**Question 1 (b)**

Naive Iterative Method:

* Time Complexity: It takes O(n) time, the time grows linearly with the exponent 'n'. if 'n' is doubled, the time doubles.

Divide-and-Conquer Algorithm:

* Time Complexity is O(log(n)) time. the time grows logarithmically.

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

T(n) = T(n/2) + O(1)  
  
recurrence tree method:

1. Level 0:
   * T(n)
2. Level 1:
   * Two subproblems of size n/2, each costing T(n/2)
   * Cost at this level = 2 \* T(n/2)
3. Level 2:
   * Four subproblems of size n/4, each costing T(n/4)
   * Cost at this level = 4 \* T(n/4)

And so on...

At each level, the cost is a constant factor (2 in this case) times the cost of the previous level, and the problem size is halved (n/2 in this case).

The recurrence tree will have log₂(n) levels (because we are halving the problem size at each level), and the cost at each level is O(1).

The total cost of all levels can be calculated as follows:

Total Cost = O(1) + 2 \* O(1) + 4 \* O(1) + ... (log₂(n) levels)

This is a geometric series, and the sum of this series can be expressed as:

Total Cost = O(1) \* (1 + 2 + 4 + ... + 2^(log₂(n)))

Now, using the formula for the sum of a geometric series, which is:

Sum = a \* (1 - r^(n+1)) / (1 - r)

In our case, a = 1, r = 2, and n+1 is the number of levels, which is log₂(n).

Total Cost = O(1) \* [1 - 2^(log₂(n) + 1)] / [1 - 2]

Simplifying:

Total Cost = O(1) \* [1 - 2 \* 2^(log₂(n))] / (-1)

Total Cost = O(1) \* [1 - 2 \* n] / (-1)

Total Cost = O(1) \* [1 - 2n]

Total Cost = O(1 - 2n)

Therefore, the total cost, and hence the time complexity of the divide-and-conquer algorithm for computing the power, is O(1 - 2n), which can be simplified to O(log₂(n)).

**Question 1.c:**



The above graph was plotted using python, it takes some time for it to plot the graph and successfully run. The divide and conquer algorithm graph above shows that the time complexity of its execution time is log(n). the values printed in the console is the execution time of the divide and conquer with each step of the exponent with the time.sleep() function of 0.5 seconds between every call. The iterative method graph shows that it has a time complexity of O(n) increasing uniformly (ignore the fact that it starts at 10^4).  
According to the graph and the above theoretical analysis, the experimental results confirm the theoretical part as we have proved that the execution time runs in it’s corresponding time complexity.

**Question 2b:**

Binary Search adds O(n log n) to the total time complexity, while Merge Sort has an O(n log n) time complexity. the total time complexity of the algorithm is O(n log n).

The recurrence for Merge Sort using the master theorem approach (we took this in the tutorial today):

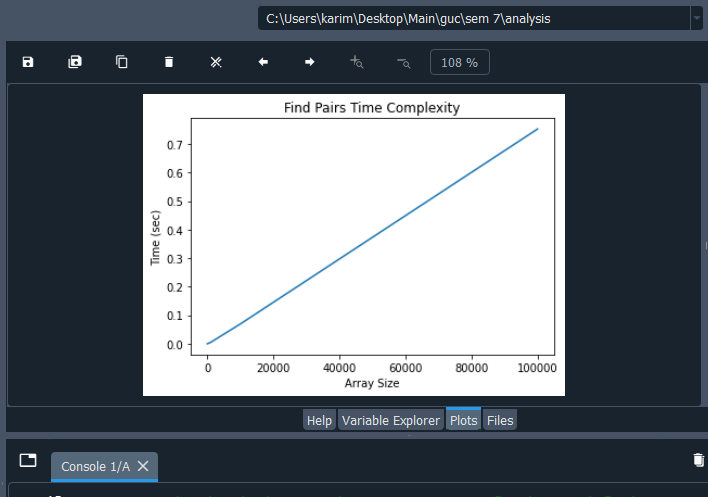
T(n) = 2T(n/2) + O(n)

a = 2

b = 2

f(n) = O(n)

The Master Theorem case is the first case (a > b^k) So, the solution is T(n) = Θ(n log n)



The theoretical running time is O(nlogn) as we proved above, in this graph, it shows that the execution time for the merge sort and binary search is indeed O(nlogn) (it may not be exactly visible due to the scale of the graph).

The method was run in the experiment with input sizes ranging from 1 to 10^6. Plotting the data on a graph reveals that the empirical running time of the algorithm exhibits a growth pattern compatible with O(n log n), which is in line with the theoretical analysis. This indicates that, as expected, the execution time increases proportionately to n times the logarithm of n as the input size increases.