Taner Giray Sönmez
2023, Fall

CS 301: Algorithms
Assignment 1

Problem 1

Part (a):

By The Master's Theorem: $a = 2 \geq 1, b = 2 \geq 1$, f(n) = n^3 (asymptotically positive)
We need $n^{\log_b a}$:
$n^{\log_b a} = n^{\log_2 2} = n$
We need to compare $n^{\log_b a}$ with f(n):
f(n)=n^3= $\Omega(n^{\log_b a+\epsilon}) = \Omega(n^{\log_2 2+\epsilon}) = \Omega(n^\epsilon)$
Case 3 may apply, we need to check:
$af(n/b) \leq$ (?) cf(n)
$2(n/2)^3 \leq$ (?) cn^3
$n^3/4 \leq cn^3$, for $1/4 \leq c < 1$
Hence T(n) $= \Theta(f(n)) = \Theta(n^3)$

Part (b):

By The Master's Theorem: $a = 7 \geq 1, b = 2 \geq 1$, f(n) = n^2 (asymptotically positive)
We need $n^{\log_b a}$:
$n^{\log_b a} = n^{\log_2 7} = n^{2.8074}$
We need to compare $n^{\log_b a}$ with f(n):
f(n)=n^2= $O(n^{\log_b a-\epsilon}) = O(n^{\log_2 7-\epsilon}) = O(n^{2.8074-\epsilon})$ for any $0 < \epsilon < 0.8074$
Case 1 applies, therefore:
Hence T(n) $= \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7}) = \Theta(n^{2.8074})$

Part (c):

By The Master's Theorem: $a = 2 \geq 1, b = 4 \geq 1$, f(n) = $\sqrt{n}$ (asymptotically positive)
We need $n^{\log_b a}$:
$n^{\log_b a} = n^{\log_4 2} = n^{0.5}$
We need to compare $n^{\log_b a}$ with f(n):
f(n)=$\sqrt{n}$= $\Theta(n^{\log_b a}) = O(n^{\log_4 2}) = O(n^{0.5})$
Case 2 applies, therefore:
Hence T(n) $= \Theta(n^{\log_b a}\log n) = \Theta(n^{\log_4 2}\log n) = \Theta(n^{0.5}\log n) = \Theta(\sqrt{n}\log n)$

Part (d):

Step 1 : Claim that T(n) = O(n^2)
Step 2 : Use induction to prove T(n) = O(n^2)
In order to prove this, we need to show: $\exists c, n_0 \geq 0$ such that $\forall n \geq n_0 : T(n) \leq cn^2$
Induction base: $T(1) \leq c1^2 \leq c$
We can always pick c big enough to satisfy this
Inductive step:
Assume that $T(k) \leq ck^2$ for all k < n (induction hypothesis)
Show that T(n) $\leq cn^2$
T(n) = T(n - 1) + n $\leq c(n-1)^2 + n$, where $c(n-1)^2 + n = cn^2 - 2cn + c + n$
T(n)$\leq cn^2 - 2cn + c + n \leq$ (?) cn^2
T(n)$\leq cn^2 - (2cn - c - n) \leq cn^2$, $2cn - c - n$ is positive when $c \geq 2$ and $n \geq 2$
Hence T(n) $\leq cn^2$ which means that **T(n) = O(n^2)**

Problem 2

Part (a):

(i):

In the **worst-case** scenario where there's **no common subsequence** between input strings X and Y, the algorithm continually makes two recursive calls at each step, leading to an exponential number of calls and making the algorithm highly inefficient for cases with minimal or no shared elements between the strings. In other words, in the worst case scenario the algorithm will always execute **else part** which is $max(lcs(X, Y, i, j-1), lcs(X, Y, i-1, j))$ until one of the i or j is getting zero. Moreover, we should add O(k) which is time complexity of the max function in python. (Let's denote length of the string X as m, and length of the string Y as n for simplicity even if they have the same length) By considering these we can write the worst case scenario recurrence relation accordingly:
$T(m, n) = T(m-1, n) + T(m, n-1) + O(k)$

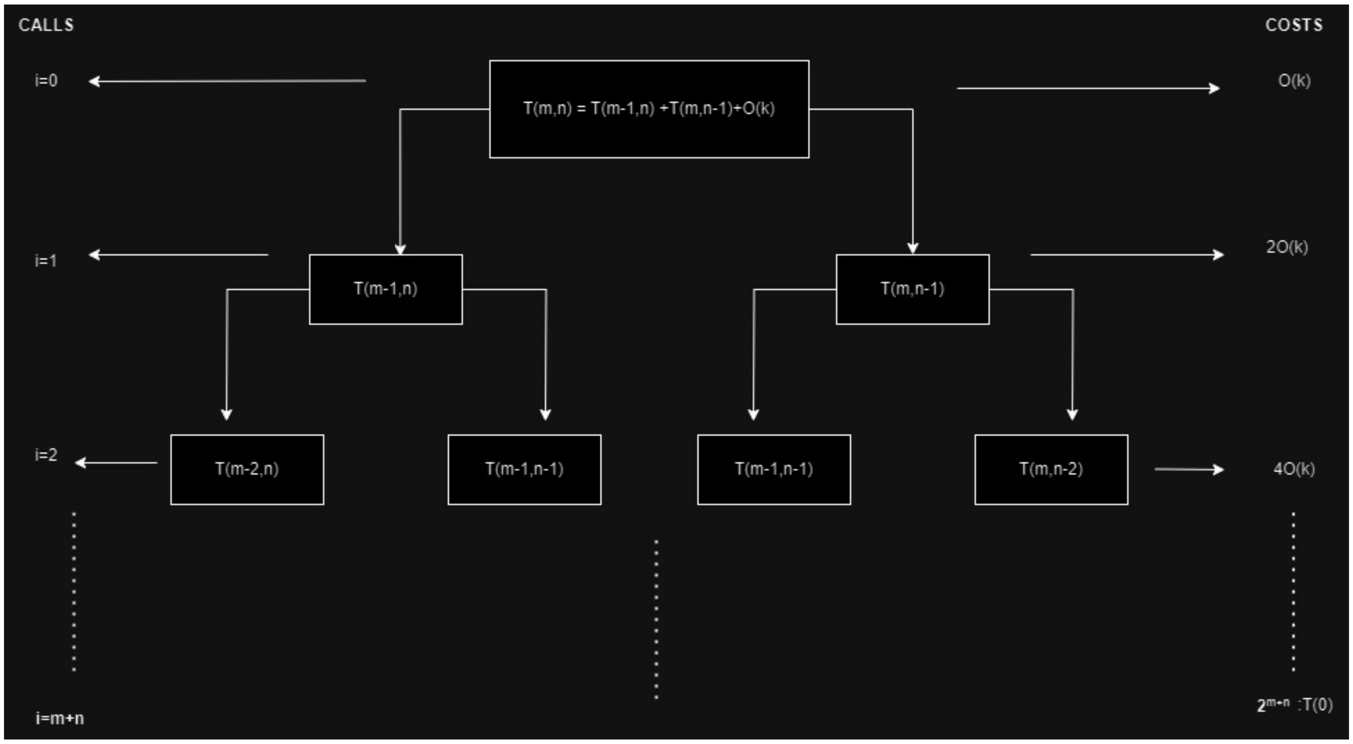

Figure 1: Recursion Tree

When drawing recursion tree, I noticed that costs are growing as 2^i. Moreover, recursive calls will stop when we get T(0), meaning that when cost reaches 2^{m+n}(total the length of strings). If we sum all of these costs, then we get the running time:
$T(m, n) = O(k) + 2O(k) + 4O(k) + ... + 2^{m+n}T(0)$
By geometric sum formula:
$T(m, n) = \sum_{k=0}^{m+n-1} 2^k + 2^{m+n}T(0) = 2^{m+n} - 1 + 2^{m+n}T(0)$
$T(m, n) = 2^{m+n}(T(0) + 1) - 1 = 2^{m+n}(c) - 1$ where c is constant real number
Hence it is $\theta(2^{m+n})$ where m and n are the length of the strings
Also, $\theta(2^{n+n})$ since m=n

(ii):

The worst case scnerio happens when there is no common subsequence between strings meaning that algorithm computes and stores the results for all possible combinations of i and j, where i and j can go from 0 to m or n where

m and n are the length of the strings. The algorithm avoids redundant calculations by reusing stored results which reduces total costs in the recursion tree. For each unique pair of i and j, the algorithm only computes the LCS length once and then retrieves it from the matrix in subsequent calls. Therefore, we only need to compute possible combinations of the pair of i and j. Since i and j get values between 0 to m and 0 to n respectively.
$0 \leq i \leq m$
$0 \leq j \leq n.$
We got m+1 options for i and n+1 options for j. In total we have $(m+1)*(n+1)$ possible combinations.Meaning that:
$T(m,n) = \theta((m+1)*(n+1)) = \theta(mn+m+n+1) = \theta(mn)$ **which is $\theta(m^2)$ where m=n**

Part (b):

(i):

Specs of my machine:
CPU: AMD Ryzen 7 3700X 8-Core Processor 3.6 GHz
RAM: 16GB
OS: x64 Windows 10
GPU: NVIDIA RTX 3070

Table 1: Worst case running times in seconds

Algorithm	m = n = 4	m = n = 8	m = n = 12	m = n = 16	m= n = 20
Naive	0.00100	0.00575	1.12875	250.385	-
Memoization	0.00021	0.00047	0.00112	0.00258	0.00412

Since naive algorithm grows exponentially with the increasing input size, I could not calculate the time for input size 20. However, memoziation algorithm relatively faster, hence i could calculate the time for input size 20.

(ii):

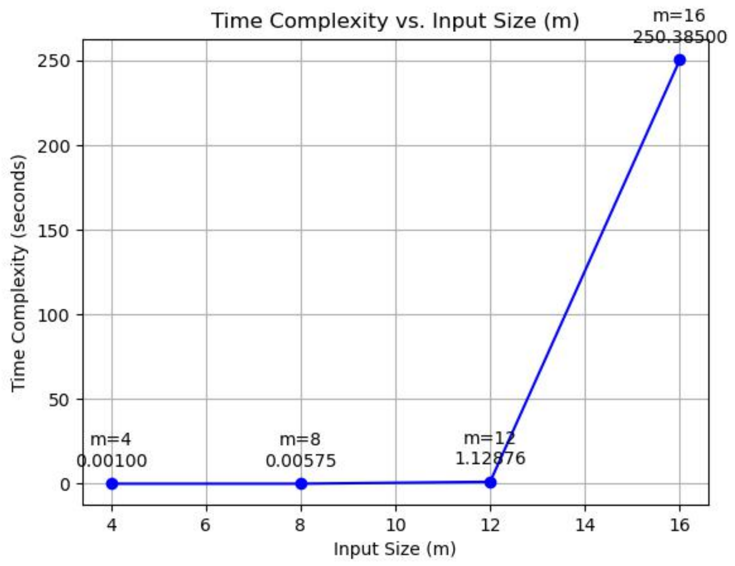

Figure 2: Naive Algorithm

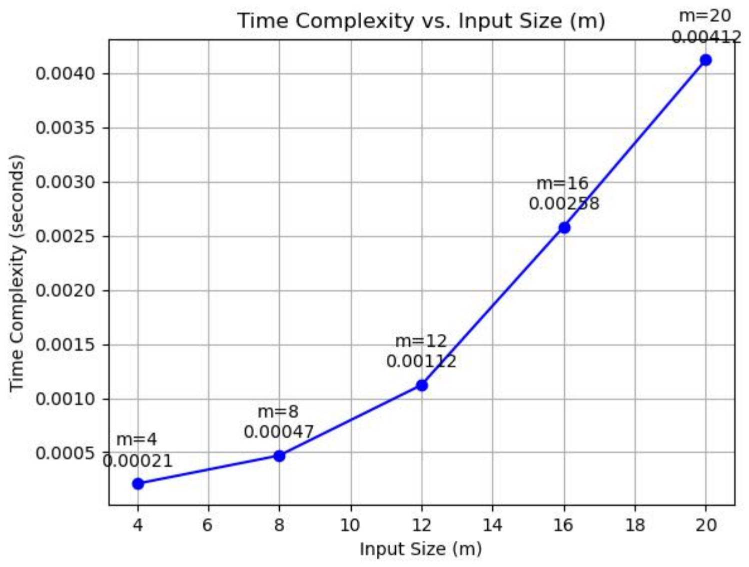

Figure 3: Memoziation Algorithm

(iii):

A memoization algorithm is a more efficient and time-saving approach. When using a naive approach, it surpasses the time limit when dealing with longer strings. This is natural since naive algorithm time complexity $\theta(2^{m+n})$ is bigger than memoization algorithm time complexity which is $\theta(mn)$. In the question above(q2 part a) where I calculated time complexity, I found an exponential growth($\theta(2^{m+n})$) for the naive algorithm. Likewise, when I put the values on the graph, I obtained an exponential graph, which showed that I had found the time complexity correctly.I couldn't calculate the time for an input size of 20 for naive algorithm, which demonstrates how quickly the algorithm's complexity increases. Furthermore, I found quadratic growth($\theta(m^2)$ where m=n) for memoization algorithm. Similarly, when I put the values on the graph, I obtained kind of a quadratic graph(if we run for different input sizes and more input sizes, we can observe that graph will look quadratic), which again showed that I had found the time complexity correctly. In other words, computation time grows exponentially when input size increases in the naive algorithm. Besides, computation time grows quadraticly when input size increases in the memoization algorithm. With these results we can conclude that memorization algorithm is more scalable (When calculating timings, I performed 10 measurements for each input size and then averaged the results to ensure more accurate and reliable results).

Part (c):

Table 2: Average running times in seconds

Algorithm	m = n = 4		m = n = 8		m = n = 12		m = n = 16		m = n = 20	
	µ	σ	µ	σ	µ	σ	µ	σ	µ	σ
Naive	0.001	0.0000004	0.0009	0.0000006	0.0092	0.0004	0.8	0.006	10.45	0.029
Memoization	0.001	0.0000006	0.0011	0.0000005	0.0013	0.00000049	0.0016	0.000001	0.0021	0.0000005

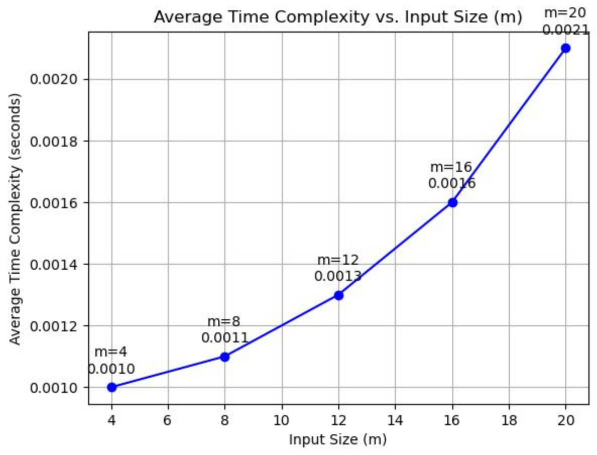

Figure 4: Average Time Complexity of Memoization Algorithm

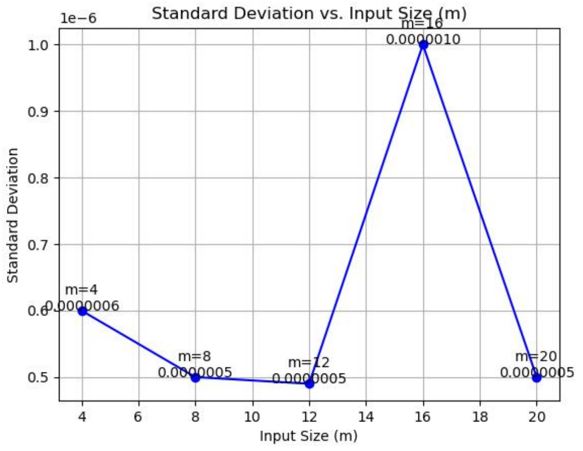

Figure 5: StDev of Memoization Algorithm

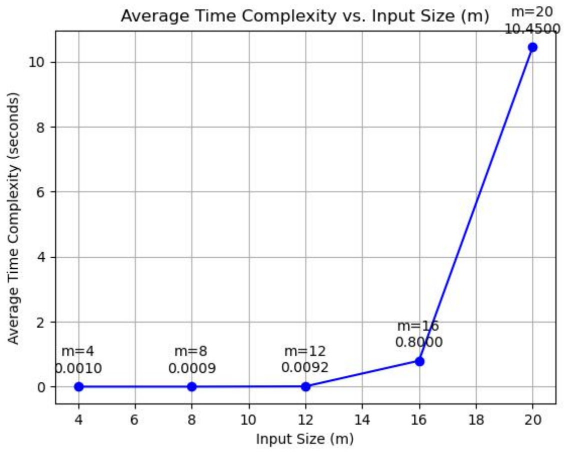

Figure 6: Average Time Complexity of Naive Algorithm

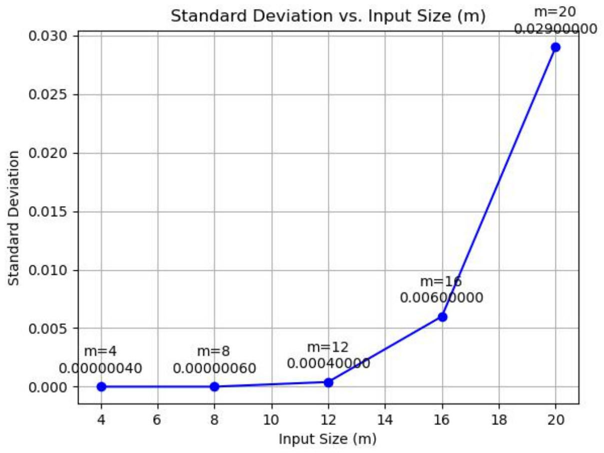

Figure 7: StDev of Naive Algorithm

In the worst case scenario, I could not even calculate the time for the naive algorithm input size 20 because it took too long, but now that I am creating random DNA, there is generally a common subsequence between the strings and this reduces the running time by preventing the worst case situation from occurring. At this point, I got shorter times than the times I found in part b. The time complexity of both algorithms depends on the input size, and since it increases as the input size increases, computation time also increases when the input size increases. While this increase looks like again exponential in the naive algorithm, it appears kind of linear in the memoization algorithm. While the computation time increases more when the input size increases in Part B, the increase in this part is lower because since we create random DNA, the worst-case scenario occurs very rarely. Experimental findings shed light on the performance of naive and memorization algorithms. In the worst-case scenario, the Naive algorithm's running times exhibit exponential growth, in line with the expected $\theta(2^{m+n})$. New results also confirm this trend. On the other hand, the Memoization algorithm, with a more favorable worst-case time complexity of $\theta(m^2)$, shows controlled growth in running times as input size increases, approaching a linear trend. This underscores the efficiency gains of memoization over the Naive approach, providing solid experimental support for the theoretical analysis.The experiments highlight input data's significant influence on algorithm efficiency. The naive algorithm, expectedly impractical for input size 20 in worst-case analysis, performed well with randomly generated DNA sequences. Meanwhile, the memoization algorithm consistently delivered practical performance for real-world data.