力扣2410题·运动员和训练师的最大匹配数

李雨

长沙学院

2025年7月13日

1. 2410·运动员和训练师的最大匹配数

1.1 知识点

贪心 | 排序 | 双指针 | 二分查找 | 优先队列

1.2 解题思路

每个运动员有一个**能力值**，每个训练师有一个**训练能力值**，如果一个运动员的能力值**小于等于**另一个训练师的的训练能力值，说明这个运动员可以和这个训练师进行匹配，我们要解出**最大**匹配数。

为了最大程度地完成匹配，我们对每位运动员寻找一个能力值不小于其本人的、并尽可能接近的训练师，以提高整体匹配数量。

1.3 方法一：排序 + 二分

1.3.1 思路

对于每个运动员来说，我们要找到和当前运动员能力值最相近的训练师，我们把训练师由小到大进行排序，利用序列的有序性使用二分查找来加速查找过程。为了找到和当前运动员能力值最相近的训练师，所以如果**从运动员的角度出发**，我们要找打大于等于当前运动员能力值的最小的训练师。对于每个训练师，我们要找到小于等于当前训练师能力值的能力值最大的运动员，同样，如果**从训练师的角度出发**，我们将运动员的能力值按照从小到大的顺序进行排序，利用序列的有序性使用二分查找来加快查找过程。两者的大致思路是一样的，不同的是具体实现的代码不同，从运动员的角度出发是使用`lower_bound`实现，从训练师的角度出发是使用`upper_bound`实现。

1.3.2 代码

- **从运动员的角度出发**

```cpp
class Solution {
public:
    int matchPlayersAndTrainers(vector<int>& players, vector<int>& trainers) {
        multiset<int> trainer_st(trainers.begin(), trainers.end());
        // 找最小的大于等于player的trainer
        int ans = 0;
        for (int x : players) {
            auto pos = trainer_st.lower_bound(x);
            if (pos != trainer_st.end()) {
                ans++;
                trainer_st.erase(pos);
            }
        }时
        return ans;
    }
};
```

- **从训练师的角度出发**

```cpp
class Solution {
public:
    int matchPlayersAndTrainers(vector<int>& players, vector<int>& trainers) {
        multiset<int> player_st(players.begin(), players.end());
        // 找小于trainer的最大player
        int ans = 0;
        for (int trainer : trainers) {
            auto pos = player_st.upper_bound(trainer);
            if (pos != player_st.begin()) {
                --pos;
                ans++;
                player_st.erase(pos);
            }
        }
        return ans;
    }
};

```

1.3.3 复杂度分析

- 时间复杂度：$O(nlogn)$。向容器中插入元素的时间复杂度为$logn$，共有n个元素，所以将训练师的训练能力值放进容器内的时间复杂度为$O(nlogn)$。对于每个运动员，二分找到最接近的训练的时间复杂度为$log(n)$,在容器内删除训练师的时间复杂度也是$log(n)$,共有n个运动员，最坏情况下，每个运动员都要二分找一个，并且删除一次，总时间复杂度还是$O(nlogn)$.
- 空间复杂度：$O(n)$.使用了一个额外的容器存储训练师的训练能力值。

1.3.4 总结

为了加快查找，我们利用了数组排序后的有序性，另外在排序加二分的方法下，如果只是简单的将数组排序，然后进行查找的话，可能会出现多个运动员匹配同一个训练师的情况，为了避免这一个情况，我们使用一个删除方便的容器`multiset`，这个容器删除一个元素的时间复杂度为$O(logn)$，在找到配对的运动员和训练师的同时将其从容器内删除，这样就可以避免上面所述情况。

问：为什么要使用二分查找加快查找?

由于我们要找到最接近每个运动员的能力值的训练师，由于暴力找到对应的训练师需要我们枚举每个运动员的能力值，然后在训练师中找到最接近运动员的能力值的训练能力值，最坏的情况下对于每个运动员可能都要枚举一遍训练师能力值数组，为了加快查找，我们可以先对训练师的能力值数组进行由小向大进行排序，然后使用二分查找的方法快速查找到最接近运动员能力值的训练师。

问：为什么使用multiset容器?

这里我们使用`multiset`存储我们排序后的数组，`multiset`是一个天然的排序容器，且可以存储存储重复元素，满足我们存储数组元素但是可能出现重复元素的情况的要求。对于有序容器`multiset`，其存在`lower_bound`内置函数，满足了我们进行二分查找的需求。在枚举每个运动员并且使用二分查找到对应的训练师后，我们将对应的训练师从我们的数据结构中删除，这样就可以避免后续查找到同一个训练师的情况。

1.4 方法二：贪心 + 排序 + 双指针

1.4.1 思路

- **从运动员的角度出发**

对于任意一种配对方案，我们都可以讲其中已经配对的运动员换成未配对的能力值更小的运动员。为了方便枚举，我们首先对$players$和$trainers$进行排序；排序后的$players$的前缀便是将要配对的运动员。

对于每个运动员$players[i]$，我们要在$trainer$中找到大于等于$players[i]$的最接近当前运动员的能力值的训练师。不然可能会导致能力值更大的运动员找不到和他配对的训练师，我们可以使用双指针实现这一功能，对于每个运动员，找到大于等于当前运动员的第一个训练师，重复这一操作，直到运动员找不到对应的训练师。

- **从训练师的角度出发**

对于每个训练师而言，如果能够尽可能的找到配对的运动员，那么最后得到的配对数量也一定是最大的，对于每个训练师，只要存在运动员的能力值小于等于当前训练师的训练能力值，那么就说明当前训练师可以找到运动员进行配对，这里我们还是让最小的运动员和训练师进行匹配。同样为了方便计算，先对$players$，和$trainers$进行排序，同样使用双指针的做法，使用一个额外的下标j来枚举运动员的数量，如果当前的运动员符合条件，就继续枚举下一个运动员，直到枚举完所有的运动员，或者枚举完所有的训练师。

1.4.2 代码

- **从运动员的视角出发**

```cpp
class Solution {
public:
    int matchPlayersAndTrainers(vector<int>& players, vector<int>& trainers) {
        sort(players.begin(), players.end());
        sort(trainers.begin(), trainers.end());
        int n = players.size();
        int j = 0;
        for (int i = 0; i < n; i++) {
            while (j < m && trainers[j] < players[i]) {
                j++;
            }
            if (j == m) {
                return i;
            }
            j++;
        }
        return n;
    }
};
```

- **从训练员的视角出发**

```cpp
class Solution {
public:
    int matchPlayersAndTrainers(vector<int>& players, vector<int>& trainers) {
        sort(players.begin(), players.end());
        sort(trainers.begin(), trainers.end());
```

```
        int n = players.size();
        int j = 0;
        for (int t : trainers) {
            if (j < n && players[j] <= t) {
                j++;
            }
        }
        return j;
    }
};
```

1.4.3 复杂度分析

- 时间复杂度：$O(nlogn) + O(mlogm)$。其中n为$players$的长度，m为$trainers$的长度，注意使用双指针$players$ 和$trainers$都最多只会遍历一次，所以双指针枚举的时间复杂度为$O(n + m)$。所以总的时间复杂度还是$O(NlogN)$,N是n和m的较大值。
- 空间复杂度：$O(1)$

1.4.4 总结

双指针解决问题的思路就是固定一方，让另一方去动态的移动，直到找到可以配对的两者，然后再去移动固定的另一方，需要注意的是，本题双指针算法的前提是基于两个序列的有序性的。

1.5 方法三：贪心 + 优先队列

1.5.1 思路

为了让更多玩家匹配到训练师，我们还可以采用如下的策略：

1. 每次优先处理能力值最大(最小)的玩家。
2. 尝试为其分配能力值最大(最小)的训练师。
3. 如果该训练师的能力值大于等于运动员的能力值，代表匹配成功
4. 如果不能匹配成功，就继续找比当前运动员能力值更小(更大)的运动员

因为每次我们要处理运动员和训练师的能力值的最大值，我们采用`“大根堆”`或者`“小根堆”`；

1.5.2 代码

- **从运动员的角度出发**

```
// 小顶堆
class Solution {
public:
    int matchPlayersAndTrainers(vector<int>& players, vector<int>& trainers) {
        priority_queue<int, vector<int>, greater<>> pq_players(players.begin(),
players.end());
        priority_queue<int, vector<int>, greater<>> pq_trainers(trainers.begin(),
trainers.end());

        int ans = 0;
        while (!pq_players.empty() && !pq_trainers.empty()) {
```

```
            int player = pq_players.top(), trainer = pq_trainers.top();
            // 如果当前的运动员和训练师匹配了，就将运动员弹出大顶堆
            if (player <= trainer) {
                pq_players.pop();
                ans++;
            }
            pq_trainers.pop();
        }
        return ans;
    }
    // 注意循环中的逻辑，如果运动员的能力值小于等于训练师，就要同时删除运动员和训练师，否则只要删除训练师就行了，所以可以简写成
    // 当满足条件的时候只删除运动员，但是每次都会删除训练师
};
```

- **从训练师的角度出发**

```
// 大顶堆
class Solution {
public:
    int matchPlayersAndTrainers(vector<int>& players, vector<int>& trainers) {
        priority_queue<int> pq_players(players.begin(), players.end());
        priority_queue<int> pq_trainers(trainers.begin(), trainers.end());

        int ans = 0;
        while (!pq_players.empty() && !pq_trainers.empty()) {
            int player = pq_players.top(), trainer = pq_trainers.top();
            // 如果当前的训练师和运动员匹配了，就将训练师从大顶堆中删除
            if (player <= trainer) {
                pq_trainers.pop();
                ans++;
            }
            pq_players.pop();
        }
        return ans;
    }
    // 注意循环中的逻辑，如果运动员的能力值小于等于训练师，就要同时删除运动员和训练师，否则只要删除运动员就行了，所以可以简写成
    // 当满足条件的时候只删除训练师，但是每次都会删除运动员
};
```

1.5.3 复杂度分析

- 时间复杂度：$O(nlogn)$.构建大根堆的时间复杂度为$O(n)$,我们要从大根堆中取出所有训练师的最大值，总的时间复杂度为$O(nlogn)$
- 空间复杂度：$O(n)$。额外使用了两个大顶堆存储运动员和训练师的能力值。

1.5.4 总结

这里使用大顶堆来处理的本质就是因为为了让训练师去匹配更多的运动员，我们让能力值最大的训练师去匹配能力值最大的运动员，为了每次取到能力值最大的运动员和训练师，所以我们构建了大顶堆来存储所有运动员和能力值和所有训练师的能力值。注意是因为我们要每次取到最大值，所以我们才使用了这个数据结构。这道题的算法逻辑是：如果运动员和训练师的大顶柜都不为空的时候我们才会继续去找可以匹配的一组，对于当前的训练师，如果当前的运动员不能和当前的训练师进行匹配，说明对于当前的训练师而言，运动员的能力值太大了，那么我们就要去找能力值相对来说更小的运动员；对应的算法的操作就是将当前的运动员的能力值从大顶堆中删除(pop)，这样下一个堆顶就是能力值更小的运动员了。如果可以匹配的话，我们就要让能力值更小的训练师去匹配下一个运动员，重复这个操作。需要注意的是，这里使用大根堆和使用小根堆处理的大概思路是一样的，大顶堆让能力值最大的训练师去处理能力值最大的运动员，而小根堆是让能力值最小的运动员去处理能力值最小的训练师。

2. 题目总结

本题是基于贪心的思想，为了让运动员尽可能多地匹配训练师，所以对于每个运动员来说，他都要匹配和和自己能力值相近的训练师，对于训练师而言，他也要匹配能力值和他相近的运动员，为了加快查找，我们可以将一方进行排序，遍历另一方的同时利用序列的有序性加快查找；而对于每个和训练师配对的运动员来说，都可以换成能力值更小的运动员和训练师进行配对，所以我们又可以从最小的运动员开始配对，让每个运动员去匹配和其能力值相近的训练师，这里使用双指针或者使用小顶堆都可以实现。同理，对于每个和运动员配对的训练师来说，我们都能找到能力值更大的训练师和运动员进行配对，所以我们可以枚举能力值最大的训练师，贪心的找到能力值最大的运动员和其进行配对。

另一点就是对于这个题目，我们发现只有序列具有有序性，那么进行查找的时候我们就可以使用二分或者双指针进行加快查找，两者的时间复杂度不相同，本题利用双指针算法在最坏情况下要遍历完两个序列，时间复杂度为$O(n+m)$.而对于二分而言，每次查找的时间复杂度为$logn$。而对于二分查找而言，最坏情况下对于每个运动员(训练师)都要查找一下训练师（运动员），时间复杂度为$O(nlogn)$。而当我们根据运动员的最小值，找训练师的最小值，或者根据训练师的最大值找到运动员的最小值，涉及到频繁的最值操作，我们就可以想到使用小顶堆或者大顶堆记性操作。

总的来说，一题多解是我们彻底搞懂一个题目的前提，当然，每种方法的使用前提我们也要直到，比方本题使用二分和双指针的前提就是利用了序列的有序性，如果没有这一个性质的话，恐怕这里并不能使用二分或者双指针。堆是一种处理最值操作很方便的数据结构，如果题目设置最值操作。我们也要想到使用这个数据结构。