1726. Tuple with Same Product

Given an array `nums` of **distinct** positive integers, return *the number of tuples* `(a, b, c, d)` *such that* `a * b = c * d` *where* `a`, `b`, `c`, *and* `d` *are elements of* `nums`, *and* `a != b != c != d`.

Example 1:

```
Input: nums = [2,3,4,6]
Output: 8
Explanation: There are 8 valid tuples:
(2,6,3,4) , (2,6,4,3) , (6,2,3,4) , (6,2,4,3)
(3,4,2,6) , (4,3,2,6) , (3,4,6,2) , (4,3,6,2)
```

Example 2:

```
Input: nums = [1,2,4,5,10]
Output: 16
Explanation: There are 16 valid tuples:
(1,10,2,5) , (1,10,5,2) , (10,1,2,5) , (10,1,5,2)
(2,5,1,10) , (2,5,10,1) , (5,2,1,10) , (5,2,10,1)
(2,10,4,5) , (2,10,5,4) , (10,2,4,5) , (10,2,5,4)
(4,5,2,10) , (4,5,10,2) , (5,4,2,10) , (5,4,10,2)
```

Constraints:

- `1 <= nums.length <= 1000`
- `1 <= nums[i] <= 104`
- All elements in `nums` are **distinct**.

1726. Tuple with Same Product

Overview

We are given an array `nums` containing `n` **distinct** positive integers. The goal is to find the number of tuples `(a, b, c, d)` such that:

- `a`, `b`, `c`, and `d` are distinct elements of the `nums` array, and
- The condition `a * b == c * d` is satisfied.

Note, that a tuple refers to an ordered list of 4 elements. This means the tuples

`(2, 3, 1, 6)` and `(3, 2, 1, 6)` are considered distinct and counted separately.

In fact, if we have two pairs of numbers `{a, b}` and `{c, d}` that satisfy `a * b == c * d`, we can generate multiple distinct tuples by varying the order of the elements and the pairs:

- `(a, b, c, d)`
- `(b, a, c, d)`
- `(a, b, d, c)`
- `(b, a, d, c)`
- `(c, d, a, b)`
- `(c, d, b, a)`
- `(d, c, a, b)`
- `(d, c, b, a)`

To understand this, observe that for every two pairs of distinct numbers `{a, b}` and `{c, d}`, there are three independent ways to reorder the elements and pairs:

1. Within each pair:

- The order of elements in `{a, b}` can be `(a, b)` or `(b, a)` (2 options).
- Similarly, the order in `{c, d}` can be `(c, d)` or `(d, c)` (2 options).

2. Between the pairs:

- The order of the two pairs can be `({a, b}, {c, d})` or `({c, d}, {a, b})`(2 options).

Since these choices are independent, the total number of distinct tuples is the product of these options: $\boxed{2\times2\times2=8}$.

1726. Tuple with Same Product

Approach#1: Brute force

A straightforward way to solve the problem is to test all possible combinations of values for a , b , c , and d and count how many satisfy the condition. This approach can be implemented using 4 ***nested for loops***, with each loop assigning a value to one of a , b , c , or d . However, this method has a time complexity of $O(n^4)$, which is inefficient for the given constraints.

```
/*
   Brute force -TLE
   Time complexity: O(nlogn+n^4)
   Space complexity: O(logn)
*/
typedef std::vector<int> vi;
class Solution {
   public:
      int tupleSameProduct(vi& nums) {
         int n=nums.size();
         std::sort(nums.begin(),nums.end());
         // Try all possibilities
         int ans=0;
         for(int i=0;i<n-3;++i){
            for(int j=i+1;j<n-2;++j){
               for(int k=j+1;k<n-1;++k){
                  for(int l=k+1;l<n;++l){
                     int a=nums[i];
                     int b=nums[l];
                     int c=nums[k];
                     int d=nums[j];
                     if(a*b==c*d) ans+=8;
                  }
               }
            }
         }
         return ans;
      }
};
```

Approach#2: Optimizing the Brute force

- ***Using binary search***

```
/*
   Brute force + Binary search - AC
   (TLE if time constraint 1s)
   Time complexity:  O(nlogn+n^3 logn)
   Space complexity:  O(logn)
*/
typedef std::vector<int> vi;
class Solution {
   public:
      int tupleSameProduct(vi& nums) {
         int n=nums.size();

         std::sort(nums.begin(),nums.end());

         // Fix a, c and d, then look for b using binary search
         // Same concept of (1. Two Sum)
         int ans=0;
         for(int i=0;i<n-2;++i){
            for(int j=i+1;j<n-1;++j){
               for(int k=j+1;k<n;++k){
                  int a=nums[i];
                  int c=nums[k];
                  int d=nums[j];

                  // If we can not make a product with actual values (a,c,d)
                  if(c*d%a) continue;

                  // Otherwise, search b in the remaining part of the array
                  if(std::binary_search(nums.begin()+k,nums.end(),c*d/a)) ans+=8;
               }
            }
         }
         return ans;
      }
};
```

- ***Using Hash map***

```
/*
   Brute force + Hashing - AC
   (TLE if time constraint 1s)
   Time complexity:  O(nlogn+n^3)
   Space complexity:  O(logn+n)
*/
typedef std::vector<int> vi;
class Solution{
   public:
      int tupleSameProduct(vi& nums) {
         int n=nums.size();
         std::sort(nums.begin(),nums.end());

         // Fix a, c and d, then look for b using hashing
         // Same concept of (1. Two Sum)
         // Store all element for later lookup
         std::unordered_map<int,int> freq;
         for(auto& e: nums) freq[e]++;
         int ans=0;
         for(int i=0;i<n-2;++i){
            for(int j=i+1;j<n-1;++j){
               for(int k=j+1;k<n;++k){
                  int a=nums[i];
                  int c=nums[k];
                  int d=nums[j];
                  // If we can not make a product with actual values (a,c,d)
                  if(c*d%a) continue;
                  // Otherwise, lookup for b in the hash map
                  if(freq[c*d/a]) ans+=8;
               }
            }
         }
         return ans;
      }
};
```

1726. Tuple with Same Product

Approach#3: Products frequencies + combinatorics

Definition: Distinct Pairs in a Set of Pairs

Given a set S , a **pair** is an unordered or ordered selection of two elements from S .

A **set of pairs** is a collection of such pairs, and **distinct pairs** are pairs that differ in at least one element.

Case 1: Unordered Pairs (Combinations)

If pairs are **unordered**, meaning (a,b) is the same as (b,a) , then two pairs (a,b) and (c,d) are **distinct** if: $(a,b) \neq (c,d)$

Example:
For $S=\{1,2,3,4\}$, the set of all **unordered pairs** is:
$\{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$

The pairs $(1,2)$ and $(2,1)$ are **not distinct** because order does not matter.

Case 2: Ordered Pairs (Permutations)

If pairs are **ordered**, meaning $(a,b) \neq (b,a)$, then two pairs (a,b) and (c,d) are **distinct** if:

$(a,b) \neq (c,d)$

Example:
For $S=\{1,2,3,4\}$, the set of all **ordered pairs** is:

$(1,2),(1,3),(1,4),(2,1),(2,3),(2,4),(3,1),(3,2),(3,4),(4,1),(4,2),(4,3)$

Here, $(1,2)$ and $(2,1)$ are **distinct** because order matters.

1726. Tuple with Same Product

/*

Hashing + Combinatorics - AC

(AC if time constraint 1s)

Time complexity: $O\left(n^2+n^2\right)$

Space complexity: $O\left(n^2\right)$

*/

```
typedef std::vector<int> vi;
typedef long long ll;
class Solution{
    public:
        int tupleSameProduct(vi& nums) {
            int n=nums.size();

            // Count the number of pairs (a*b) given the same product
            std::unordered_map<ll,int> freq;
            for(int i=0;i<n-1;++i){
                for(int j=i+1;j<n;++j){
                        freq[nums[i]*nums[j]]++;
                }
            }

            // For each product
            int ans=0;
            for(auto& [prod,f]: freq){
                // Count the number of distinct pairs from these f pairs
                int count_distinct_pairs_with_same_product=f*(f-1)/2;

                // Cumulate the answer
                ans+=count_distinct_pairs_with_same_product*8;
            }

            return ans;
        }
};
```