CSEN703: Analysis and Design of Algorithms

Assignment 1

Report

Author: Maram Osama
52-4968
8001779

Question 1:

The program segments from part a) could be found on the repository in the file titled ‘Q1.py’.

b) i- Finding the asymptotic running time complexity for the iterative method is quite simple. The algorithm is made up of a loop that is entered ‘n’ times. Inside the loop, a single computation is done: r=r*a. This on its own has time complexity $\theta(1)$.

The best and worst case scenarios are the same, as the loop is entered in all cases and runs for n times, so the asymptotic time complexity relies completely on the size of n, thus it has an asymptotic time complexity of **$\theta(n)$**.

ii- As for the divide and conquer approach, we can solve it in two ways. We can find the running time complexity by analyzing the code: it consists of a base case ‘return 1’ that is only carried out once, when n reaches value zero. So this line has complexity $\theta(1)$.

Then, we divide the list into a sublist of size n/2. So, it’s very simple to find the time complexity. The number of times we can continuously divide a list into half its size, until it reaches 1, is log(n) of base 2. This is the same for both the best and worst case scenarios. Thus, it has an asymptotic time complexity of **$\theta(\log_2 n)$**.

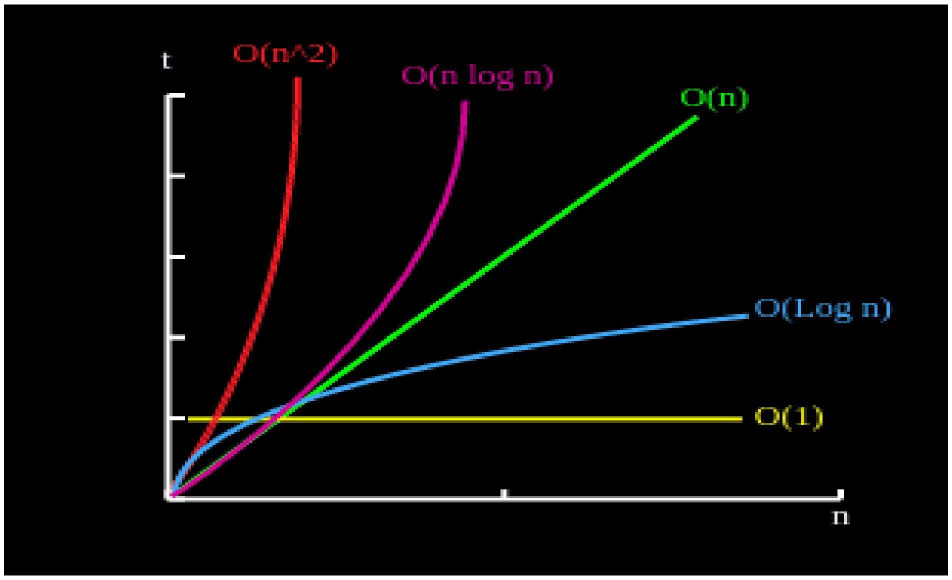
The second approach is by solving the recurrence function $T(n)= T(n/2) + f(n)$. In our case, $f(n)= \theta(1)$. Thus, we obtain this recurrence function: $T(n)= T(n/2) + \theta(1)$.

Solving this using the master theorem, when we have the form $T(n)= a\,T(n/b) + f(n)$.

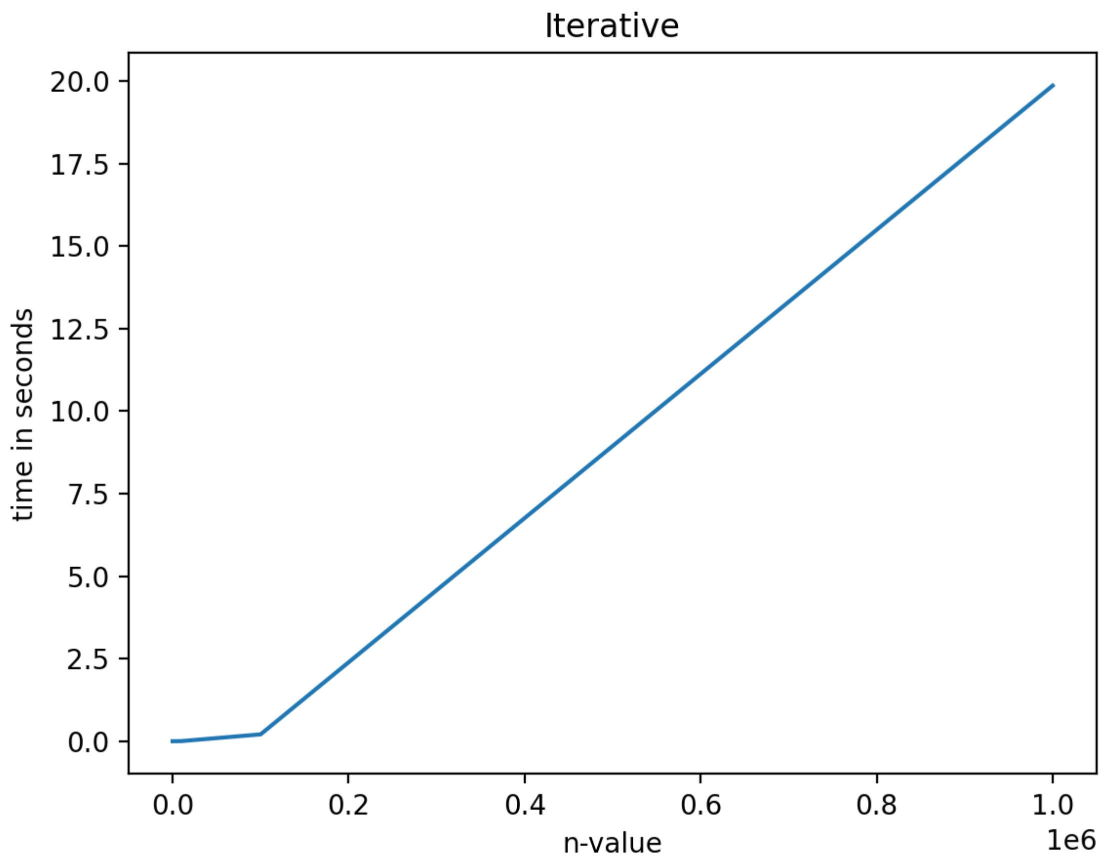
As we can see: a=1. b=2. f(n)= θ(1).
We can also say that f(n)= n^0 (as n^0=1).
Since 0= $\log_2 1 = \log_b a$, this shows that f(n)= θ(n^$\log_b a$), thus we are in case 2 of the master theorem, which provides us with the solution:

T(n)= θ(n^ $\log_b a$ logn)= θ(n^ 0 log n) = **θ(logn)**.

For reference: this image contains the diagram for the predicted running times throughout this report, which is θ(n) for Q1.a.i, θ(logn) for Q1.a.ii, and θ(nlogn) for Q2.

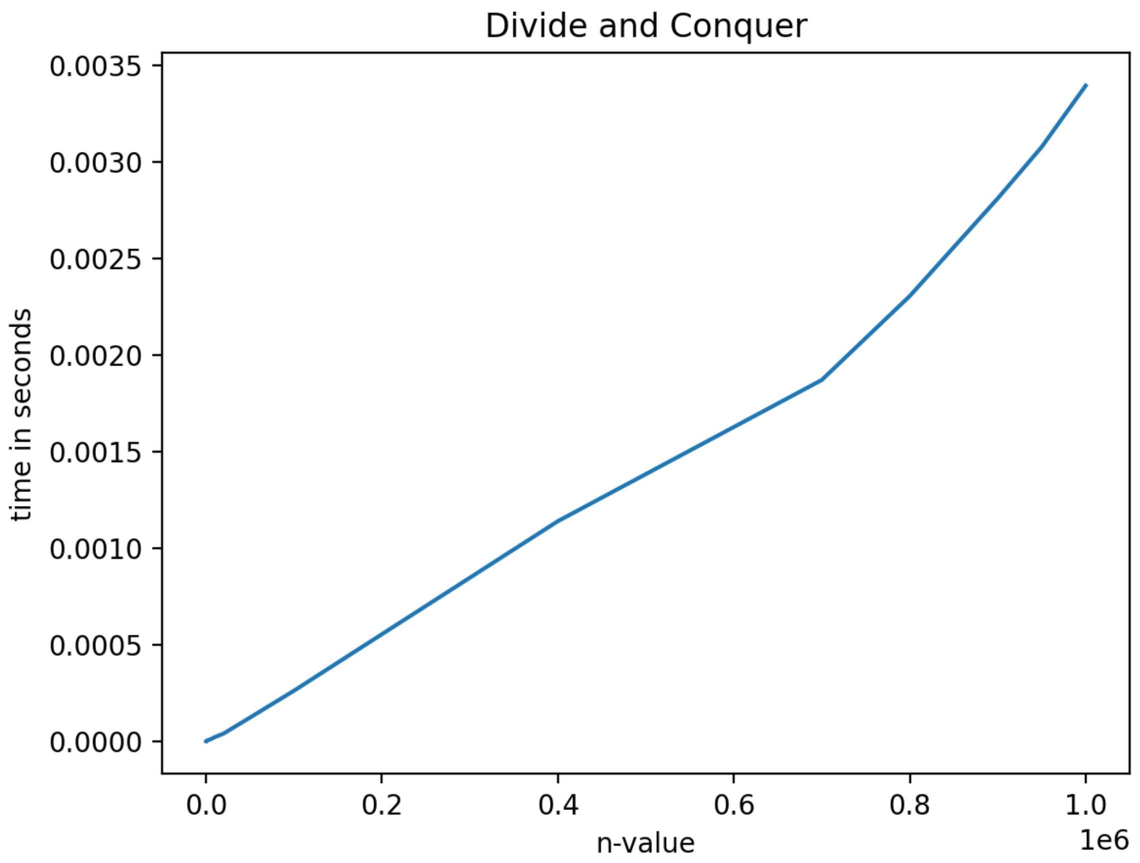


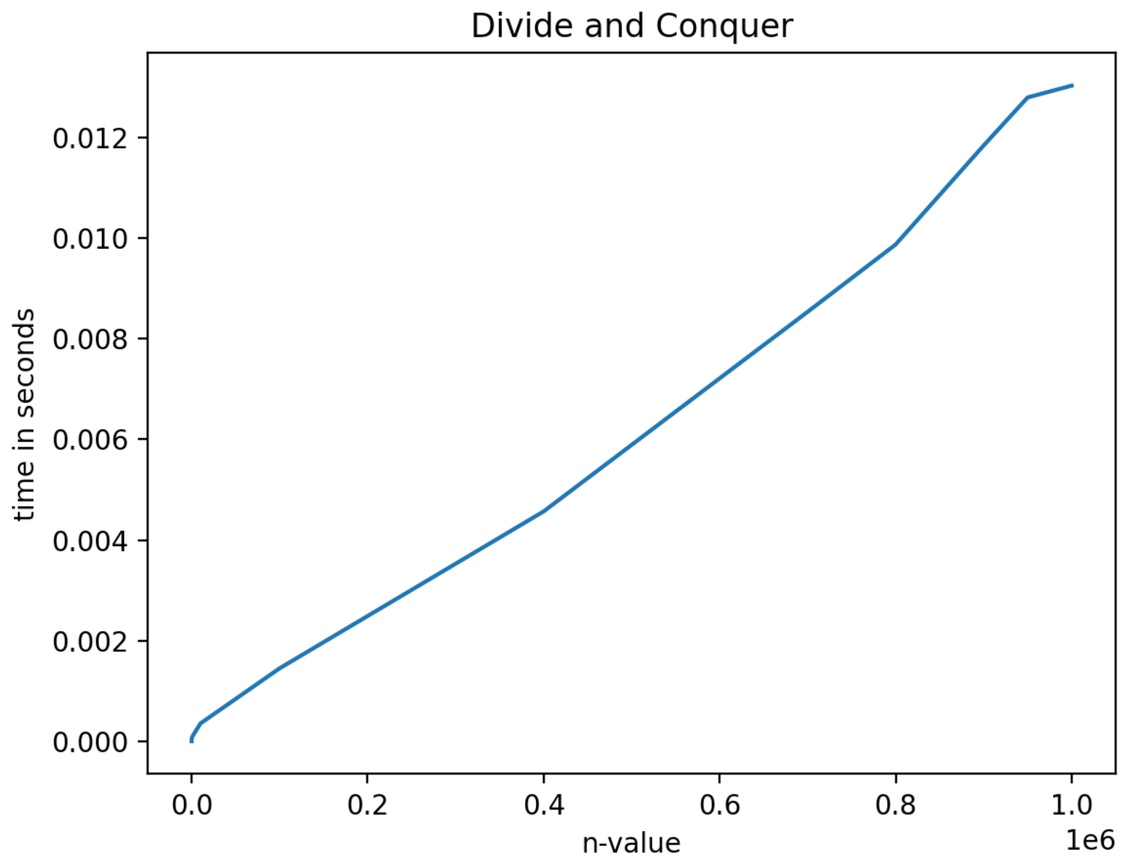
c) i- When running the program on python for different values of n, and using the python matplotlib to plot the graph, this is the obtained graph for inputting n values=[1,100,1000,10000,100000,1000000]:



The graph most resembles the one for θ(n), thus the values certainly match, with few variations due to different processes running and so on.

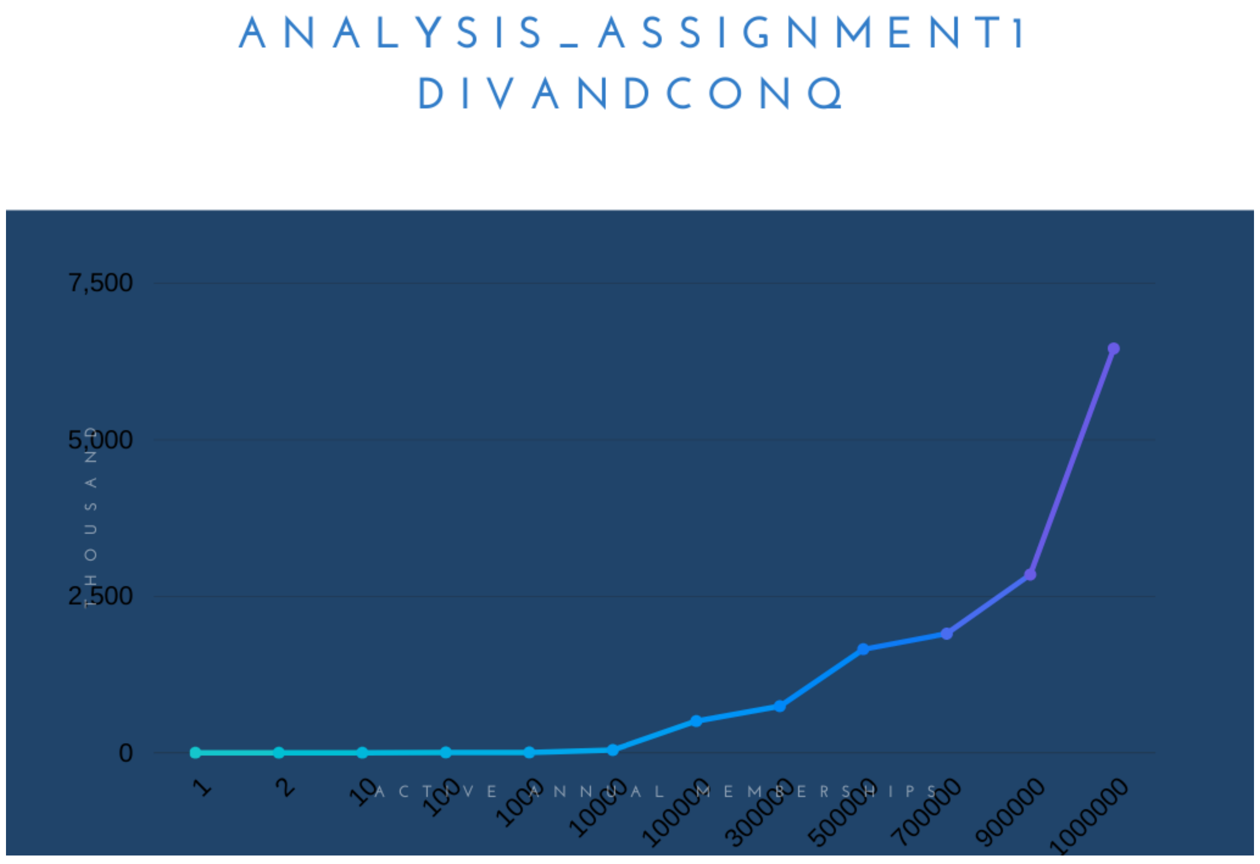
ii- As for the divide and conquer loop, I had to input more values for n to get a more accurate graph. So, when inputting the following n values= [1,100,1000,10000,100000,400000,700000,800000,900000,95000,1000000], these graphs were obtained:


Divide and Conquer
time in seconds
0.0000
0.0005
0.0010
0.0015
0.0020
0.0025
0.0030
0.0035
0.0
0.2
0.4
0.6
0.8
1.0
1e6
n-value



The difference is that in the second graph, I added a time delay between the plotting of each point.
It most resembles the graph for $\log_2(n)$, except for some variations especially when n approaches the value '800,000', but overall it seems to confirm our hypothesis in part b).

To gain more results, I ran the divide and conquer algorithm manually for the following inputs: n= [1,2,10,100,1000,10000,100000,300000,500000,700000,900000,1000000].
The n-time corresponding value in milliseconds were the following: [(1, 2.875), (2, 3.5868), (10, 4.7998), (100, 7.0832), (1000, 10.633), (10000, 48.925), (100000, 510.100), (300000, 747.334), (500000, 1655.541), (700000, 1906.167), (900000, 2848.833), (1000000, 6457.0918)]
I ran it 5 times for each input and recorded the average time, then used Canva to plot a graph with the results:



This graph can show the θ(log$_2$n) resemblance a bit more clearly, thus confirming our predicted running time in part b).

Question 2:

The program segments from part a) could be found on the repository in the file titled ‘Q2.py’.

First, I passed the input list to a ‘mergeSort’ method that sorts it. Then, I used binary search to find the sublist containing inputs smaller than or equal the input value ‘a’.

Finally, I defined and used two pointers, one pointing to beginning and one to the end of the list, to find all pairs that sum up to the given value ‘a’.

b) As for the time complexity, we need to analyze each method we call in our 'sumPairs' method.

'MergeSort' is a recursive method that has a time complexity of θ(nlogn). The reason for this is the following: we divide the list continuously into lists that are half of its size, thus this step has a time complexity of θ(logn). At the end, we merge the lists together, which is a linear time operation, and this is done $\log_2 n$ times, thus the total time complexity becomes $\theta(n\log_2 n)$.

To solve the recurrence for this we can do the following:

Since this is the recurrence function: T(n)= 2 T(n/2) + n, and the function in master theorem has this form: T(n)= a T(n/b) + f(n). Then a=2, b=2, f(n)= n.

Since $\log_b a=\log_2 2=1$, therefore n^$\log_b a$= n^1, therefore f(n)= θ(n^$\log_b a$), thus we're in case 2 of the master theorem. Then we can easily obtain that the solution for this will be T(n)= θ(n^$\log_b a$ log n)= θ(n^$\log_2 2$ log n)= **θ(n log n)**.

As for the 'binarySearch' step, it's also a recursive method that divides my list into a sublist half its size and searches in it instead, thus it's easy to deduce the time complexity in this case which is θ(logn).

Using the master theorem to solve this recurrence: our function has the form: T(n)= T(n/2) + f(n). In our case, f(n)= θ(1). Thus, we obtain this recurrence function: T(n)= T(n/2) + θ(1).

Solving this using the master theorem, when we have the form T(n)= a T(n/b) + f(n). a= 1. b=2. f(n)=1. Comparing n^$\log_b a$= n^$\log_2 1$= n^0=1, we deduce that f(n)= θ(n^$\log_b a$), thus we are in case 2 of the master theorem, which provides us with the solution:

T(n)= θ(n^ $\log_b a$ logn)= θ(n^ 0 log n) = **θ(logn)**.

Finally, the while loop for actually finding the pairs is a linear time operation, but its value depends on the length of the list. The list could have a length of '0' (if all elements in the list are greater than my sum), or it could have length equal to the original list, if all values are less than or equal to the input sum.

In all cases, its value depends on its length 'n', thus it has a time complexity of θ(n).

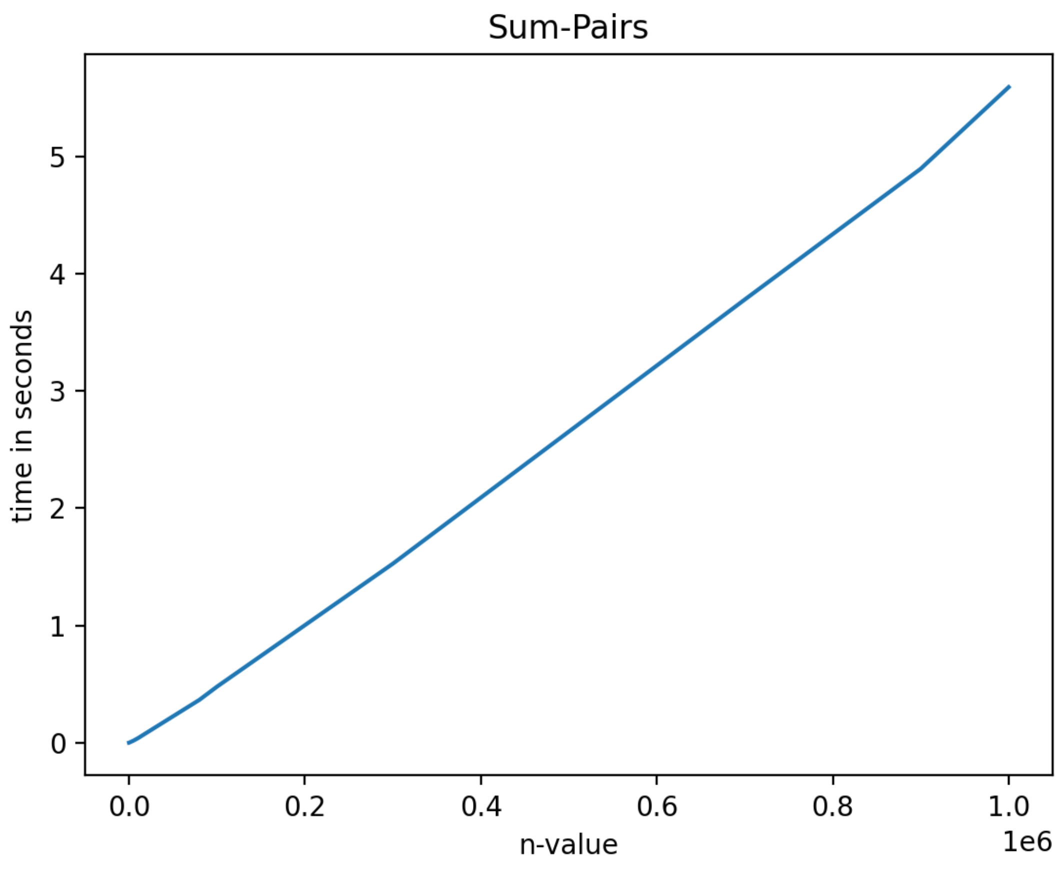
To determine the final asymptotic running time for my algorithm, it is easy to deduce that the time complexity of the 'MergeSort' step will overshadow the rest of the algorithm, thus this algorithm will have a running time complexity of **θ(n logn)**.

c) In order to plot the graph, i filled a list with the values of ‘n’ i wanted to use, which were the following:
[1,100,1000,5000,10000,80000,100000,300000,700000,900000,1000000]

Then in a loop, I iterated over the list, and in each iteration created an array of size of corresponding ‘n’, and filled it with random values ranging between 1 and 100.
Then I chose the sum also to be a random number ranging between the same range (1 and 100).

Finally, I calculated the time before and after calling the method ‘sumPairs’ on the given array and sum value.

This was the obtained graph:



From the graph, we can conclude that it most resembles the (nlogn) plot.

Thus, the empirically observed running time mostly matches with the predicted running time in part b), with a few disparities due to python and different processes running at the same time, plus variations in results.