1726. Tuple with Same Product

Medium | Array | Hash Table

Problem Description

The problem revolves around finding a specific type of combinations within an array of distinct positive integers. Our goal is to determine how many groups of four elements, `(a, b, c, d)`, exist within the array such that the product of `a` and `b` is equal to the product of `c` and `d`. There are two constraints in choosing these elements: firstly, all elements `a`, `b`, `c`, and `d` must be different from each other; secondly, they must be elements from the input array `nums`. The result we're looking for is the count of these unique tuples within given constraints.

Intuition

When considering the solution to this problem, the key lies in understanding that the equation `a * b = c * d` suggests pairs (a, b) and (c, d) have the same product, and we need to count the number of such pairs. Given that the pairs (a, b) and (c, d) are interchangeable and the order within a pair doesn't matter (since multiplication is commutative), each product can potentially be involved in several such tuples.

We can approach this problem by using a hash table (or dictionary) to store the counts of products. Here's a step-by-step breakdown:

1. Iterate over the array using two nested loops to find all possible pairs (a, b) where `a != b`.
   - For each pair, compute the product and store it in a hash table with the product as the key and the occurrence count as the value.
2. After we have the hash table prepared, we look at the occurrence counts. If a certain product occurs 'n' times, this means there are n pairs of numbers with this product. We can select any 2 pairs out of these n to form the combination mentioned above, which can be calculated using the combination formula `C(n, 2)`, which is `n * (n - 1) / 2`.
3. For each distinct product with `n` pairs, we can form combinations that satisfy the equation. Since each combination can produce 8 different tuples `(a, b, c, d)`, `(a, b, d, c)`, `(b, a, c, d)`, etc. (because the pairs can be flipped and their elements can be swapped), we need to multiply the number of possible combinations by 8, which is the same as left-shifting by 3 bits `(v * (v - 1) // 2) << 3`.
4. Finally, we sum up these results for all different products to get the final answer.

Solution Approach

The reference solution provided leverages a `defaultdict` from Python's collections module, which is a dictionary-like data structure that provides all methods provided by a dictionary but takes a first argument (`default_factory`) as a default data type for the dictionary. Using `int` as `default_factory`, it ensures that if a key is not found in the dictionary, it will be automatically initialized with a default integer value of `0`.

Here is a step-by-step explanation of the logic implemented in the solution:

1. A nested for-loop structure is used to generate all unique pairs `(a, b)` of the input array `nums` whose indices are `(i, j)`. Python range function excludes the ending index, which avoids using the same element twice and ensures distinctness:

```python
for i in range(1, len(nums)):
    for j in range(i):
        # Calculate the product and update the count.
```

2. The product `x` of two distinct numbers `nums[i] * nums[j]` from the input array is computed:

```python
x = nums[i] * nums[j]
```

3. The product `x` is used as a key in a `cnt` defaultdict. The value for this key represents the number of times this product `x` has been computed and observed so far:

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

4. After the for-loops have finished executing, each entry in the `cnt` dictionary will hold a key-value pair, where the key is a distinct product and the value is the count of pairs that resulted in that product.

5. An answer is calculated by iterating through the values of the `cnt` dictionary. Each value `v` represents the number of pairs with the same product. As mentioned earlier, `C(n, 2)` combinations are possible for these pairs, and hence `v * (v - 1) // 2` gives us the number of ways to choose 2 pairs:

```python
v * (v - 1) // 2
```

6. Multiplying the number of pair combinations by 8 for each product's tuples, which is achieved by a left shift by 3 (`<< 3`), provides the count of valid 4-number tuples for that product.

7. Summing all these counts together gives the final result; the total number of tuples `(a, b, c, d)` that satisfy the condition `a * b = c * d`:

```python
return sum(v * (v - 1) // 2 for v in cnt.values()) << 3
```

The key algorithmic principle used in this solution is the exploitation of the mathematical properties of combinations and the use of a hash map to efficiently tabulate and retrieve the number of occurrences of each product. This way, the solution maximizes the efficiency of the search process and reduces what could have been a complex problem to a calculation of combinatorial values based on the contents of the dictionary.

Example Walkthrough

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

Suppose we have an input array `nums` with distinct positive integers `[2, 3, 4, 6]`.

Here's how the solution finds the number of groups of four elements `(a, b, c, d)` where the product of `a * b` is equal to the product of `c * d`, given the constraints that `a`, `b`, `c`, and `d` are distinct elements from the `nums` array:

1. We start by iterating over the `nums` array to find all possible pairs `(a, b)` using nested loops:

```python
cnt = defaultdict(int)
for i in range(1, len(nums)):
    for j in range(i):
        x = nums[i] * nums[j]
        cnt[x] += 1
```

2. As we go forward, we compute the product of each pair and track the count of each product:

   Iteration 1: Pair (3, 2), product 6, count becomes `cnt[6] = 1`. Iteration 2: Pair (4, 2), product 8, count becomes `cnt[8] = 1`. Iteration 3: Pair (4, 3), product 12, count becomes `cnt[12] = 1`. Iteration 4: Pair (6, 2), product 12, and since this product already existed, we increase the count to `cnt[12] = 2`. Iteration 5: Pair (6, 3), product 18, count becomes `cnt[18] = 1`. Iteration 6: Pair (6, 4), product 24, count becomes `cnt[24] = 1`.

3. We have the following `cnt` products and their occurrence counts:

```
{6: 1, 8: 1, 12: 2, 18: 1, 24: 1}
```

4. Now, we evaluate the number of tuples for each product. For the product `12`, which has 2 occurrences, there can be `C(2, 2) = 2 * (2 - 1) / 2 = 1` combination to choose two pairs, which translates into `1 << 3 = 8` different tuples `(a, b, c, d)`.

5. Since other products have only 1 occurrence, they do not contribute to the count of tuples as we need at least 2 pairs - meaning that product `12` is the only one that allows for tuples considering our constraints.

6. The final answer would be the sum of the number of tuples of all products, which in this case is only from the product `12`, leading to `8` tuples total.

```python
return sum(v * (v - 1) // 2 for v in cnt.values()) << 3
# This will return 8 for our example.
```

In summary, for our example, the number of valid 4-element tuples `(a, b, c, d)` with the condition `a * b = c * d` is 8.

Solution Implementation

Python

```python
from collections import defaultdict

class Solution:
    def tupleSameProduct(self, nums):
        # Initialize a dictionary to store the frequency of product occurrences
        product_count = defaultdict(int)

        # Calculate all possible products for unique pairs of numbers
        for i in range(1, len(nums)):
            for j in range(i):
                product = nums[i] * nums[j]
                product_count[product] += 1  # Increment the count for this product

        # Sum up the number of tuples that can be formed with the same product.
        # Each product that appeared v times contributes v choose 2 tuples
        # Each tuple corresponds to 8 permutations, hence the left shift by 3 (equivalent to multiplying by 8)
        total_tuples = sum(count * (count - 1) // 2 for count in product_count.values()) << 3

        return total_tuples
```

Java

```java
class Solution {
    public int tupleSameProduct(int[] nums) {
        Map<Integer, Integer> countMap = new HashMap<>(); // a map to count the frequencies of the products

        // Iterate over all possible pairs of elements in the array
        for (int i = 1; i < nums.length; ++i) {
            for (int j = 0; j < i; ++j) {
                // Calculate the product of the current pair
                int product = nums[i] * nums[j];

                // Increment the count of this product in the map by 1,
                // or set it to 1 if it's the first time seeing this product
                countMap.merge(product, 1, Integer::sum);
            }
        }

        int answer = 0; // Initialize the answer to store the number of tuples

        // Iterate over the countMap values to calculate the number of tuples with the same product
        for (int frequency : countMap.values()) {
            // For every product, the number of 4-tuple combinations is frequency choose 2
            answer += frequency * (frequency - 1) / 2;
        }

        // Each combination can produce 8 tuples since (a*b = c*d) corresponds to 8 tuples
        // (a, b, c, d), (a, b, d, c), (b, a, c, d), (b, a, d, c), (c, d, a, b), (c, d, b, a), (d, c, a, b), (d, c, b, a)
        // Hence, we multiply the answer by 8 to get the total number of tuples
        return answer << 3; // The left shift by 3 is equivalent to multiplying by 8
    }
}
```

C++

```cpp
class Solution {
public:
    int tupleSameProduct(vector<int>& nums) {
        unordered_map<int, int> productCount;  // Map to store the count of each product

        // Calculate the product of each unique pair of numbers
        for (int i = 0; i < nums.size(); ++i) {
            for (int j = i + 1; j < nums.size(); ++j) {
                int product = nums[i] * nums[j];
                ++productCount[product];  // Increment the count of this product
            }
        }

        int answer = 0;
        // Iterate through the map to calculate the number of tuples from the count of each product
        for (auto& [product, count] : productCount) {
            // Each product that has been found more than once can form
            // a certain number of tuples. This is a combinatorial problem:
            // if a product is the result of `count` pairs, then the number
            // of tuples is the number of ways to pick 2 pairs out of `count`,
            // which is `count` choose 2.
            answer += count * (count - 1) / 2;
        }

        // Each tuple can be permuted in 8 different ways since the tuple (a, b, c, d)
        // can give us 2 distinct equations by swapping the pairs: a*b = c*d AND a*c = b*d,
        // and each equation has 4 permutations (a * b, b * a, c * d, d * c).
        return answer * 8;
    }
};
```

TypeScript

```typescript
function tupleSameProduct(nums: number[]): number {
    // Create a map to store the frequency of the product of any two elements.
    const productFrequency: Map<number, number> = new Map();

    // Loop over all pairs of elements in the array to populate the map.
    for (let i = 1; i < nums.length; ++i) {
        for (let j = 0; j < i; ++j) {
            // Calculate the product of the current pair.
            const product = nums[i] * nums[j];

            // Update the map with the new count. If the product is not in the map, initialize with 0 then add 1.
            productFrequency.set(product, (productFrequency.get(product) ?? 0) + 1);
        }
    }

    let tuplesCount = 0; // This will hold the total count of all unique quadruplets.

    // Iterate through the map to count how many distinct tuple pairs can be formed.
    for (const [_, frequency] of productFrequency) {
        // For each product, if there are 'n' pairs that have this product,
        // there are (n * (n - 1)) / 2 tuple pairs, as each pair can form a tuple with 'n - 1' other pairs.
        tuplesCount += (frequency * (frequency - 1)) / 2;
    }

    // Each tuple pair represents 8 tuples since the tuple (a, b, c, d) can be permuted to (b, a, d, c), etc.
    // Thus, we left shift by 3 (same as multiplying by 8) to get the total number of tuples.
    return tuplesCount << 3;
}
```


```python
from collections import defaultdict

class Solution:
    def tupleSameProduct(self, nums):
        # Initialize a dictionary to store the frequency of product occurrences
        product_count = defaultdict(int)

        # Calculate all possible products for unique pairs of numbers
        for i in range(1, len(nums)):
            for j in range(i):
                product = nums[i] * nums[j]
                product_count[product] += 1  # Increment the count for this product

        # Sum up the number of tuples that can be formed with the same product.
        # Each product that appeared v times contributes v choose 2 tuples
        # Each tuple corresponds to 8 permutations, hence the left shift by 3 (equivalent to multiplying by 8)
        total_tuples = sum(count * (count - 1) // 2 for count in product_count.values()) << 3

        return total_tuples
```


Time and Space Complexity

The given Python class `Solution` includes a method `tupleSameProduct` that calculates the number of tuples (i, j, k, l) such that `i < j < k < l` and `nums[i] * nums[j] = nums[k] * nums[l]`, with a different ordering of the same numbers not considered as a different tuple.

Time Complexity:

The time complexity of the method is `O(n^2)` where `n` is the length of the input `nums`. This complexity arises as the method uses double nested loops to iterate through pairs of elements in `nums`. With each loop, the algorithm performs a constant time operation of multiplying the elements and updating a dictionary.

- The outer loop runs `(n-1)` times (from `1` to `n-1`).
- The inner loop runs `i` times for each iteration of the outer loop where `i` ranges from `0` to `n-2`. This gives us a total of approximately `(n-1)(n/2)` pairs, which simplifies to `n(n-1)/2` pairs. Considering big O notation, this simplifies further to `O(n^2)`.

Space Complexity:

The space complexity is `O(n^2)` because in the worst case, the `cnt` dictionary can grow to store a unique product for every pair of elements in `nums`. The maximum number of entries in the `cnt` dictionary is the number of unique pairs of elements which is the same as the total number of pairs `(n(n-1)/2)`, which simplifies to `O(n^2)` in big O notation when considering the worst-case scenario.

In conclusion:

- The **Time Complexity** is `O(n^2)`
- The **Space Complexity** is `O(n^2)`