CS301-HW1

Gülsen Görkem Köse - 25359

February 2020

1 Question 1

(a) T(n) = $\theta(n^3)$

Explanation: Since $a = 2 \geq 1$, $b = 2 > 1$ *and* n^3 *is asymptotically positive, we can use the Master Theorem. Since* $f(n) = n^3 = \Omega(n^{\log_2 2+\epsilon})$ *for some* $\epsilon > 0$ *and* $2.f(\frac{n}{2}) = 2.(\frac{n}{2})^3 \leq c.f(n) = c.n^3$ *for some* c *such that* $\frac{1}{4} \leq c < 1$ *and for large n, the third case applies. Then* $T(n) = \theta(f(n)) = \theta(n^3)$*, which is the asymptotic tight bound for T(n).*

(b) T(n) = $\theta(n^{\log_2 7})$

Explanation: Since $a = 7 \geq 1$, $b = 2 > 1$ *and* n^2 *is asymptotically positive, we can use the Master Theorem. Since* $f(n) = n^2 = O(n^{\log_2 7-\epsilon})$ *for some* $\epsilon > 0$*, the first case applies. Then* $T(n) = \theta(f(n)) = \theta(n^{\log_2 7})$*, which is the asymptotic tight bound for T(n).*

(c) T(n) = $\theta(\sqrt{n}.\log n)$

Explanation: Since $a = 2 \geq 1$, $b = 4 > 1$ *and* $\sqrt{n}$ *is asymptotically positive, we can use the Master Theorem. Since* $\sqrt{n} = \theta(n^{log_4 2})$*, the second case applies. Then Then* $T(n) = \theta(n^{\log_4 2}.\log n) = \theta(\sqrt{n}.\log n)$*, which is the asymptotic tight bound for T(n).*

(d) T(n) = $\theta(n^2)$

Explanation: Since this recursion is not in the form of $T(n) = a.T(\frac{n}{b}) + f(n)$*, we cannot use the Master Theorem. Therefore, we will use Substitution Method for part (d). In this method, tight bounds can be shown by:* $T(n) = O(g(n))$ *and* $T(n) = \Omega(g(n)) \Leftrightarrow T(n) = \theta(g(n))$*. So, in order to find the asymptotic tight*

bound, we will both prove upper and lower bound.
Upper bound: Claim that $T(n) = O(n^2)$. *Then we will use induction to prove, and to prove that we need to show:* $\exists c, n_0 \geq 0$ *such that* $\forall n \geq n_0 : T(n) \leq cn^3$.
Base case: $T(1) \leq c1^2 \leq c$
Inductive step: Assume that $T(n) \leq ck^2$ *for all* $k < n$ *as an inductive hypothesis. We'll show that* $T(n) \leq cn^2$.

$$T(n) = T(n-1) + n$$

$$\leq c(n-1)^2 + n$$

$$= cn^2 - 2cn + c + n$$

$$= cn^2 - (2cn - c - n)$$

and $2cn - c - n$ *is nonnegative if* $c \geq 1, n \geq 1$.
Therefore, $T(n) \leq cn^2$ *so,* $T(n) = O(n^2)$
Lower bound: Claim that $T(n) = \Omega(n^2)$. *Then we will use induction to prove.*
Base case: $T(1) \geq c1^2 \geq c$
Inductive step: Assume that $T(n) \geq ck^2$ *for all* $k \leq n$ *as an inductive hypothesis. We'll show that* $T(n) \geq cn^2$.

$$T(n) = T(n-1) + n$$

$$\geq c(n-1)^2 + n$$

$$= cn^2 - 2cn + c + n$$

$$= cn^2 + (c + n - 2cn)$$

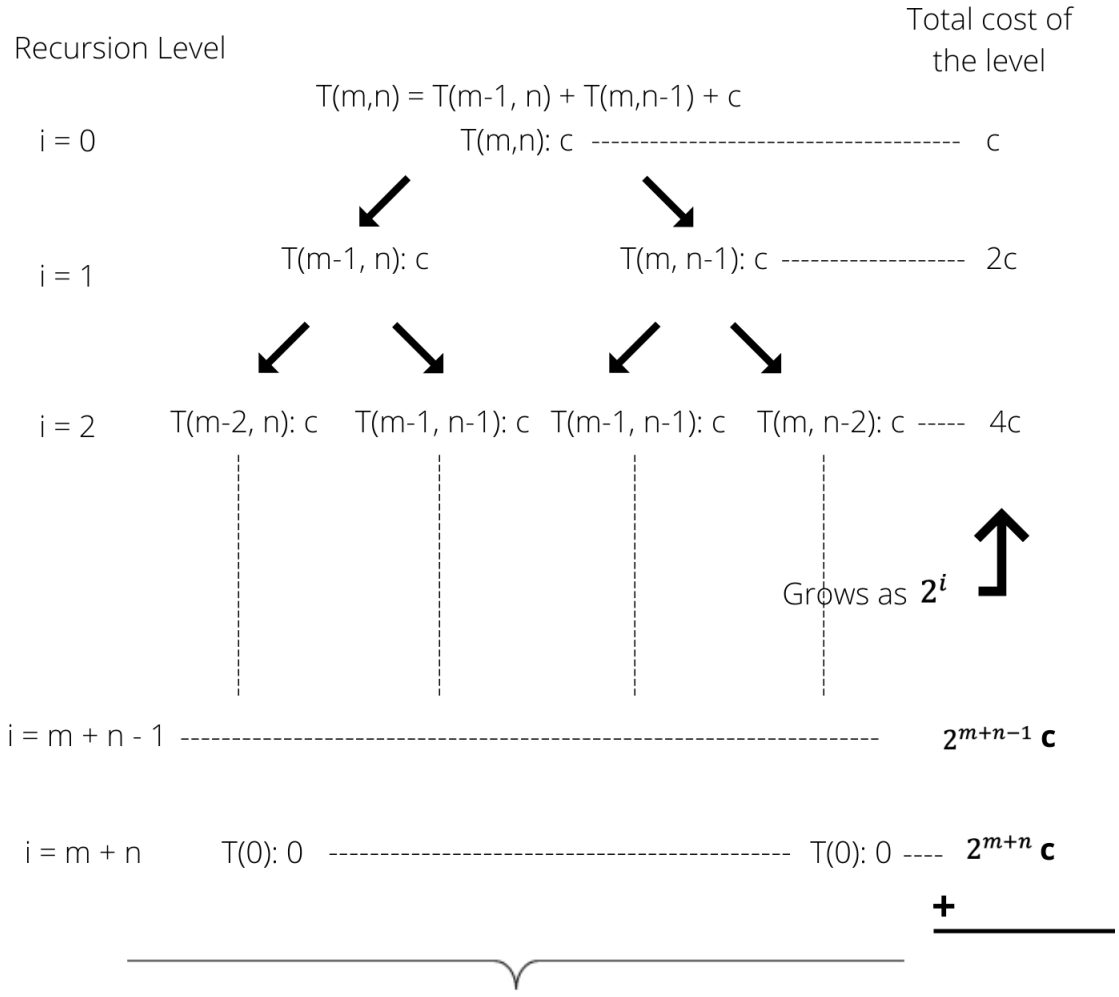
and $c + n - 2cn$ *is nonnegative if* $c \leq \frac{1}{2}, n \geq 1$.
Therefore $T(n) \geq cn^2$ *so,* $T(n) = \Omega(n^2)$.
Since we both proved that $T(n) = O(n^2) and T(n) = \Omega(n^2), T(n) = \theta(n^2)$.

2 Question 2 (a)

(i) *The worst case happens when there is no common subsequence, because in this case at each iteration rightmost characters become never same, so algorithm makes 2 recursive calls, instead of 1. To find the best asymptotic worst-case running time, we will write a recurrence relation to express the work done by the algorithm and then proceed the recursion using a recursion tree in order to find the best asymptotic worst-case running time.*
We can express the algorithm as the following recurrence relation.

$$T(m, n) = T(m-1, n) + T(m, n-1) + O(2)$$

since at each step we make 2 recursive calls and take their max, that's why we have additional work of O(2) at the end.
In the following recursion tree, O(2) is denoted by c.



number of leaves = 2^i = 2^{m+n}

When we add the cost of levels:

$$T(m,n) = c + 2c + 4c + \ldots + 2^{m+n-1}c + 2^{m+n}T(0)$$

$$= \sum_{k=0}^{i-1} 2^k = 2^i - 1 = 2^{m+n} - 1$$

T(m,n) = $\theta(2^{m+n})$

Therefore, the best asymptotic worst-case running time is $\theta(2^{m+n})$ for the naive algorithm.

(ii)

The worst case happens when there is no common subsequence, because in this case at each iteration rightmost characters become never same, so algorithm makes 2 recursive calls, instead of 1. But since the memoization technique divides the problem into smaller subproblems and store the solutions for that subproblems so that they would not be computed several times, the only thing that we need to do is to find an upper bound for the number of possible subproblems that the algorithm encounters during the execution. The maximum number of subproblems acts like a worst-case scenario here.

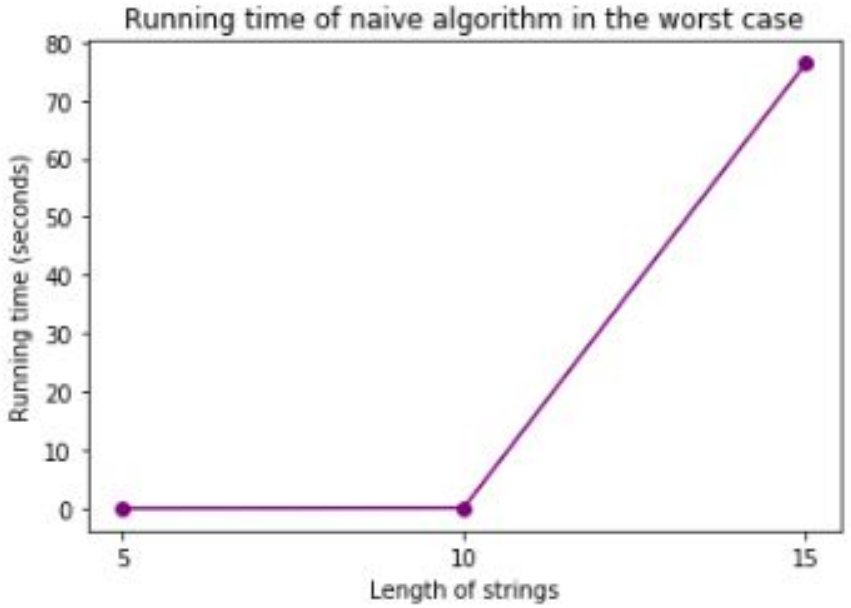
In the array c, the allocated memory is $(m+1)(n+1)$; which will be the cells of the matrix that the subproblems would be stored. So the maximum number of subproblems is $(m+1)(n+1)$. Each time the memoized function is called, if the current subproblem is already computed and stored in the matrix, it returns the result in O(1) time. The first times when there is a return from cascaded calls, is when either $i = 0$ or $j = 0$, and the value of the corresponding cell is updated as 0. The other returns through the cascaded calls will be always the maximum of the already computed two cells and taking the maximum takes O(2) in this case. Hence, the total running time is equal to the number of cells of the matrix, which is the number of subproblems computed: $\theta((m+1)(n+1)) = \theta(mn)$

3 Question 2 (b)

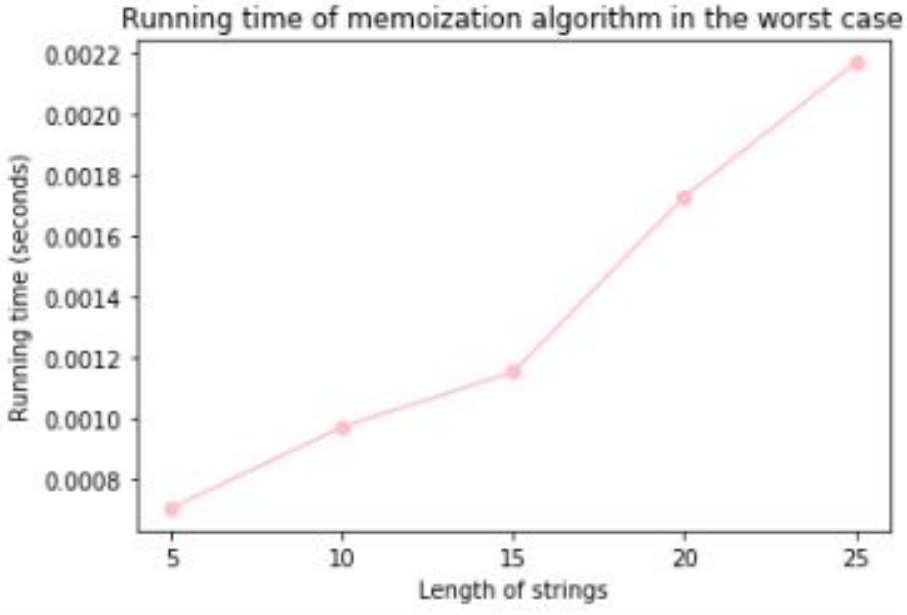
(i) *The following table consists of the running seconds of the specified algorithms with the specified input sizes. Both of the algorithms were executed in the same machine with CPU 2.50 GHz - 2.71 GHz and 8 GB RAM. The algorithms were executed on Google Colab. Unfortunately, there is no running time information for the naive algorithm with input sizes 20 and 25, due to the exponential time complexity.*

Algorithm	m = n = 5	m = n = 10	m = n = 15	m = n = 20	m = n = 25
Naive	0.0019794259	0.0992063469	76.3548994200	-	-
Memoization	0.0007034780	0.0009704389	0.0011508489	0.0017260639	0.0021668070

(ii) *The experimental results for the naive algorithm are plotted in the following graph.*



The experimental results for the memoization algorithm are plotted in the following graph.



(iii) *Naive algorithm:*
The naive implementation for Longest Common Subsequence is definitely not scalable since even with the input sizes $m = n = 20$ or $m = n = 25$ which are not considerably huge input sizes, it is not possible to get the output for hours. Moreover, for the input sizes $m = n = 5$, $m = n = 10$, and $m = n = 15$ that I can get the output, the graph is also exponential. And this confirms the theoretical results from part (a), which is also exponential $\theta(2^{m+n})$.

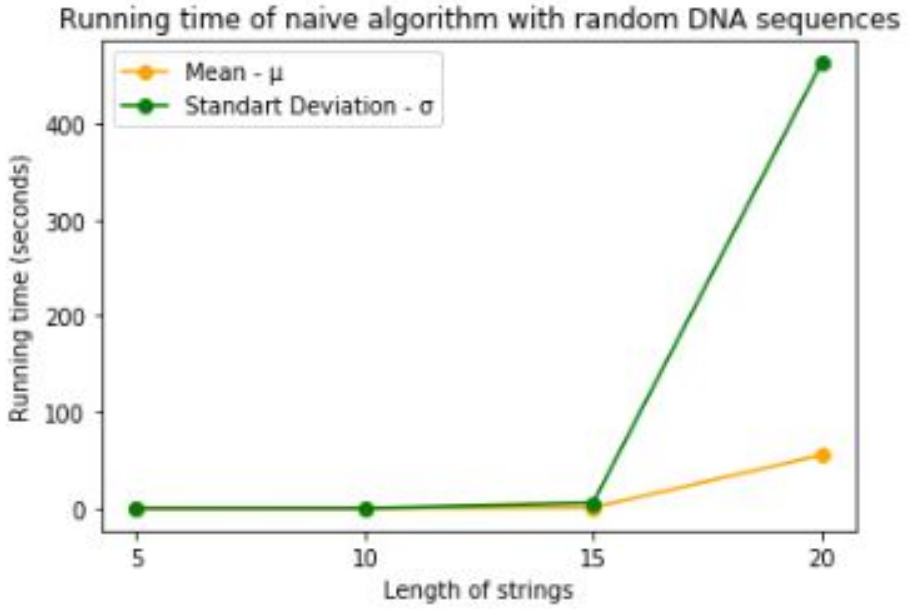
Memoization algorithm:
The memoization implementation for the Longest Common subsequence is much more scalable than the Naive implementation since getting the outputs for the specified input sizes did not take long as the naive version. And this confirms the theoretical results from part (a) since the graph that is formed based on the experimental results looks like a polynomial of degree 2, which is same as the theoretical result $\theta(mn)$. But the fact that the graph is indeed quadratic is not really obvious to notice, since the graph is plotted out of just 1 instance of data for each input size.

4 Question 2 (c)

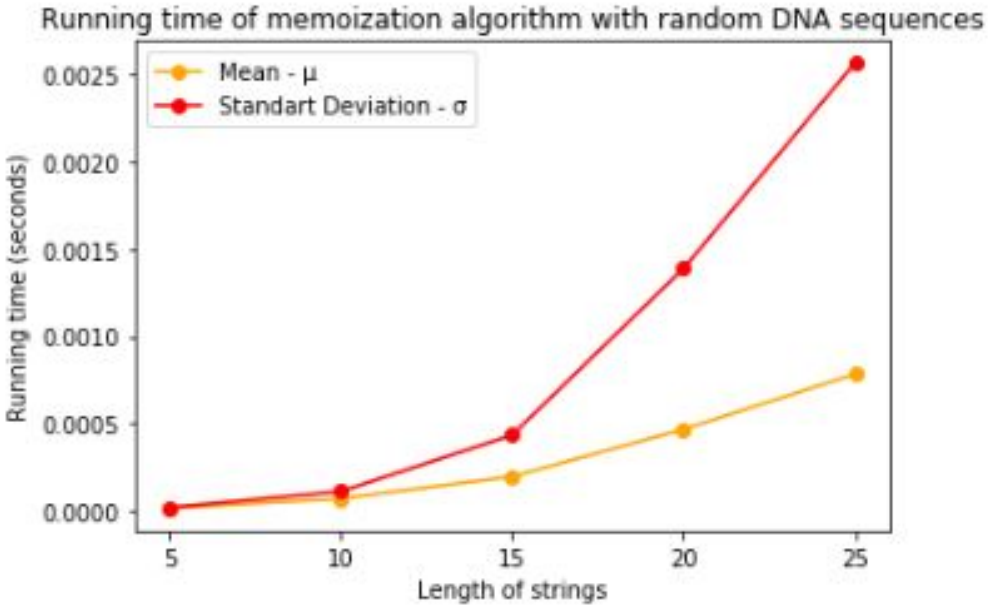
(i) *The following table consists of the average running seconds and standart deviations of the specified algorithms with the specified input sizes. Both of the algorithms were executed in the same machine with CPU 2.50 GHz - 2.71 GHz and 8 GB RAM. The algorithms were executed on Google Colab. Unfortunately, there is no average running time and standart deviation information for the naive algorithm with input size 25, due to the exponential time complexity.*

	m = n = 5		m = n = 10		m = n = 15		m = n = 20		m = n = 25	
Algorithm	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Naive	7.89E-05	0.0006311669	0.0061883001	0.0348317726	0.6216943136	5.579793516	55.2880639	462.8124499	-	-
Memoization	1.43E-05	1.82E-05	6.94E-05	0.0001100639	0.000195537	0.0004335797	0.0004663063	0.0013876501	0.000782715	0.0025647844

(ii) *The experimental results for the naive algorithm that is executed with 30 pair of randomly generated DNA sequences are plotted in the following graph.*



The experimental results for the memoization algorithm that is executed with 30 pair of randomly generated DNA sequences are plotted in the following graph.



(iii) *Naive Algorithm: The average running time of naive algorithm again grows rapidly as plotted in the graph above. But compared with the worst-case in part(b), the average running time for randomly generated 30 pair of DNA sequences; I got the output for the input size* $m = n = 20$ *and this stems for the fact that since there are only 4 bases that a DNA sequence could consist of, at some of the recursive calls; instead of new 2 additional recursive calls, there would be 1. Therefore, it may not possible to worst-case taking place all the time. That's why it is possible to get the output for* $m = n = 20$*, but still there is no output for* $m = n = 25$ *because the average running time grows rapidly likewise. The naive algorithm is not scalable in this case as well, but it is better compared to the worst-case version in part(b). It may still be exponential since the graph looks like an exponential graph, yet there is no theoretical result to prove that. We can only say that even for the cases which the strings have common subsequences, naive algorithm is not applicable and does not work well.*

Memoization Algorithm: The average running time of memoization algorithm again grows as quadratic as plotted in the graph above. Compared to the worst-case in part (b), this time it is much more obvious that graph grows as quadratic, since the graph is formed out of the average running time of the algorithm with a sample set of 30 pair of randomly generated DNAs, instead of just 1 instance. Like I stated above, the average running time is much smaller than the worst-case version in part(b) for each input size. This is caused by the fact that it is not possible for the worst case to take place all the time since there are only 4 bases to form a DNA sequence. And for the sample data set of 30 pair of DNA sequences, the memoization algorithm is much more scalable.

For this homework, I collaborated with Deniz Cangı. Yet the solutions, proofs and explanations reflect my own understanding.