2343. Query Kth Smallest Trimmed Number

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

Problem Description

In this LeetCode problem, we're given an array of strings called `nums`, with each string representing a number with leading zeros if required so that all numbers are of the same length. Along with this array, we're given a 2D array of queries. Each query is a pair `[k, trim]`. For each query, we need to do the following:

1. Trim each number in the `nums` array to keep only the rightmost `trim` digits. This is equivalent to removing digits from the beginning (left side) of the string until we are left with `trim` number of digits.
2. Find the `k`th smallest number in this trimmed list. If two numbers are the same, the one whose index is smaller in the original `nums` array is considered the smaller one.
3. After the answer is determined for each individual query, we reset each number in the `nums` array to its original value before proceeding with the next query.

The result is an array of indices indicating the positions of the `k`th smallest trimmed numbers in the original `nums` array for each query.

Intuition

The given problem can be approached by looking at each query and processing the `nums` array based on that query. This involves transforming the original array into a form where it can be sorted, and then picking the `k`th smallest element.

Here's the intuitive breakdown of the solution:

1. For each query, begin by trimming all the numbers in `nums`. This task involves taking the last `trim` digits of each number.
2. After the array is trimmed, each number along with its original index must be kept together so that after sorting, we can refer back to where the number came from. Tuple pairs can be used for this, with the trimmed number first for sorting purposes, followed by the original index.
3. We sort the collection of these pairs to arrange the trimmed numbers in ascending order along with their original indices. Sorting in Python is stable, which means that if two elements are equal, their order relative to each other will remain the same as it was in the input.
4. Once the array is sorted, we simply pick the element that is at the `k-1` position (due to the 0-indexing) which corresponds to the `k`th smallest number along with its index.
5. This index is added to the result list, which is returned at the end after processing all queries.

Utilizing the sort functionality of Python in this way allows us to address the problem in a straightforward manner.

Solution Approach

The implementation follows directly from the intuitive approach that was discussed. Here's a step-by-step walkthrough using the provided Python solution:

1. Initializing an empty list named `ans` to store the indices of the `k`th smallest trimmed numbers for each query.

2. Looping through each `[k, trim]` pair in the `queries` list:

   - The key operation in the loop is to create a list of tuples for each number in `nums`. The tuple consists of the trimmed number (`v[-trim:]`) and its original index (`i`). This is facilitated by Python's list comprehension and slicing features.
   - For the trimming operation, `v[-trim:]` is used to get the substring of `v` from the `-trim` position (counting from the right) to the end. This essentially gives us the rightmost `trim` digits of each number.
   - Along with the trimmed number, the original index is also stored in the tuple to keep track of its position in the original `nums` array.

3. The list of tuples is then sorted. As mentioned earlier, Python's sorting is stable, so if two trimmed numbers are equal, their relative order will remain the same and thus the number from the lower index in the original list will be deemed smaller. The sorting algorithm typically used in Python is Timsort, which is efficient for the kind of partial orderings that appear often in practice.

4. After the list is sorted, the `k-1` indexed tuple in the sorted list gives us the `k`th smallest number because Python uses 0-based indexing. From this tuple, `[1]` is used to extract the second element, which is the original index of the trimmed number in the `nums` array.

5. The extracted index is appended to the `ans` list.

6. This process is repeated for each query, and the original `nums` array remains unaltered as we only use substrings and do not modify the original array.

7. After processing all queries, the `ans` list is returned, containing the indices of the `k`th smallest trimmed number for each query.

The solution's time complexity is dominated by the sorting step inside each query loop. If `n` is the length of `nums` and `m` is the length of `queries`, the expected time complexity is O(m * n * log(n)), with additional space complexity of O(n) for the list of tuples created during each query.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose `nums` is `["102","473","251","814"]` and the queries array is `[[2,1],[1,2]]`.

1. For the first query `[2,1]`, trim value is `1`. We trim each element in `nums` to the last digit:

   - `102` becomes `2`
   - `473` becomes `3`
   - `251` becomes `1`
   - `814` becomes `4` The array of trimmed numbers paired with their indices is `[('2', 0), ('3', 1), ('1', 2), ('4', 3)]`.

2. We then sort these pairs to get `[('1', 2), ('2', 0), ('3', 1), ('4', 3)]`.

3. The second smallest number after trimming is at index `0` because the pair `('2', 0)` is the second element. Thus, for the first query, the result is `0`.

4. For the next query `[1,2]`, the trim value is `2`, so we don't actually trim since all `nums` elements are already of length `2` or smaller.

5. The sorted array of numbers paired with their indices is `[('02', 0), ('14', 3), ('51', 2), ('73', 1)]`.

6. The smallest number (after trimming to two digits, which in this case changes nothing) is at index `0`. So, for the second query, the result is `0`.

As a result, after processing both queries, we get an answer array of `[0, 0]`.

This walkthrough matches the steps provided in the solution approach:

1. A list of tuples is created for each number and its index with the numbers trimmed to the correct size.
2. The list of tuples is sorted based on the trimmed numbers.
3. The index of the `k`th smallest trimmed number is then found and appended to the result list.
4. This process is repeated for each query without altering the original `nums` array.

Hence the final indices of the `k`th smallest trimmed numbers for each query are determined and returned in a list.

Python Solution

```python
from typing import List

class Solution:
    def smallestTrimmedNumbers(self, numbers: List[str], queries: List[List[int]]) -> List[int]:
        # Initialize an empty list to store the answer
        answer = []

        # Iterate through each query in the list of queries
        for k, trim in queries:
            # Create a sorted list of tuples where each tuple contains
            # a trimmed number (last 'trim' characters) and its original index
            trimmed_sorted_list = sorted((num[-trim:], index) for index, num in enumerate(numbers))

            # From the sorted list, select the k-th smallest trimmed number
            # (as sorting starts with 0, we use k-1 as the index)
            # and append its original index to the answer list
            answer.append(trimmed_sorted_list[k - 1][1])

        # Return the list of original indices of the k-th smallest trimmed numbers
        return answer

```

Java Solution

```java
import java.util.Arrays;

class Solution {

    public int[] smallestTrimmedNumbers(String[] nums, int[][] queries) {
        // Get the number of strings in the nums array and the total number of queries.
        int numOfStrings = nums.length;
        int numOfQueries = queries.length;

        // Create an array to store the answers for each query.
        int[] answers = new int[numOfQueries];

        // Initialize a 2D array to store both the trimmed string and its original index.
        String[][] trimmedAndIndices = new String[numOfStrings][2];

        // Iterate over each query
        for (int i = 0; i < numOfQueries; ++i) {
            // Extract the k-th value and the trim length from the current query.
            int k = queries[i][0];
            int trimLength = queries[i][1];

            for (int j = 0; j < numOfStrings; ++j) {
                // Trim each string in nums from the end by the given trim length and store the result along with the original index.
                trimmedAndIndices[j] = new String[] {
                    nums[j].substring(nums[j].length() - trimLength),
                    String.valueOf(j)
                };
            }

            // Sort the array of trimmed strings and indices.
            Arrays.sort(trimmedAndIndices, (a, b) -> {
                // Compare trimmed strings.
                int comparison = a[0].compareTo(b[0]);
                // If equal, compare their original indices.
                return comparison == 0 ? Integer.compare(Integer.parseInt(a[1]), Integer.parseInt(b[1])) : comparison;
            });

            // Get the index of the k-th smallest trimmed string.
            answers[i] = Integer.parseInt(trimmedAndIndices[k - 1][1]);
        }

        // Return the array of answers.
        return answers;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

class Solution {
public:
    // Function to return the indices of the smallest trimmed numbers for each query
    vector<int> smallestTrimmedNumbers(vector<string>& numbers, vector<vector<int>>& queries) {
        int numOfNumbers = numbers.size(); // Total number of strings in the numbers vector
        vector<pair<string, int>> trimmedNumbers(numOfNumbers); // Pair to hold trimmed strings and original indices
        vector<int> answer; // Vector to hold the final results

        // Iterate through each query
        for (auto& query : queries) {
            int k = query[0];  // kth smallest number to find after trimming
            int trimLength = query[1]; // Length of the number to consider after trimming

            // Prepare the trimmed numbers along with their original indices
            for (int i = 0; i < numOfNumbers; ++i) {
                // Trim the number keeping the last 'trimLength' digits and store the original index
                trimmedNumbers[i] = {numbers[i].substr(numbers[i].size() - trimLength), i};
            }

            // Sort the trimmed numbers, (since we're using pairs, it sorts by the first element (trimmed string), and uses the secor
            sort(trimmedNumbers.begin(), trimmedNumbers.end());

            // Add the original index of the kth smallest trimmed number to the answer vector
            answer.push_back(trimmedNumbers[k - 1].second);
        }

        // Return the final result after processing all queries
        return answer;
    }
};

```

Typescript Solution

```typescript
// In TypeScript, we can simply use arrays and strings, and TypeScript has built-in types for these.

// Function to return the indices of the smallest trimmed numbers for each query
function smallestTrimmedNumbers(numbers: string[], queries: number[][]): number[] {
    const numOfNumbers = numbers.length; // Total number of strings in the numbers array
    const trimmedNumbers: { trimmed: string; originalIndex: number }[] = []; // Array to hold trimmed strings and original indices
    const answer: number[] = []; // Array to hold the final results

    // Iterate through each query
    for (const query of queries) {
        const k = query[0];  // kth smallest number to find after trimming
        const trimLength = query[1]; // The number of characters to consider from the end of the string after trimming

        // Prepare the trimmed numbers along with their original indices
        for (let i = 0; i < numOfNumbers; ++i) {
            // Trim the number, keeping the last 'trimLength' characters, and store along with the original index
            trimmedNumbers[i] = {
                trimmed: numbers[i].slice(-trimLength),
                originalIndex: i
            };
        }

        // Sort the trimmed numbers (the array gets sorted by the 'trimmed' property, and uses the 'originalIndex' for tie-breaking)
        trimmedNumbers.sort((a, b) =>
            a.trimmed !== b.trimmed
                ? a.trimmed.localeCompare(b.trimmed)
                : a.originalIndex - b.originalIndex
        );

        // Add the original index of the kth smallest trimmed number to the answer array
        answer.push(trimmedNumbers[k - 1].originalIndex);
    }

    // Return the final result after processing all queries
    return answer;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the solution consists of two main operations:

1. Trimming and sorting the strings: For each query, we trim the strings to the last `trim` characters and sort them. This operation has a complexity of `O(n * m * log(n))`, where `n` is the length of `nums` and `m` is the length of the trimmed string. The `m` factor comes from the time to create the substring for each number, and `log(n)` is for the sorting.

2. The loop runs for each query, adding a factor of the number of queries `q`.

Thus, the total time complexity is `O(q * n * m * log(n))`.

Space Complexity

Space complexity pertains to the extra space required by the algorithm. Here's what it includes:

1. The trimmed and tuples list `t`, which has a size `O(n)` for each query since we store the trimmed strings and their original indices.

2. The answer list `ans` that holds one value for each query, giving it a size of `O(q)`.

Thus, the total space complexity would be `O(n + q)`.