406-根据身高重建队列

题述

406. 根据身高重建队列

难度 中等 1153

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。

思路

本题有两个维度，h和k，看到这种题目一定要想如何确定一个维度，然后在按照另一个维度重新排列。

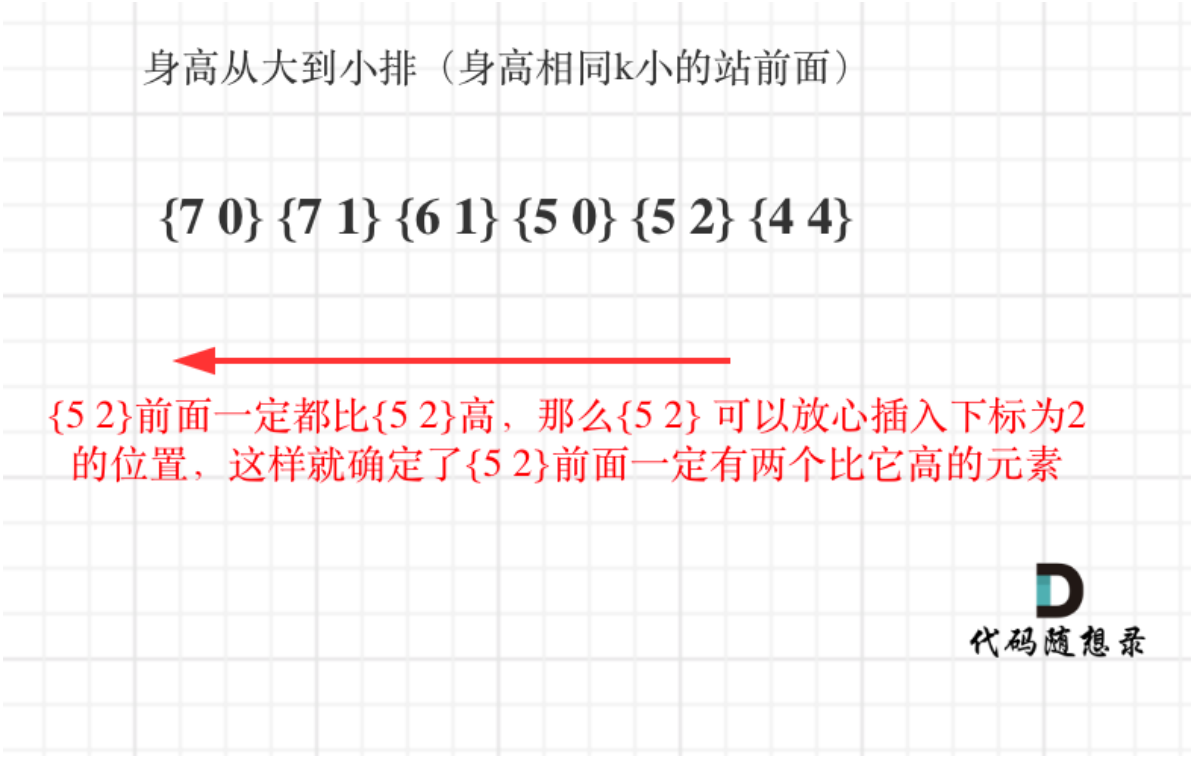
如果两个维度一起考虑一定会顾此失彼

如果按照k来从小到大排序，排完之后，会发现k的排列并不符合条件，身高也不符合条件，两个维度哪一个都没确定下来。

那么按照身高h来排序呢，身高一定是从大到小排（身高相同的话则k小的站前面），让高个子在前面。

此时我们可以确定一个维度了，就是身高，前面的节点一定都比本节点高！

那么只需要按照k为下标重新插入队列就可以了



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

局部最优可推出全局最优，找不出反例，那就试试贪心。

题解

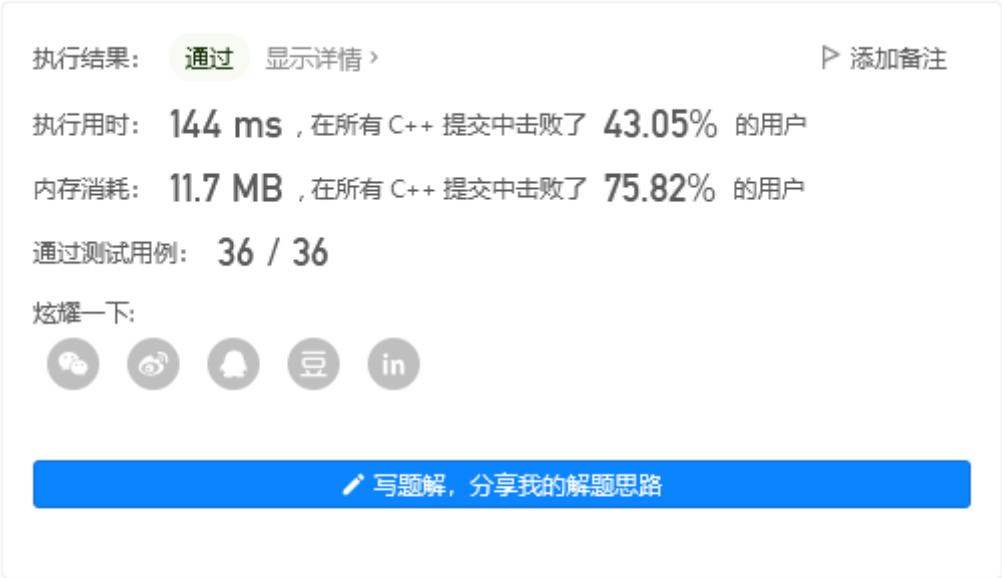
C++解法一

```
class Solution {
public:
    static bool cmp(const vector<int>& a,const vector<int>& b)
    {
        //排序规则
        if(a[0] == b[0])
        {
            return a[1]<b[1];
        }
        return a[0] > b[0];
    }

    vector<vector<int>> reconstructQueue(vector<vector<int>>& people)
    {
        //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。
        //输入序列为乱序
        //按其规则进行排列
```

```
        //两个排序的维度   先确定一个维度
        //以身高 hi为第一维度

        //局部最优：优先按照身高由大到小进行排序
        //全局最优：满足题目规则
        sort(people.begin(),people.end(),cmp);  //按照身高从大到小的规则进行排序   以hi
为维度
        vector<vector<int>> que;    //答案序列
        for(int i=0;i<people.size();i++)
        {
            int pos = people[i][1];     //第二维度
            que.insert(que.begin()+pos,people[i]);
        }
        return que;
    }
};
```



但使用vector是非常费时的，C++中vector（可以理解是一个动态数组，底层是普通数组实现的）如果插入元素大于预先普通数组大小，vector底部会有一个扩容的操作，即申请两倍于原先普通数组的大小，然后把数据拷贝到另一个更大的数组上。

所以使用vector（动态数组）来insert，是费时的，插入再拷贝的话，单纯一个插入的操作就是$O(n^2)$了，甚至可能拷贝好几次，就不止$O(n^2)$了。

C++解法二

```
// 版本二
class Solution {
public:
    // 身高从大到小排（身高相同k小的站前面）
    static bool cmp(const vector<int>& a, const vector<int>& b) {
        if (a[0] == b[0]) return a[1] < b[1];
        return a[0] > b[0];
```

```
    }
    vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
        sort (people.begin(), people.end(), cmp);
        list<vector<int>> que; // list底层是链表实现，插入效率比vector高的多
        for (int i = 0; i < people.size(); i++) {
            int position = people[i][1]; // 插入到下标为position的位置
            std::list<vector<int>>::iterator it = que.begin();
            while (position--) { // 寻找在插入位置
                it++;
            }
            que.insert(it, people[i]);
        }
        return vector<vector<int>>(que.begin(), que.end());
    }
};
```


```
// 版本二
class Solution {
public:
    // 身高从大到小排（身高相同k小的站前面）
    static bool cmp(const vector<int>& a, const vector
b) {
        if (a[0] == b[0]) return a[1] < b[1];
        return a[0] > b[0];
    }
    vector<vector<int>> reconstructQueue(vector<vector
people) {
        sort (people.begin(), people.end(), cmp);
        list<vector<int>> que; // list底层是链表实现，
比vector高的多
        for (int i = 0; i < people.size(); i++) {
            int position = people[i][1]; // 插入到下标
position的位置
            std::list<vector<int>>::iterator it = que.
            while (position--) { // 寻找在插入位置
                it++;
            }
            que.insert(it, people[i]);
        }
        return vector<vector<int>>(que.begin(), que.er
    }
};
```

思考

关于出现两个维度一起考虑的情况，我们已经做过两道题目了

其技巧都是确定一边然后贪心另一边，两边一起考虑，就会顾此失彼。

可以明显看是使用C++中的list（底层链表实现）比vector（数组）效率高得多。

对使用某一种语言容器的使用，特性的选择都会不同程度上影响效率。