贪心法

jask

10/03/2024

1005 k 次取反后最大化的数组和

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。

```
class Solution {
public:
    int largestSumAfterKNegations(vector<int>& nums, int k) {
        sort(nums.begin(),nums.end(),[](int a,int b){
            return abs(a)>abs(b);
        });
        for(int i=0;i<nums.size();i++){
            if(nums[i]<0&&k>0){
                nums[i]=-nums[i];
                k--;
            }
        }
        if(k%2==1) nums[nums.size()-1]*=-1;
        int result=0;
        for(auto& a: nums){
            result+=a;
        }
        return result;
    }
};
```

45 跳跃游戏 2

给定一个长度为 n 的 0 索引整数数组 nums。初始位置为 nums[0]。

每个元素 nums[i] 表示从索引 i 向前跳转的最大长度。换句话说，如果你在 nums[i] 处，你可以跳转到任意 nums[i + j] 处：

0 <= j <= nums[i]

i + j < n

返回到达 nums[n - 1] 的最小跳跃次数。生成的测试用例可以到达 nums[n - 1]。

```
class Solution {
public:
    int jump(vector<int>& nums) {
        if (nums.size() == 1) {
            return 0;
        }
        int cur_dist = 0;
        int ans = 0;
        int next_dist = 0;
        for (int i = 0; i < nums.size(); i++) {
```

```
            next_dist = max(next_dist, nums[i] + i);
            if (i == cur_dist) {
                if (cur_dist != nums.size() - 1) {
                    ans++;
                    cur_dist = next_dist;
                    if (next_dist >= nums.size() - 1)
                        break;
                } else {
                    break;
                }
            }
        }
        return ans;
    }
};
```

134 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

```
class Solution {
public:
int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
int curSum = 0;
int totalSum = 0;
int start = 0;
for (int i = 0; i < gas.size(); i++) {
curSum += gas[i] - cost[i];
totalSum += gas[i] - cost[i];
if (curSum < 0) {
// 当前累加 rest[i] 和 curSum 一旦小于 0
start = i + 1;// 起始位置更新为 i+1
curSum = 0;// curSum 从 0 开始
}
}
if (totalSum < 0) return -1; // 说明怎么走都不可能跑一圈了
return start;
}
};
```

135 分发糖果

`n` 个孩子站成一排。给你一个整数数组 `ratings` 表示每个孩子的评分。

你需要按照以下要求，给这些孩子分发糖果：

每个孩子至少分配到 1 个糖果。
相邻两个孩子评分更高的孩子会获得更多的糖果。

请你给每个孩子分发糖果，计算并返回需要准备的最少糖果数目。

思路：

这道题目一定是要确定一边之后，再确定另一边，例如比较每一个孩子的左边，然后再比较右边，如果两边一起考虑一定会顾此失彼。先确定右边评分大于左边的情况（也就是从前向后遍历）此时局部最优：只要右边评分比左边大，右边的孩子就多一个糖果，全局最优：相邻的孩子中，评分高的右孩子获得比左边孩子更多的糖果局部最优可以推出全局最优。如果 `ratings[i] > ratings[i - 1]` 那么 `[i]` 的糖一定要比 `[i - 1]` 的糖多一个，所以贪心：`candyVec[i] = candyVec[i - 1] + 1`

再确定左孩子大于右孩子的情况（从后向前遍历）遍历顺序这里有同学可能会有疑问，为什么不能从前向后遍历呢？ 因为如果从前向后遍历，根据 `ratings[i + 1]` 来确定 `ratings[i]` 对应的糖果，那么每次都不能利用上前一次的比较结果了。所以确定左孩子大于右孩子的情况一定要从后

向前遍历！ 如果 ratings[i] > ratings[i + 1]，此时 candyVeci就有两个选择了，一个是 candyVec[i + 1] + 1(从右边这个加 1 得到的糖果数量)，一个是 candyVeci。那么又要贪心了，局部最优：取 candyVec[i + 1] + 1 和 candyVec[i] 最大的糖果数量，保证第 i 个小孩的糖果数量即大于左边的也大于右边的。全局最优：相邻的孩子中，评分高的孩子获得更多的糖果。局部最优可以推出全局最优。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。

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        vector<int> candyVec(ratings.size(),1);
        //从前向后
        for(int i=1;i<ratings.size();i++){
            if(ratings[i]>ratings[i-1]) candyVec[i]=candyVec[i-1]+1;
        }
        //从后向前
        for(int i=ratings.size()-2;i>=0;i--){
            if(ratings[i]>ratings[i+1]){
                candyVec[i]=max(candyVec[i],candyVec[i+1]+1);
            }
        }
        return accumulate(candyVec.begin(), candyVec.end(),0);
    }
};
```

406 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）。

思路：

按照身高排序之后，优先按身高高的 people 的 k 来插入，后序插入节点也不会影响前面已经插入的节点，最终按照 k 的规则完成了队列。所以在按照身高从大到小排序后：局部最优：优先按身高高的 people 的 k 来插入。插入操作过后的 people 满足队列属性全局最优：最后都做完插入操作，整个队列满足题目队列属性

```
class Solution {
public:
    vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
        auto cmp=[](vector<int>& a,vector<int>& b)->bool{
            if(a[0]==b[0]) return a[1]<b[1];
            return a[0]>b[0];
        };
        sort(people.begin(),people.end(),cmp);
        vector<vector<int>> queue;
        for(int i=0;i<people.size();i++){
            int position=people[i][1];
            queue.insert(queue.begin()+position, people[i]);
        }
        return queue;
    }
};
```

452 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。

```
class Solution {
public:
    int findMinArrowShots(vector<vector<int>>& points) {
        auto cmp=[](vector<int>& a,vector<int>& b)->bool{
            return a[0]<b[0];
        };
        if(points.size()==0) return 0;
        sort(points.begin(),points.end(),cmp);
        int result=1;
        for(int i=1;i<points.size();i++){
            if(points[i][0]>points[i-1][1]){
                result++;
            }else{
                points[i][1]=min(points[i-1][1],points[i][1]);
            }
        }
        return result;
    }
};
```

435 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。

```
class Solution {
public:
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        auto cmp=[](vector<int>& a,vector<int>& b)->bool{
        return a[1]<b[1];
        };
        sort(intervals.begin(),intervals.end(),cmp);
        int count=1;
        int end=intervals[0][1];
        for(int i=1;i<intervals.size();i++){
            if(intervals[i][0]>=end){
                end=intervals[i][1];
                count++;
            }
        }
        return intervals.size()-count;
    }
};
```

763 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。

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。

```
class Solution {
public:
    vector<int> partitionLabels(string s) {
        int hash[27]{};
        for(int i=0;i<s.size();i++){
            hash[s[i]-'a']=i;
        }
        vector<int> result;
        int left=0,right=0;
```

```
        for(int i=0;i<s.size();i++){
            right=max(right,hash[s[i]-'a']);
            if(i==right){
                result.push_back(right-left+1);
                left=i+1;
            }
        }
        return result;
    }
};
```

56 合并区间

以数组 `intervals` 表示若干个区间的集合，其中单个区间为 `intervals[i] = [starti, endi]` 。请你合并所有重叠的区间，并返回一个不重叠的区间数组，该数组需恰好覆盖输入中的所有区间。

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        auto cmp=[](vector<int>& a,vector<int>& b)->bool{
            return a[0]<b[0];
        };
        sort(intervals.begin(),intervals.end(),cmp);
        auto left=intervals[0][0];
        auto right=intervals[0][1];
        vector<vector<int>> result;
        for(int i=1;i<intervals.size();i++){
            if(intervals[i][0]>right){
                result.push_back({left,right});
                left=intervals[i][0];
                right=intervals[i][1];
            }
            right=max(right,intervals[i][1]);
        }
         result.push_back({left,right});
        return result;
    }
};
```

738 单调递增的数字

当且仅当每个相邻位数上的数字 `x` 和 `y` 满足 `x` 小于等于 `y` 时，我们称这个整数是单调递增的。

给定一个整数 `n` ，返回小于或等于 `n` 的最大数字，且数字呈单调递增。

```
class Solution {
public:
    int monotoneIncreasingDigits(int n) {
        auto num=to_string(n);
        int flag=num.size();
        for(int i=num.size()-1;i>0;i--){
            if(num[i-1]>num[i]){
                flag=i;
                num[i-1]--;
            }
        }
        for(int i=flag;i<num.size();i++){
            num[i]='9';
        }
        return stoi(num);
    }
```

```
};
```

968 监控二叉树

给定一个二叉树，我们在树的节点上安装摄像头。

节点上的每个摄影头都可以监视其父对象、自身及其直接子对象。

计算监控树的所有节点所需的最小摄像头数量。

所以我们要从下往上看，局部最优：让叶子节点的父节点安摄像头，所用摄像头最少，整体最优：全部摄像头数量所用最少！

```
class Solution {
    int result=0;
    int traversal(TreeNode *cur){
        if(!cur) return 2;
        int left=traversal(cur->left);
        int right=traversal(cur->right);
        if(left==2&&right==2) return 0;
        else if(left==0||right==0){
            result++;
            return 1;
        }else return 2;
    }
public:
    int minCameraCover(TreeNode* root) {
        if(traversal(root)==0){//root 无覆盖
            result++;
        }
        return result;
    }
};
```