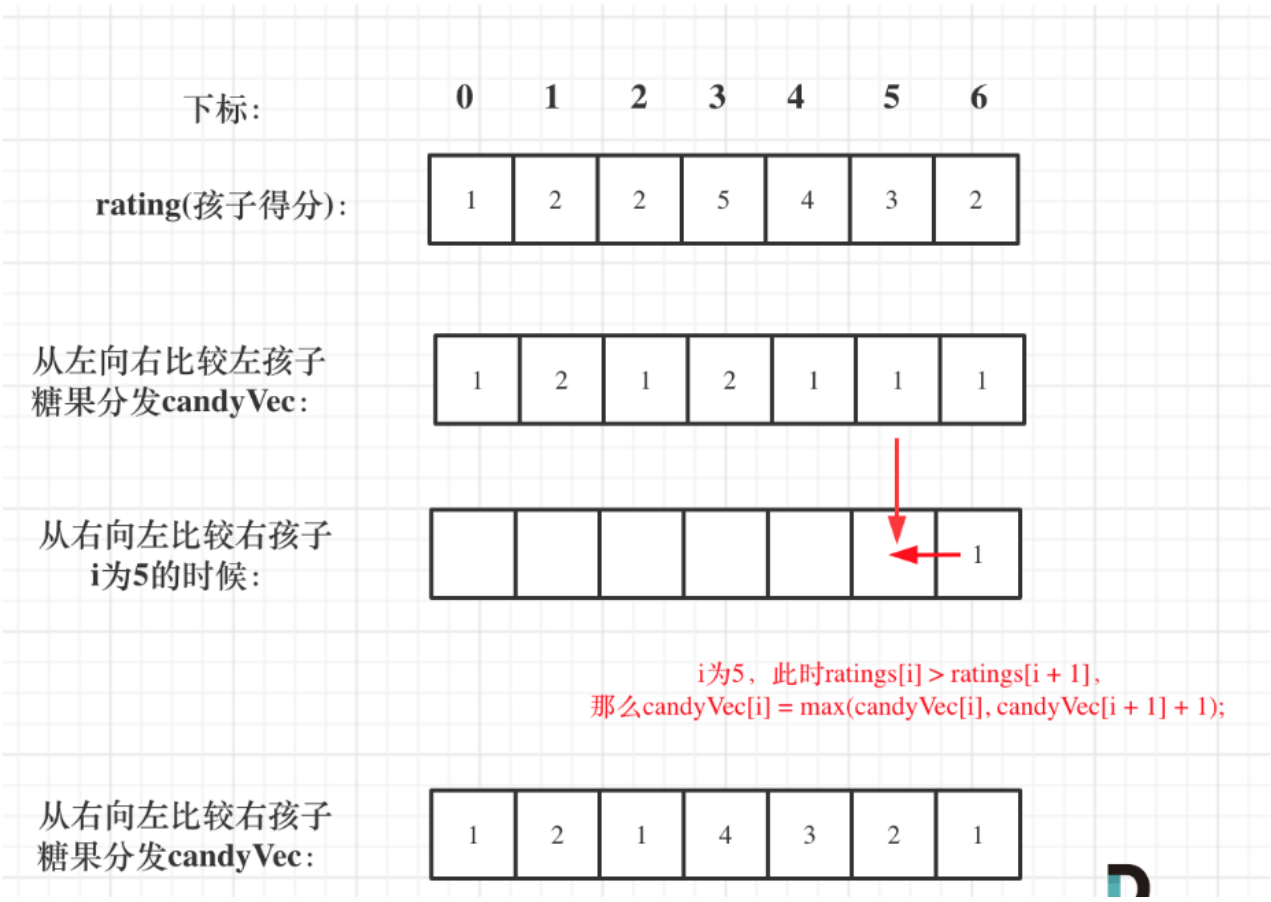
两个维度权衡问题

135.分发糖果（hard）

本题要从两个维度权衡问题，即从左至右来一次遍历，和从右到左来一次遍历。不能同时考虑一个点的左右情况，容易顾此失彼。
一次是从左到右遍历，只比较右边孩子评分比左边大的情况。
一次是从右到左遍历，只比较左边孩子评分比右边大的情况，
取`candy[i + 1] + 1`和`candy[i]`两者间大的值，这样才能保证得分比左右都高的点分发的糖果比左右都多。

所以就取candyVec[i + 1] + 1 和 candyVec[i] 最大的糖果数量，**candyVec[i]只有取最大的才能既保持对左边candyVec[i - 1]的糖果多，也比右边candyVec[i + 1]的糖果多。**

如图：



406.根据身高重建队列

此题的思想就是先根据身高大小，从大到小对vector中的元素进行一次排序。
然后根据k值，再插入到指定的位置上。
然后注意用list<vector<int>>代替vector<vector<int>>进行插入操作，能够减少时间复杂度。

回归本题，整个插入过程如下：

排序完的people： [[7,0], [7,1], [6,1], [5,0], [5,2]， [4,4]]

插入的过程：

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

此时就按照题目的要求完成了重新排列。

有点难度题目

区间问题

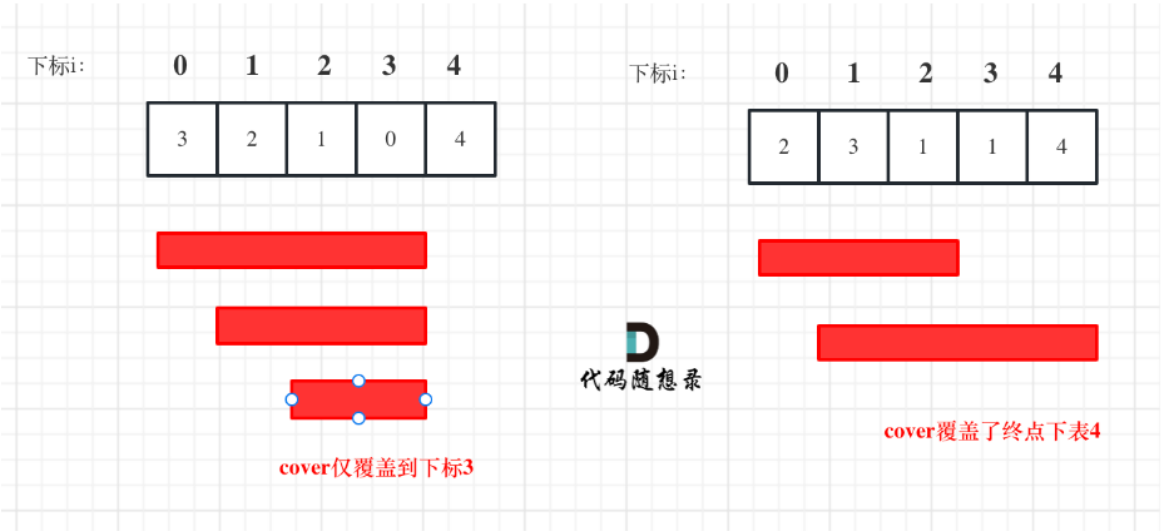
55.跳跃游戏

每次移动取最大跳跃步数（得到最大的覆盖范围），每移动一个单位，就更新最大覆盖范围。

贪心算法局部最优解：每次取最大跳跃步数（取最大覆盖范围），整体最优解：最后得到整体最大覆盖范围，看是否能到终点。

局部最优推出全局最优，找不出反例，试试贪心！

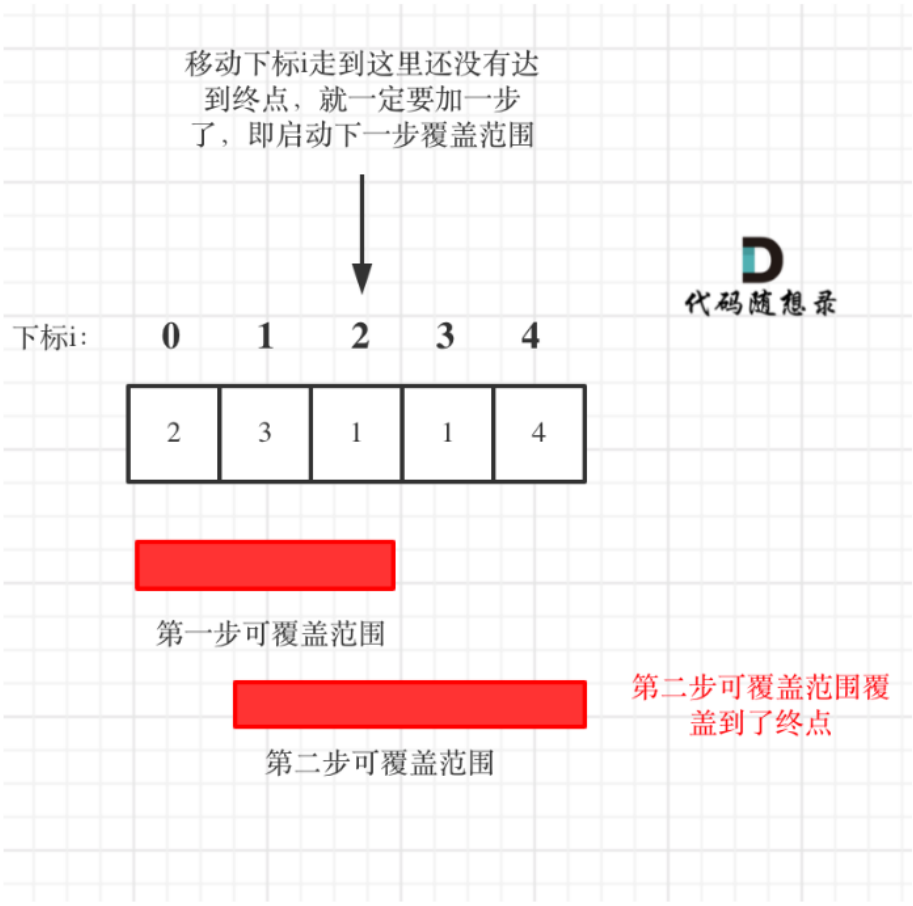
如图：



i 每次移动只能在 cover 的范围内移动，每移动一个元素，cover 得到该元素数值（新的覆盖范围）的补充，让 i 继续移动下去。

注意for循环是再cover的范围，不是num.size()的范围，只要覆盖的范围能到 最后一个元素的位置就返回true。所以for循环要在cover的范围内往后移动

45.跳跃游戏ii



图中覆盖范围的意义在于，只要红色的区域，最多两步一定可以到！（不用管具体怎么跳，反正一定可以跳到）

这里还是有个特殊情况需要考虑，当移动下标达到了当前覆盖的最远距离下标时

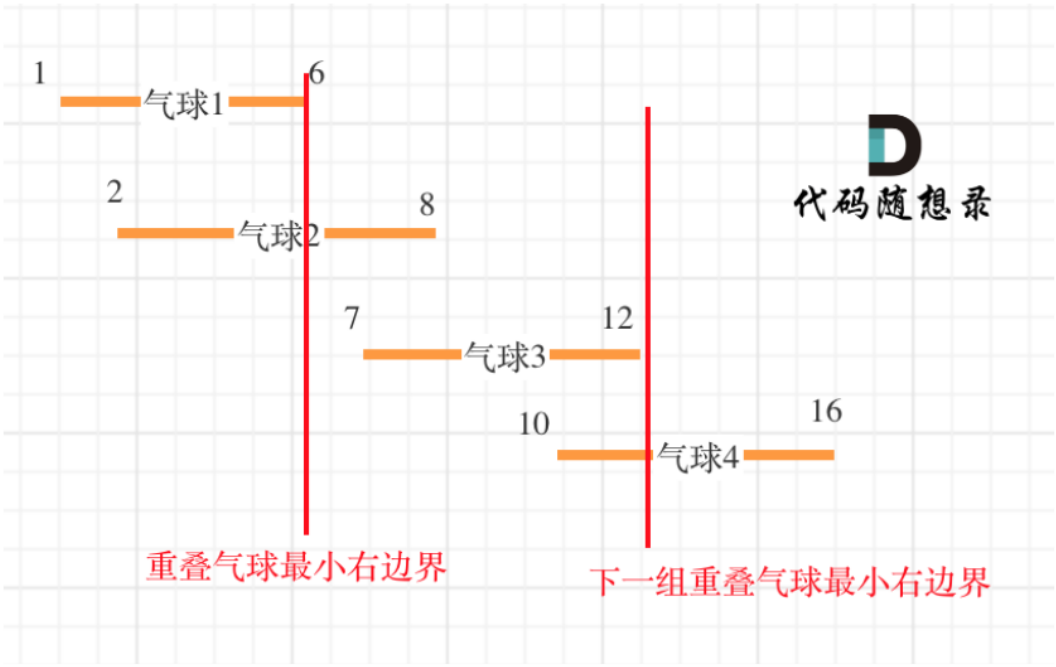
如果当前覆盖最远距离下标不是是集合终点，步数就加一，还需要继续走。
如果当前覆盖最远距离下标就是是集合终点，步数不用加一，因为不能再往后走了。

452.用最少数量的箭引爆气球

局部最优：当气球出现重叠，一起射，所用弓箭最少。全局最优：把所有气球射爆所用弓箭最少。

如果气球重叠了，重叠气球中右边边界的最小值 之前的区间一定需要一个弓箭。

以题目示例：[[10,16],[2,8],[1,6],[7,12]]为例，如图：（方便起见，已经排序）



可以看出首先第一组重叠气球，一定是需要一个箭，气球3，的左边界大于了 第一组重叠气球的最小右边界，所以再需要一支箭来射气球3了。

```
//气球i和气球i - 1不挨着
if(points[i][0] > points[i - 1][1]) {
    res++;
}
//气球i和气球i - 1挨着
else{
    points[i][1] = min(points[i][1], points[i - 1][1]);
}
```

**435.无重叠区间 **

方法1：
本题其实和（452.用最少数量的箭引爆气球）非常像，弓箭的数量就相当于是非交叉区间的数量，只要把弓箭那道题目代码里射爆气球的判断条件加个等号（认为[0，1][1，2]不是交叉区间，即为非交叉区间），然后用总区间数减去弓箭数量 就是要移除的区间数量了。

```
    int res = 1;
    sort(points.begin(), points.end(), cmp);
    for(int i = 1; i < points.size(); i++) {
        if(points[i][0] >= points[i - 1][1]) {
            res++; //记录重叠区间个数，即非交叉区间的数量
        }
        else{
            points[i][1] = min(points[i][1], points[i - 1][1]);
        }
```

```
    }
    return points.size() - res;
```

方法2：
从左边从小到大排序

```
static bool cmp(vector<int>& a, vector<int>& b) {
    //从小到大
    return a[0] < b[0];
}
int eraseOverlapIntervals(vector<vector<int>>& intervals) {
    sort(intervals.begin(), intervals.end(), cmp);
    int count = 0; //记录重叠区间个数
    int end = intervals[0][1]; //记录区间分割点
    for(int i = 1; i < intervals.size(); i++) {
        //无重叠
        if(intervals[i][0] >= end) {
            end = intervals[i][1];
        }
        //重叠
        else{
            count++;
            end = min(end, intervals[i][1]);
        }

    }
    return count;

}
```

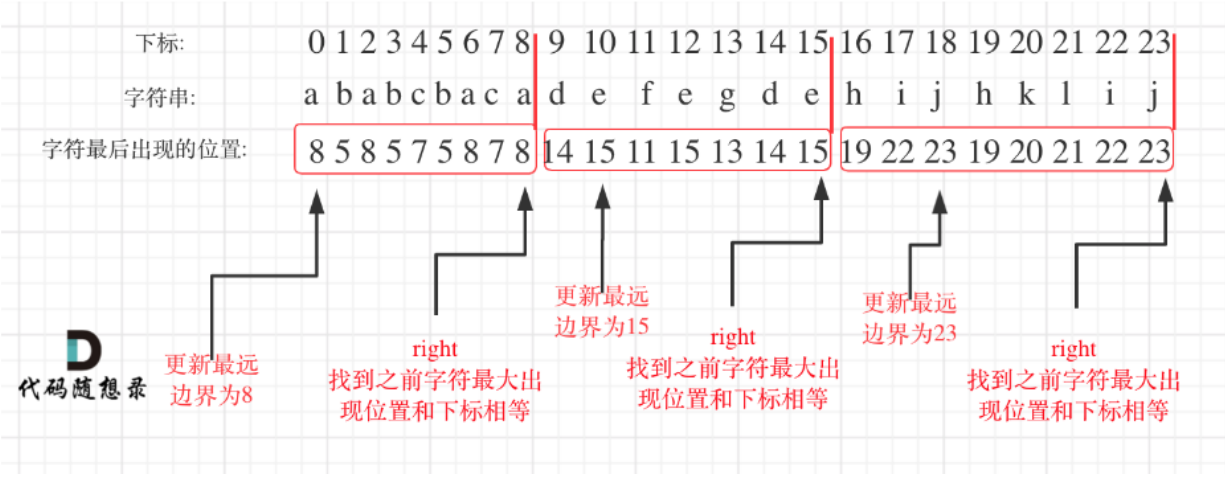
763.划分字母区间

在遍历的过程中相当于是要找每一个字母的边界，**如果找到之前遍历过的所有字母的最远边界，说明这个边界就是分割点了**。此时前面出现过所有字母，最远也就到这个边界了。

可以分为如下两步：

- 统计每一个字符最后出现的位置
- 从头遍历字符，并更新字符的最远出现下标，如果找到字符最远出现位置下标和当前下标相等了，则找到了分割点

如图：

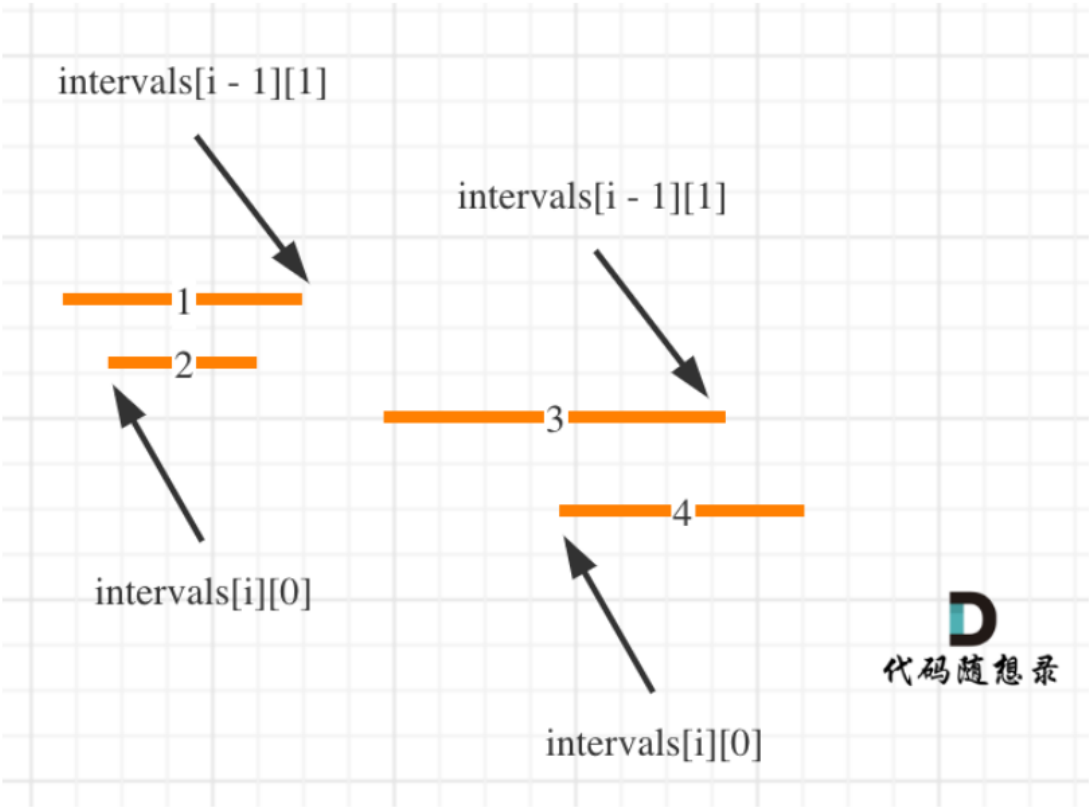


```cpp
class Solution {
public:
    vector<int> partitionLabels(string s) {
        int hash[26] = {0};
        //记录字符出现的最远位置
        for(int i = 0; i < s.size(); i++) {
            hash[s[i] - 'a'] = i;
        }

        int left = 0;
        int right = 0;
        vector<int> res;
        for(int i = 0; i < s.size(); i++) {
            //找到字符出现的最远边界
            right = max(right, hash[s[i] - 'a']);
            if(i == right) {
                res.push_back(right - left + 1);
                left = right + 1;
            }

        }
        return res;
    }
};
```

56.合并区间

按照左边界从小到大排序之后，如果 `intervals[i][0] <= intervals[i - 1][1]` 即intervals[i]的左边界 <= intervals[i - 1]的右边界，则一定有重叠。（本题相邻区间也算重贴，所以是<=）

这么说有点抽象，看图：（**注意图中区间都是按照左边界排序之后了**）



知道如何判断重复之后，剩下的就是合并了，如何去模拟合并区间呢？

其实就是用合并区间后左边界和右边界，作为一个新的区间，加入到result数组里就可以了。如果没有合

```cpp
class Solution {
public:
    static bool cmp(vector<int> a, vector<int> b) {
        return a[0] < b[0];
    }
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        vector<vector<int>> res;
        sort(intervals.begin(), intervals.end(), cmp);

        res.push_back(intervals[0]);

        for(int i = 1; i < intervals.size(); i++) {
            //有重叠直接在res上改右边界就可以
            if(intervals[i][0] <= res.back()[1]) {
                res.back()[1] = max(res.back()[1], intervals[i][1]);
            }
            else{
                //没有重叠直接加入res
                res.push_back(intervals[i]);
            }
        }
        return res;
    }
};
```

区间问题结束

53.最大子数组和

```cpp
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int res = INT32_MIN;
        int count = 0;
        for(int i = 0; i < nums.size(); i++) {
            count += nums[i];
            //相当于不断调整最大子数组和的结束位置
            if(res < count) {
                res = count;
            }
            //子数组累加和为负数了 重新确定起点 即i + 1
            if(count < 0) {
                count = 0;
            }
        }
        return res;
    }
};
```

134.加油站

此题暴力法容易超时，贪心的想法很巧妙

贪心算法（方法一）

直接从全局进行贪心选择，情况如下：

- 情况一：如果gas的总和小于cost总和，那么无论从哪里出发，一定是跑不了一圈的
- 情况二：rest[i] = gas[i]-cost[i]为一天剩下的油，i从0开始计算累加到最后一站，如果累加没有出现负数，说明从0出发，油就没有断过，那么0就是起点。
- 情况三：如果累加的最小值是负数，汽车就要从非0节点出发，从后向前，看哪个节点能把这个负数填平，能把这个负数填平的节点就是出发节点。

贪心算法（方法二）

可以换一个思路，首先如果总油量减去总消耗大于等于零那么一定可以跑完一圈，说明 各个站点的加油站 剩油量rest[i]相加一定是大于等于零的。

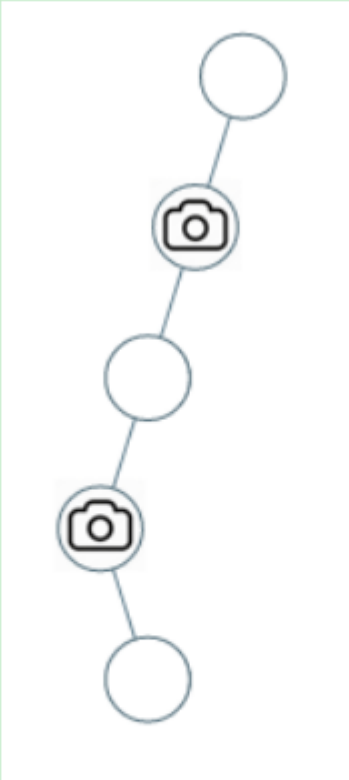
每个加油站的剩余量rest[i]为gas[i] - cost[i]。

i从0开始累加rest[i]，和记为curSum，一旦curSum小于零，说明[0, i]区间都不能作为起始位置，因为这个区间选择任何一个位置作为起点，到i这里都会断油，那么起始位置从i+1算起，再从0计算curSum。

```
int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
    int cursum = 0;
    int sum = 0;
    int start = 0;
    for(int i = 0; i < gas.size(); i++) {
        cursum += gas[i] - cost[i];
        sum += gas[i] - cost[i];
        //当前油量断油了 应该从下一个位置出发，并重新统计cursum
        if(cursum < 0) {
            start = i + 1;
            cursum = 0;
        }
    }
    // sum += gas[i] - cost[i]小于0 说明油量不够 不能走一圈
    if(sum < 0) {
        return -1;
    }
    return start;
}
```

968.监控二叉树（hard太难了）

示例 2：



输入：[0,0,null,0,null,0,null,null,0]
输出：2
解释：需要至少两个摄像头来监视树的所有节点。 上图显示了摄像头放置的有效位置之一。

摄像头可以覆盖上中下三层，如果把摄像头放在叶子节点，就浪费了一层的覆盖。所以把摄像头放在**叶子节点的父节点**，才能充分利用摄像头的覆盖面积。

大体思路就是从下到上，先给叶子节点父节点放个摄像头，然后隔两个节点放一个摄像头，直到二叉树头节点。

此时这道题目还有两个难点：

- 二叉树的遍历
- 如何隔两个节点放一个摄像头

二叉树的遍历选择后序遍历。

如何隔两个节点放一个摄像头：

每个节点有几种状态用以下数字表示：

0.该节点无覆盖

1.本节点有摄像头

2.本节点有覆盖\

```
class Solution {
public:
```

```
    struct TreeNode{
        int val;
        TreeNode* left;
        TreeNode* right;
        TreeNode() : val(0), left(nullptr), right(nullptr) {}
        TreeNode(int x):val(x),left(nullptr),right(nullptr){}
    };
    /*
    0.无覆盖
    1.安装摄像头
    2.有覆盖
    */
    int res = 0;
    int traversal(TreeNode* root) {
        /*
        为了让摄像头数量最少，尽量让叶子节点的父节点安装摄像头，
        这样才能摄像头的数量最少，所以空节点的状态只能是有覆盖
        这样就可以在叶子节点的父节点放摄像头了。
        */
        if(root == nullptr) return 2;
W
        int left = traversal(root->left);
        int right = traversal(root->right);
        //1.左右节点都有覆盖，那么此时中间节点就应该是无覆盖的状态
        if(left == 2 && right == 2) {
            return 0;
        }

        //2.左右节点至少有一个无覆盖的情况 则中间节点（父节点）应该
        //安装摄像头
        if(left == 0 || right == 0) {
            res++;
            return 1;
        }

        //3.左右节点至少有一个摄像头，那么其父节点应该是2（覆盖状态）
        //其它情况前段代码均已覆盖
        if(left == 1 || right == 1) {
            return 2;
        }


        //这个return -1 逻辑不会走到这里
        return -1;

    }
    int minCameraCover(TreeNode* root) {
        res = 0;
        //4.头节点无覆盖
        if(traversal(root) == 0) {
            res++;
        }
        return res;
```

```
        }
    };
```