2261. K Divisible Elements Subarrays

Medium Trie Array Hash Table Enumeration Hash Function Rolling Hash Leetcode Link

Problem Description

The problem provides an integer array named `nums`, along with two integers `k` and `p`. The objective is to find out how many distinct subarrays exist within `nums` such that each subarray contains at most `k` elements divisible by `p`.

Subarrays are contiguous parts of the array, and they have to be non-empty. Distinct subarrays are those that differ in size or have at least one differing element at any position when compared.

It's important to understand:

- What a subarray is: a continuous sequence from the array.
- A distinct subarray: when compared to another, it has a different length or at least one different element at any index.
- The condition to be met: having at most `k` elements divisible by `p` in each subarray.

The challenge is to consider all possible subarrays and count only the unique ones matching the divisibility constraint.

Intuition

The intuition behind the solution comprises two main aspects: iteration and uniqueness tracking.

- **Iteration**: To find all possible subarrays, we iterate through the original array. Starting from each index in `nums`, we add elements one by one to the current subarray and check if they meet the condition (at most `k` elements divisible by `p`).
- **Uniqueness Tracking**: As we keep extending these subarrays, we convert them into a string representation, by concatenating elements separated by commas, in order to record each unique one. This string acts as a unique key for the particular subarray.
- **Break Condition**: An important aspect is knowing when to stop extending a subarray. This is determined by the count of elements divisible by `p`. As soon as we have more than `k` such elements, we break the inner loop and proceed with the next starting index.
- **Set for Uniqueness**: We use a set data structure, since it automatically ensures that only unique subarrays are counted. We add the string representation of each valid subarray (up to `k` divisible elements) to this set.

The key lies in sequential scanning and the use of a set to ensure distinct subarrays are tracked and counted. The iteration ensures that every possible subarray configuration is considered, while the set data structure helps in avoiding duplicates and keeping the count of distinct subarrays only.

Solution Approach

The solution approach uses a brute force method to find all possible subarrays and utilizes a set to store unique subarrays satisfying the condition. To understand the algorithm, let's break down the code:

- We have a nested loop; the outer loop starts with index `i` from `0` to the end of the array.
- For each position in the array, the inner loop adds elements to the subarray and checks if it still satisfies our condition.
- We keep track of the number of elements divisible by `p` using the variable `cnt`.
- To maintain the uniqueness, we create a string `t` to represent the current subarray. As we iterate with the inner loop, we append the current element `x` concatenated with a comma to `t`.
- After appending each element `x`, we check the divisibility condition. If `cnt` exceeds `k`, we stop extending that subarray and break the inner loop.
- If `cnt` is still less than or equal to `k`, we add `t` to our set `s`.
- The set 's' maintains all unique subarrays encountered during the loops.
- Finally, we return the size of the set `s`, which gives us the number of unique subarrays.

Key Components:

- **Brute Force Enumeration**: The solution uses a brute force approach to generate all possible subarrays starting from each index in the original array.
- **Set for Uniqueness**: A set is employed to handle the uniqueness of subarrays, which means any duplicate representation is automatically neglected.
- **String Representation**: Subarrays are represented as strings, which allows for an easy comparison of uniqueness.
- **Breaking Condition**: The algorithm includes a breaking condition within the inner loop that halts further extension of a subarray once it no longer satisfies the divisibility condition.

The time complexity of this algorithm is O(n^3) in the worst case, where `n` is the length of `nums`. This is because we have a nested loop (O(n^2)) and string concatenation within the inner loop (O(n)), which can be quite inefficient for very large arrays. However, for smaller arrays or with constraints on `k`, this approach is sufficient to find the correct answer.

Example Walkthrough

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

Suppose we have the following inputs:

- `nums = [3, 1, 2, 3]`
- `k = 2`
- `p = 3`

Now, we will walk through the algorithm step by step:

1. Initialize a set `s` to keep track of unique subarrays.
2. Initialize an outer loop that starts with index `i` at `0` and goes up to the length of `nums`.
3. For each iteration of the outer loop, we start from the `i-th` element and initialize an inner loop that creates subarrays by adding one element at a time to the current subarray.
4. Inside the inner loop, we have a `cnt` variable to count the number of elements divisible by `p` and a string `t` for the subarray representation.
5. Starting with index `i = 0` and `nums[i] = 3`:
   - For the first element, `cnt` is incremented as `3` is divisible by `p = 3`.
   - The string `t` becomes `"3,"` (representation of the subarray `[3]`), and we add it to set `s`.
6. The inner loop extends the subarray to include the next element `1`, making it `[3, 1]`:
   - `cnt` remains `1` since `1` is not divisible by `3`.
   - The new subarray is represented by `"3,1,"`, and we add it to set `s`.
7. We continue adding elements to the subarray and update `cnt` and `t` accordingly:
   - Next, we include element `2`, resulting in subarray `[3, 1, 2]`. As `2` is not divisible by `3`, `cnt` stays the same. The string `t` is updated to `"3,1,2,"`, and it is added to `s`.
   - Then, we add the last element `3`, forming subarray `[3, 1, 2, 3]`. The `cnt` now becomes `2` since `3` is divisible by `3`. The string `t` is `"3,1,2,3,"`, and it is also added to `s`.
8. At this point, if there were more elements to add, and if adding another element divisible by `p` made `cnt` exceed `k`, the inner loop would break, and we would not add the new subarray representation to `s`.
9. The outer loop moves to the next starting index (`i = 1`) and repeats the process, generating all subarrays starting from `nums[1]`.
10. The algorithm continues iterating this way, ensuring that all subarrays are considered, with only unique subarrays that have at most `k` divisible elements are added to `s`.
11. Once the loops are finished, the distinct subarrays in set `s` will be:

- `"3,"`
- `"3,1,"`
- `"3,1,2,"`
- `"3,1,2,3,"`
- `"1,"`
- `"1,2,"`
- `"1,2,3,"`
- `"2,"`
- `"2,3,"`
- `"3,"` (which already exists and isn't counted again)

Thus, the size of set `s` gives us the count of distinct subarrays, which is `9` in this example.

This walkthrough demonstrates how the brute force approach and set data structure come together to solve the given problem by considering every single possible subarray and maintaining their uniqueness.

Python Solution

```python
class Solution:
    def countDistinct(self, nums: List[int], k: int, p: int) -> int:
        # Initialize the length of 'nums' list
        num_count = len(nums)
        # Initialize a set to store unique sequence tuples
        unique_sequences = set()

        # Iterate through 'nums' list starting at each index
        for start_index in range(num_count):
            # Counter for numbers divisible by 'p'
            divisible_count = 0
            # Temporary string to build sequences
            temp_sequence = ""

            # Iterate through the rest of the list from current start_index
            for number in nums[start_index:]:
                # Increment the divisible count if the number is divisible by 'p'
                divisible_count += number % p == 0
                # If divisible_count exceeds k, break the loop as the condition is not met
                if divisible_count > k:
                    break

                # Append the current number to the sequence, separated by a comma
                temp_sequence += str(number) + ","
                # Add the current sequence to the set of unique sequences
                unique_sequences.add(temp_sequence)

        # Return the number of unique sequences found
        return len(unique_sequences)

```

Java Solution

```java
class Solution {
    public int countDistinct(int[] nums, int k, int p) {
        // Initialize the number of elements in the array
        int n = nums.length;
        // Use a HashSet to store unique subarrays
        Set<String> uniqueSubarrays = new HashSet<>();
        // Loop through the array to start each possible subarray
        for (int i = 0; i < n; ++i) {
            int countDivisibleByP = 0; // Count of elements divisible by p
            StringBuilder subarrayBuilder = new StringBuilder(); // Use StringBuilder to build the string representation of subarrays
            // Loop to create subarrays starting at index i
            for (int j = i; j < n; ++j) {
                // If the current element is divisible by p, increment the count
                if (nums[j] % p == 0) {
                    ++countDivisibleByP;
                }
                // If the count exceeds k, stop considering more elements in the subarray
                if (countDivisibleByP > k) {
                    break;
                }
                // Add the current element to the subarray representation and include a separator
                subarrayBuilder.append(nums[j]).append(",");
                // Add the current subarray representation to the HashSet
                uniqueSubarrays.add(subarrayBuilder.toString());
            }
        }
        // The size of the HashSet gives the count of distinct subarrays
        return uniqueSubarrays.size();
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <unordered_set>
#include <string>

class Solution {
public:
    int countDistinct(vector<int>& nums, int maxDivisibleCount, int p) {
        // Initialize a set to store the unique subarrays.
        unordered_set<string> uniqueSubarrays;

        // Size of the input array.
        int numSize = nums.size();

        // Iterate through the input array to start each subarray.
        for (int startIdx = 0; startIdx < numSize; ++startIdx) {
            // Initialize count for divisible numbers.
            int divisibleCount = 0;

            // Use a string to represent the subarray.
            string subarrayStr;

            // Continue to add elements to the subarray.
            for (int endIdx = startIdx; endIdx < numSize; ++endIdx) {
                // If the current element is divisible by p increment the count.
                if (nums[endIdx] % p == 0 && ++divisibleCount > maxDivisibleCount) {
                    // If the divisible count exceeds 'k', we break out of the inner loop.
                    break;
                }

                // Append the current element and a comma to the subarray string.
                subarrayStr += std::to_string(nums[endIdx]) + ",";

                // Insert the current subarray into the set (this ensures uniqueness).
                uniqueSubarrays.insert(subarrayStr);
            }
        }

        // Return the number of unique subarrays.
        return uniqueSubarrays.size();
    }
};

```

Typescript Solution

```typescript
/**
 * Counts the number of distinct subarrays where the number of elements
 * divisible by `p` does not exceed `k`.
 * @param nums - An array of numbers to be processed.
 * @param k - The maximum allowable number of elements divisible by `p` in a subarray.
 * @param p - The divisor used to determine divisibility of elements in the subarray.
 * @returns The count of distinct subarrays meeting the criteria.
 */
function countDistinct(nums: number[], k: number, p: number): number {
    // Get the length of the input array.
    const numsLength = nums.length;
    // Initialize a Set to store unique subarray representations.
    const uniqueSubarrays = new Set<string>();

    // Iterate over the array to consider different starting points for subarrays.
    for (let startIndex = 0; startIndex < numsLength; ++startIndex) {
        // Initialize count for tracking the number of elements divisible by `p`.
        let divisibleCount = 0;
        // Initialize a temporary string to represent the subarray.
        let subarrayRepresentation = '';

        // Iterate over the array to consider different ending points for subarrays.
        for (let endIndex = startIndex; endIndex < numsLength; ++endIndex) {
            // If the current element is divisible by `p` and we've exceeded `k`, break out of the loop.
            if (nums[endIndex] % p === 0 && ++divisibleCount > k) {
                break;
            }
            // Add the current element to the subarray representation, followed by a comma.
            subarrayRepresentation += nums[endIndex].toString() + ',';
            // Add the current subarray representation to the Set of unique subarrays.
            uniqueSubarrays.add(subarrayRepresentation);
        }
    }

    // Return the size of the Set, which represents the number of unique subarrays.
    return uniqueSubarrays.size;
}

```

Time and Space Complexity

Time Complexity

The given Python code has a nested loop structure where the outer loop runs `n` times (where `n` is the length of the `nums` list) and the inner loop can run up to `n` times in the worst case. For each iteration of the inner loop, the code checks whether the current number is divisible by `p` and breaks the loop if the count of numbers divisible by `p` exceeds `k`.

The `+=` operator on a string inside the inner loop has a time complexity of `O(m)` each time it is executed, where `m` is the current length of the string `t`. Since `t` grows linearly with the number of iterations, the string concatenation inside the loop can have a quadratic effect in terms of the number of elements encountered.

Considering all these factors, the overall worst-case time complexity is `O(n^3)` - this happens when all numbers are less than `p` or when `k` is large, allowing the inner loop to run for all elements without breaking early.

Space Complexity

The space complexity mainly comes from the set `s` that stores unique strings constructed from the list elements. In the worst case, it can store all subarrays, and the size of the subarrays are incremental starting from 1 to `n`. The space required for these strings could be considered as summing an arithmetic progression from 1 to `n`, giving a space complexity of `O(n*(n+1)/2)`, which simplifies to `O(n^2)`.

Additionally, there is a minor space cost for the variables `cnt` and `t`, but they don't grow with `n` in a way that would affect the overall space complexity.

Therefore, the overall space complexity of the code is `O(n^2)`.