2053. Kth Distinct String in an Array

Easy | Array | Hash Table | String | Counting

Leetcode Link

Problem Description

In this problem, we are given an array of strings, `arr`, where we need to identify strings that appear exactly once in the array, which we refer to as "distinct strings." Our goal is to find the `k`th distinct string in the array, considering the order in which the strings appear. If the number of distinct strings in the array is less than `k`, we should return an empty string `""`. Essentially, the problem is asking us to process the array and extract a specific element based on its distinctness and order of occurrence.

Intuition

The solution for this problem involves two steps:

1. Counting the occurrence of each string in the array.
2. Iterating through the array to find the `k`th string that occurs exactly once.

To efficiently count occurrences, we use a data structure known as a **counter** (which can be provided by Python's `collections.Counter` class). This counter keeps track of how many times each string appears in the array.

Once we have the occurrences counted, the next step is to iterate through the array while keeping track of the number of distinct strings encountered so far. A string is considered distinct if its counted occurrence is equal to one. We sequentially check each string's occurrence count, decreasing `k` each time we find a distinct string.

When `k` becomes 0, that means we've encountered the `k`th distinct string and can return it immediately. If the end of the array is reached and `k` has not reached 0, we return an empty string because there aren't enough distinct strings in the array.

Solution Approach

The solution is implemented in Python and follows these steps:

1. **Counting Occurrences**: We first create a `counter` object from Python's `collections.Counter` class to count the occurrences of every string in the array `arr`. The `Counter` class generates a dictionary-like object where each key is a unique string from `arr`, and the corresponding value is the count of that string's occurrences.

```
counter = Counter(arr)
```

2. **Finding the kth Distinct String**: We then iterate over the original array `arr` since we need to respect the order of strings. For each string `v` in `arr`, we look at its count in the `counter`.

```
for v in arr:
    if counter[v] == 1:
```

If the count is 1, it signifies that `v` is a distinct string. We decrement `k` for each distinct string found.

3. **Checking the kth Position**: If during iteration `k` becomes 0, this implies that we have found the `k`th distinct string, and we immediately return this string `v`.

```
        k -= 1
        if k == 0:
            return v
```

4. **Returning an Empty String**: If the loop finishes and no string has made `k` reach 0, this means that there are fewer than `k` distinct strings in the array. Hence, the function returns an empty string `''`.

```
return ''
```

This implementation is efficient because it traverses the list only once to count the elements and a second time to find the kth distinct element. The counter object provides an O(1) access time to find an element's count, ensuring that the solution is linear with respect to the size of the input array, which is optimal for this problem.

Example Walkthrough

Let's illustrate the solution approach with a small example. Imagine we are given the following array of strings `arr` and we want to find the `2nd` distinct string:

```
arr = ["apple", "banana", "apple", "orange", "banana", "kiwi"]
```

Counting Occurrences: First, we use the counter to count the occurrences of each string:

```
counter = Counter(arr)  # {'apple': 2, 'banana': 2, 'orange': 1, 'kiwi': 1}
```

Finding the kth Distinct String: The counter tells us that "apple" and "banana" are not distinct (both appear twice). However, "orange" and "kiwi" are distinct (each appears once). As we wish to find the 2nd distinct string, we start iterating through `arr`:

- We encounter "apple" first. Its occurrence count is 2, so it's not distinct.
- We move to "banana" with the same result as "apple".
- Next is "apple" again, still not distinct.
- Then we encounter "orange", which is distinct since its count is 1.
  - We set `k` to 2 initially. Now we decrement `k` to 1 as we have found our 1st distinct string.
- We move on to "banana" once more, which is also not distinct.
- Lastly, we find "kiwi", which has a count of 1 and is therefore distinct.
  - We decrement `k` again and now `k` is 0, which means "kiwi" is our 2nd distinct string.

Checking the kth Position: Since we found the 2nd distinct string and `k` is now `0`, we return "kiwi".

If instead `k` was set to `3` initially, after going through the array, we would still be left with `k` equals `1`, meaning there wasn't a 3rd distinct string. In that case, we'd return an empty string `""`.

Returning an Empty String: Since in this example there are only 2 distinct strings and we found the 2nd, there's no need to return an empty string. If we were looking for the 3rd distinct string which does not exist in our `arr`, our result would be `""`.

By following this method, we call the Counter class once to build our occurrence dictionary and then iterate through the array only once more, making this a very efficient way to solve the problem.

Python Solution

```
from collections import Counter  # Import the Counter class from collections module

class Solution:
    def kthDistinct(self, arr: List[str], k: int) -> str:
        # Create a counter for all items in arr
        # Counter will store words as keys and their occurrences as values
        occurrence_counter = Counter(arr)

        # Iterate over each word in arr
        for word in arr:
            # Check if the current word occurs exactly once
            if occurrence_counter[word] == 1:
                # Decrement k as we've found one distinct word
                k -= 1
                # If k reaches 0, we've found the kth distinct word
                if k == 0:
                    return word

        # If the kth distinct word is not found, return an empty string
        return ''
```

Java Solution

```
class Solution {

    // Method to find the k-th distinct string in the array
    public String kthDistinct(String[] arr, int k) {
        // Create a HashMap to store the frequency of each string
        Map<String, Integer> frequencyMap = new HashMap<>();

        // Count the occurrences of each string in the array
        for (String element : arr) {
            frequencyMap.put(element, frequencyMap.getOrDefault(element, 0) + 1);
        }

        // Iterate over the array to find the k-th distinct string
        for (String element : arr) {
            // If the frequency of the string is 1, it is distinct
            if (frequencyMap.get(element) == 1) {
                k--; // Decrement k for each distinct string found

                // If k reaches zero, we found the k-th distinct string
                if (k == 0) {
                    return element;
                }
            }
        }

        // If k distinct strings are not found, return an empty string
        return "";
    }
}
```

C++ Solution

```
#include <string>
#include <vector>
#include <unordered_map>
using namespace std;

class Solution {
public:
    // Function to find the k-th distinct string in the array.
    string kthDistinct(vector<string>& arr, int k) {
        // Create a hash map to store the frequency of each string.
        unordered_map<string, int> frequencyMap;

        // Count the frequency of each string in the array.
        for (const string& value : arr) {
            ++frequencyMap[value];
        }

        // Iterate through the array to find the k-th distinct string.
        for (const string& value : arr) {
            // Check if the current string is distinct (frequency is 1).
            if (frequencyMap[value] == 1) {
                // Decrement k and check if we have found the k-th distinct string.
                --k;
                if (k == 0) {
                    // If k reaches 0, the current string is the k-th distinct string.
                    return value;
                }
            }
        }

        // If the k-th distinct string is not found, return an empty string.
        return "";
    }
};
```

Typescript Solution

```
// Importing required types for TypeScript
import { string } from "prop-types";

// Function to find the k-th distinct string in the array.
function kthDistinct(arr: string[], k: number): string {
    // Create a map to store the frequency of each string.
    const frequencyMap: Record<string, number> = {};

    // Count the frequency of each string in the array.
    for (const value of arr) {
        // Increase the frequency count for the string in the map.
        frequencyMap[value] = (frequencyMap[value] || 0) + 1;
    }

    // Iterate through the array to find the k-th distinct string.
    for (const value of arr) {
        // Check if the current string is distinct (frequency is 1).
        if (frequencyMap[value] === 1) {
            // Decrement k and check if we have found the k-th distinct string.
            k--;
            if (k === 0) {
                // If k reaches 0, the current string is the k-th distinct string.
                return value;
            }
        }
    }

    // If the k-th distinct string is not found, return an empty string.
    return "";
}

// Example usage:
// const strings = ["a", "b", "a", "a"];
// const result = kthDistinct(strings, 2); // Should return "b" if called
```

Time and Space Complexity

The given Python code snippet defines a method `kthDistinct` which finds the k-th distinct string in the provided `arr` list. The computational complexity analysis is as follows:

Time Complexity

The time complexity of the code can be broken down into the following steps:

1. **Counter Creation**: `counter = Counter(arr)` creates a counter object which counts the occurrences of each distinct value in `arr`. Constructing this counter takes `O(n)` time, where `n` is the number of elements in `arr`.

2. **Iteration and Checks**: The code then iterates over each value in `arr`, this iteration takes `O(n)` time. Within the loop, it performs a constant-time check `if counter[v] == 1` for each value `v`, which does not affect the overall O(n) time complexity.

Overall, since both steps are sequential, the total time complexity is `O(n) + O(n)` which simplifies to `O(n)`.

Space Complexity

The space complexity of the code also involves two major components:

1. **Counter Storage**: Storing counts of each unique value in `arr` requires `O(m)` space, where `m` is the number of distinct elements in `arr`.

2. **Loop Variables**: The loop variables (`v` and `k`) and the space for storing function arguments use constant `O(1)` space.

Thus, the combined space complexity is `O(m)`.

The markdown results display the formulas within "`" to properly markup the complexity notations.