贪心算法

分发饼干

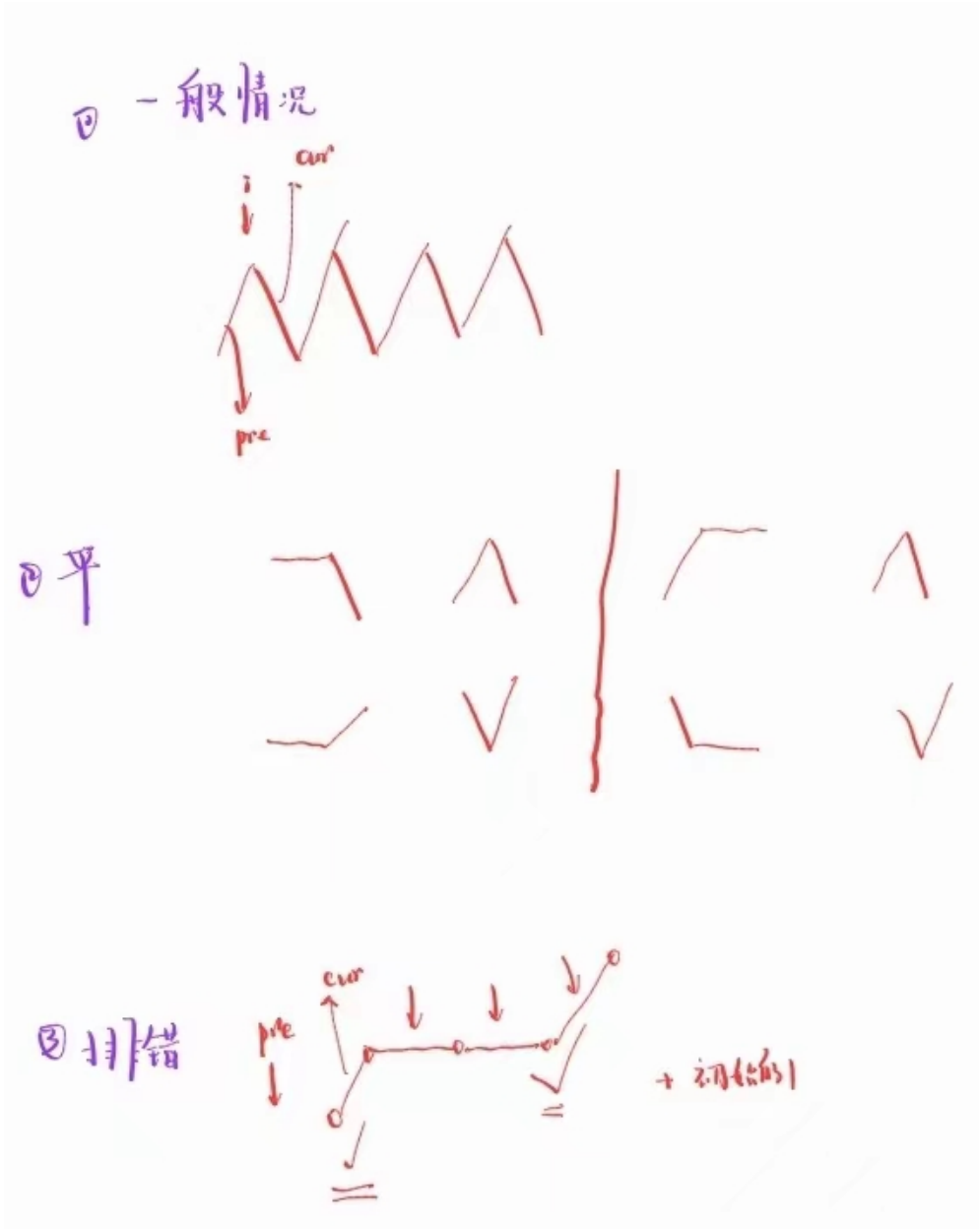
算法：先排序，从后面开始遍历，当满足条件的时候ret+1

C++

```
class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        int ret = 0;
        sort(g.begin(), g.end());
        sort(s.begin(), s.end());

        for (int i = g.size() - 1,j = s.size() - 1;i >= 0&&j >= 0;) {//这里要使用&&，而不是i>=0,j>=0
            if (s[j] >= g[i]) {//先让大孩子满足，再让小孩子满足
                i--;
                j--;
                ret++;
            }
            else {
                i--;
            }
        }

        return ret;
    }
};
```

摆动序列

pre_directioin记录之前的坡向，cur记录当前的差值




```
#include <vector>
#include <algorithm>
using namespace std;

class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {

        if (nums.size() <= 1) {
            return nums.size();
        }
        else if (nums.size() == 2) {
            if (nums[0] == nums[1]) {
```

```
                return 1;
            }
            else {
                return 2;
            }
        }

        int ret = 1;
        int pre_direction = 0;
        int cur = 0;

        for (int i = 0;i < nums.size() - 1;i++) {
            cur = nums[i + 1] - nums[i];
            if ((pre_direction >= 0 && cur < 0) || (pre_direction <= 0 && cur > 0))
{
                ret++;
                pre_direction = cur;
            }
        }

        return ret;
    }
};
```

最大子序和

算法思路：

1.如果sum的值小于0，就抛弃调用sum，并且让sum初始化为0

2.如果ret<sum，说明要更新ret

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        int Max = INT_MIN;
        int sum = 0;
        for (auto it : nums) {

            sum += it;
            Max = max(sum, Max);
            if (sum < 0) {//如果sum小于0，说明会拖累后面的数,直接让sum初始化
                sum = 0;
            }//贪心的地方
        }
```

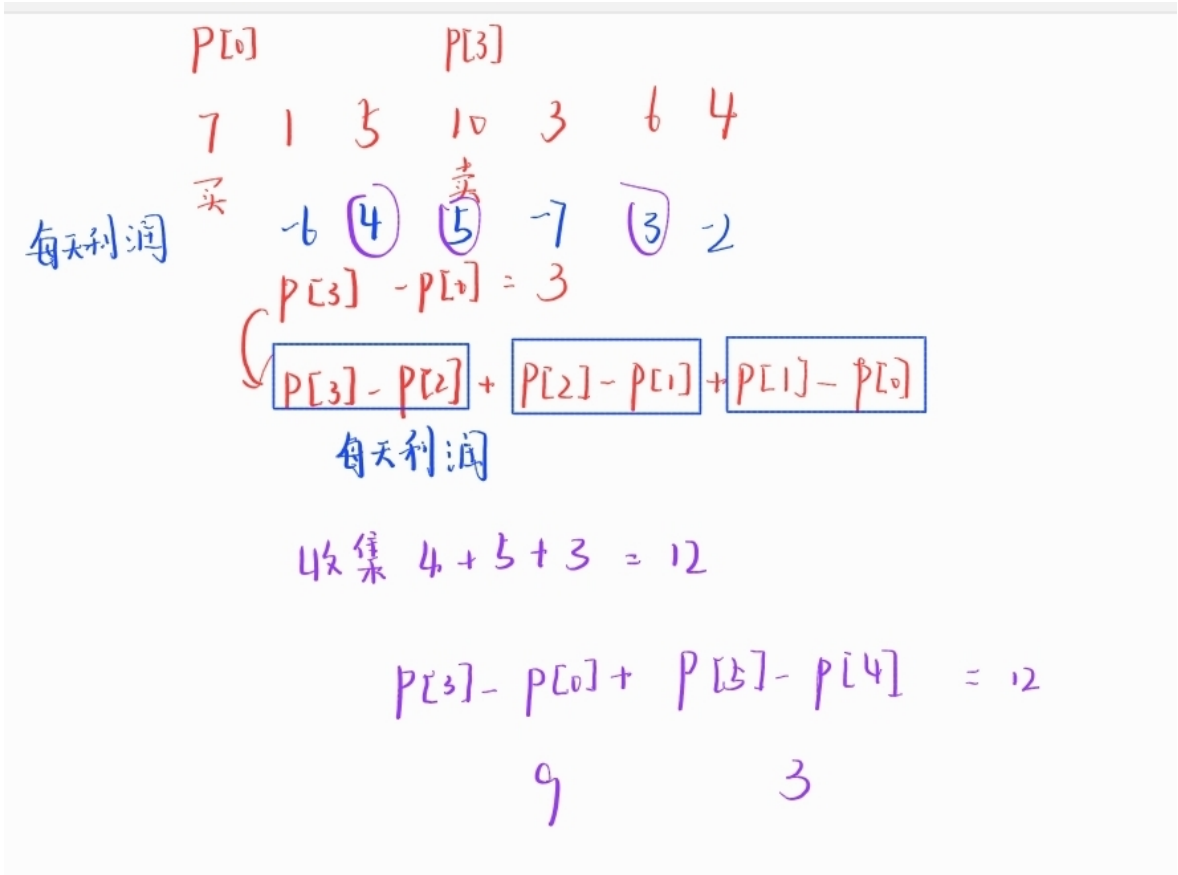
```
        return Max;
    }
};
```

买卖股票的时间

算法思路：

任意两天的差值可以看作每两天的买入以及卖出，当每只收集每两天为正数的情况，即可以得到最优解

得到一个每天利润数组，然后收集期中的正数




```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int ret = 0;
        for (int i = 0;i < prices.size() - 1;i++) {
            if (prices[i + 1] - prices[i]>0) {
                ret += prices[i + 1] - prices[i];
            }
        }
```

```
        return ret;
    }
};
```

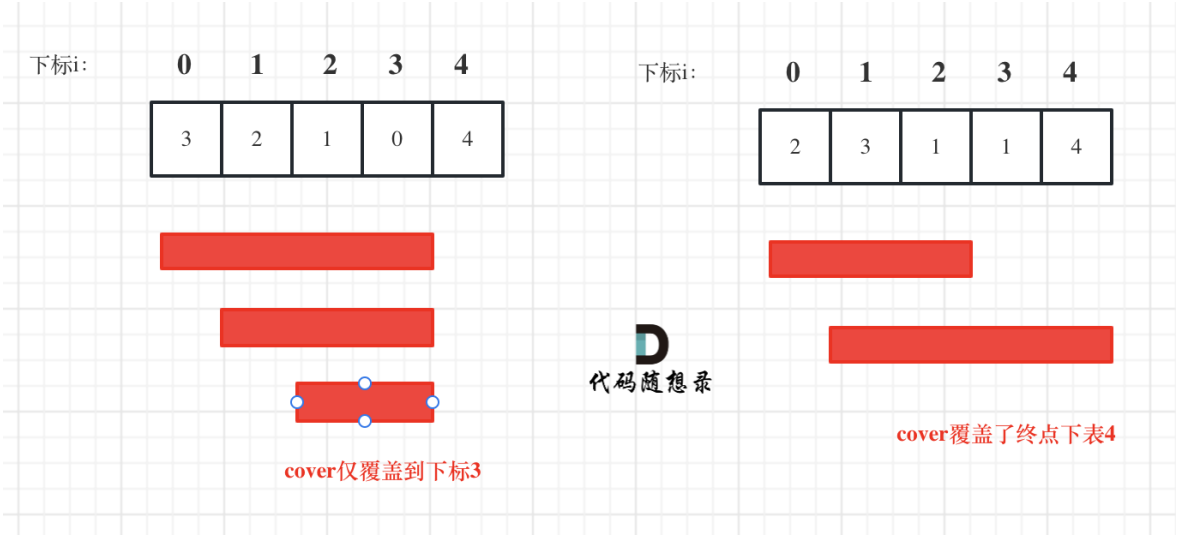
跳跃游戏

判断是否能够跳到终点

算法思路：

确定跳跃的最右端的范围。当可以跳跃的最右端的范围大于等于nums.size()-1的时候，则可以跳跃到末尾。

难点是更新RightRange值以及for循环，贪心的思想体现在那个max求最大右边界的过程上.



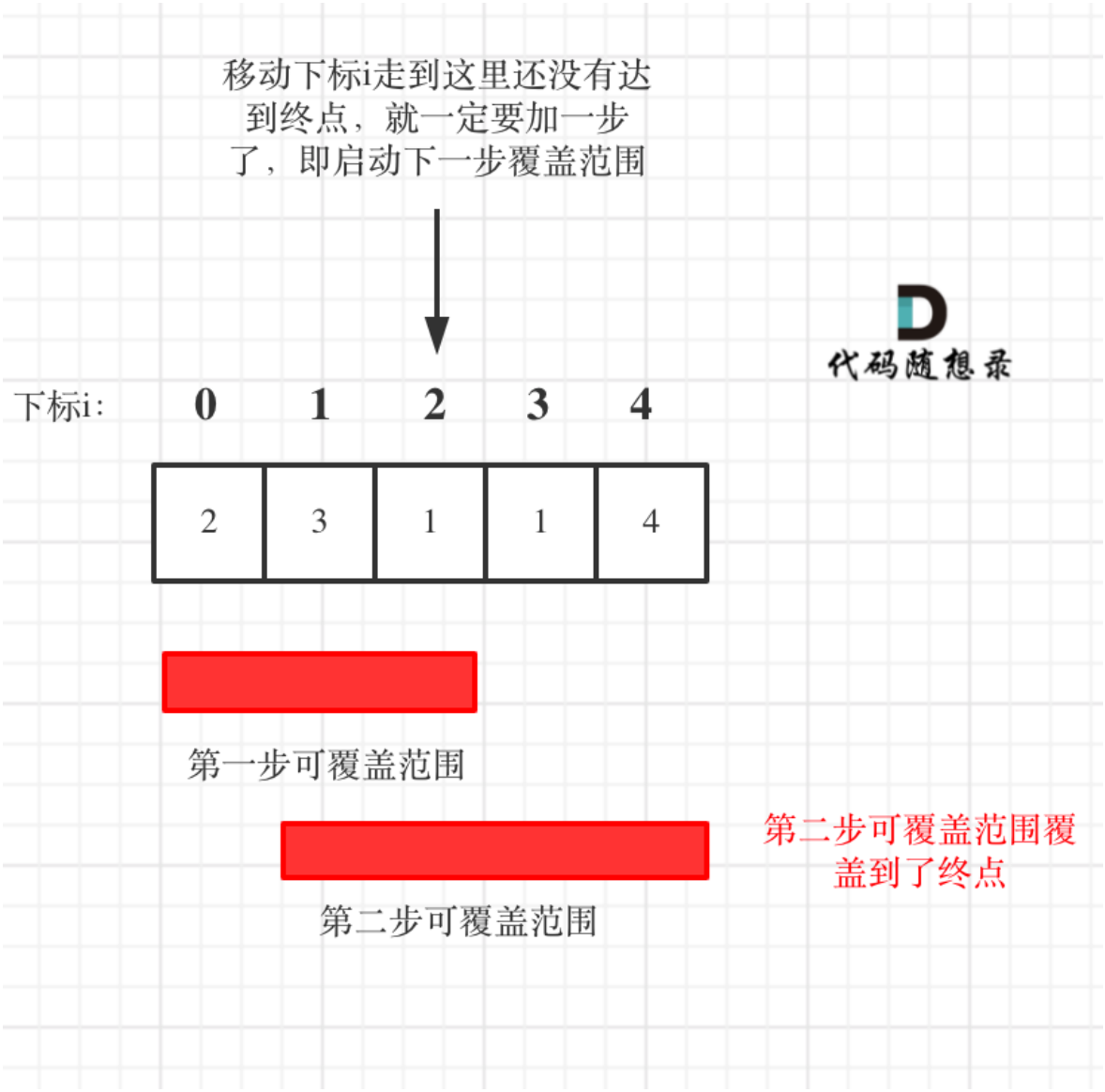

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        if (nums.size() == 1) {
            return true;
        }

        int RightRange = 0;
        for (int i = 0;i <= RightRange;i++) {
            RightRange = max(nums[i] + i, RightRange);
            if (RightRange >= nums.size() - 1) {
                return true;
            }
        }
```

```
        return false;
    }
};
```

跳到终点的最小步数

算法思路：记录当前的最大范围以及下一步的最大范围

如果不能覆盖，则将cur更新为next，并且让ret++。以此来模拟跳跃的情况




```
class Solution {
public:
    int jump(vector<int>& nums) {

        if (nums.size() == 1) {
```

```
            return 1;
        }

        int cur = 0;//当前可以跳跃的最远范围
        int next = 0;//下一步可以跳跃的最远范围
        int ret = 0;

        for (int i = 0;i < nums.size();i++) {
            next = max(nums[i] + i, next);

            if (i == cur) {
                if (cur != nums.size() - 1) {
                    ret++;
                    cur = next;
                    if (cur >= nums.size() - 1) {//更新范围后覆盖到终点
                        break;
                    }
                }
                else {//直接覆盖终点
                    break;
                }
            }
        }

        return ret;
    }
};
```

K次取反后最大的数组和

取小于等于k次

思路：首先按照绝对值从小到大排序，然后让k发挥作用。当不满足k > 0 && right >= 0的时候终止。

```
class Solution {
private:
    bool compare(int a, int b) {
        return abs(a) < abs(b);
    }
public:
    int largestSumAfterKNegations(vector<int>& nums, int k) {
        sort(nums.begin(), nums.end(),compare);

        int right = nums.size() - 1;
```

```
        while (k > 0 && right >= 0) {
            if (nums[right] < 0) {
                nums[right] = abs(nums[right]);
                k--;
                right--;
            }
            else {
                right--;
            }
        }

        int ret = 0;
        for (int i = 0;i < nums.size();i++) {
            if (nums[i] > 0) {
                ret += nums[i];
            }
        }

        return ret;
    }
};
```

只能取k次

如果全部都变成正数后，k还没有被消耗完，则让最小的正数反复变号消掉k

```
class Solution {//取等于k时
private:
    bool compare(int a, int b) {
        return abs(a) < abs(b);
    }
public:
    int largestSumAfterKNegations(vector<int>& nums, int k) {
        sort(nums.begin(), nums.end(), compare);

        int right = nums.size() - 1;
        while (k > 0 && right >= 0) {
            if (nums[right] < 0) {
                nums[right] = abs(nums[right]);
                k--;
                right--;
            }
            else {
                right--;
            }
        }
```

```
        if (k > 0) {//让最小的数反复变号消掉k
            while (k--) {
                nums[0] *= -1;
            }
        }

        int ret = 0;
        for (int i = 0;i < nums.size();i++) {
            ret += nums[i];
        }

        return ret;
    }
};
```

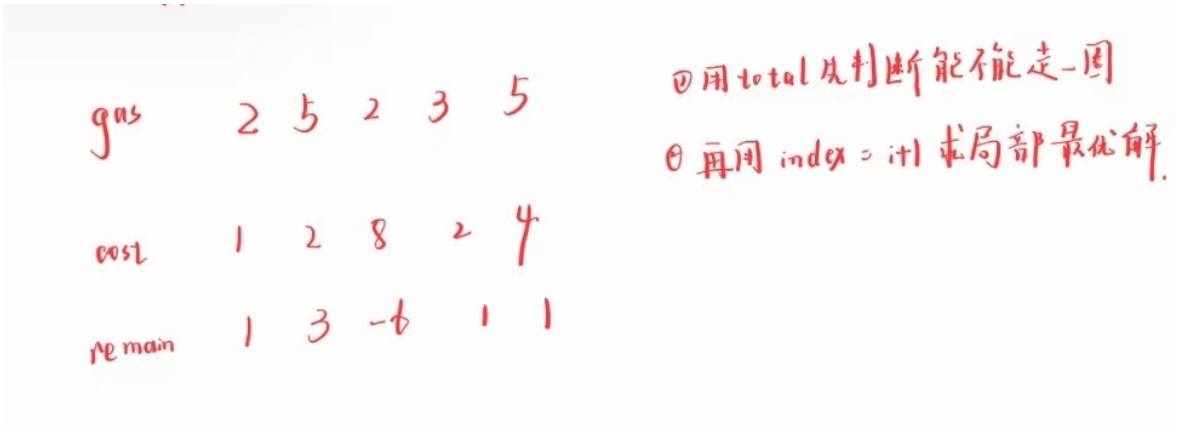
加油站

算法思路：

只能从前往后开车，循环数组

gas[i]-cost[i]得到正数，说明能够为接下来补充石油，小于0说明会亏欠石油

先补充石油，再消耗石油，贪心贪在此处




```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {

        int cur = 0;
        int total = 0;
        int index = 0;
```

```
        for (int i = 0;i < gas.size();i++) {
            cur += gas[i] - cost[i];
            total += gas[i] - cost[i];
            if (cur < 0) {
                index = i + 1;
                cur = 0;//初始化cur
            }
        }

        if (total < 0) {//total是用来判断是否能走完，如果total小于0，说明补充上的油小于消耗的油，直接返回-1
            return -1;
        }
        else {
            return index;
        }
    }
};
```

分发糖果

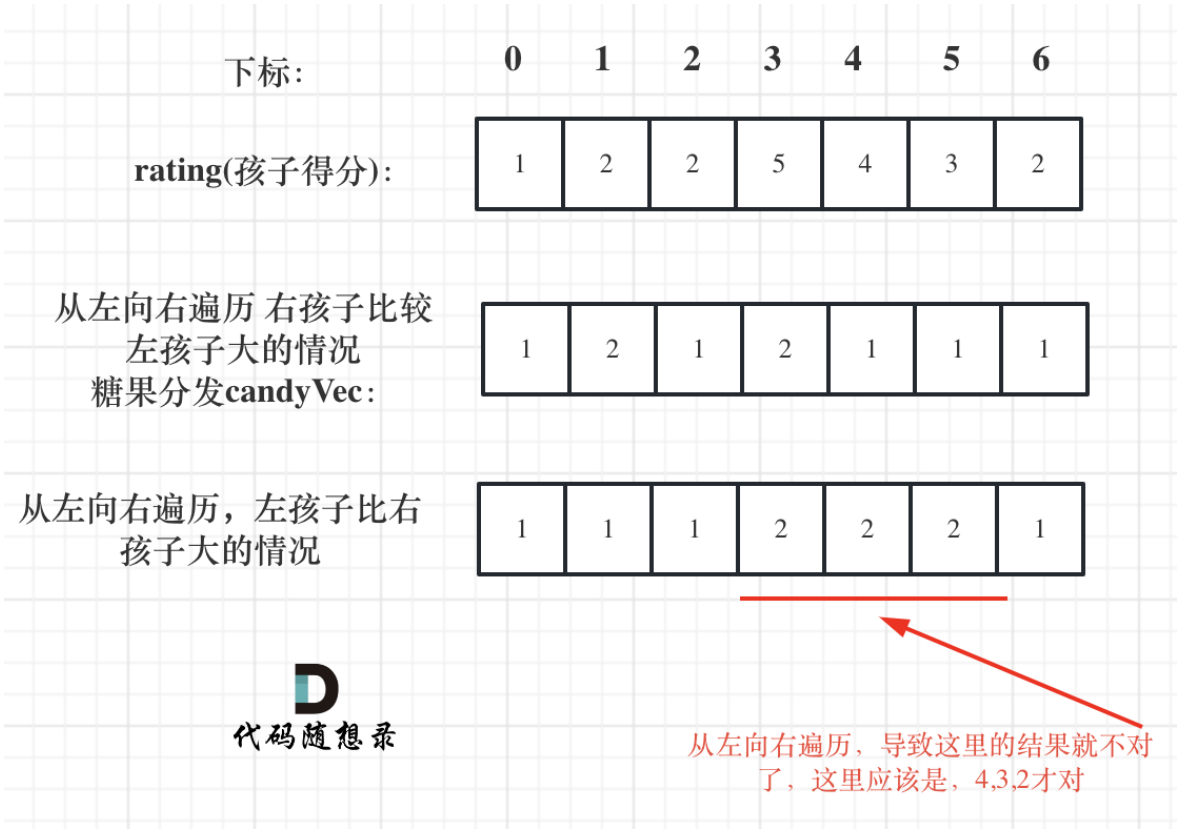
算法思路：

这种两边都要比较的题目，先比较一边，让其一边最优解后再比较另外一边。

首先每一个人都分发一个糖果

从左向右遍历，如果大于则加1；从右向左遍历，如果大于则加1。取两者中最大数

这样恰好刚刚能够符合最基本的条件，而且最开始都只发了一个糖果。贪心贪在此处




```
class Solution {
public:
    int candy(vector<int>& ratings) {

        vector<int> candy(ratings.size(), 1);//初始化为1，让每一个人至少拿到一个糖果
        candy[0] = 1;//第一个初始化为0

        for (int i = 1;i < ratings.size();i++) {//从左向右遍历，如果大于则加1
            if (ratings[i] > ratings[i - 1]) {
                candy[i] = candy[i - 1] + 1;
            }
        }

        for (int i = ratings.size() - 2;i >= 0;i--) {//从右向左遍历，如果大于则加1
            if (ratings[i] > ratings[i + 1]) {
                candy[i] = max(candy[i + 1] + 1, candy[i]);//此处取了最大值
            }
        }

        int sum = 0;
        for (auto it : candy) {
            sum += it;
        }
```

```
        return sum;
    }
};
```

找零的策略

算法思路：

总共分为3种情况。

贪心贪在当it等于20的时候，先把10与5组合在一起花出去，实在不行才找3张5块。贪心贪在此处。

```
class Solution {
public:
    bool lemonadeChange(vector<int>& bills) {
        int five = 0, ten = 0, twenty = 0;
        for (auto it : bills) {
            if (it == 5) {
                five++;
            }
            else if (it == 10) {
                if (five >= 1) {
                    ten++;
                    five--;
                }
                else {
                    return false;
                }
            }
            else if (it == 20) {
                if (ten >= 1 && five >= 1) {//先把10花出去，留着5好找零
                    twenty++;
                    ten--;
                    five--;
                }
                else if (five >= 3) {
                    five -= 3;
                    twenty++;
                }
                else {
                    return false;
                }
            }
        }
```

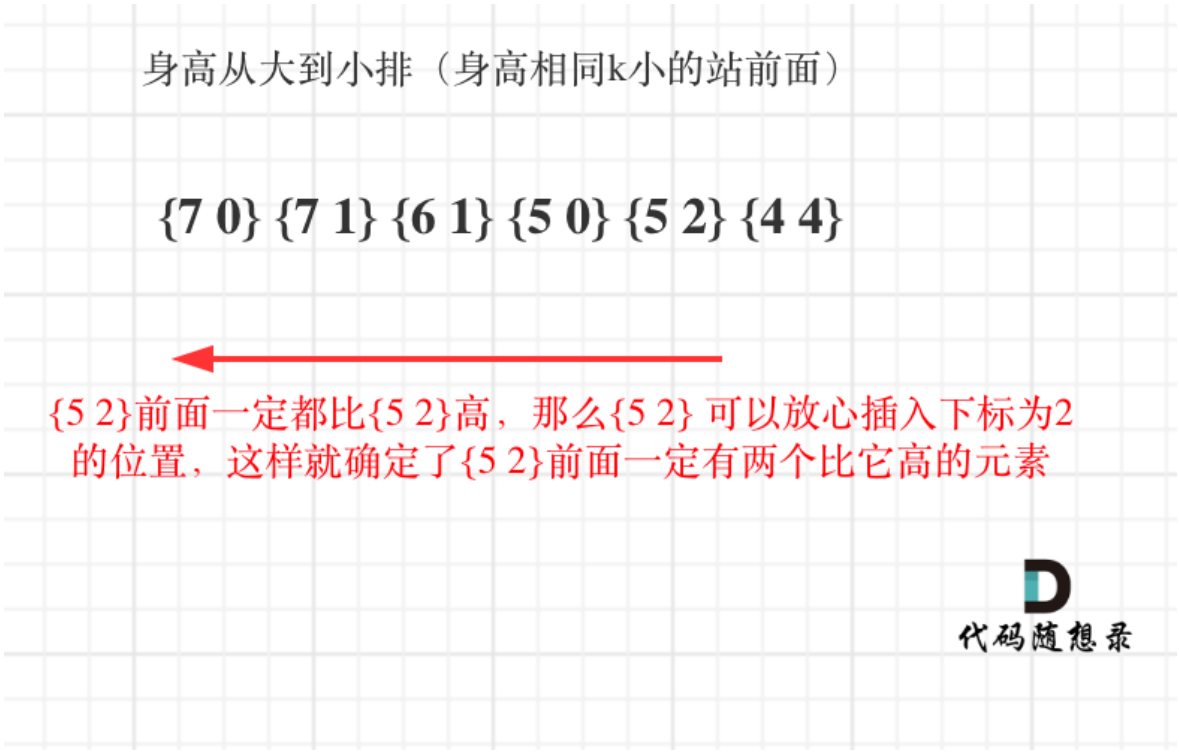
```
        return true;
    }
};
```

根据身高重建独列

算法思路：
首先根据身高从大到小排列，如果相同则k小的排在前面（贪心，更容易符合条件）

然后根据k的值以及在ret中插入数据

(当同时要处理两个维度，两个方向的时候，贪心先处理一边，再处理另一边。例如本体是先处理身高问题，后处理个数问题)




```
class Solution {
private:
    bool compare(vector<int> a, vector<int> b) {
        if (a[0] == b[0]) {
            return a[1] < b[1];
        }
        else {
            return a[0] > b[0];
        }
    }
```

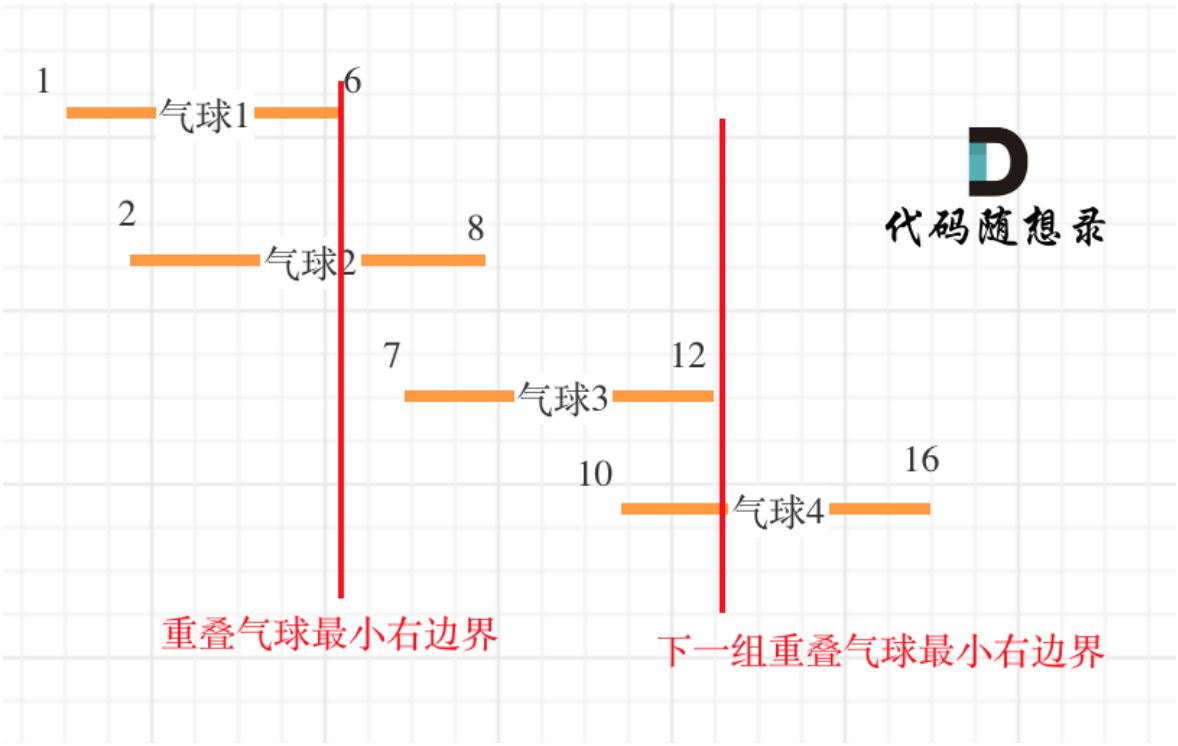
```
public:
    vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {

        sort(people.begin(), people.end(), compare);
        vector<vector<int>> ret;

        for (int i = 0;i < people.size();i++) {
            int pos = people[i][1];//前面高于他的人数
            ret.insert(ret.begin() + pos, people[i]);//vector的insert函数的使用，第一个参数为位置迭代器，第二个参数为数据
        }

        return ret;
    }
};
```

使用最少的弓箭射爆气球

算法思路：

根据左边界的大小进行排序，让其尽可能的集中到一起

如果前一个气球的右边界小于后一个气球的左边界，说明需要多一支。否则更新右边界，并在下一轮中作为前一个气球进行比较.



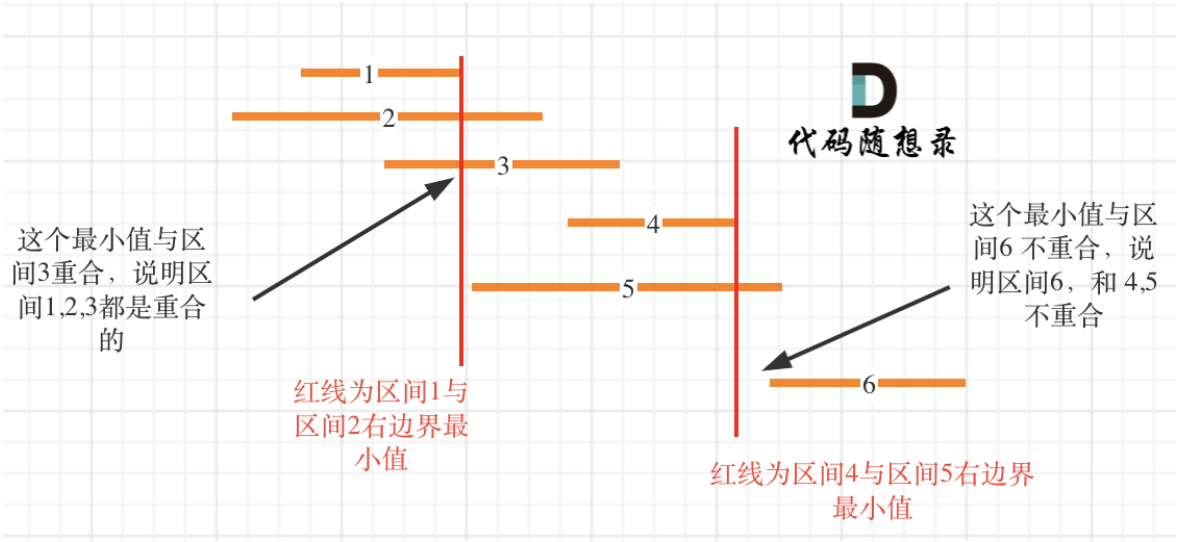
```cpp
class Solution {
private:
    static bool compare(const vector<int>& a,const vector<int>& b) {
        return a[0] < b[0];
    }
public:
    int findMinArrowShots(vector<vector<int>>& points) {

        if (points.size() <= 1) {
            return points.size();
        }
        sort(points.begin(), points.end(), compare);

        int ret = 1;
        for (int i = 1;i < points.size();i++) {
            if (points[i][0] > points[i - 1][1]) {//无重叠部分
                ret++;
            }
            else {
                points[i][1] = min(points[i][1], points[i - 1][1]);
                //区间合并，注意这里取的是最小值
                //能同时射中1，2，不一定能同时射中2，3。画图理解
            }
        }

        return ret;
    }
};
```

无重叠区间

算法思路：

将区间排序，并且更新右区间

当重叠时，每一次都取最小的右区间（相当于将右区间大的删除），这样更不可能重叠（贪心的体现）




```
class Solution {
    static bool compare(const vector<int>& a, const vector<int>& b) {
        return a[0] < b[0];
    }
public:
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        if (intervals.size() <= 1) {
            return 0;
        }

        sort(intervals.begin(), intervals.end(), compare);

        int count = 0;
        for (int i = 1;i < intervals.size();i++) {
            if (intervals[i][0] > intervals[i - 1][1]) {//前区间与后区间不重合
                continue;
            }
            else {
                intervals[i][1] = min(intervals[i][1], intervals[i - 1][1]);
                //重叠区间，删除右边范围最大的（其更有可能重叠）,并没有实际上删除
                //更新右边界
                count++;
            }
        }
        return count;
    }
};
```

划分字母区间

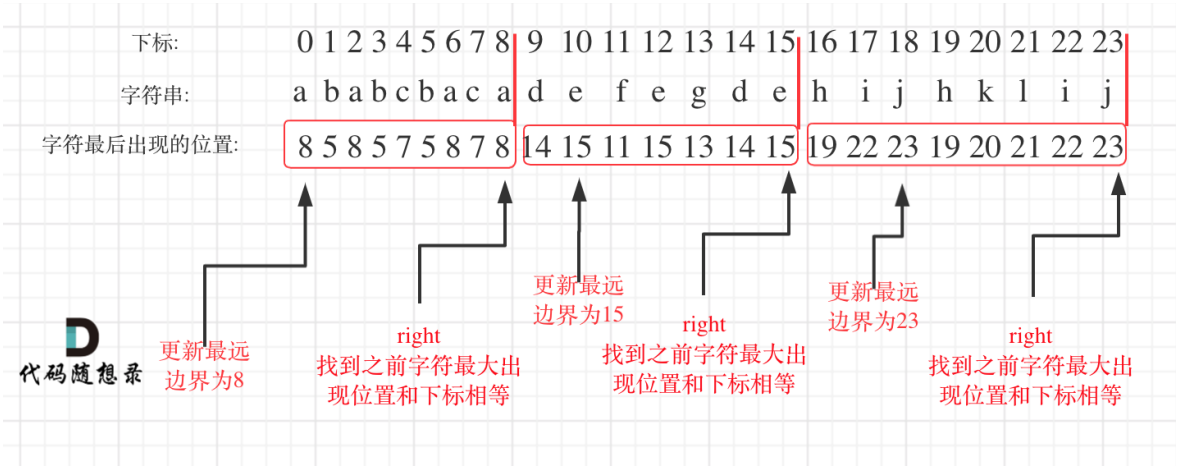
本体与跳跃区间II的解法很相似

思路：

1.寻找每一个字母的最右端出现的位置（使用哈希表进行记录）

2.使用left以及right双指针来控制区间长度，使用i来遍历数组（相当于三指针）

3.更新right的值，当i==right说明以及遍历完成这个区间，此时需要初始化left以及right




```
class Solution {
public:
    vector<int> partitionLabels(string s) {
        int MaxDistance[26];
        for (int i = 0;i < s.size();i++) {//记录最远出现的下标
            MaxDistance[s[i] - 'a'] = i;
        }

        int left = 0, right = 0;//用来控制区间的长度

        vector<int> ret;
        for (int i = 0;i < s.size();i++) {
            right = max(right, MaxDistance[s[i] - 'a']);
            if (i == right) {
                ret.push_back(right - left);
                left = right + 1;
            }
            else if (i == s.size() - 1) {//遍历到终点的时候
                ret.push_back(right - left);
            }
        }

        return ret;
```

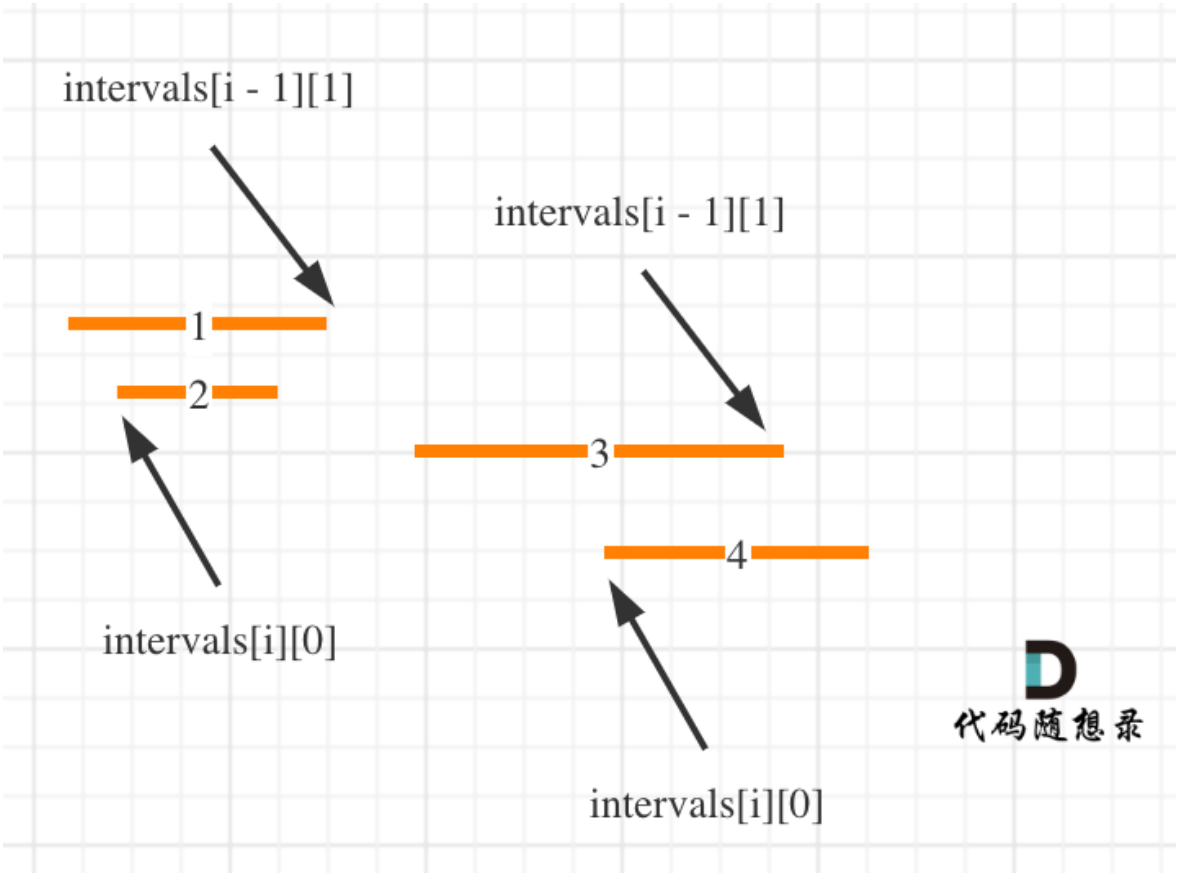
```
    }
};
```

合并区间

算法思想：

首先先插入intervals[0]的数据，并以此为基础更新有边界（这一点很重要，否则一直更新删除会很麻烦，可能会出现插入的区间与新的区间重叠的情况）

当intervals[i][0]小于result.back().[1]的时候，更新右边界，否则说明区间不重合，插入新的区间




```
class Solution {
private:
    static bool compare(const vector<int>& a, const vector<int>& b) {
        return a[0] < b[0];
    }

public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {

        if (intervals.size() <= 1) {
```

```
            return intervals;
        }

        sort(intervals.begin(), intervals.end(), compare);

        vector<vector<int>> result;
        result.push_back(intervals[0]);

        for (int i = 1;i < intervals.size();i++) {//不要在原数组上动刀，intervals.size()会变化，你把握不住
            if (intervals[i][0] <= result.back()[1]) {//重合
                int right = max(result.back()[1], intervals[i][1]);
                result.back()[1] = right;//更新右边界
            }
            else {
                result.push_back(intervals[i]);
            }
        }

        return result;
    }
};
```

每位上都单调递增的最大数字

暴力解法

```
class Solution {
private:
    // 判断一个数字的各位上是否是递增
    bool checkNum(int num) {
        int max = 10;
        while (num) {
            int t = num % 10;
            if (max >= t) max = t;
            else return false;
            num = num / 10;
        }
        return true;
    }
public:
    int monotoneIncreasingDigits(int N) {
        for (int i = N; i > 0; i--) { // 从大到小遍历
            if (checkNum(i)) return i;
        }
```

```
        return 0;
    }
};
```

贪心算法

思路：

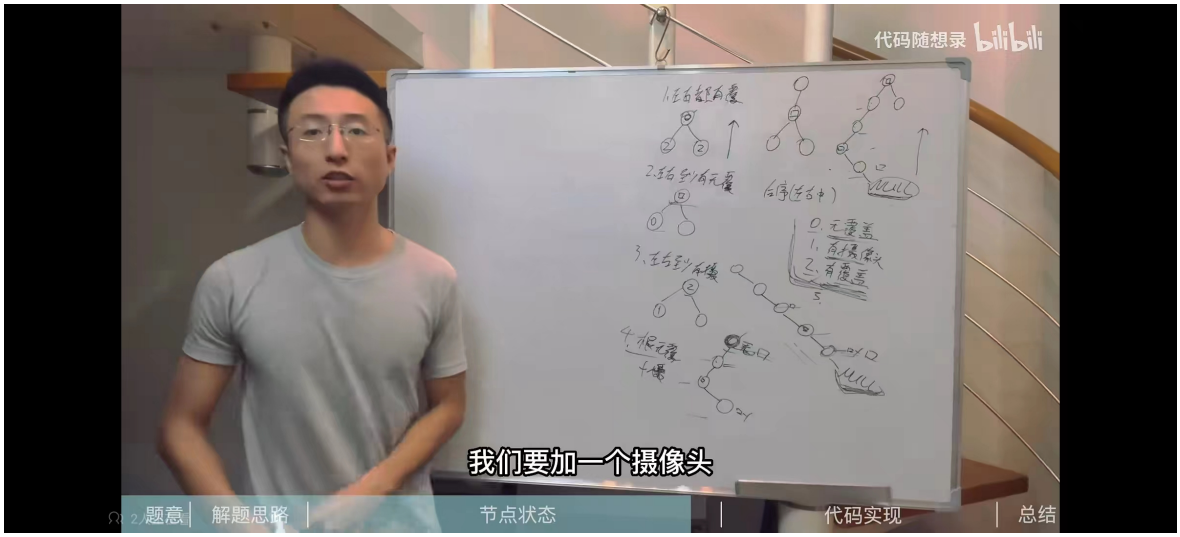
倒序，从后向前遍历。如果前大于后，则让前减一，并且记录此时后的位置

最后得到的index，将index以及后面所有的字符都设置为9

例如1000。最终的答案为0999（字符串），转换为int类型999。index记录位置为1，从1到str.size()-1都会被设置为9

```
class Solution {
public:
    int monotoneIncreasingDigits(int n) {

        string str = to_string(n);
        int index = str.size();

        for (int i = str.size() - 1;i > 0;i--) {//从后往前遍历，如果前大于后，则前减1，并且记录后的位置
            if (str[i - 1] > str[i]) {
                str[i - 1]--;
                index = i;
            }
        }

        for (int i = index;i < str.size();i++) {//将index后面的字符都设置为'9'
            str[i] = '9';
        }

        return stoi(str);//将字符串转换为数字
    }
};
```

二叉树与贪心的结合（不会）




```
#include <vector>
using namespace std;

struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode() : val(0), left(nullptr), right(nullptr) {}
    TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
    TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(ri
ght) {}
};

class Solution {
private:
    int result;
    int traversal(TreeNode* cur) {

        // 空节点，该节点有覆盖
        if (cur == NULL) return 2;

        int left = traversal(cur->left);    // 左
        int right = traversal(cur->right);  // 右

        // 情况1
        // 左右节点都有覆盖
        if (left == 2 && right == 2) return 0;

        // 情况2
        // left == 0 && right == 0 左右节点无覆盖
        // left == 1 && right == 0 左节点有摄像头，右节点无覆盖
        // left == 0 && right == 1 左节点有无覆盖，右节点摄像头
```

```
        // left == 0 && right == 2 左节点无覆盖，右节点覆盖
        // left == 2 && right == 0 左节点覆盖，右节点无覆盖
        if (left == 0 || right == 0) {
            result++;
            return 1;
        }

        // 情况3
        // left == 1 && right == 2 左节点有摄像头，右节点有覆盖
        // left == 2 && right == 1 左节点有覆盖，右节点有摄像头
        // left == 1 && right == 1 左右节点都有摄像头
        // 其他情况前段代码均已覆盖
        if (left == 1 || right == 1) return 2;

        // 以上代码我没有使用else，主要是为了把各个分支条件展现出来，这样代码有助于读者理解
        // 这个 return -1 逻辑不会走到这里。
        return -1;
    }

public:
    int minCameraCover(TreeNode* root) {
        result = 0;
        // 情况4
        if (traversal(root) == 0) { // root 无覆盖
            result++;
        }
        return result;
    }
};
```