1512. Number of Good Pairs

Easy Array Hash Table Math Counting

Leetcode Link

Problem Description

The problem gives us an array `nums` of integers. We need to find the total number of 'good pairs' in this array. A 'good pair' is defined as a pair of indices `(i, j)` such that `i < j` and `nums[i] == nums[j]`. In simple terms, we must count how many pairs of elements have the same value, where the first element comes before the second element in the array.

Intuition

To solve this problem, we use a hashmap (dictionary in Python) to keep track of the number of times each value appears in the array as we iterate through it. This approach helps us to find 'good pairs' efficiently.

Here's the thinking process for arriving at this solution:

1. We initialize a counter `cnt` to keep the frequency of each element we've seen so far.
2. We initialize a variable `ans` to keep the running count of good pairs.
3. We iterate over each element `x` in `nums`.
4. For every element `x`, we add `cnt[x]` to `ans`. Why? Because if `cnt[x]` is the number of times we've seen `x` so far, then there are `cnt[x]` ways to form a 'good pair' with `x` being the second element.
5. After counting the 'good pairs' for `x`, we increment `cnt[x]` since we've just seen another `x`.
6. After the loop ends, `ans` will hold the total number of 'good pairs'.

By counting incrementally with each new element, we avoid the need for nested loops, which reduces the time complexity significantly from O(n^2) to O(n).

Solution Approach

The solution for counting the number of good pairs uses a hashmap as an auxiliary data structure to store the frequency of each element in the array. In Python, we use the `Counter` class from the `collections` module for this purpose. The approach taken in the solution is both efficient and straightforward to implement.

Here are the details of the implementation:

1. We define a class `Solution` with a method `numIdenticalPairs` that takes a list of integers `nums` as input and returns an integer.

2. Within the method, we initialize our answer variable `ans` to 0, which will eventually hold the total number of good pairs.

3. We then initialize our counter `cnt` as an instance of `Counter`, which is a subclass of the dictionary in Python, specifically designed to count hashable objects.

4. We begin a loop over each element `x` in `nums`:

   - For each element `x`, we first increment our answer `ans` by the current count of `x` in `cnt`:

     ```
     ans += cnt[x]
     ```

     This is based on the idea that if we have already encountered `x` 'n' times, then there are 'n' pairs that can be formed with this current 'x' as the second element of the pair.

   - We then increment the count of `x` in our counter:

     ```
     cnt[x] += 1
     ```

     This is necessary to reflect that we have come across another instance of `x`.

5. After completing the loop, we have counted all good pairs, and the `ans` variable now contains the correct answer.

6. Lastly, we return `ans` as the result.

The key algorithmic idea here is to efficiently keep track of past occurrences of elements to calculate the number of good pairs without needing to compare each pair of elements individually, which would otherwise result in a much slower algorithm.

Example Walkthrough

Let's consider an example to illustrate the solution approach:

Suppose we have the array `nums = [1, 2, 3, 1, 1, 3]`.

Now apply the steps described in the solution approach:

1. Initialize `ans` to 0, as we have not yet counted any 'good pairs'.
2. Initialize `cnt` as an empty `Counter` object.

Now let's iterate over each element `x` in `nums`:

- First element is `1`:
  - `ans += cnt[1]` which is 0 since `1` has not appeared before.
  - `cnt[1] += 1` so now `cnt` is `{1:1}`.
- Second element is `2`:
  - `ans += cnt[2]` which is 0 since `2` has not appeared before.
  - `cnt[2] += 1`, now `cnt` is `{1:1, 2:1}`.
- Third element is `3`:
  - `ans += cnt[3]`, which is 0 since `3` has not appeared before.
  - `cnt[3] += 1`, now `cnt` is `{1:1, 2:1, 3:1}`.
- Fourth element is `1` again:
  - `ans += cnt[1]` which is 1, reflecting the first appearance of `1`.
  - `cnt[1] += 1`, now `cnt` is `{1:2, 2:1, 3:1}` and `ans` is 1.
- Fifth element is `1` once more:
  - `ans += cnt[1]` which is 2, as we've previously encountered `1` twice.
  - `cnt[1] += 1`, so `cnt` becomes `{1:3, 2:1, 3:1}` and `ans` updates to 3.
- Last element is `3` again:
  - `ans += cnt[3]` which is 1, reflecting the first appearance of `3`.
  - `cnt[3] += 1`, now `cnt` is `{1:3, 2:1, 3:2}` and `ans`'s final value is 4.

At the end of the loop, `ans` holds the total number of good pairs which is 4. These pairs are `(0, 3)`, `(0, 4)`, `(1, 5)`, and `(3, 4)` since they comply with the condition that `i < j` and `nums[i] == nums[j]`.

Finally, we return the value of `ans`, which is `4` in this example.

This example adequately demonstrates how the algorithm works as intended, efficiently counting the number of good pairs in the array using the hashmap `cnt` to keep track of the occurrences of each element.

Python Solution

```python
from collections import Counter

class Solution:
    def numIdenticalPairs(self, nums: List[int]) -> int:
        # Initialize the count of good pairs to zero
        good_pairs_count = 0

        # Create a Counter object to track the occurrences of each number in the list
        occurrences = Counter()

        # Iterate over each number in the input list
        for number in nums:
            # For each number, add the current count of that number to good_pairs_count
            # This utilizes the property that a pair is formed for each previous occurrence of the same number
            good_pairs_count += occurrences[number]

            # Increment the count for this number
            occurrences[number] += 1

        # Return the final count of good pairs
        return good_pairs_count

```

Java Solution

```java
class Solution {
    public int numIdenticalPairs(int[] nums) {
        int goodPairs = 0; // This will hold the count of good pairs
        int[] count = new int[101]; // Array to store the frequency of numbers (since the max number is 100)

        for (int number : nums) {
            goodPairs += count[number]; // Add the count of the current number to the good pairs count
            count[number]++; // Increment the frequency of the current number
        }

        return goodPairs; // Return the total count of good pairs
    }
}

```

C++ Solution

```cpp
#include <vector>

class Solution {
public:
    int numIdenticalPairs(std::vector<int>& nums) {
        int goodPairsCount = 0; // Initialize a count for good pairs
        int counts[101] = {0};  // Initialize an array to store the frequency of each number, assuming numbers fall within 1 to 100

        // Iterate over the input vector 'nums'
        for (int num : nums) {
            // For each number 'num', increment the good pairs count by the number of times 'num' has already appeared
            goodPairsCount += counts[num];

            // Increment the count for the current number in 'counts' array
            counts[num]++;
        }

        // Return the total count of good pairs
        return goodPairsCount;
    }
};

```

Typescript Solution

```typescript
// This function calculates the number of good pairs in an array.
// A good pair is defined as pairs (i, j) where nums[i] == nums[j] and i < j.
function numIdenticalPairs(nums: number[]): number {
    // Initialize an array with 101 elements all set to zero
    // as the problem constraints suggest numbers between 1 and 100.
    const count = new Array(101).fill(0);

    // This will hold the total number of good pairs.
    let totalPairs = 0;

    // Iterate over each number in the input array.
    for (const number of nums) {
        // A good pair is found for each prior occurrence of the same number,
        // so we increase the totalPairs by the count of the current number seen so far.
        totalPairs += count[number];

        // Increment the count for the current number for tracking future pairs.
        count[number]++;
    }

    // Return the total number of good pairs found.
    return totalPairs;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the given code is `O(n)`, where `n` is the length of the input list `nums`. This is because the code iterates through each element of `nums` exactly once, and operations within the loop (accessing and updating the `Counter` dictionary) are `O(1)` on average due to the hashing.

Space Complexity

The space complexity of the code is `O(m)`, where `m` is the number of unique elements in `nums`. In the worst case, if all elements are unique, `m` would equal `n`. The `Counter` object - a dictionary in Python - holds count data for each unique element. Therefore, the storage required grows with the number of unique elements.