Assignment 1

Moamen Ahmed Labib

10000369

Question 1

A. Python CODE

B.

T
he divide-and-conquer strategy breaks the problem up into smaller ones and computes the power using a recursive algorithm. This algorithm's recurrence relation is as follows:

n = (n/2) + O(1) = T(n)

The time required to calculate the power of a number n/2 is denoted by T(n/2).
The constant time required for multiplication and other operations at each stage is denoted by O(1).
This recurrence relation can be resolved by applying the Master Theorem:

One recursive call,
a = 1.
b = 2 (half the size of the problem)
For constant time operations, f(n) = O(1).

Three circumstances apply to the Master Theorem:

Example 2:

When a = b^k for some k, where k >= 0, and f(n) = Θ (1), as it is in this instance, the temporal complexity is Θ(log_b(n) * log^k(n)).

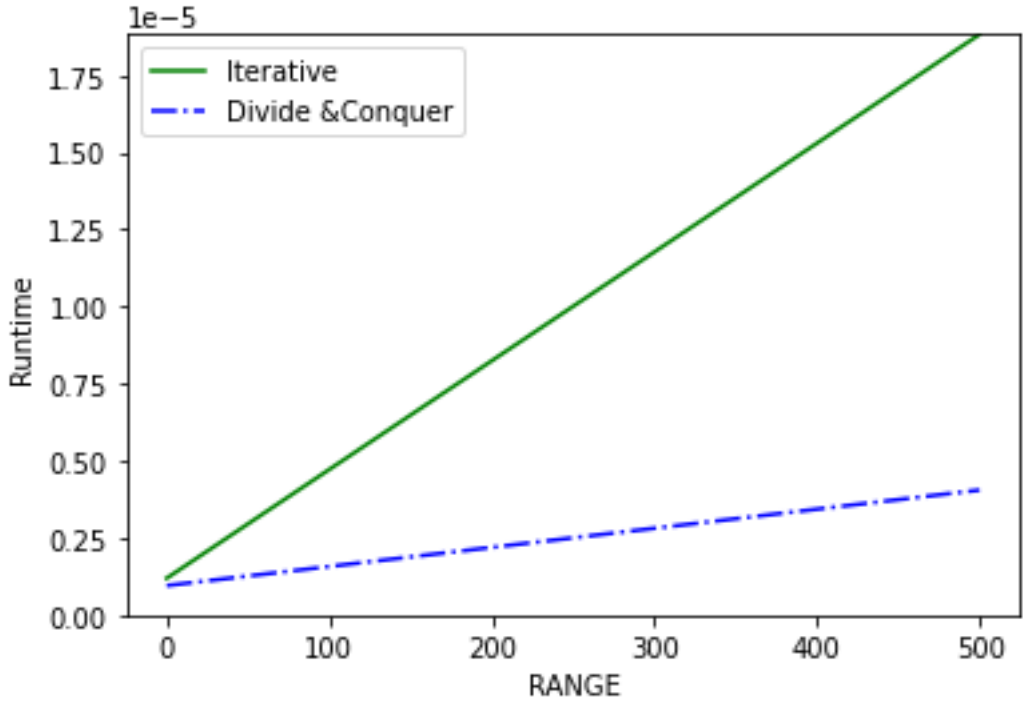
Since an is not a power of b in this instance, a = 1, b = 2, and k = 0. Consequently, Θ(log_2(n) * log^0(n)) = Θ(log_2(n)). is the time complexity.

Thus, the divide-and-conquer strategy has a running time complexity of Θ (log(n)).

In summary:

Simple Iterative Approach: Θ(n)
Method of Division and Conquer: Θ (log(n))
When computing huge powers of a number, the divide-and-conquer strategy outperforms the simple iterative approach.

c.



D. the divide-and-conquer algorithm (logarithmic complexity) is more scalable and efficient than the naive iterative method (linear complexity) for larger values of "n." The graph should confirm the theoretical analysis results.

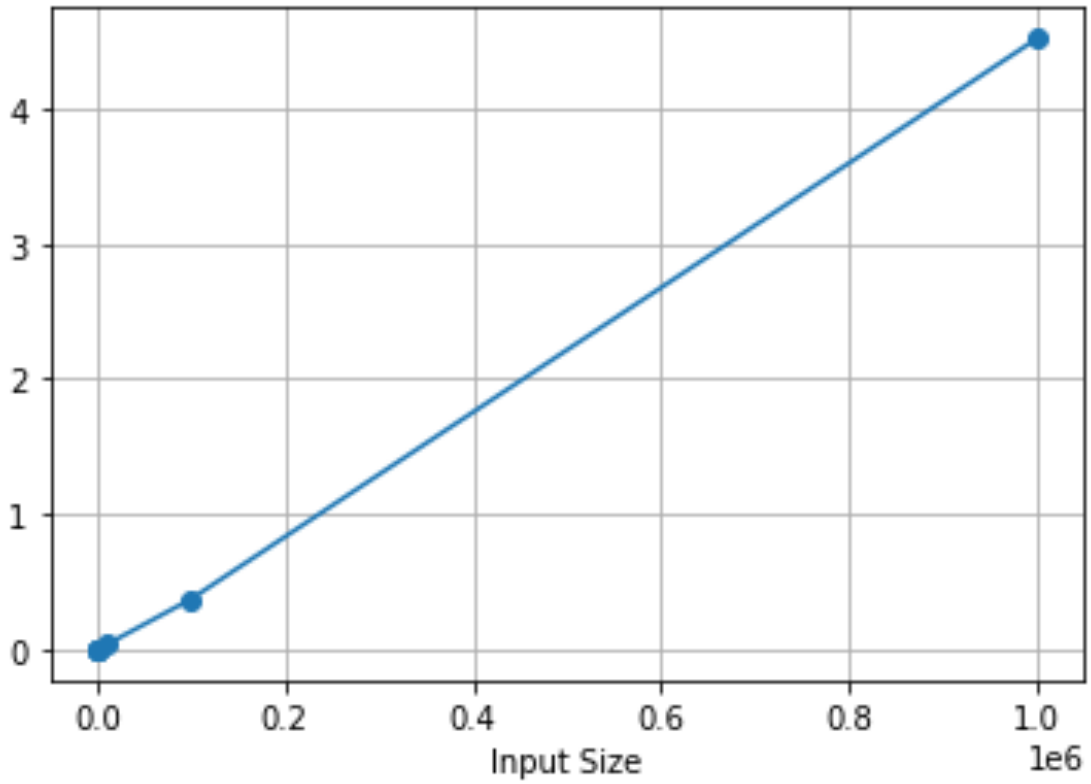
Question 2

A. Code

B.

Merge Sort: The Merge Sort algorithm has a time complexity of O(n log n) in the worst case.
Binary Search: In the worst case, Binary Search has a time complexity of O(log n). In your algorithm, Binary Search is called for each element of the sorted array, resulting in a total time complexity of O(n log n).
The overall time complexity of the algorithm is determined by the dominating factor, which is the Merge Sort operation (O(n log n)). Therefore, the asymptotic running time complexity of the algorithm is Θ(n log n) in the worst case.
Now, let's solve the recurrence relation for the Divide-and-Conquer algorithm used in Merge Sort:
T(n) = 2T(n/2) + O(n)
The recurrence relation for Merge Sort states that to sort an array of size 'n', you divide it into two subproblems of size 'n/2', sort each subproblem, and then merge the two sorted subarrays. The merge step takes O(n) time.
Using the Master Theorem, let's analyze this recurrence relation:
a = 2 (two recursive calls for subproblems)
b = 2 (problem size is halved)
f(n) = O(n) (time taken for merging)
The Master Theorem has three cases, and our recurrence fits into Case 2:
Case 2: If f(n) is Θ(n^c) for some constant c > 0, and if a = b^k for some k ≥ 0, then the time complexity is Θ(n^c * log^k(n)).
In our case, a = 2, b = 2, and c = 1. Since a is equal to b^1, k is 1.
So, the time complexity of Merge Sort is Θ(n * log(n)).
The Binary Search component, being O(log n), doesn't change the overall complexity of Merge Sort, as Merge Sort is the dominating factor. Therefore, the time complexity of the entire algorithm is Θ(n log n).

C.



We measure the execution time of the algorithm for each input size.
We plot the empirical execution times and compare them with the theoretical analysis, which is n * log2(n).

By running this program, you can observe how the algorithm's performance scales with increasing input size and compare it to the theoretical prediction. The resulting graph will show the empirical scalability of the algorithm and whether it aligns with the expected theoretical performance.