1985. Find the Kth Largest Integer in the Array

Medium Array String Divide and Conquer Quickselect Sorting Heap (Priority Queue) Leetcode Link

Problem Description

You are given an array of strings called `nums` where each string is a non-negative integer without leading zeros. Alongside this array, you're given an integer `k`. The task is to determine the `kth` largest integer in the array, **when the integers are considered in their numeric value rather than their string order**, and return it as a string.

The problem stipulates that duplicate numbers should count as separate entities in the ordering. For example, if the given array contains multiple instances of the same number, each instance should be considered separately in the ranking of largest numbers. Essentially, every string is a distinct entry when determining the `kth` largest number, even if its numeric value is identical to that of another string.

It's important to remember that integers represented as longer strings are intrinsically larger than shorter strings when comparing numeric values, regardless of the characters within those strings. For instance, "100" is larger than "99" simply because it represents a larger integer, even though "9" comes after "1" in lexicographic order.

Intuition

The intuition behind the solution is based on custom sorting. Since we're dealing with strings representing numbers, we can't directly sort them as strings because the lexicographic sorting would yield incorrect results for numeric comparisons. For example, "30" would come before "12" in lexicographic order, even though it's numerically larger.

Therefore, we employ a custom comparison function that first compares the lengths of the strings. If two strings have different lengths, the longer one represents a larger number and should come first in a descending sort. When the string lengths are equal, we compare them lexicographically, as numeric equality in string form means the comparative order of digits will determine the larger number.

The `cmp_to_key` method from the `functools` module in Python is used to convert this custom comparator into a sorting key. With this method, we use Python's built-in `sort` function which sorts based on the comparisons defined in our custom function, ensuring that the largest numbers (highest numeric values) come first in the array.

Once the array is sorted in descending numerical order, retrieving the `kth` largest number is straightforward: we access the element at the position `k - 1` since arrays are zero-indexed in Python. The resulting element, which is still in string form, is our desired solution and is returned directly.

Solution Approach

The solution provided uses Python's built-in sorting algorithm with a custom comparison function to address the need for a numerical rather than lexicographical order of the strings in the `nums` array.

The solution is outlined as follows:

1. Define a custom comparator `cmp` that takes two strings `a` and `b`:
   - The comparator first checks the lengths of the two strings. If they have different lengths, the function returns the difference between the lengths of `b` and `a` to ensure that the longer string is considered larger (since we want a descending sort).
   - If both strings are of the same length, lexicographical comparison is used to determine which one is larger. It returns `1` if `b` is greater than `a`, and `-1` if `b` is less than or equal to `a`.
2. Use the `cmp_to_key` function from the `functools` module to convert the custom comparator `cmp` to a key function. This key function is then used with the `sort` method on `nums`. The `sort` method will internally use the key function to perform comparisons between elements during the sort process.
3. With the array now sorted in descending numerical order by the values represented by the strings, the solution directly accesses the `k-1` indexed element to account for zero-based indexing.

Here is how the custom comparison and sorting work out in code:

```python
def cmp(a, b):
    if len(a) != len(b):
        return len(b) - len(a)
    return 1 if b > a else -1

nums.sort(key=cmp_to_key(cmp))
```

The `cmp` function is critical in achieving the desired order without converting strings to integers, enabling efficient sorting even for very large numbers.

After correctly sorting `nums`, the `kth` largest number is simply retrieved using:

```python
return nums[k - 1]
```

By ensuring all steps adhere to the problem's requirements, the solution successfully retrieves the `kth` largest integer in its string representation from `nums` array.

Example Walkthrough

Let's walk through a small example to illustrate the given solution approach.

Imagine we are given an array of strings `nums = ["3", "123", "34", "30", "5"]` representing integers, and we want to find the `2nd` largest integer among these. We will follow the steps outlined in the solution approach.

1. First, we need to construct the custom comparator. The `cmp` function defined will compare the strings based on their lengths to decide which string represents a larger integer, and if the lengths are equal, it will compare them lexicographically.
2. We utilize the `cmp_to_key` method from the `functools` module to use this comparator in the sorting process. With the sorting key ready, we will apply the `sort` method to `nums`.
3. After sorting, the array of strings should reflect the correct numerical order, descending from the largest number. The `nums` array would look like this after sorting: `["123", "34", "30", "5", "3"]`.
4. Finally, to get the `2nd` largest number, we select the element at index `k - 1`, which is `1` in our zero-indexed array (since `k = 2`). The element at index `1` is `"34"`, which is the `2nd` largest number in our array.

Here is how the example would be implemented in Python based on our solution approach:

```python
from functools import cmp_to_key

# Define the comparator function
def cmp(a, b):
    if len(a) != len(b):
        return len(b) - len(a)  # Strings with longer length are numerically bigger
    return 1 if b > a else -1  # For equal lengths, compare lexicographically

# Example array and k value
nums = ["3", "123", "34", "30", "5"]
k = 2

# Sort the strings using the custom comparator converted to a key
nums.sort(key=cmp_to_key(cmp))

# Getting the kth largest number in its string representation
kth_largest = nums[k - 1]

print(kth_largest)  # Output should be "34"
```

The output of the program is `"34"`, which is what we expected for the `2nd` largest number. The custom sorting ensures we are directly comparing the numbers not just as strings but as numerical values they represent, adhering to the desired outcome of the given task.

Python Solution

```python
from functools import cmp_to_key

class Solution:
    def kthLargestNumber(self, nums: List[str], k: int) -> str:
        # Define a comparison function to use in sorting.
        # This function will compare two numbers based on their length first,
        # and then by their value if the lengths are equal.
        def compare_numbers(num1: str, num2: str) -> int:
            if len(num1) != len(num2):
                # Sort primarily by length of the string representation of the numbers.
                return len(num2) - len(num1)
            else:
                # If lengths are equal, sort by the string representation itself.
                # -1 if num1 should come before num2, 1 if num2 should come before num1.
                return -1 if num2 > num1 else 1

        # Sort the list of numbers using our custom comparison function.
        nums.sort(key=cmp_to_key(compare_numbers))

        # Return the k-th largest number as a string.
        # Since the list is sorted from largest to smallest,
        # the k-th largest number is at index k-1.
        return nums[k - 1]

```

Java Solution

```java
import java.util.Arrays; // Import necessary class for sorting

class Solution {

    // Method to find the kth largest number in the form of a string from a given array of strings
    public String kthLargestNumber(String[] nums, int k) {
        // Sort the array using a custom comparator
        Arrays.sort(nums, (firstNumber, secondNumber) -> {
            // If the lengths of numbers are equal, compare them lexicographically in descending order
            if(firstNumber.length() == secondNumber.length()) {
                return secondNumber.compareTo(firstNumber);
            } else {
                // Sort based on length of strings to handle large numbers accurately
                // Longer numbers should come first since they are larger
                return secondNumber.length() - firstNumber.length();
            }
        });

        // Return the kth element in the sorted array which is the kth largest number
        // (k - 1) is used because array indices start from 0
        return nums[k - 1];
    }
}

```

C++ Solution

```cpp
#include <string>   // Include the string library
#include <vector>   // Include the vector library
#include <algorithm> // Include the algorithms library for the sort function

// Define the Solution class with the public member function 'kthLargestNumber'
class Solution {
public:
    // Define the 'kthLargestNumber' function that returns the k-th largest number from a vector of strings
    string kthLargestNumber(vector<string>& nums, int k) {
        // Define a custom comparator lambda function to sort the numbers based on length and value
        auto comparator = [](const string& a, const string& b) {
            // Compare sizes first. If sizes are equal, compare strings lexicographically
            return a.size() == b.size() ? a > b : a.size() > b.size();
        };

        // Sort the vector of strings using the custom comparator
        sort(nums.begin(), nums.end(), comparator);

        // Return the k-th largest number, which is at index k-1 after sorting
        return nums[k - 1];
    }
};

```

Typescript Solution

```typescript
// Import necessary functionalities from util libraries (not applicable in TypeScript as it's often executed in a browser or Node.js
// However, TypeScript has built-in sort functionality for arrays

// Define the 'kthLargestNumber' function that returns the k-th largest number from an array of strings
function kthLargestNumber(nums: string[], k: number): string {
    // Define a custom comparator function to sort the numbers based on length and value
    const comparator = (a: string, b: string): number => {
        // Compare sizes first. If sizes are equal, compare strings lexicographically
        if (a.length === b.length) {
            return b.localeCompare(a);
        }
        return b.length - a.length;
    };

    // Sort the array of strings using the custom comparator
    nums.sort(comparator);

    // Return the k-th largest number, which is at index k-1 after sorting
    return nums[k - 1];
}

// The TypeScript function can be used as follows:
// let result = kthLargestNumber(["3", "6", "2", "10"], 2); // result would be "6"

```

Time and Space Complexity

Time Complexity

The given code defines a custom comparator and sorts a list of strings representing numbers based on their numeric values. The time complexity of this algorithm is dominated by the sorting step.

The `sort()` function in Python typically has a time complexity of `O(n log n)` where `n` is the number of elements in the list. However, because a custom comparator is used, there is an additional overhead of comparing each pair of strings. The comparison involves checking the length of the strings and possibly comparing the strings themselves.

The worst-case time complexity for comparing two strings is `O(m)` where `m` is the length of the longer string among the two being compared.

Thus, considering both the sorting of `n` elements and the custom comparison, the total time complexity is `O(n * m * log n)`, where `n` is the length of the input list `nums` and `m` is the maximum length of a string within `nums`.

Space Complexity

The space complexity is mainly due to the space required for sorting the strings. Python's sort function can be `O(n)` in the worst case for space, since it may require allocating space for the entire list. Moreover, since we are dealing with strings, the space complexity does not only depend on the number of elements `n`, but also on the total space required to hold all of the strings.

If we let `k` be the total amount of space required to store all strings in the list (sum of the lengths of all strings), the overall space complexity will be `O(k)`.

The other space overhead in the solution is the space required for the function `cmp_to_key(cmp)`. This is a function object that consumes constant space, so it does not significantly impact the overall space complexity, which remains `O(k)`.