406. Queue Reconstruction by Height Medium <u>Greedy</u> Binary Indexed Tree Segment Tree Array

**Problem Description** In this problem, we are given an array of people's attributes, where each person is represented by a pair of values [h, k]. The h

represents the height of the person and k represents the number of people who are in front of this person in a queue and are

The reconstructed queue should also be represented as an array where each element follows the same pair representation [h, k], with each pair denoting the attributes of the people in the queue. To be more concrete, if  $people[i] = [h_i, k_i]$ , we need to place the ith person in such a position in the queue that  $k_i$  is the exact count of people ahead of this person with heights greater than or equal to h\_i.

taller or of the same height. The array people does not necessarily represent the order of people in a queue. Our task is to

reconstruct the queue in such a way that the attributes of people in the reconstructed queue align with the original array people.

The intuition behind the solution lies in the fact that it's easier to place the tallest people first and then insert the shorter people

Intuition

descending height and x[1] for ascending k values. Sorting is done with this line:

already in the list, since those have been inserted earlier due to the sorting strategy.

front first.

at their appropriate positions based on the k value. Here's how we can think of the approach: Sort people by their height in descending order; if two people have the same height, sort them by their k value in ascending order. This way, we start with the tallest people, and for those with the same height, we place those with fewer people in

- Initialize an empty list ans that will represent our queue. Iterate over the sorted list, inserting each person into the ans list at the index equal to their k value. It works because by inserting the tallest people first, their k value corresponds exactly to the position they should be in the queue. As we insert the shorter people, we displace the taller ones only if needed, which maintains the integrity of the k values.
- By following this approach, we ensure that each person is placed in accordance with the number of taller or equally tall people that are supposed to be in front of them according to the original array people.

people.sort(key=lambda x: (-x[0], x[1]))

Solution Approach The solution uses a greedy algorithm approach along with list operations to reconstruct the queue. Here is a step-by-step

walkthrough of the implementation based on the solution code provided: Sort the people array in a way such that we first sort the people by height in descending order and then, if the heights are

equal, sort by the k value in ascending order. The lambda function in the sort method accomplishes this by using -x[0] for

for p in people:

return ans

**Example Walkthrough** 

ans.insert(p[1], p)

By sorting in this manner, we prioritize taller individuals and, among those of identical height, prioritize those with a smaller k value, which is the count of people in front of them.

ans = [] Iterate over the sorted people array. For each person p, we insert them into the ans list at the index specified by their k value. This is possible because at the time of their insertion, there will be p[1] (which is k) taller or equal height people

The insert operation places the person at the correct position in the queue, shifting elements that are already there.

Return the ans list which now represents the reconstructed queue.

Let's consider a small example to illustrate the solution approach.

Suppose we are given the following array people:

Following the steps outlined in the solution approach:

Initialize an empty list ans which will be used to construct the queue.

```
and the count of people in front of them with equal or greater height. The use of sorting and list insertions are key aspects of the
algorithm, ensuring that the resulting ans list satisfies the requirements that are stated in the problem.
```

Using this greedy strategy, we are able to re-order the array so that each person is positioned in accordance with their height

people = [[7, 0], [4, 4], [7, 1], [5, 0], [6, 1], [5, 2]]

1. We first sort the array people by height in descending order and k value in ascending order when the heights are equal. After sorting, our

```
2. We initialize an empty list ans:
```

ans = []

people array looks like this:

Solution Implementation

queue = []

import java.util.Arrays;

import java.util.List;

class Solution {

import java.util.ArrayList;

for person in people:

**Python** 

Java

We then iterate over the sorted people array and insert each person into the ans list at the index specified by their k value:

We insert [7, 1] at index 1: ans = [[7, 0], [7, 1]]

ans = [[5, 0], [7, 0], [5, 2], [6, 1], [4, 4], [7, 1]]

def reconstructQueue(self, people: List[List[int]]) -> List[List[int]]:

# Initialize an empty list to hold the final queue reconstruction

# Sort the 'people' array in descending order of height (h),

people.sort(key=lambda person: (-person[0], person[1]))

# Iterate over the sorted 'people' list

public int[][] reconstructQueue(int[][] people) {

Arrays.sort(people, (person1, person2) ->

// at the index specified by their k-value.

// Sort the array. First, sort by height in descending order.

// Initialize a list to hold the final queue reconstruction,

List<int[]> reconstructedQueue = new ArrayList<>(people.length);

// Iterate over the sorted array, and insert each person into the list

return reconstructedQueue.toArray(new int[reconstructedQueue.size()][]);

arrangedQueue.insert(arrangedQueue.begin() + person[1], person);

// Sort the `people` array to arrange them by their height in descending order.

// The index at which to insert is determined by the person's k-value.

// let peopleInput: number[][] = [[7, 0], [4, 4], [7, 1], [5, 0], [6, 1], [5, 2]];

def reconstructQueue(self, people: List[List[int]]) -> List[List[int]]:

# Initialize an empty list to hold the final queue reconstruction

# Sort the 'people' array in descending order of height (h),

people.sort(key=lambda person: (-person[0], person[1]))

// If two people have the same height, order them by their k-value in ascending order.

// which indicates the number of people in front of them with a height greater than or equal.

# and in the case of a tie, in ascending order of the number of people in front (k).

# Insert each person into the queue. The index for insertion is the k-value.

# which represents the number of people in front of them with equal or greater height.

// which allows us to insert people at specific indices.

// Convert the list back to an array before returning it.

for (const std::vector<int>& person : people) {

// Return the final arrangement of the queue.

function reconstructQueue(people: number[][]): number[][] {

// Initialize an empty array for the final arrangement.

return b[0] - a[0] || a[1] - b[1];

// Insert each person into the `arrangedQueue`.

// Return the final arrangement of the queue.

# Iterate over the sorted 'people' list

queue.insert(person[1], person)

# Return the reconstructed queue

// let queue = reconstructQueue(peopleInput);

arrangedQueue.splice(person[1], 0, person);

let arrangedQueue: number[][] = [];

for (let person of people) {

return arrangedQueue;

return arrangedQueue;

people.sort((a, b) => {

• We insert [7, 0] at index 0: ans = [[7, 0]]

people = [[7, 0], [7, 1], [6, 1], [5, 0], [5, 2], [4, 4]]

```
We insert [6, 1] at index 1: ans = [[7, 0], [6, 1], [7, 1]]
• We insert [5, 0] at index 0: ans = [[5, 0], [7, 0], [6, 1], [7, 1]]
○ We insert [5, 2] at index 2: ans = [[5, 0], [7, 0], [5, 2], [6, 1], [7, 1]]
```

• Finally, we insert [4, 4] at index 4: ans = [[5, 0], [7, 0], [5, 2], [6, 1], [4, 4], [7, 1]]

The final ans list is our reconstructed queue which aligns with the original people array attributes:

the k value is the exact count of people ahead of them with heights greater than or equal to their own.

# and in the case of a tie, in ascending order of the number of people in front (k).

// If heights are equal, sort by the number of people in front (k-value) in ascending order.

person1[0] == person2[0] ? person1[1] - person2[1] : person2[0] - person1[0]);

```
Using the greedy solution approach of sorting the individuals by height and then inserting them into the queue based on their k
```

value, we have successfully reconstructed a queue that satisfies the requirements of the problem. In this queue, for every person,

```
from typing import List
class Solution:
```

# Insert each person into the queue. The index for insertion is the k-value. # which represents the number of people in front of them with equal or greater height. queue.insert(person[1], person) # Return the reconstructed queue return queue

```
for (int[] person : people) {
   // The second value of each person array (person[1])
   // specifies the index at which this person should be added in the queue
   reconstructedQueue.add(person[1], person);
```

```
C++
#include <vector>
#include <algorithm>
class Solution {
public:
    std::vector<std::vector<int>> reconstructQueue(std::vector<std::vector<int>>& people) {
        // Sort the people to arrange them according to their height in descending order.
        // If two people are of the same height, order them by their "k-value" in ascending order.
        std::sort(people.begin(), people.end(), [](const std::vector<int>& personA, const std::vector<int>& personB) {
            return personA[0] > personB[0] || (personA[0] == personB[0] && personA[1] < personB[1]);</pre>
        });
        // Initialize an empty vector for the final arrangement.
        std::vector<std::vector<int>> arrangedQueue;
        // Insert each person into the arrangedQueue.
        // The index to insert the person is determined by the person's "k-value",
        // which indicates the number of people in front of this person who have a height greater than or equal to this person's heig
```

```
// console.log(queue); // Outputs arranged queue based on the given conditions.
from typing import List
```

class Solution:

queue = []

return queue

**Time Complexity** 

for person in people:

Time and Space Complexity

Sorting the people array:

elements of the list.

The ans list:

// Example of usage:

**TypeScript** 

});

```
This operation uses a sorting algorithm, typically Timsort in Python, which has a time complexity of 0(n log n) for an array
of n elements.
```

The time complexity of the code can be broken down into two major operations:

Since insert is called for each person in the sorted array and the list can grow up to n elements, this operation will have a worst-case complexity of  $O(n^2)$  as it performs n insertions into a structure that can be up to n in size.

Reconstructing the queue by inserting elements:

the dominating term when n is large. **Space Complexity** 

The space complexity of the code is the amount of extra space used by the algorithm, not counting the space occupied by the

By combining these, the overall time complexity of the code is  $0(n \log n) + 0(n^2)$ , which simplifies to  $0(n^2)$  since  $0(n^2)$  is

The insert operation within a list has a time complexity of O(n) in the worst case because it may require shifting the

```
input itself. In this case:
   Sorted people array:
```

Since the sorting operation is done in-place, it does not contribute to additional space complexity beyond what is taken by the input people array.

This list is created to store the output and grows to contain n elements, which are the same elements from the input people

array. Thus, the space required is O(n).

Therefore, the overall space complexity of the code is O(n).