2418. Sort the People

Easy Array Hash Table String Sorting

Problem Description

In this problem, we are given two arrays: `names` and `heights`. Both arrays have the same length, `n`, and for any given index `i`, `names[i]` represents the name of a person, and `heights[i]` represents their respective height. The `heights` array is made up of distinct positive integers, which means no two people can have the same height.

Our task is to sort the people by their heights in descending order, i.e., from the tallest to the shortest person. The output should be a reordered `names` array that reflects this sorting by height.

Understanding the problem involves recognizing that we need to maintain the association between each person's name and their height while sorting, which is commonly known as a "sort by keys" problem where the heights serve as keys, and the names are the values associated with those keys.

Intuition

The solution to this problem requires a sorting strategy. Since each person's height is unique (there are no duplicates), we can sort the people based on their heights in descending order. However, rather than sorting `heights` directly, we need to sort indices that can then be used to reorder the `names`.

Here's the intuitive breakdown of the solution:

1. Generate a list of indices `idx` from `0` to `n-1` where `n` is the length of the heights array.
2. Sort the `idx` list, but instead of sorting it in the usual ascending order, we use a custom sorting key. The key is a function that takes an index `i` and returns the negative of `heights[i]`. By using the negative value, we effectively sort the indices in order of descending heights since the usual sorting is in ascending order.
3. After sorting the indices in this way, we can create a new list with names in the proper order. We do this by iterating through the sorted `idx` list and fetching the corresponding name from the `names` list for each sorted index.

This approach allows us to sort the `names` list according to the descending order of the heights without losing the association between names and their corresponding heights.

Solution Approach

The implementation of the solution involves a few key steps, and it employs basic list manipulation and sorting techniques in Python. Here's a step-by-step walkthrough of the algorithm and patterns used:

1. **Initialization of index list**: We create a list called `idx` which contains indices from `0` to `n-1`, where `n` is the length of the `heights` array. This is accomplished with the `range` function in Python, which returns a sequence of numbers. We convert it to a list using the built-in `list` function.

   ```
   idx = list(range(len(heights)))
   ```

2. **Custom Sorting**: We want to sort the indices based on the values in `heights` array but in descending order. To do this, we use the `sort` method of the list. The `sort` method is modified by a custom key function, which is passed as a lambda function. The lambda function takes each index `i` and returns `-heights[i]`, which ensures that we are sorting in descending order based on height values.

   ```
   idx.sort(key=lambda i: -heights[i])
   ```

3. **Reordering names by sorted indices**: With the indices sorted in the correct order, the final step is to reorder the `names` according to these indices. We accomplish this by iterating over the `idx` and using a list comprehension to create a new list. For each `i` in `idx`, we fetch `names[i]` and add it to the new list.

   ```
   return [names[i] for i in idx]
   ```

This approach leverages a key aspect of Python's sorting capabilities, where sort is stable and can be customized easily with a key function. By inverting the heights in the key function (using negative heights), we avoid the need to write a custom comparison function; the sort method automatically handles it for us.

This solution is efficient because it does not involve any complicated data structures or algorithms beyond basic list operations and sorting, and it maintains a time complexity of O(n log n), where `n` is the number of people in the list. The custom key function does not significantly affect this complexity, as it simply changes the values used for comparison during the sort process.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have the following lists of names and heights:

```
names = ['Alice', 'Bob', 'Charlie', 'Dana']
heights = [155, 165, 160, 170]
```

We can see that Dana is the tallest, followed by Bob, then Charlie, and finally Alice. We want to sort the `names` array so that it matches the descending order of `heights`. Following the steps in the solution approach:

1. **Initialization of index list**: We create a list of indices which will correspond to the indices of the `heights` list.

   ```
   idx = list(range(len(heights)))  # idx will be [0, 1, 2, 3]
   ```

2. **Custom Sorting**: Now we sort `idx` by the heights in descending order by using the negative heights as the key for sorting.

   ```
   idx.sort(key=lambda i: -heights[i])  # After sorting, idx will be [3, 1, 2, 0]
   ```

   The index `3` comes first as `heights[3]` (which is 170) is the greatest, and so on.

3. **Reordering names by sorted indices**: Using our sorted `idx`, we reorder `names` by mapping each index to the corresponding name.

   ```
   sorted_names = [names[i] for i in idx]  # This gives us ['Dana', 'Bob', 'Charlie', 'Alice']
   ```

Now, when we look at `sorted_names`, we have the names ordered by descending height:

```
print(sorted_names)  # Output: ['Dana', 'Bob', 'Charlie', 'Alice']
```

This matches our expectations based on the heights. The `sorted_names` list is the resultant list where the corresponding names are ordered from the tallest to the shortest person. This example clearly demonstrates the efficacy of the solution approach for the problem described.

Solution Implementation

Python

```python
from typing import List

class Solution:
    def sortPeople(self, names: List[str], heights: List[int]) -> List[str]:
        # This method sorts the people based on descending order of their heights.
        # It then returns a list of names of the people sorted by their heights.

        # Generate an index list that matches the indexes of the heights list.
        indices = list(range(len(heights)))

        # Sort the indices list based on descending values of heights.
        # The lambda function returns the negated height value for each index, sorting in descending order.
        indices.sort(key=lambda i: -heights[i])

        # Map the sorted indices to their corresponding names.
        # This results in a list of names ordered by descending height.
        sorted_names = [names[i] for i in indices]

        # Return the list of sorted names.
        return sorted_names

# Example usage:
# If we have two lists, one of names and one of the corresponding heights:
# names = ["Mary", "John", "Emma"]
# heights = [180, 165, 165]
# solution = Solution()
# sorted_people = solution.sortPeople(names, heights)
# The output will be: ["John", "Emma", "Mary"]
```

Java

```java
import java.util.Arrays;

class Solution {
    public String[] sortPeople(String[] names, int[] heights) {
        // Extract the number of elements from the names array, which also applies to the heights array.
        int numberOfPeople = names.length;

        // Create an array of indices representing each person.
        Integer[] indices = new Integer[numberOfPeople];

        // Initialize the indices array with values from 0 to numberOfPeople - 1.
        for (int i = 0; i < numberOfPeople; ++i) {
            indices[i] = i;
        }

        // Sort the indices array based on the descending order of heights using a custom comparator.
        // The comparator computes the difference between the heights at indices j and i.
        Arrays.sort(indices, (i, j) -> heights[j] - heights[i]);

        // Create a result array to store the sorted names based on the heights.
        String[] sortedNames = new String[numberOfPeople];

        // Build the result array by mapping sorted indices to their respective names.
        for (int i = 0; i < numberOfPeople; ++i) {
            sortedNames[i] = names[indices[i]];
        }

        // Return the array of names sorted by descending heights.
        return sortedNames;
    }
}
```

C++

```cpp
#include <algorithm> // include the algorithm header for std::sort and std::iota functions
#include <numeric>   // include the numeric header for std::iota function
#include <string>    // include the string header for using std::string
#include <vector>    // include the vector header for using std::vector

class Solution {
public:
    // This function sorts people by their heights in descending order
    // Arguments:
    //   names - a vector of strings representing people's names
    //   heights - a vector of integers representing corresponding people's heights
    // Returns:
    //   A vector of strings rearresting people's names sorted by their heights in descending order
    vector<string> sortPeople(vector<string>& names, vector<int>& heights) {
        int numPeople = names.size(); // get the number of people
        vector<int> indices(numPeople); // create a vector to hold indices
        iota(indices.begin(), indices.end(), 0); // fill the vector with consecutive numbers starting at 0

        // Use sort function with custom comparator to sort the indices based on descending order of heights
        sort(indices.begin(), indices.end(), [&](int i, int j) {
            return heights[i] > heights[j];
        });

        vector<string> sortedNames; // create a vector to hold the sorted names
        sortedNames.reserve(numPeople); // reserve space in the vector for numPeople elements

        // Construct the sorted names vector based on the sorted indices
        for (int index : indices) {
            sortedNames.push_back(names[index]); // add name to the result vector in sorted order
        }

        return sortedNames; // return the sorted names
    }
};
```

TypeScript

```typescript
function sortPeople(names: string[], heights: number[]): string[] {
    // Get the number of people from the length of the names array.
    const numberOfPeople = names.length;
    // Initialize an index array to keep track of original indices.
    const indices = new Array(numberOfPeople);
    // Fill the index array with indices from 0 to numberOfPeople - 1.
    for (let i = 0; i < numberOfPeople; ++i) {
        indices[i] = i;
    }
    // Sort the index array based on heights in descending order.
    // The sort compares the heights at the indices and sorts the indices array.
    indices.sort((index1, index2) => heights[index2] - heights[index1]);
    // Initialize an array to store the sorted names.
    const sortedNames: string[] = [];
    // Iterate over the sorted indices and push the corresponding name
    // into the sortedNames array.
    for (const index of indices) {
        sortedNames.push(names[index]);
    }
    // Return the array of sorted names.
    return sortedNames;
}
```

```python
from typing import List

class Solution:
    def sortPeople(self, names: List[str], heights: List[int]) -> List[str]:
        # This method sorts the people based on descending order of their heights.
        # It then returns a list of names of the people sorted by their heights.

        # Generate an index list that matches the indexes of the heights list.
        indices = list(range(len(heights)))

        # Sort the indices list based on descending values of heights.
        # The lambda function returns the negated height value for each index, sorting in descending order.
        indices.sort(key=lambda i: -heights[i])

        # Map the sorted indices to their corresponding names.
        # This results in a list of names ordered by descending height.
        sorted_names = [names[i] for i in indices]

        # Return the list of sorted names.
        return sorted_names

# Example usage:
# If we have two lists, one of names and one of the corresponding heights:
# names = ["Mary", "John", "Emma"]
# heights = [180, 165, 165]
# solution = Solution()
# sorted_people = solution.sortPeople(names, heights)
# The output will be: ["John", "Emma", "Mary"]
```

Time and Space Complexity

Time Complexity

The time complexity of the given code is primarily determined by the sorting operation. The `sort` method on the `idx` list is using a custom key, which is based on the values of the `heights` list. Sorting in Python is implemented with the Timsort algorithm, which has a time complexity of `O(n log n)` in the average and worst case, where `n` is the number of elements to be sorted.

In the given code, the list `idx` has a length equal to that of `names` and `heights`, so there will be `n` items to sort, where `n` represents the length of either `names` or `heights`.

Hence, the overall time complexity is `O(n log n)`.

Space Complexity

The space complexity of this function can be analyzed by looking at the additional memory used by the program.

1. The list `idx` is created which will hold `n` integers. This contributes `O(n)` space.
2. The sorting operation may temporarily require additional space to hold elements when sorting, contributing an additional `O(log n)` space due to the recursion stack of Timsort.
3. After sorting, a new list comprehension is used to generate the result list based on the sorted indices. This list will also be of length `n`, which means an additional `O(n)` space.

As the dominant term in space complexity is `O(n)` (the space used for storing indices and the final output list), we can conclude the overall space complexity of the given code is `O(n)`.