Problem 1

A) T(n)= 2T(N/2)+n^3-> *Master Theorem*

$a = 2, b = 2$, $n^3 = f(n)$-> $n^{log_b a} = n^{log_2 2} = n$ $n < n^3$
therefore $n^{log_b a+\epsilon}$(*Case*3)

*Case*3 – $\Theta(f(n)) = \Theta(n^3)$ if $2(n/2)^3 \leq cn^3$

$n^3/4 \leq cn^3$ for $c = 1/4$ therefore $\Theta(f(n)) = \Theta(n^3)$

B) T(n)= 7T(N/2)+n^2-> *Master Theorem*

$a = 7, b = 2$, $n^{log_2 7} > n^2$ because $log_2 7 > 2$

(Case 1) $n^{log_b a-\epsilon}$ therefore T(n)=$\Theta(n^{log_2 7})$

C) T(n)= 2T(N/4)+$\sqrt{n}$-> *Master Theorem*

$a = 2, b = 4$, $n^{log_4 2} = n^{1/2} = \sqrt{n} = (f(n))$ (Case 2)
therefore $T(n)$ =$\Theta(n^{log_b a} lgn) = \Theta(n^{log_4 2} lgn)$

D) T(n)= 2T(N/4)+$\sqrt{n}$ *Master Theorem isn't applicable.*

Guess - $O(n^2)$ (*SubstitutionMethod*)
We assume that $T(k) \leq ck^2$ *for all* $k < n$
and show that $T(n) \leq cn^2$
$T(n) \leq c(n-1)^2 + n = c(n^2 - 2n + 1) + n$
$= cn^2 - 2cn + c + n \leq cn^2$
cn^2 (*Desired*) – $(2cn - c - n)$ (*Residual*)
$2cn - c - n \geq 0$ $c(2n-1)-n \geq 0$ for $c \geq 1$ and $n \geq 1$
Therefore T(n)=$O(n^2)$

Ural Sarp Sipahi 28093

Problem 2

A)

1) The best asymptotic worst-case running time of the naive recursive algorithm is O(2^n) because of the fact that in the naive approach the algorithm checks every single subsequence the strings have. The worst case in the LCS algorithm occurs when none of the characters of both strings match. In our case, the line;

Else:

return(max(lcs(X,Y,i,j-1),lcs(X,Y,i-1,j))

keeps getting executed when none of the characters of X and Y match. In every recursive call, the problem gets divided to 2 subproblems for each lcs() call therefore the running time grows exponentially.

As an example if X = “abc” and if Y= “cde” these subsequences would be divided to 2 subproblems such as lcs(abc,cd) and (ab,cde) in the first call. After that lcs(abc,cd) would be divided to 2 subproblems: (abc,c) and (ab,cd). Also lcs(ab,cde) would be divided to 2 subproblems likewise: (ab,cd) and (a,cde) and so on.

2) In the naive approach there could be several repetitions. When we are dividing subsequences into smaller subsequences we could be solving one subsequence that we have already solved before. For the algorithm with memoization this will not be the case. In the algorithm with the memoziation, we create a table of size (m x n) -> (m being the length of the first string and n being the length of the second string) and fill it up recursively as shown in Figure 2. In Figure 2 all of the if, else if and else conditions of function lcs() take O(1) time. Therefore the best asymptotic worst-case running time of the recursive algorithm with memoization is O(mn) -> (The cost of creating the table).

B)

1)

Computer Specs:

CPU: AMD Ryzen 5 2600 (3.4 ghz)

Memory: 16 GB

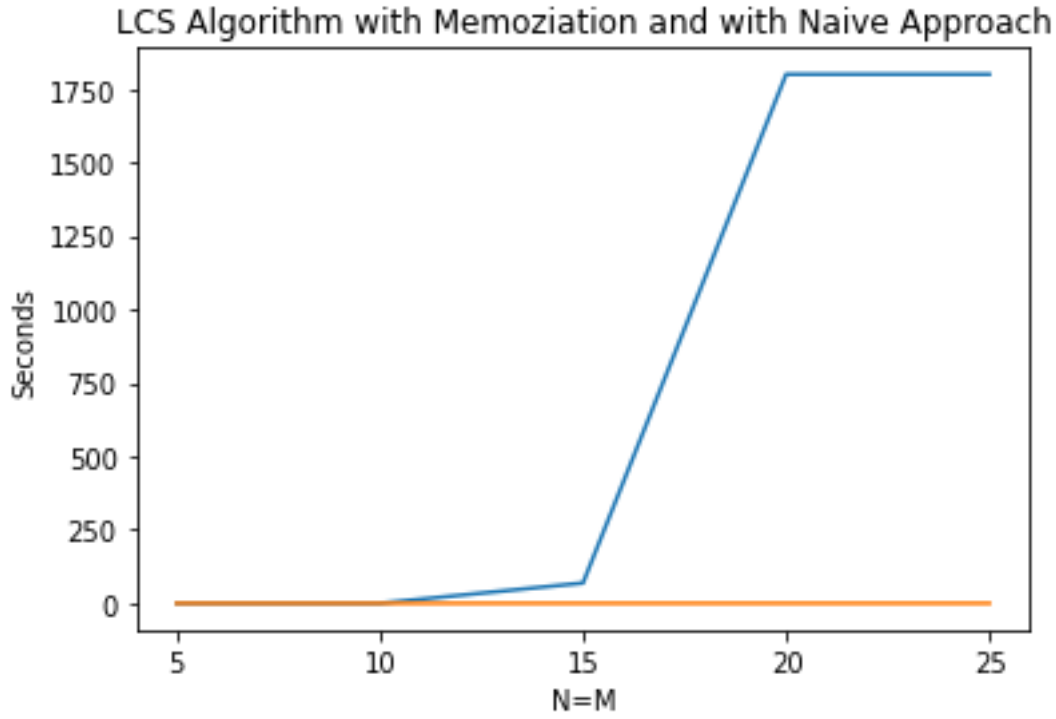
Storage: 1 TB HDD + 256 GB SSD

OS: Win 10 Pro

Algorithm	n=m=5	n=m=10	n=m=15	n=m=20	n=m=25
Naive	0.001000642	0.079018592	69.00250244	T.O.	T.O.
Memoziation	0.001999855	0.000999689	0.001001596	0.001000881	0.001001119

(T.O. = Timed Out. I put a time limit for all the input sizes. Time limit was 30 minutes. The runtimes that are greater than 30 minutes are specified as T.O.)

2)



(Orange line represents LCS Algorithm with Memoziation. Blue line represents LCS Algorithm with Naive approach.) (The seconds limit is set to 1800 due to the reason that I have explained above.)

For small input sizes both of the algorithms' run times are close to each other. However as the input sizes get bigger the algorithm with memoziation is way faster compared to the algorithm with naive approach as can be seen from the graph.

Yes they confirm the result I found on a) for the algorithm with the naive approach. I found that the naive approach algorithm was a O(2^n) algorithm and it can be seen from the graph that the values of the blue line are exponentially increasing. In addition the values for the input sizes 20 and 25 are not shown in the graph for the blue line. They would be much higher than what is shown in the graph (Especially n=m=25).

I also found that the LCS algorithm with memoziation was a O(m*n) algorithm. For our case n=m therefore it is an O(n^2) algorithm. However by looking at this graph we cannot conclude that the memoization algorithm is growing quadratically. This algorithm needs to be tested with much bigger input sizes to see if the theoretical results hold.

C)

1)

Algorithm	n=m=5		n=m=10		n=m=15		n=m=20		n=m=25	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Naive	0.001000380516052246	6.17766099364569e-07	0.005634657541910807	0.007314966446400386	0.33487413724263 51	0.25285954332817956	36.950972723960874	37.19870396745467	**T.O.**	**T.O**
Memoization	0.001000181833902994 8	4.150048498776158e-07	0.001000332832336425 7	6.420379993703221e-07	0.001033528645833333	0.000182600766382 97343	0.001000102361043 2943	5.667433150863746e-07	0.001000507672627767	5.995167087615311e-07

(For the average case analysis I put a time limit for all the input sizes again. The naive algorithm with input size n=m=25 took more than 30 minutes to run for 5 different random DNA sequences therefore it is marked as timed out.) (All the average time values in the graph are measured in seconds.)

95% confidence intervals are:

For Naive Algorithm:

n=m=5 -> 0.001000380516052246 ± 2.3031338724634353e-07

n=m=10 -> 0.005634657541910807 ± 0.0027271400965458137

n=m=15 -> 0.3348741372426351 ± 0.09427020676819076

n=m=20 -> 36.950972723960874 ± 13.868290151775605

For Memoization:

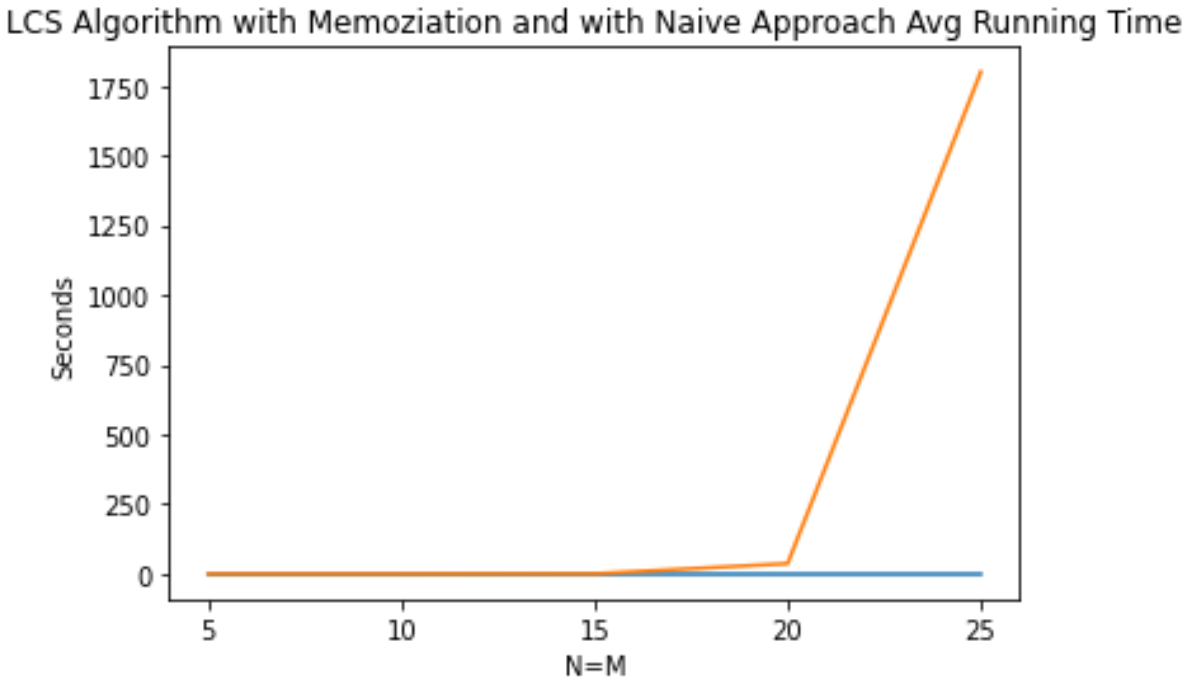
n=m=5 -> 0.0010001818339029948 ± 1.5472065041653838e-07

n=m=10 -> 0.0010003328323364257 ± 2.3936235175083727e-07

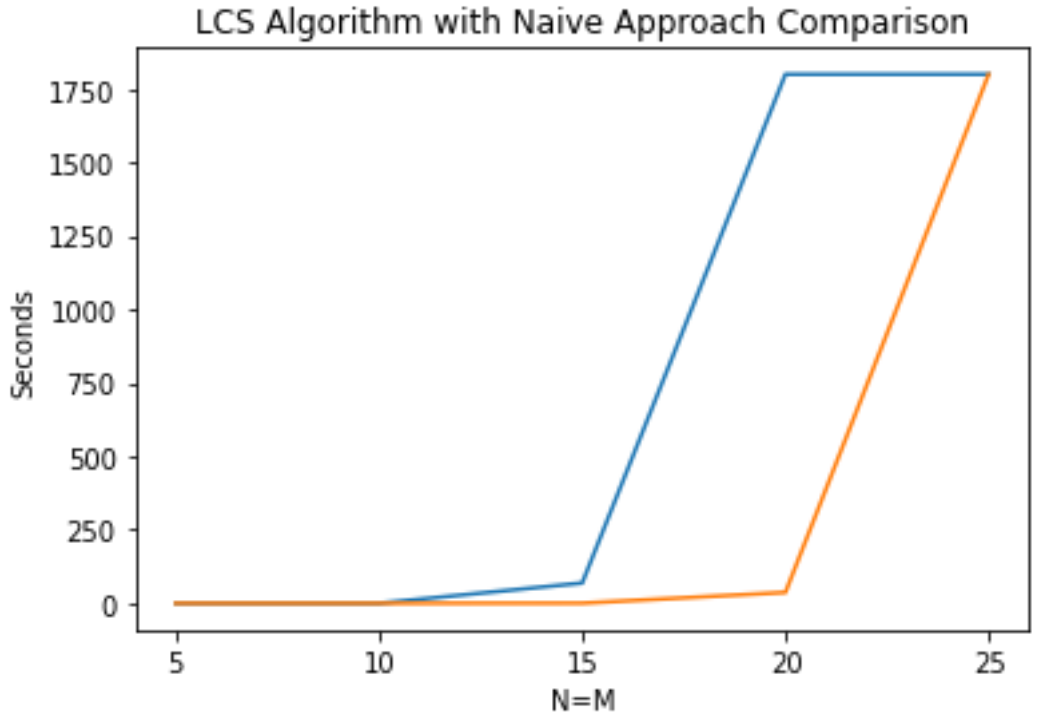
n=m=15 -> 0.0010335286458333333 ± 6.807657633317663e-05

n=m=20 -> 0.0010001023610432943 ± 2.1129125203054337e-07

n=m=25 -> 0.0010000507672627767 ± 2.2350971354315632e-07



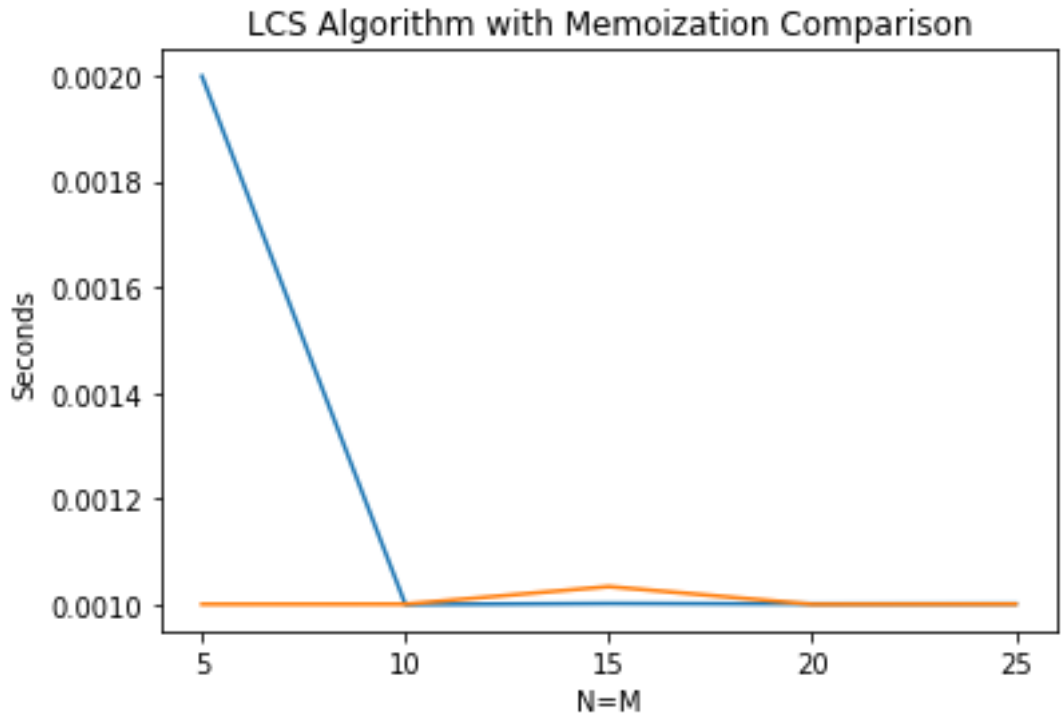
As can be seen from the graph the growth of the functions are similar to the first graph on the previous question. Orange line grows exponentially again.



(Blue line represent worst case, orange line represent average case)

As can be seen from the graph, the average case of the naive algorithm is faster than the worst case but eventually both of them are growing exponentially. (For the worst case n=m=20, n=m=25 and for

the average case n=m=25 are capped at 1800 seconds as explained earlier due to the fact that these algorithms get timed out after 30 minutes.)



(Blue line represent worst case, orange line represent average case)

As can be seen from the graph for the algorithm with memoization, all the input sizes take way less than 1 second and the values are very close to each other. The instances that we have generated are not sufficient to measure the growth of the function for the memoization approach. Possibly with greater input sizes, the growth of the function can be seen as quadratic which is the theoretical result that was found on part b.