Assignment #1 Report

Question 1:

(a) Naive iterative method:
The naive iterative method performs multiplication 'n' times in a loop, where 'n' is the exponent. Therefore, the running time complexity is $\Theta(n)$.

(b) Divide-and-conquer method:
The divide-and-conquer method reduces the exponent size by half recursively. We can represent the running time complexity of the divide-and-conquer algorithm using a recurrence relation:
$T(n) = T(n/2) + O(1)$

To solve this recurrence relation, I used the Master Theorem. In this case, the recurrence has the form:

$T(n) = a * T(n/b) + O(n^d)$

Here, $a = 1$, $b = 2$, and $d = 0$. Since $b^d = 2^0 = 1$, we can compare it with a.

According to the Master Theorem:

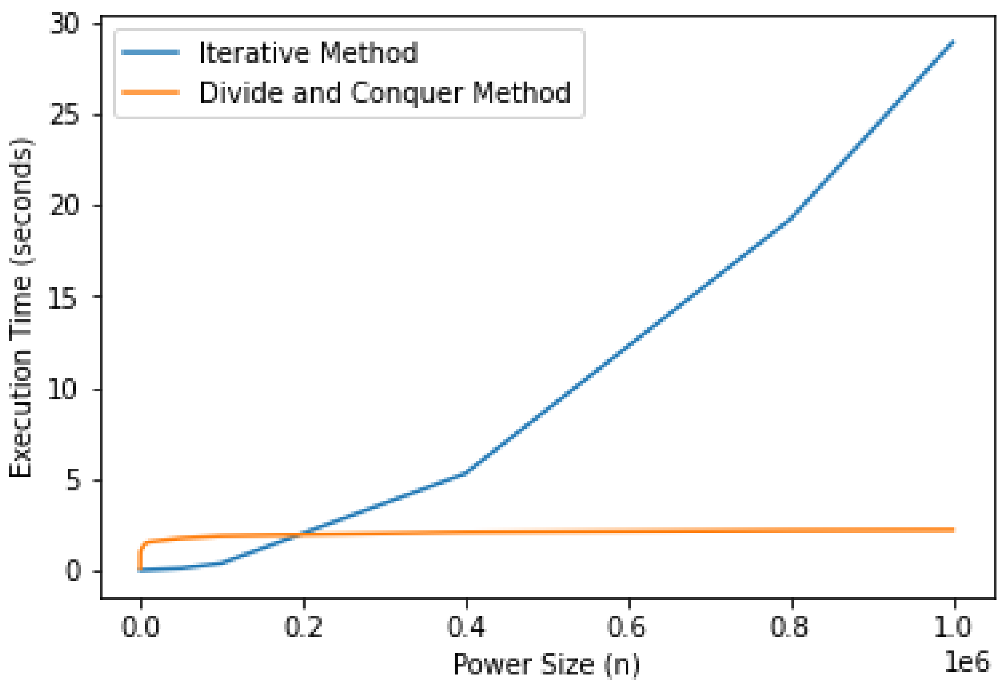
1. If $a > b^d$, the running time complexity is $\Theta(n^{\log_b(a)})$.
2. If $a = b^d$, the running time complexity is $\Theta(n^d * \log n)$.
3. If $a < b^d$, the running time complexity is $\Theta(n^d)$.

In this case, $a = 1$, $b = 2$, and $d = 0$. Since $a = b^d$, the running time complexity of the divide-and-conquer algorithm is $\Theta(n^0 * \log n) = \Theta(\log n)$.

The running time complexity of the divide-and-conquer algorithm is $\Theta(n^0 * \log n) = \Theta(\log n)$.

Therefore, the asymptotic running time complexity for the algorithms is:
(a) Naive iterative method: Θ(n)
(b) Divide-and-conquer method: Θ(log n)

<u>Diagram for Q1:</u>



Question 2:

a) <u>Merge Sort:</u>

- The time complexity of merge sort can be expressed by the recurrence relation:

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

- Using the master theorem to solve this recurrence relation to determine the time complexity

In the given recurrence relation:

- a = 2 (the number of recursive calls)
- b = 2 (the size of subproblems)
- f(n) = O(n) (the time taken to merge subproblems)

Comparing the values of a, b, and f(n) in the master theorem:

- If f(n) = O(n^c) where c < log_b(a), then T(n) = Θ(n^log_b(a)).
- If f(n) = Θ(n^c) where c = log_b(a), then T(n) = Θ(n^c * log n).
- If f(n) = Ω(n^c) where c > log_b(a), and if a * f(n/b) <= k * f(n) for some constant k < 1 and sufficiently large n, then T(n) = Θ(f(n)).

In merge sort, a = 2, b = 2, and f(n) = O(n). Since f(n) = Θ(n) = Θ(n^1), which falls under the second case of the master theorem, the time complexity of merge sort is:

T(n) = Θ(n^1 * log n) = Θ(n log n).

b) <u>Binary Search:</u>

- The time complexity of binary search is O(log n), where n is the size of the input set.

-Since binary search is performed for each element in the sorted set, the overall time complexity is O(n log n).

c) <u>Finding Pairs:</u>

- After sorting the set `S` using merge sort, the algorithm performs a binary search for each element in the sorted set. This search takes O(log n) time.

- Since there are n elements in the sorted set, the overall time complexity for finding pairs is O(n log n).

Combining the time complexities of the individual components, the overall time complexity of the algorithm is:

O(n log n) (Merge Sort) + O(n log n) (Binary Search) = O(n log n)

Therefore, the asymptotic running time complexity of the proposed algorithm is O(n log n).

<u>Diagram for Q2:</u>