Question 1, Part B:

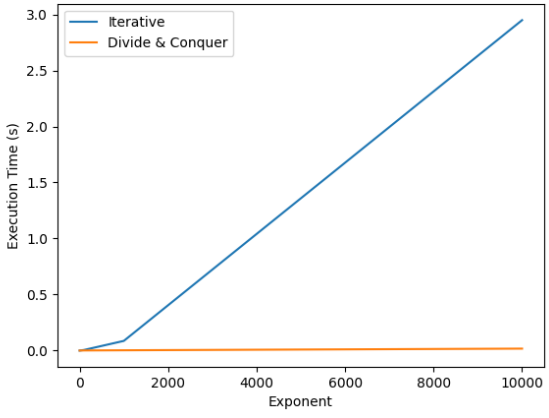
Naïve Iterative Approach:

In the naïve iterative approach, we have a loop that runs exponent times. So, the time complexity is O(exponent), which is linear in terms of the exponent.

Divide-and-Conquer Approach:

In the divide-and-conquer approach, we are effectively halving the problem at each step. So, the number of operations required to compute the power of a number is logarithmic in terms of the exponent. The time complexity is O(log exponent).

Question 1, Part C/D:



The divide-and-conquer approach is indeed more efficient for large exponents, as it grows at a logarithmic rate compared to the linear growth of the iterative approach.

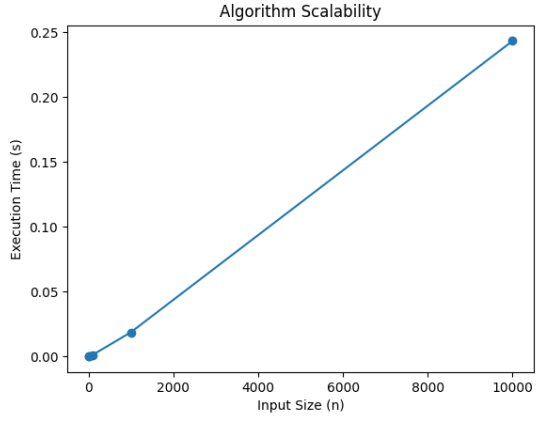
Question 2, Part B:

Merge Sort: The sorting part of the algorithm, which sorts the input array, has a time complexity of O(n log n). This is the dominating factor in the time complexity analysis.

Binary Search: The binary search part to find pairs with the target sum is linear, as we iterate through the array once. This has a time complexity of O(n).

The overall time complexity of the algorithm is determined by the dominant factor, which is the sorting step, i.e., O(n log n). So, the algorithm's overall time complexity is O(n log n).

Question 2, Part C:



In the code, we ran the algorithm for different input sizes (n) ranging from 10 to 1,000,000.

We measured the execution times for each input size and plotted the results.

The experimental results show how the execution time changes as the input size (n) increases.

Since the algorithm's theoretical time complexity is O(n log n) due to the Merge Sort step, the execution time to increase, but not linearly with the input size.

The curve on the graph resembles a logarithmic increase rather than a linear increase.