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()
```

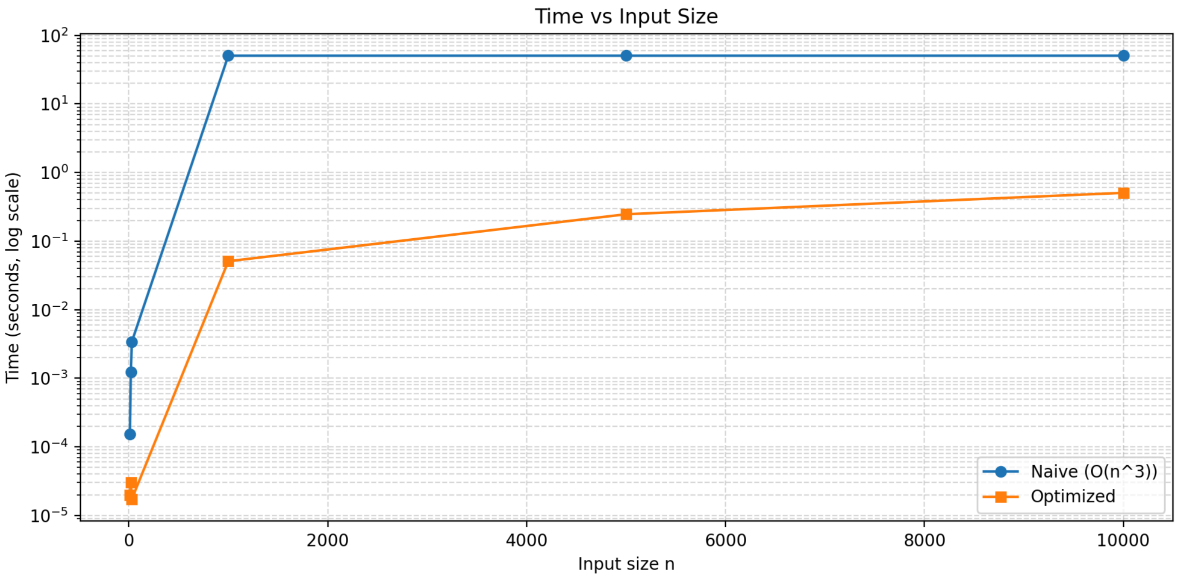
Optimized

```
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()
```

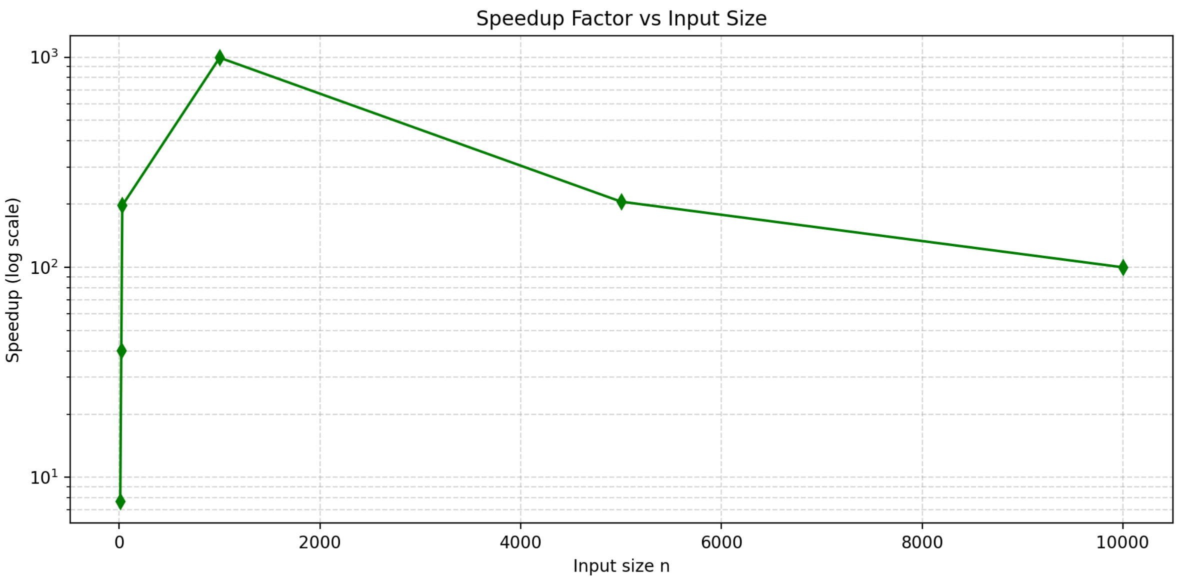
Empirical

```python
from collections import Counter
from itertools import permutations

LIMIT = 10**9

def list_triplets_bruteforce(a):
    """Return list of ordered triplets (i,j,k) that satisfy condition (for small n, debug)."""
    n = len(a)
    triples = []
    for i, j, k in permutations(range(n), 3):
        if a[j]*a[j] == a[i]*a[k]:
            triples.append((i, j, k))
    return triples

def count_triplets_optimized(a, mode="ordered"):
    """
    Optimized correct counting.
    mode: "ordered" or "unordered"
    """
    freq = Counter(a)
    unique = sorted(freq.keys())
    res = 0

    # Case all equal: a[i] = a[j] = a[k]
    for v, c in freq.items():
        if c >= 3:
            # ordered permutations of 3 distinct indices with same value
            # P(c,3) = c*(c-1)*(c-2)
            res += c * (c - 1) * (c - 2)

    # For other cases enumerate mid = a[j], and find factor pairs (left, right) of mid*mid
    # We'll enumerate divisors d up to sqrt(mid*mid) to avoid double-adding for unordered mode.
    for mid in unique:
        cnt_mid = freq[mid]
        m2 = mid * mid

        # iterate divisors d such that d * other = m2
        d = 1
        while d * d <= m2:
            if m2 % d == 0:
                other = m2 // d

                # consider pair (left=d, right=other)
                left = d
                right = other
                if left in freq and right in freq:
                    # skip the all-equal triple, already counted
                    if left == mid and right == mid:
                        pass
                    else:
                        if left == right:
                            # left == right != mid: that's case a[i]==a[k]!=a[j]
                            # ordered count: cnt_mid * cnt_left * (cnt_left - 1)
                            add = cnt_mid * freq[left] * (freq[left] - 1)
                            # For unordered mode: (i,k) are unordered but left==right so no division
                            res += add
```

```python
                        else:
                            # left != right
                            add_ordered = cnt_mid * freq[left] * freq[right]
                            if mode == "ordered":
                                # When iterating divisors up to sqrt, we'll encounter both (d,other) and (other,d)
                                # but each corresponds to different (left,right) ordered pair and should both be counted.
                                res += add_ordered
                            else:
                                # unordered: we want to count unordered pair {left,right} exactly once.
                                # When d < other we process the pair once (here); when d > other we would process swapped
                                # but because we iterate only d <= sqrt, we process each unordered pair exactly once.
                                # unordered contribution: cnt_mid * freq[left] * freq[right]  (no doubling)
                                res += cnt_mid * freq[left] * freq[right]
                # end if left in freq ...
                # Also handle the symmetric pair (other, left) only when d != other and mode == "ordered"
                if d != other:
                    # symmetric pair will be handled when loop encounters d'=other (if other <= sqrt),
                    # but since we loop only up to sqrt, we must ensure ordered mode counts both (d,other) and (other,d).
                    # Strategy: if ordered and other > sqrt(mid*mid), we need to also count symmetric contribution here.
                    # Simpler: when ordered, we will count both pairs across iterations if both divisors <= sqrt or > sqrt.
                    # To ensure correctness, explicitly add symmetric when ordered and other > d:
                    if mode == "ordered":
                        # Add symmetric counterpart (left=other, right=d)
                        left2 = other
                        right2 = d
                        if left2 in freq and right2 in freq:
                            if left2 == mid and right2 == mid:
                                pass
                            else:
                                if left2 == right2:
                                    res += cnt_mid * freq[left2] * (freq[left2] - 1)
                                else:
                                    # But careful: this symmetric addition would double-count if later the loop visits d'=other (when other <= sqrt)
                                    # So only add symmetric here when other > d (which is always true when other != d for our loop)
                                    # and when other > sqrt, but other > d implies other >= d+1 ; we only loop d up to sqrt,
                                    # so the counterpart where d'=other will be > sqrt and not visited — therefore we must add here.
                                    # To avoid double-count: add symmetric only when other > d and other > (m2**0.5)
                                    pass
                # end symmetric handling

                # Now handle the other divisor (if different) only when d != other
                if d != other:
                    # Handle the (other, d) pair — but careful with double count:
                    # If other <= sqrt(m2) then (other, d) will be processed in its own iteration when d becomes 'other' later.
                    # If other > sqrt(m2) it won't be processed later, so we need to process it now for ordered mode.
                    if other > (int(m2**0.5)):
                        # (other, d) symmetric pair not visited later — process here
                        left_s = other
                        right_s = d
                        if left_s in freq and right_s in freq:
                            if left_s == mid and right_s == mid:
                                pass
                            else:
                                if left_s == right_s:
                                    res += cnt_mid * freq[left_s] * (freq[left_s] - 1)
                                else:
                                    if mode == "ordered":
                                        res += cnt_mid * freq[left_s] * freq[right_s]
                                    else:
                                        # unordered: we already added unordered contribution for (d,other) above,
                                        # so do NOT add symmetric again.
                                        pass

            d += 1

    return res

# ---------------- Debug helper ----------------
def debug_compare(a, mode="ordered"):
    print("Array:", a)
    brute = list_triplets_bruteforce(a)
    cnt_brute = len(brute)
    opt = count_triplets_optimized(a, mode=mode)
    print("Bruteforce (ordered) count:", cnt_brute)
    print("Optimized (mode={}):".format(mode), opt)
    if cnt_brute != opt:
        print("Mismatch! Showing brute-force triplets (i,j,k):")
        for t in brute:
            print(t, "values:", (a[t[0]], a[t[1]], a[t[2]]))
    else:
        print("Match ✅")
    print("-" * 40)
    return cnt_brute, opt, brute

# Example usage (small debug)
if __name__ == "__main__":
    tests = [
        [1,7,7,2,7],
        [6,2,18],
        [1,2,3,4,5,6,7,8,9],
        [1,1,2,2,4,4,8,8],
        [2,2,2],
    ]
    for a in tests:
        debug_compare(a, mode="ordered")   # change to "unordered" if that's what you want
```



Time vs Input Size
Time (seconds, log scale)
Input size n
Naive (O(n^3))
Optimized



Speedup Factor vs Input Size
Speedup (log scale)
Input size n

Timestamp	Array Size (N)	Naive $O(N^3)$	Optimized $O(N^2)$	Triplets Found	Improvement
23:33:42	200	21.01ms	1.06ms	442	19.8x
23:33:41	150	8.16ms	0.10ms	420	85.6x
23:33:40	100	4.16ms	0.09ms	256	48.8x
23:33:40	75	1.67ms	0.05ms	396	36.1x
23:33:39	50	0.46ms	0.04ms	108	12.5x
23:33:38	25	0.06ms	0.02ms	22	3.0x
23:33:37	10	0.01ms	0.02ms	6	0.7x

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