Project Idea: Magic Triples (Geometric Progression in Arrays)

Team Members / Section [5]

[Omar Fathy Mohammed] (Leader)

[Omar Tamer Omar Mohammed]

[Essam Abdulaziz Mohammed Al-Lawandi]

---

1. Problem Identification

1.1) Problem Description:

The objective is to find the number of triples `(i, j, k)` in an array of `n` integers that form a geometric progression, i.e., the middle element squared equals the product of the first and last elements:

`a[j]^2 = a[i] * a[k]`

The indices `(i, j, k)` must be **distinct**.
This problem is designed to test algorithmic efficiency on **large datasets**, where `n` can reach 200,000 and element values can be up to 1,000,000,000.

1.2 Input-Output Examples:

Sample Input	Sample Output	Explanation

[1,1,1]	6	All permutations of these 3 identical elements satisfy `a[j]^2 = a[i]*a[k]`
[1,2,4]	1	The triple `(1,2,4)` satisfies `2^2 = 1*4`
[1,2,3]	0	No valid triples exist

---

2. Algorithm Development

2.1) Naïve Algorithm (Algorithm 1):

- **2.1.1) Description:**
  Use **three nested loops** to iterate through all possible triples `(i, j, k)`.
- For each triple, check whether `a[j]^2 == a[i] * a[k]`.
- Count all valid triples.

Implementation

Naive.py

```python
def naive():
    n = int(input())
    a = list(map(int, input().split()))

    ret = 0
    for i in range(n):
        for j in range(n):
            if j == i:
                continue
            for k in range(n):
                if k == i or k == j:
                    continue
                if a[j] * a[j] == a[i] * a[k]:
                    ret += 1
    print(ret)

t = int(input())
for _ in range(t):
    naive()
```

Analysis:

Feature	Naive (Triple Loop)
Time Complexity	$O(n^3)$

Space Complexity	O(n)
Redundancy	High

2.1.3 Pros & Cons:

- Simple and easy to implement.
- Extremely slow for large n.

2.2 Optimized Algorithm (Hashmap + Arithmetic Analysis)

Description:

- Count occurrences of each number in a **hashmap** to avoid recomputation.
- Two main cases:
    1. **b = 1:** Triples with identical elements.
    2. **b > 1:** Check divisors and products to find valid triples efficiently.
- Special handling if 1 exists because multiplication by 1 behaves differently.

Implementation

Optimized.py

```python
from collections import defaultdict

MAX_VAL = 10**9
K = 10**6

def solve():
    n = int(input())
    a = list(map(int, input().split()))

    cnt = defaultdict(int)
    for x in a:
        cnt[x] += 1

    ans = 0

    # b = 1
    for x in a:
        if cnt[x] >= 3:
            ans += (cnt[x]-1)*(cnt[x]-2)

    # b > 1
    for num in cnt:
        val = cnt[num]
        if num > K:
            b = 2
            while b * num <= MAX_VAL:
                if num % b == 0 and (num//b) in cnt and (num*b) in cnt:
                    ans += val * cnt[num//b] * cnt[num*b]
                b += 1
        else:
            b = 2
            while b * b <= num:
                if num % b == 0:
                    if num * b <= MAX_VAL and (num//b) in cnt and (num*b) in cnt:
                        ans += val * cnt[num//b] * cnt[num*b]
                    if b*b != num and num//b * num <= MAX_VAL and b in cnt and (num//b * num) in cnt:
                        ans += val * cnt[b] * cnt[num//b * num]
                b += 1
            if num > 1 and num*num <= MAX_VAL and 1 in cnt and (num*num) in cnt:
                ans += val * cnt[1] * cnt[num*num]

    print(ans)

t = int(input())
for _ in range(t):
    solve()
```

Analysis:

Feature	Optimized (Hashmap)

Time Complexity	O(n * sqrt(max(a[i])))
Space Complexity	O(n)
Redundancy	Zero

2.2.3 Pros & Cons:

- Much faster than Naive for large arrays.
- Slightly more complex to implement.

Pseudo Code

Naive

Function NaiveMagicTriples().txt

```
Function NaiveMagicTriples()
    // Step 1: Read the input
    Read n  // number of elements in the array
    Read array a[1..n]  // array elements

    // Step 2: Initialize the counter
    ret = 0  // stores number of valid triples

    // Step 3: Iterate over all possible triples (i, j, k)
    For i = 1 to n:
        For j = 1 to n:
            // Skip if middle element is same as first
            If j == i: continue
            For k = 1 to n:
                // Skip if last element is same as first or middle
                If k == i OR k == j: continue
                // Step 4: Check if the triple satisfies condition
                If a[j] * a[j] == a[i] * a[k] Then
                    // Increment the counter
                    ret = ret + 1

    // Step 5: Print the result for this test case
    Print ret
End Function

// Step 6: Main driver to handle multiple test cases
Read t  // number of test cases
For test_case = 1 to t:
    Call NaiveMagicTriples()
```

Optimized

```
Function OptimizedMagicTriples()
    // Step 1: Read input
    Read n
    Read array a[1..n]

    // Step 2: Count occurrences of each number
    Initialize empty map cnt
    For each element x in array a:
        cnt[x] = cnt[x] + 1

    // Step 3: Initialize answer variable
    ans = 0

    // Step 4: Handle special case b = 1
    // Triples where all three elements are identical
    For each x in array a:
        If cnt[x] >= 3 Then
            // Combinatorial count for middle element
            ans = ans + (cnt[x] - 1) * (cnt[x] - 2)

    // Step 5: Handle general case b > 1
    For each num in cnt:
        val = cnt[num]  // count of current number

        // Step 5a: Large numbers (num > K)
        If num > K Then
            b = 2
            While b * num <= MAX_VAL:
                // Check if num is divisible by b
                If num % b == 0 Then
                    // Check if the other two numbers exist in cnt
                    If (num / b) exists in cnt AND (num * b) exists in cnt Then
                        ans = ans + val * cnt[num / b] * cnt[num * b]
                b = b + 1

        // Step 5b: Small numbers (num ≤ K)
        Else
            b = 2
            While b * b <= num:
                If num % b == 0 Then
                    // First condition: middle element j corresponds to divisor b
                    If num * b <= MAX_VAL AND (num / b) in cnt AND (num * b) in cnt Then
                        ans = ans + val * cnt[num / b] * cnt[num * b]

                    // Second condition: avoid double counting
                    If b * b != num AND num / b * num <= MAX_VAL AND b in cnt AND (num / b * num) in cnt Then
                        ans = ans + val * cnt[b] * cnt[num / b * num]
                b = b + 1

            // Step 5c: Special case when 1 exists
            If num > 1 AND num*num <= MAX_VAL AND 1 in cnt AND num*num in cnt Then
                ans = ans + val * cnt[1] * cnt[num*num]

    // Step 6: Print the total answer
    Print ans
End Function

// Step 7: Main driver
Read t  // number of test cases
For test_case = 1 to t:
    Call OptimizedMagicTriples()
```

3. Comparison Summary

Feature	Naive	Optimized	Notes
Time Complexity	$O(n^3)$	O(n*sqrt(max))	Optimized avoids unnecessary checks
Space Complexity	O(1)	O(n)	Both store array/counters
Redundancy	High	Zero	Optimized counts only valid triples