2006. Count Number of Pairs With Absolute Difference K

Easy Array Hash Table Counting

Problem Description

The goal of this problem is to find the total number of unique pairs `(i, j)` in a given integer array `nums` wherein the absolute difference between the numbers at positions `i` and `j` is exactly `k`. The condition is that `i` should be less than `j`, which implies that we're looking at pairs where the first element comes before the second element in the array order. The absolute value here means that if the result of the subtraction is negative, we consider its positive value instead.

To put it simply, we iterate over the array, and for each element, we check how many elements that come after it have a difference of `k`.

Intuition

The solution makes use of a hash map to efficiently track the counts of numbers we have seen so far. This is a common strategy in problems where we need to quickly access the count or existence of elements based on their value, which is often referred to as the frequency map pattern.

When we look at an element `num` in the array, there are two numbers that could form a valid pair with it: `num + k` and `num - k`. For each `num`, the solution checks if `num + k` and `num - k` have been seen before (i.e., they are in the hash map). If they are, it adds the count of how many times they've been seen to our answer because each of those instances forms a valid pair with our current `num`.

We then update the count of the current `num` in the hash map, increasing it by 1, to keep track of how many times it has been seen for future iterations.

Solution Approach

The solution makes use of the `Counter` data structure from Python's `collections` module, which is essentially a hash map (or dictionary) designed for counting hashable objects. The keys in this hash map are the distinct elements from `nums` and the values are the counts of how many times they appear. Here's how the solution is implemented:

1. Initialize a variable `ans` to count the number of valid pairs found. It starts at `0`.
2. Create a `Counter` object `cnt` which will store the frequency of each number encountered in `nums`.
3. Iterate over each number `num` in the `nums` array:
   - For the current number `num`, check if `num - k` is in the counter. If it is, it means there are numbers previously seen that, when subtracted from `num`, give `k`. We add the count of `num - k` to `ans`.
   - Similarly, check if `num + k` is in the counter. If it is, add the count of `num + k` to `ans`. This counts the cases where the previous numbers were smaller than `num` and had a difference of `k`.
   - After checking for pairs, increment the count of `num` in the `cnt` Counter to account for its occurrence.
4. After finishing the loop, return the value of `ans`, which now contains the total number of valid pairs.

The algorithm operates in O(n) time complexity, where n is the number of elements in `nums`. This is because the operation of checking and updating the counter is O(1), and we only iterate through the array once.

The key algorithms and data structures used in this solution include:

- **Looping through Arrays**: The for loop iterates through each element in `nums` to check for possible pairs.
- **Hash Map (Counter)**: Utilizes the `Counter` data structure to store and access frequency of elements in constant time (O(1)).
- **Incremental Counting**: Maintains the count of valid pairs in variable `ans` as the array is processed.

By employing the Counter, we are able to maintain a running total of pair counts as the `nums` array is iterated over, thus avoiding the need for nested loops that would significantly increase the computational complexity (potential O(n^2) if using brute force approach).

Example Walkthrough

Let's assume we are given a small integer array `nums = [1, 5, 3, 4, 2]` and we must find the number of unique pairs `(i, j)` such that the absolute difference between `nums[i]` and `nums[j]` is `k = 2`. Following the solution approach:

1. Initialize `ans` to `0` as no pairs have been counted yet.
2. Create a `Counter` object `cnt` which is initially empty.

Now, let's iterate over each number `num` in `nums`:

- For the first number `1`, we check if `1 - 2` (which is `-1`) and `1 + 2` (which is `3`) are in the counter. Neither are because the counter is empty, so we don't change `ans`. Then we add `1` to the counter, so `cnt` becomes `Counter({1: 1})`.
- Moving to the second number `5`, we look for `5 - 2` (equals `3`) and `5 + 2` (equals `7`). Neither are in the counter, so `ans` remains `0`. Then we update `cnt`, now `Counter({1: 1, 5: 1})`.
- Next, `3` is checked against the counter. We look for `3 - 2` (which is `1`) and `3 + 2` (which is `5`). We find `1` in the counter with a count of `1`. So we increment `ans` by `1`. We do not find `5` because we only count pairs where `i < j`, to avoid re-counting. Update the counter with `3`, now `Counter({1: 1, 5: 1, 3: 1})`.
- For `4`, we do the same. We find `4 - 2 = 2` is not in the counter but `4 + 2 = 6` isn't in the counter either. So, `ans` is still `1`. Update `cnt` to `Counter({1: 1, 5: 1, 3: 1, 4: 1})`.
- Lastly, for `2`, `2 - 2` equals `0` (not present in the counter) but `2 + 2` equals `4` which is in the counter with a count of `1`. Thus, we increment `ans` by `1` making it `2`. Final update to the counter leaves it as `Counter({1: 1, 5: 1, 3: 1, 4: 1, 2: 1})`.

After finishing the iteration, `ans` is `2`, implying there are two unique pairs where the difference is exactly `k = 2`: these are `(1, 3)` and `(2, 4)` based on the original positions in the array (`nums[0]` and `nums[2]`, `nums[4]` and `nums[3]` respectively).

The solution has efficiently counted the pairs without re-counting or using nested loops, showcasing the advantage of using a `Counter` to keep track of frequencies and significantly simplifying the search process for complements that result in the required difference `k`.

Solution Implementation

Python

```python
from collections import Counter

class Solution:
    def countKDifference(self, nums: List[int], k: int) -> int:
        # Initialize the answer to 0
        pair_count = 0

        # Initialize the counter that will keep track of the occurrences of elements
        num_counter = Counter()

        # Loop through each number in the input list
        for num in nums:
            # For the current number, add the count of the number that is 'k' less and 'k' more than the current number
            # This is because we're looking for pairs that have a difference of k
            pair_count += num_counter[num - k] + num_counter[num + k]

            # Increment the count of the current number in our counter
            num_counter[num] += 1

        # Return the total count of pairs that have a difference of k
        return pair_count
```

Java

```java
class Solution {

    /**
     * Counts the number of unique pairs in the array with a difference of k.
     *
     * @param nums The array of integers to process.
     * @param k The difference to look for between pairs of numbers.
     * @return The count of pairs with the specified difference.
     */
    public int countKDifference(int[] nums, int k) {
        // Initialize answer to 0 to keep count of pairs
        int countPairs = 0;

        // Array to store counts of each number, considering the constraint 1 <= nums[i] <= 100
        int[] countNumbers = new int[110];

        // Iterate through each number in the input array
        for (int num : nums) {

            // If current number minus k is non-negative, add the count of that number to the total
            // as it represents a pair where num - (num - k) = k
            if (num >= k) {
                countPairs += countNumbers[num - k];
            }

            // If current number plus k is within the allowed range (less than or equal to 100),
            // add the count of that number to the total as it represents a pair where (num + k) - num = k
            if (num + k <= 100) {
                countPairs += countNumbers[num + k];
            }

            // Increment the count for the current number
            ++countNumbers[num];
        }

        // Return total count of pairs
        return countPairs;
    }
}
```

C++

```cpp
class Solution {
public:
    int countKDifference(vector<int>& nums, int k) {
        int countPairs = 0; // Initialize a variable to store the number of pairs
        int countNumbers[110] = {}; // Initialize an array to count occurrences of numbers

        // Iterate through each number in the input vector
        for (int number : nums) {
            // Check if the (number - k) is non-negative as array indices cannot be negative
            if (number >= k) {
                // Add the count of (number - k) to the number of pairs as they satisfy the condition of having a difference of k
                countPairs += countNumbers[number - k];
            }
            // Check if the (number + k) is within the bounds of the countNumbers array
            if (number + k <= 100) {
                // Add the count of (number + k) to the number of pairs as they satisfy the condition of having a difference of k
                countPairs += countNumbers[number + k];
            }
            // Increment the count of the current number
            ++countNumbers[number];
        }

        // Return the total number of pairs with a difference of k
        return countPairs;
    }
};
```

TypeScript

```typescript
function countKDifference(nums: number[], k: number): number {
    let countPairs = 0; // Initialize count of pairs with difference k
    let numberFrequency = new Map<number, number>(); // Initialize a map to keep track of frequencies of numbers

    // Iterate over each number in the array
    for (let num of nums) {
        // Increment countPairs by the count of numbers that are k less than the current number (if any)
        countPairs += (numberFrequency.get(num - k) || 0);
        // Increment countPairs by the count of numbers that are k more than the current number (if any)
        countPairs += (numberFrequency.get(num + k) || 0);

        // Update the frequency map for the current number
        numberFrequency.set(num, (numberFrequency.get(num) || 0) + 1);
    }

    // Return the total count of pairs with difference k
    return countPairs;
}
```

```python
from collections import Counter

class Solution:
    def countKDifference(self, nums: List[int], k: int) -> int:
        # Initialize the answer to 0
        pair_count = 0

        # Initialize the counter that will keep track of the occurrences of elements
        num_counter = Counter()

        # Loop through each number in the input list
        for num in nums:
            # For the current number, add the count of the number that is 'k' less and 'k' more than the current number
            # This is because we're looking for pairs that have a difference of k
            pair_count += num_counter[num - k] + num_counter[num + k]

            # Increment the count of the current number in our counter
            num_counter[num] += 1

        # Return the total count of pairs that have a difference of k
        return pair_count
```

Time and Space Complexity

The given Python code implements a function `countKDifference` to count pairs of elements in an array `nums` that have a difference of `k`.

Time Complexity

The time complexity of the given solution can be analyzed as follows:

- The function iterates over each element in the array `nums` exactly once.
- For each element `num`, it performs a constant-time operation to check and update the counts in the `Counter`, which is an implementation of a hash map.
- Therefore, the time complexity is linear with regard to the number of elements in the list, which is `O(n)` where `n` is the length of the `nums` list.

Space Complexity

The space complexity of the solution can be analyzed as follows:

- A `Counter` is used to keep track of the occurrences of each number in the list.
- In the worst case, if all elements in the list are unique, the size of the `Counter` will grow linearly with the number of elements.
- Therefore, the space complexity of the solution is `O(n)` where `n` is the number of unique elements in `nums`.

In summary, both the time complexity and the space complexity of the given code are `O(n)`.