📘 Reverse Pairs Problem Documentation

📑 Table of Contents

1. Problem Statement

Given an integer array nums, return the number of reverse pairs.

A reverse pair is defined as a pair (i, j) such that:

- 0 <= i < j < nums.length
- nums[i] > 2 * nums[j]

2. Intuition

This problem is similar to counting inversions in an array, but with a specific condition: nums[i] > 2 * nums[j]. A brute-force solution would require checking all possible pairs — which becomes inefficient for large arrays. Instead, we use merge sort, which not only sorts the array but also helps count valid reverse pairs while merging.

3. Key Observations

- When two subarrays are sorted, the number of reverse pairs between them can be found efficiently using two pointers.
- Sorting the array helps because we can skip many comparisons once we find a number that violates the condition.

4. Approach

We use a divide-and-conquer strategy with the help of merge sort:

- Divide the array into two halves recursively.
- Conquer by counting:
  - The number of reverse pairs in the left half.
  - The number of reverse pairs in the right half.
  - The number of reverse pairs between the left and right halves.
- Merge the two sorted halves and return the total count.

During the merge step:

- Use two pointers to efficiently count valid pairs where nums[i] > 2 * nums[j].
- Merge the sorted halves to maintain order for higher-level comparisons.

5. Edge Cases

- All elements are equal → no reverse pair.
- Sorted in ascending order → no reverse pair.
- Sorted in descending order → maximum reverse pairs.
- Array of size 1 → no reverse pair.

6. Complexity Analysis

 Time Complexity

- O(n log n)
  Merge sort divides the array in log n levels and does O(n) work at each level.

💾 Space Complexity

- O(n)
  Due to the temporary arrays used in the merge step.

7. Alternative Approaches

- Brute Force (Nested Loops)
  - Check all pairs (i, j) and count if nums[i] > 2 * nums[j].
  - Time: $O(n^2)$, not suitable for large n.
- Binary Indexed Tree (Fenwick Tree) or Segment Tree
  - Compress coordinates and track counts efficiently.
  - Requires advanced data structures and is more complex to implement.

8. Test Cases

```
# Test case 1
    nums = [1,3,2,3,1]
    # Expected Output: 2

# Test case 2
    nums = [2,4,3,5,1]
    # Expected Output: 3

# Test case 3
    nums = [5,4,3,2,1]
    # Expected Output: Multiple pairs (6 total)
```

Test case 4

nums = [1,2,3,4,5]

Expected Output: 0

Test case 5

nums = [1]

Expected Output: 0

9. Final Thoughts

- This problem is an excellent application of merge sort beyond just sorting.
- It demonstrates how we can use divide-and-conquer to count pairs satisfying complex conditions.
- For interviews and coding rounds, this pattern is useful for problems involving pairwise comparison with a condition.
- A Segment Tree-based approach may be needed if more frequent updates or range queries are involved.