Analysis Assignment

Question 1 :

(B):

Naive Iterative Method (power1):

This algorithm uses a simple iterative approach to compute the power. It has a loop that runs n times, performing multiplication in each iteration. Therefore, the time complexity can be expresses an O(n) in big theta notation.

So for the “Naive Iteration Method”, the asymptotic running time complexity is $\theta(n)$.

Divide-and-Conquer Approach (power2):

This algorithm uses a divide-and-conquer approach to compute the power more efficiently. It recursively divides the problem into smaller subproblems. The recurrence relation for this algorithm can be expresses as $T(n) = T(n/2) + O(1)$, where the work done at each level is constant.

Using the recursive tree analysis, we can see that the time complexity of this divide-and-conquer algorithm is $\theta(\log n)$.

So, for the “Divide-and-Conquer Approach”, the asymptotic running time complexity is $\theta(\log n)$. This means that the divide-and-conquer approach is significantly more efficient than the naive iterative method, especially for the large values of n.

To solve the recurrence for the divide-and-conquer algorithm:

- In the naïve iterative method, we use a loop to multiply the base 'n' times. Therefore, the time complexity is $\theta(n)$ in the big theta notation. It has a linear time complexity, where 'n' represents the exponent.

- In the divide-and-conquer method we divide the problem into subproblems and use recursion. We'll analyze the time complexity by solving the recurrence relation:
- $T(n) = T(n/2) + O(1)$
- In this method, we divide the problem into two subproblems of size n/2 and perform some constant amount of work at each level of recursion.

Using the Master Theorem, we can solve this recurrence:

$a = 1$ (one subproblem)

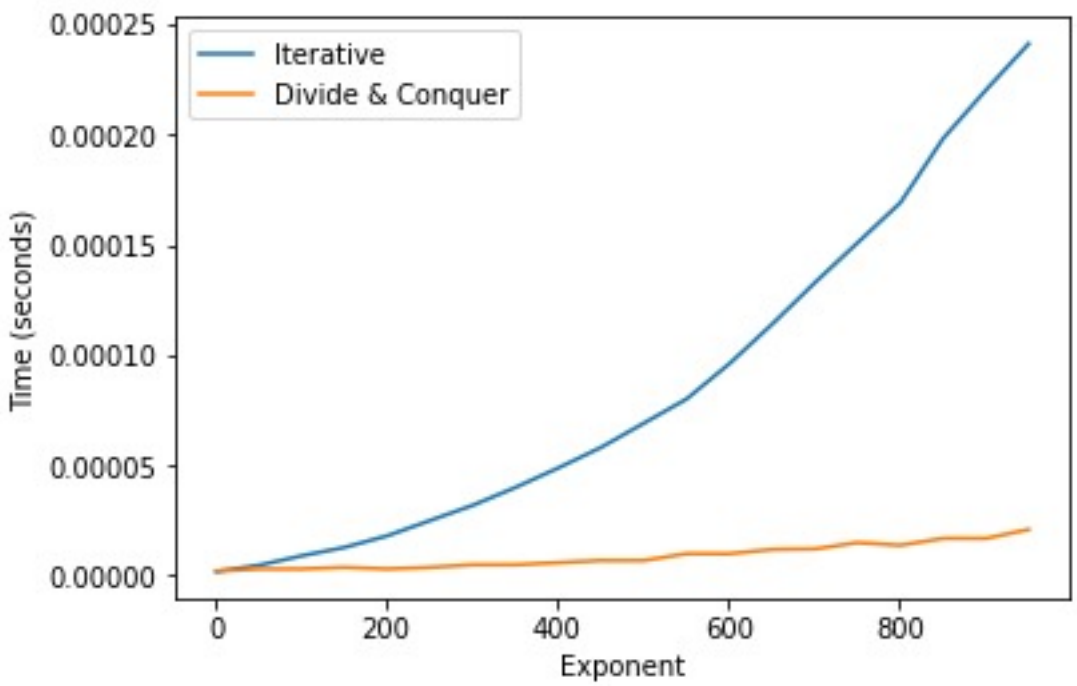
$b = 2$ (subproblem size is halved)

$f(n) = O(1)$ (constant work)

The Master Theorem's case 2 applies since $\log_b(a) = \log_2(1) = 0$, which is not greater than 0. In this case, the time complexity is:

$T(n) = \theta(n^0 * \log n) = \theta(\log n)$

So, the divide-and-conquer method has a time complexity of $\theta(\log n)$ in the big theta notation, which is more efficient than the naïve iterative method for large values of 'n'.



(D): According to the graph, it shows that the time complexity is θ(log n) which confirms the theoretical analysis results in 1.(b).

Question 2 :

(B):

The algorithm provided for finding pairs of integers in a set S whose sum equals a given integer has the following time complexities:

Merge Sort: O(n log n)

Binary Search: O(n)

Since Merge Sort is the more time-consuming part, the overall time complexity of the algorithm is dominated by the Merge Sort step, and it is O(n log n).

To solve the recurrence for the divide-and-conquer algorithm, which is the Merge Sort part:

The Merge Sort algorithm divides the array into two halves and recursively sorts them and then merges the two halves. The recurrence relation for Merge Sort can be expressed as:

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

Here:

T(n) is the time it takes to sort an array of size n.

The 2 represents the two recursive calls on the two halves.

T(n/2) represents the time to sort each half.

O(n) represents the time to merge the two halves, which is a linear operation.

Using the Master Theorem, we can solve this recurrence:

a = 2 (number of subproblems)

b = 2 (subproblem size is halved)

f(n) = O(n) (time to merge)

The Master Theorem's case 2 applies since log_b(a) = log_2(2) = 1. In this case, the time complexity is:

T(n) = θ(n^1 * log n) = θ(n log n)

So, the recurrence for Merge Sort gives us a time complexity of θ(n log n), which confirms that the overall time complexity of the algorithm is O(n log n), as previously determined.

(C): According to the graph, it shows that the time complexity is O(n log n) which confirms the theoretical analysis results in 2.(b).