2913. Subarrays Distinct Element Sum of Squares I

Easy

Problem Description

The task is to calculate a particular sum related to all subarrays of a given array `nums` that is 0-indexed (meaning that the indices of the elements start at 0). Specifically, you must find the sum of the squares of the "distinct counts" of all subarrays of `nums`. A "distinct count" refers to the number of unique elements present in a subarray.

A subarray is defined as any contiguous sequence of elements from the array. Consequently, for an array of length `n`, there are `n * (n + 1) / 2` possible subarrays since each element can be the starting point of a subarray and can extend to any of the remaining elements, including itself.

To clarify, consider a subarray `nums[i..j]` where `i` and `j` represent the starting and ending indices, respectively, and adhere to `0 <= i <= j < nums.length`. The distinct count of such a subarray is the count of unique values within it.

The problem asks for the sum of the squares of each subarray's distinct count. This means that for each subarray, we count how many different numbers it has, square this count, and then add all these squares together to get the final answer.

Intuition

To solve this problem, a straightforward approach is to look at each possible subarray, calculate its distinct count, square it, and then sum these values.

We start by iterating over all possible starting points of subarrays. For each starting point, we create a new subarray that begins at this index. We then incrementally add elements to the end of this subarray one by one, extending it until we reach the end of the array.

As we add each new element to our current subarray, we maintain a set that holds the distinct elements found so far. This set allows us to easily keep track of the count of unique elements, as adding a duplicate to a set does not change its size. The size of this set, squared, represents the contribution of the current subarray to the overall answer.

So the overall process is:

1. Enumerate the starting point `i` of each subarray.
2. For each start point, iterate through the array to extend the subarray until the end of the array, each time keeping track of unique elements using a set.
3. With each addition to the subarray, calculate the size of the set (the distinct count), square it, and add this to a running total.
4. Once we have done this for all subarrays, return the total sum as the answer.

Solution Approach

To implement the solution for the above-defined problem, the following steps solidify the approach using the Python programming language.

- We initiate by establishing a class named `Solution` which contains the function `sumCounts(self, nums: List[int]) -> int` to execute the solution logic.
- We define a variable `ans` to store our accumulated result and start it with a value of 0. Similarly, we assign `n` to the length of `nums` to keep track of the array's size, which we will repeatedly use in our iterations.
- We traverse each element in the array using a for loop, where our iterator `i` goes from 0 up to, but not including, `n`. This iterator signifies the beginning of each subarray we are going to evaluate.
- Inside this loop, we instantiate an empty set `s`. This will hold the distinct elements of the current subarray originating from index `i`.
- We then nest another for loop and set our second iterator `j` to range from `i` to `n`. This loop will consider every possible endpoint for the subarray starting at `i`. In other words, we are examining every subarray `nums[i..j]`.
- We add the current element `nums[j]` into our set `s`. The unique property of a set ensures that it will only contain distinct elements.
- Finally, we calculate the contribution of the current subarray to our answer by taking the size of the set (our distinct count) and squaring it, leading to `len(s) * len(s)`.
- That squared value is then added to `ans`, which is incrementally growing to encompass the sum of the squares of distinct counts of all subarrays considered so far.
- Once we have completed the enumeration of all subarrays and their contributions, we return `ans` as the final sum.

In terms of data structures and algorithms, this solution leverages a set to efficiently track unique elements, and it uses nested loops to enumerate all possible subarrays. The algorithm's runtime would depend on the number of subarrays it has to consider, which is on the order of `O(n^2)`, as for each starting index `i`, we could potentially look at n-i endpoints.

Summarized, the implementation is direct and makes use of simple data structures to deliver the desired outcome.

Example Walkthrough

Let's illustrate the solution approach with a small example. Suppose we have an array `nums` with the elements `[1,2,1]`. We will walk through the algorithm step by step to calculate the sum of the squares of the distinct counts of all subarrays.

1. **Initialize variables**:
   - `ans` is initialized to 0.
   - `n` is assigned the value 3 (since the array has 3 elements).
2. **Iterate over each possible start index `i` of subarrays**:
   - For `i = 0`: Initialize `s` as an empty set.
     - For `j = 0`: Add `nums[j]` (which is 1) to the set. Now, `s = {1}`. The distinct count is 1 with a square of 1. Add this to `ans`, so `ans = 1`.
     - For `j = 1`: Add `nums[j]` (which is 2) to the set. Now, `s = {1, 2}`. The distinct count is 2 with a square of 4. Add this to `ans`, so `ans = 1 + 4 = 5`.
     - For `j = 2`: Add `nums[j]` (which is 1) to the set. However, 1 is already in the set, so `s = {1, 2}`. The distinct count remains 2 with a square of 4. Add this to `ans`, so `ans = 5 + 4 = 9`.
   - For `i = 1`: Reset `s` as an empty set.
     - For `j = 1`: Add `nums[j]` (which is 2) to the set. Now, `s = {2}`. The distinct count is 1 with a square of 1. Add this to `ans`, so `ans = 9 + 1 = 10`.
     - For `j = 2`: Add `nums[j]` (which is 1) to the set. Now, `s = {1, 2}`. The distinct count is 2 with a square of 4. Add this to `ans`, so `ans = 10 + 4 = 14`.
   - For `i = 2`: Reset `s` as an empty set.
     - For `j = 2`: Add `nums[j]` (which is 1) to the set. Now, `s = {1}`. The distinct count is 1 with a square of 1. Add this to `ans`, so `ans = 14 + 1 = 15`.
3. **Final Result**: After considering all subarrays, the sum of the squares of distinct counts is equal to `ans = 15`.

Thus, following the solution approach, we've processed each subarray of `nums = [1,2,1]`, computed the square of the distinct count for each subarray, and added them together to produce the final result. The sets formed for each subarray were helpful in tracking the unique elements, and the nested for loops made sure that we considered every possible subarray.

Solution Implementation

Python

```python
class Solution:
    def sumCounts(self, nums: List[int]) -> int:
        # Initialize the result variable
        total_sum = 0
        # Compute the length of the input list once for efficiencies
        num_elements = len(nums)

        # Iterate over each element in nums
        for i in range(num_elements):
            # Initialize a set to store unique elements
            unique_elements = set()
            # Look at each subarray starting from the current element
            for j in range(i, num_elements):
                # Add the current element to the set of unique elements
                unique_elements.add(nums[j])
                # Add the square of the count of unique elements so far to the total_sum
                total_sum += len(unique_elements) * len(unique_elements)

        # Return the total sum calculated
        return total_sum
```

Java

```java
class Solution {
    // Function to calculate the sum of the squares of the distinct number counts in all subarrays
    public int sumCounts(List<Integer> nums) {
        int answer = 0; // Initialize the answer to 0
        int n = nums.size(); // Get the length of the list nums

        // Iterate over all possible starting points of subarrays
        for (int i = 0; i < n; ++i) {
            int[] count = new int[101]; // Array to count occurrences of numbers; assumes numbers in nums are in range [0, 100]
            int distinctCount = 0; // Counter to track the number of distinct numbers in the current subarray

            // Iterate over all possible ending points of subarrays, starting from i
            for (int j = i; j < n; ++j) {
                // If the number has not been seen in the current subarray, increment the distinct number count
                if (++count[nums.get(j)] == 1) {
                    distinctCount++;
                }

                // Add the square of the current number of distinct elements to the answer
                answer += distinctCount * distinctCount;
            }
        }

        // Return the computed answer
        return answer;
    }
}
```

C++

```cpp
#include <vector>

class Solution {
public:
    // Function to calculate the sum of counts of unique numbers in all subarrays.
    int sumCounts(vector<int>& nums) {
        int answer = 0; // Variable to store the final answer
        int size = nums.size(); // Get the size of the input vector

        // Iterate over all starting points of subarrays.
        for (int start = 0; start < size; ++start) {
            int counts[101] = { 0 }; // Initialize counts of all numbers to 0.
            int uniqueCount = 0; // Variable to store the number of unique numbers in a subarray

            // Iterate over all possible ending points of subarrays beginning at 'start'.
            for (int end = start; end < size; ++end) {
                // If this is the first occurrence of nums[end] in the current subarray,
                // increment the uniqueCount. Otherwise, this step just counts the occurrence.
                if (++counts[nums[end]] == 1) {
                    ++uniqueCount;
                }

                // Add the square of the current count of unique numbers to the answer.
                // This is done for each subarray.
                answer += uniqueCount * uniqueCount;
            }
        }

        // Return the final computed answer.
        return answer;
    }
};
```

TypeScript

```typescript
// This function calculates the sum of the squares of unique counts of an array slice
function sumCounts(nums: number[]): number {
    let answer = 0; // The result of the sum of the squares of unique counts.
    const length = nums.length; // The length of the input array.

    // Loop through each element in the array.
    for (let startIndex = 0; startIndex < length; ++startIndex) {
        // Initialize an array to keep count of numbers, considering the constraint (1 <= nums[i] <= 100).
        const counts: number[] = Array(101).fill(0);

        // Variable to keep track of unique numbers in the current slice.
        let uniqueCount = 0;

        // Slice the 'nums' array from the current index to the end and iterate over it.
        for (const value of nums.slice(startIndex)) {
            // Increase the count of the current number.
            // If it is the first occurrence, also increment 'uniqueCount'.
            if (++counts[value] === 1) {
                ++uniqueCount;
            }

            // Add the square of the unique count to the answer after each number addition.
            answer += uniqueCount * uniqueCount;
        }
    }

    // Return the final answer
    return answer;
}
```


```python
class Solution:
    def sumCounts(self, nums: List[int]) -> int:
        # Initialize the result variable
        total_sum = 0
        # Compute the length of the input list once for efficiencies
        num_elements = len(nums)

        # Iterate over each element in nums
        for i in range(num_elements):
            # Initialize a set to store unique elements
            unique_elements = set()
            # Look at each subarray starting from the current element
            for j in range(i, num_elements):
                # Add the current element to the set of unique elements
                unique_elements.add(nums[j])
                # Add the square of the count of unique elements so far to the total_sum
                total_sum += len(unique_elements) * len(unique_elements)

        # Return the total sum calculated
        return total_sum
```


Time and Space Complexity

Time Complexity

The provided code consists of two nested loops: the outer loop goes through each element in the input list `nums`, and the inner loop iterates over every subsequent element, adding them to a set `s`. The inner operation in which the set size `len(s)` is squared and added to the running `ans` has a constant time complexity since finding the length of a set and performing arithmetic operations are both done in constant time.

The time complexity can be understood as follows. For each index `i`, the inner loop runs `n-i` times, where `n` is the length of `nums`. So, the total number of iterations of the inner loop is `n + (n-1) + (n-2) + ... + 1`, which simplifies to `n*(n+1)/2`. Since each iteration involves a constant-time set addition and calculation, this algorithm is quadratic in nature. Using Big O notation, we write the time complexity as `O(n^2)`.

Space Complexity