CS301 Assignment-1

Görkem Yar (Student-27970)

March 2022

1 Question-1

1.1 Give an asymptotic tight bound for T (n) in each of the following recurrences. Assume that T (n) is constant for n 2. No explanation is needed.

(a) $T(n) = 2T(n/2) + n^3$

By Master Theorem

$\rightarrow$ f(n)=$n^3, b = 2, a = 2$

$\rightarrow$f(n) =$\Theta(n^{\log_b a+\epsilon})$

$= \Theta(n^{\log_2 2+\epsilon}) \rightarrow \epsilon = 2 \rightarrow \epsilon > 0$

$\rightarrow$ T(n)= $\Theta(f(n)) = \Theta(n^3)$

(b) T(n) = 7T(n/2) +n^2

By Master Theorem

$\rightarrow f(n) = n^2, b = 2, a = 7$

$\rightarrow f(n) = \Theta(n^{\log_b a-\epsilon})$

$= \Theta(n^{\log_2 7-\epsilon}) \rightarrow \epsilon < 0$

$\rightarrow T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7})$

(c) T(n) = 2T(n/4) + $\sqrt{n}$

By Master Theorem

$\rightarrow f(n) = \sqrt{n}, b = 4, a = 2$

$\rightarrow f(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 4})$

$\rightarrow T(n) = \Theta(n^{\log_4 2} * \log n) = \Theta(\sqrt{n} * \log n)$

(d) T(n)=T(n-1)+n ?

T(n)=T(n-1)+n
T(n)=T(n-2)+n-1+n
T(n)=T(n-3)+n-2 +n-1+n
.....
T(n)=T(1)+ $\Sigma_{i=2}^{n} i$
T(n)= T(1) + $n^2/2 + n/2 - 1$
T(n)= $\Theta(n^2)$

2 Question-2 (Longest Common Subsequence - Python)

2.1 Cost of Algorithms:

2.1.1 Naive Algorithm:

```
def lcs(X,Y,i,j):
    if (i == 0 or j == 0):
       return 0
    elif X[i-1] == Y[j-1]:
       return 1 + lcs(X,Y,i-1,j-1)
    else:
        return max(lcs(X,Y,i,j-1),lcs(X,Y,i-1,j))
```

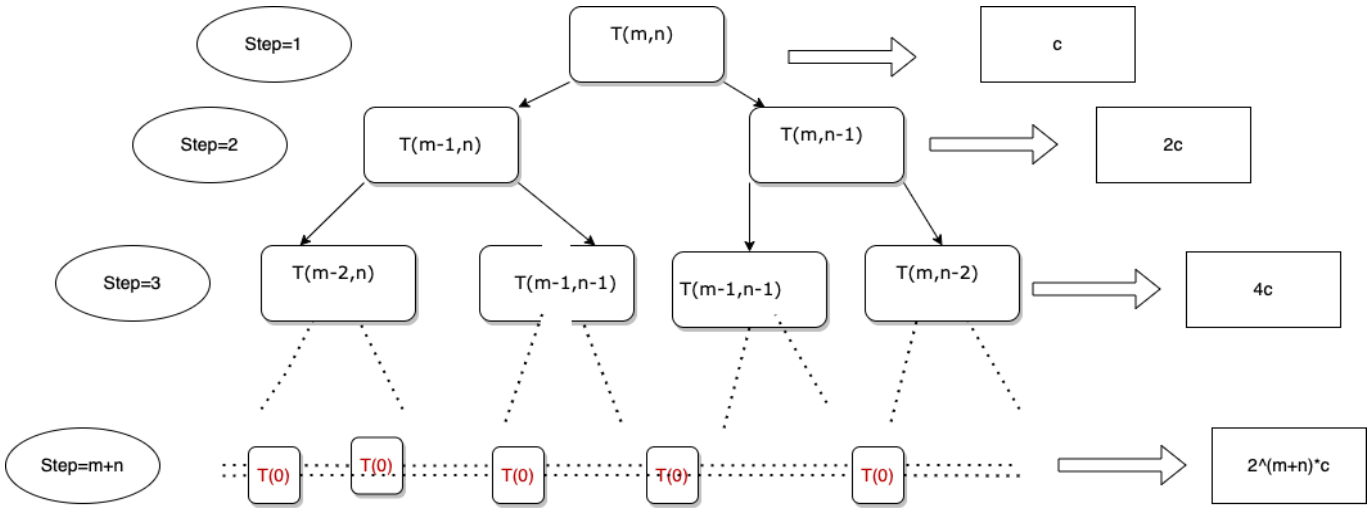
Explanation:

The worst case running time happens when the length of two strings is same, since the algorithm will make 2 recursive calls for each strings each character. If one of the inputs has a smaller length then the amount of the recursive calls would decrease. Also, the two strings should not have any common character since if there is a common character they will make 1 recursive call whereas if all the characters are differ they will make 2 recursive calls. To express the running time of the algorithm the best option is a recursive tree:

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

There is a constant amount of work done in each iteration via if and assigning operations. From this moment on, I will use a constant c instead of O(1).

$$\rightarrow \mathbf{T(m,n) = T(m-1,n) + T(m,n-1) + c}$$



It has got 2^(m+n) leaves as a result the algorithm will iterate 2^(m+n) times recursively. T(0) takes constant time therefore, T(m,n) =θ(2^(m+n))

The best, worst case running time of Naive-Algorithm is $T(m,n) = \Theta(2^{m+n})$

if we take m=n then the worst case running time of T(n,n) = $\Theta(2^{n+n}) = \Theta(4^n)$

2.1.2 Memoization-Algorithm:

```
def lcs(X,Y,i,j):
    if c[i][j] >= 0:
       return c[i][j]
    if (i == 0 or j == 0):
       c[i][j] = 0
    elif X[i-1] == Y[j-1]:
       c[i][j] = 1 + lcs(X,Y,i-1,j-1)
    else:
       c[i][j] = max(lcs(X,Y,i,j-1),lcs(X,Y,i-1,j))
    return c[i][j]
```

Explanation:

Similar to the Naive Algorithm worst case happens when there is not any common subsequence among the two strings since, if there is no common subsequence the algorithm needs to start 2 recursions; whereas, if there is something common between them 1 recursion is adequate. Due to memoization, problem divided into subproblems which are sharing subsubproblems. And the array enable algorithm to store these subsubproblems and preventing algorithm to compute same thing again and again. As a result, algorithm compute each position of m,n (the input strings) only once. If any position is required again, then it takes the data from the array. Since we need to store the base step in which either n or m is equal to 0, the size of array is (m+1)*(n+1). So there are (m+1)(n+1) subproblems, When we call already called subproblem it takes constant time O(1), and each subproblem enables to solve next subproblem. As a result the algorithmic complexity of the algorithm in the worst case is $\Theta((m+1)*(n+1)) = \Theta(mn)$.

2.2 Implementation and Comparison

2.2.1 Table

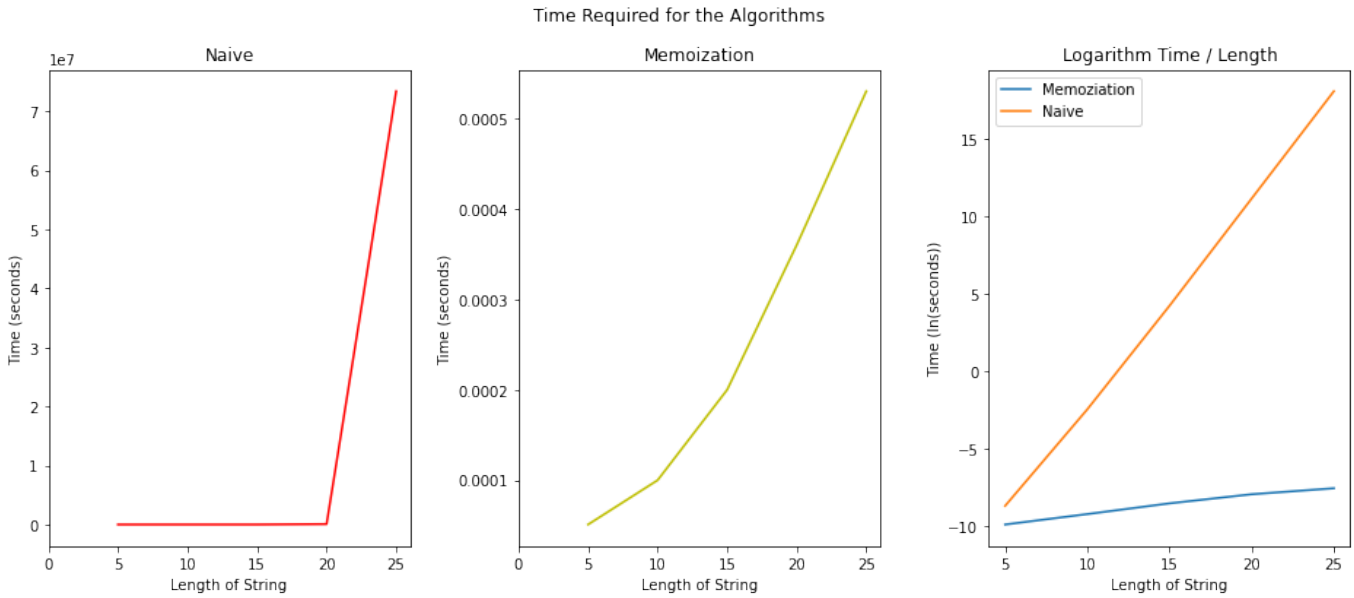
Algorithm	m = n = 5	m = n = 10	m = n = 15	m = n = 20	m = n = 25
Naive	0.00017s.	0.087 s.	70 s.	71680 s.	73400000 s.
Memoziation	0.000051 s.	0.0001 s.	0.0002 s.	0.00036 s.	0.00053 s.

```
Experimentally my machine worked perfectly fine with the memozaition algorithm.
However, after setting the input as 17 with the naive algorithm the amount
of time that taken has drastically increased. As a result, I could not run
it for input size=20 and 25. Those results are my guesses.

OS: MACOS
CPU: 2 GHz Quad-Core Intel Core i5
RAM: 16GB
GRAPHICS: Intel Iris Plus Graphics 1536 MB

I run the code in terminal with python3.
```

2.2.2 Plot Graphs



The first plot is Naive algorithms, second plot for the Memoization, and the third plot is logarithm(time) plot of both of these plots. The third chart is plotted to show both graphs in a single chart.

2.2.3 Analyze of Algorithms

Naive-Algorithm

As can be seen by the experiments the algorithm takes exponential time. To further investigate, I run the some other length strings such as 12, 13, 14, 15, 16, 17 whenever ı add one more character to both of the strings, time multiplied by 4. So the finding, graphs are indicating that worst case algorithmic analyze in 2.1.1 part which showed algorithmic complexity as $\Theta(2^{m+n})$ is correct. Unfortunately, I could not run the n=20 and n=25 due to time it takes.

Memoization-Algorithm

In contrast to Naive-Algorithm, Memoization-Algorithm is much more scalable and understandable by looking its graph. In the 2.1.2 part the Algorithmic complexity found as $\Theta(n * m)$ if we assume n=m then it is $\Theta(n^2)$. The graph show that the algorithm is growing polynomially. I looked at the numbers and they are similar to $\Theta(n^2)$.

2.3 Average Running Time

2.3.1 Table

Algorithm	m=n=5		m=n=10		m=n=15		m=n=20		m=n=25	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Naive	0.0015	0.0077	0.016	0.016	0.53	0.63	31.04	28.86	-	-
Memoization	3.8e-05	1.03e-05	6.28e-05	3.54e-05	0.00015	4.62e-05	0.0002	3.4e-05	0.00037	0.0001

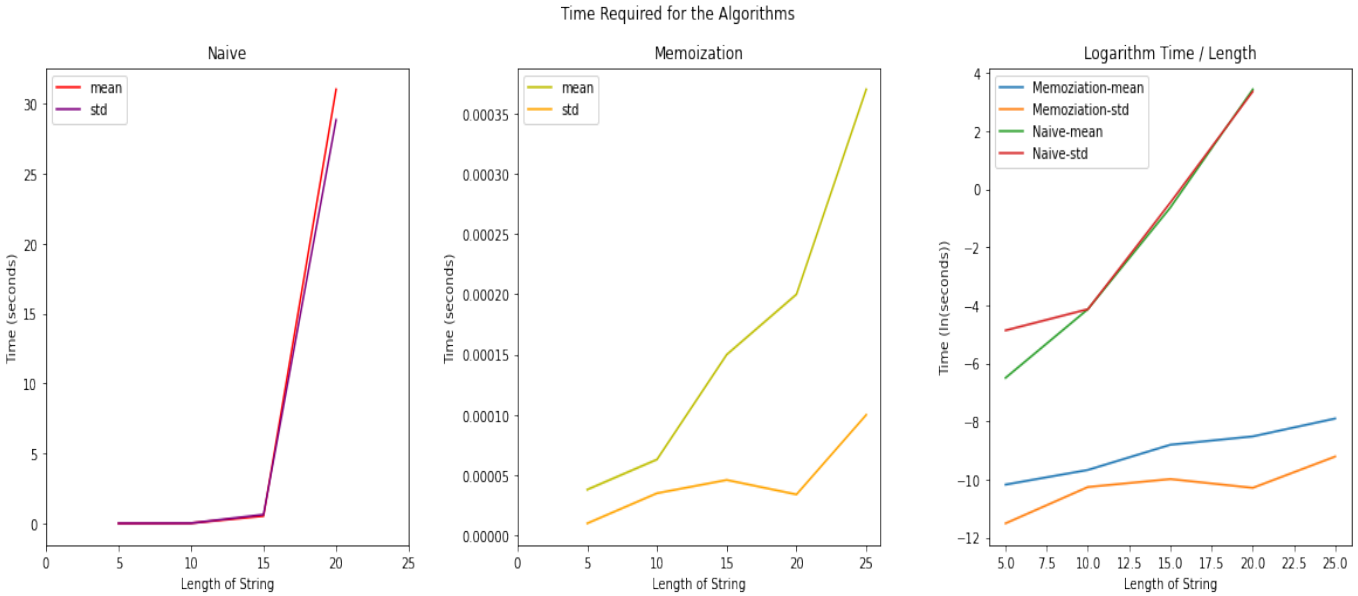
Due to exponentially growing time complexity of the Naive algorithm, I could not find the mean or standard deviation when n=m=25.

OS: MACOS

CPU: 2 GHz Quad-Core Intel Core i5

RAM: 16GB
GRAPHICS: Intel Iris Plus Graphics 1536 MB
I run the code in VSCODE.

2.3.2 Plot Graphs



The first plot is Naive algorithms (showing mean and standard deviation), second plot for the Memoization (showing mean and standard deviation), and the third plot is logarithm(time) plot of both of these plots. The third chart is plotted to show both graphs in a single chart.

2.3.3 Compare Average and Best Case Running Time

Naive-Algorithm

Compared to the worst case scenarios, it is working much faster. For instance, in the worst case scenario when n=m=20, one test case could not be run due to time it takes. However, in the average test case scenario my machine could run 30 test cases when n=m=20. This stems from the recursive manner of the algorithm, in the worst case algorithm always created 2 recursions. On the other hand, in average cases there are always 4 different DNA characters which created common elements. As a result of this, the algorithm did not pick the 2 recursion option each time. The overall working time of the algorithm is still exponentially as graphs and outputs indicated.

Memoization-Algorithm

Like the Naive-Algorithm, it worked faster when run with the DNA sequence. The reason it became faster is same as the Naive-Algorithm which is common subsequences that enables algorithm to create 1 recursion instead of 2. Also, the quadratic behaviour of the algorithm is more visible then the worst case scenarios.