2176. Count Equal and Divisible Pairs in an Array

Easy Array

Leetcode Link

Problem Description

The task requires us to find the number of unique pairs `(i, j)` from a given integer array `nums`, where the first element of the pair is at an index less than the second element of the pair (`i < j`). The condition for these pairs to count is two-fold:

1. The values at these indices must be the same (`nums[i] == nums[j]`).
2. The product of the indices `i * j` must be evenly divisible by a given integer `k` (which means `(i * j) % k == 0` should be true).

In other words, for array `nums` with `n` integers and the given integer `k`, we need to count how many distinct pairs of indices have the same value in `nums` and their index product is a multiple of `k`.

This is a combinatorial problem that involves both array traversal and elementary number theory.

Intuition

The intuition behind the solution is based on a brute-force approach where we evaluate all possible pairs to see if they satisfy the two conditions mentioned. This means we:

1. Iterate through the array using two loops:
    - An outer loop, where we pick every element `i` starting from index `0` to `n-2`
    - An inner loop, where we pick every element `j` starting from index `i+1` to `n-1`
2. For each pair `(i, j)` found through this iteration, check if `nums[i]` and `nums[j]` are equal.
3. If the numbers at `i` and `j` are equal, further check if the product `(i * j)` is divisible by `k`.
4. If both conditions hold true, count this pair towards our total count.

The solution uses a single return statement with a sum of a generator expression. This generator expression iterates over all pairs as described and uses a conditional expression to count only those pairs that satisfy the two conditions. Since it loops through half of the possible combinations (since `i < j`), it ensures that each unique pair is only considered once. This approach is simple and direct but can be inefficient for large arrays since it performs checks for every index pair.

Solution Approach

The implementation of the solution uses a brute-force algorithm with nested loops to iterate over all possible pairs of indices in the given array. There is no use of additional data structures; the solution operates directly on the input array `nums`. Here is how the solution is implemented:

1. We define a function `countPairs` that takes in an array `nums` and an integer `k` as input.
2. Inside the function, we calculate the length of the array `n`. This is necessary to know the range for our iterations.
3. We use a nested loop construct to generate all possible pairs:
    - The outer loop runs from `0` to `n-1`. The variable `i` in this loop represents the first index of the pair.
    - The inner loop runs from `i+1` to `n`. The variable `j` represents the second index of the pair.
4. For each pair `(i, j)`, the algorithm performs two checks encapsulated in a conditional statement:
    - First, it checks if `nums[i]` is equal to `nums[j]`. This is done by `nums[i] == nums[j]`, ensuring that the values at these indices are the same.
    - Second, it checks if the product of `i` and `j` (`i * j`) is divisible by `k`. This is achieved through `(i * j) % k == 0`, confirming that the product is divisible by `k` without a remainder.
5. The conditional statement is part of a generator expression that evaluates to `True` for each pair that meets the above conditions and `False` otherwise.
6. The `sum` function is then applied directly to this generator expression. In Python, `True` equates to `1` and `False` to `0`, so summing the generator expression effectively counts the number of `True` (valid pairs) instances, which is the answer we need to return.

One key reason why this is a brute-force algorithm is because it does not attempt to optimize by using any patterns or early exit conditions. The solution evaluates every index pair before reaching the count. This solution is straightforward and easy to understand, but it has a time complexity of O(n^2), which means that the algorithm's running time increases quadratically with the size of the input array. Such time complexity is not ideal for large datasets.

The algorithm does not use additional space beyond the input and intermediate variables, thus has a space complexity of O(1).

To summarize, the implementation uses simple iteration and basic conditional checks to find and count the pairs that satisfy both conditions. This brute-force approach is naturally chosen for its simplicity when dealing with small to moderately sized arrays.

Example Walkthrough

Let's illustrate the solution approach with a small example.

Suppose we have an array `nums = [1, 2, 1, 3, 2]` and our given integer `k = 2`. We are interested in finding pairs `(i, j)` where `nums[i] == nums[j]` and `(i * j) % k == 0`.

Following the steps outlined in the solution approach:

1. We define our function: `countPairs(nums, k)`.
2. We determine the length of `nums` which is `5`. We will need this for our loop ranges.
3. We start our loops:
    - The outer loop will have `i` iterate from `0` to `4` (the range is `0` to `n-1`).
    - The inner loop will begin from `i+1` and go up to `5`. So for `i = 0`, the inner loop will start at `1`.
4. Now, we need to check every pair `(i, j)`:
    - For `i = 0`, our pairs would be `(0, 1)`, `(0, 2)`, `(0, 3)`, and `(0, 4)`.
    - We see that `nums[0] == nums[2]` which is `1 == 1`, and `(0 * 2) % 2 == 0`. So `(0, 2)` is a valid pair.
    - No other pairs with `i = 0` satisfy both conditions, so we move to `i = 1`.
5. With `i = 1`, we consider pairs `(1, 2)`, `(1, 3)`, and `(1, 4)`.
    - None of these satisfy both conditions. Either `nums[1] != nums[j]` or `(i * j) % 2 != 0`.
    - Continue to the next index `i = 2`.
6. With `i = 2`, we consider pairs `(2, 3)` and `(2, 4)`.
    - Again, none of these are valid because the numbers at the indices don't match or their product isn't divisible by `2`.
    - Move on to `i = 3`.
7. At `i = 3`, the only pair to consider is `(3, 4)`.
    - Since `nums[3] != nums[4]`, this is not a valid pair.

Putting it all together, the only valid pair we have found is `(0, 2)`. Therefore, the function `countPairs(nums, k)` will return `1`.

This example helps to illustrate how the solution iteratively checks each possible pair according to the algorithm's conditions, incrementing the count when both conditions are met. The result is a count of unique index pairs where the array values are the same and their product is divisible by the given `k`.

Python Solution

```python
class Solution:
    def count_pairs(self, numbers: List[int], k: int) -> int:
        """
        Counts the number of pairs in 'numbers' where the elements are equal
        and the product of their indices is divisible by 'k'.

        :param numbers: List of integers to find pairs in
        :param k: The divisor used to check if the product of indices is divisible
        :return: The count of such pairs
        """

        # Get the length of the numbers list
        num_length = len(numbers)

        # Initialize the pairs count to zero
        pair_count = 0

        # Generate all unique pairs using two-pointer approach
        for i in range(num_length):
            for j in range(i + 1, num_length):

                # Check the condition: numbers at index i and j are the same
                # and the product of indices i and j is divisible by k
                if numbers[i] == numbers[j] and (i * j) % k == 0:
                    # If the condition satisfies, increment the count of pairs
                    pair_count += 1

        # Return the total pairs count that satisfies the condition
        return pair_count

# Note: The function name 'countPairs' has been modified to 'count_pairs'
# to follow the snake_case naming convention typical of Python function names.
```

Java Solution

```java
class Solution {
    // Method to count the number of pairs (i, j) such that nums[i] == nums[j] and the product of i and j is divisible by k.
    public int countPairs(int[] nums, int k) {
        int length = nums.length; // Store the length of the input array nums.
        int pairCount = 0;        // Initialize the counter for the number of valid pairs to 0.

        // Iterate over the elements using two nested loops to consider all possible pairs (i, j) where i < j.
        for (int i = 0; i < length; ++i) {
            for (int j = i + 1; j < length; ++j) {
                // Check if the values at index i and j are equal and if the product of i and j is divisible by k.
                if (nums[i] == nums[j] && i * j % k == 0) {
                    pairCount++; // If condition met, increment the count of valid pairs.
                }
            }
        }
        return pairCount; // Return the total number of valid pairs found.
    }
}
```

C++ Solution

```cpp
#include <vector> // Include the vector header for vector usage

class Solution {
public:
    // Count pairs in an array where the elements are equal and their product is divisible by k
    int countPairs(std::vector<int>& nums, int k) {
        int count = 0; // Initialize count of pairs
        int size = nums.size(); // Get the size of the vector to avoid multiple size() calls within loop

        // Iterate over all possible pairs
        for (int i = 0; i < size; ++i) {
            for (int j = i + 1; j < size; ++j) {
                // Check if the elements are equal and if their product is divisible by k
                if (nums[i] == nums[j] && (i * j) % k == 0) {
                    ++count; // Increment count if the conditions are met
                }
            }
        }

        return count; // Return the total pair count
    }
};

// Note that this code uses <std::vector> which requires the vector header file.
// The naming has been standardized to use camelCase for variable names.
// Comments are added to explain the code in English.
```

Typescript Solution

```typescript
function countPairs(numbers: number[], k: number): number {
    // Get the length of the numbers array.
    const length = numbers.length;

    // Initialize a variable to count the number of valid pairs.
    let count = 0;

    // Iterate over all unique pairs of indices (i, j) where i < j.
    for (let i = 0; i < length - 1; i++) {
        for (let j = i + 1; j < length; j++) {
            // Check if the elements at indices i and j are equal.
            if (numbers[i] === numbers[j]) {
                // Check if the product of indices i and j is divisible by k.
                if ((i * j) % k === 0) {
                    // If both conditions are met, increment the count of valid pairs.
                    count++;
                }
            }
        }
    }

    // Return the total count of valid pairs that satisfy the conditions.
    return count;
}

```

Time and Space Complexity

Time Complexity

The provided code has a nested loop, where the outer loop runs from `0` to `n-1` and the inner loop runs from `i+1` to `n-1`. Here, `n` is the length of the `nums` list.

In the worst case, the number of iterations will be the sum of the series from `1` to `n-1`, which is `(n-1)*(n)/2` or `O(n^2/2)`. Since constants are dropped in Big O notation, the time complexity is `O(n^2)`.

Space Complexity

The code doesn't use any additional data structures that grow with the input size. The only variables used are for loop counters and temporary variables for calculations, which occupy constant space. Therefore, the space complexity is `O(1)`.