CS 301 - Assignment 1

Oytun Kuday Duran / 28357

November 6, 2022

1 Problem 1 (Recurrences)

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

T(n) = Θ (n^3)

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

T(n) = Θ ($n^{\log_2 7}$)

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

T(n) = Θ ($\sqrt{n} \log n$)

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

T(n) = Θ (n^2)

2 Problem 2 (LCS - Python)

2.1 (a) - Costs of Worst Case

2.1.1 (i) Naive Recursive 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))
```

Solution:

Since this function calls itself by individually decreasing the indexes of both strings, worst case is when both of the strings have the same length. This function is better for strings with a huge difference and worse when strings have a very similar length. In addition, since it makes 2 recursive calls instead of one when two characters that are compared in strings are not the same, The worst case is when the strings has no common sequence and in equal length.

To show the worst-case running time, we can calculate:

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

In the equation, the T(m-1,n) and T(m,n-1) represents the recursive function calls while the O(1)

represents the operations that take constant amount of time such as comparisons and if statements. Since O(1) is constant, we can express it as c like we usually do.

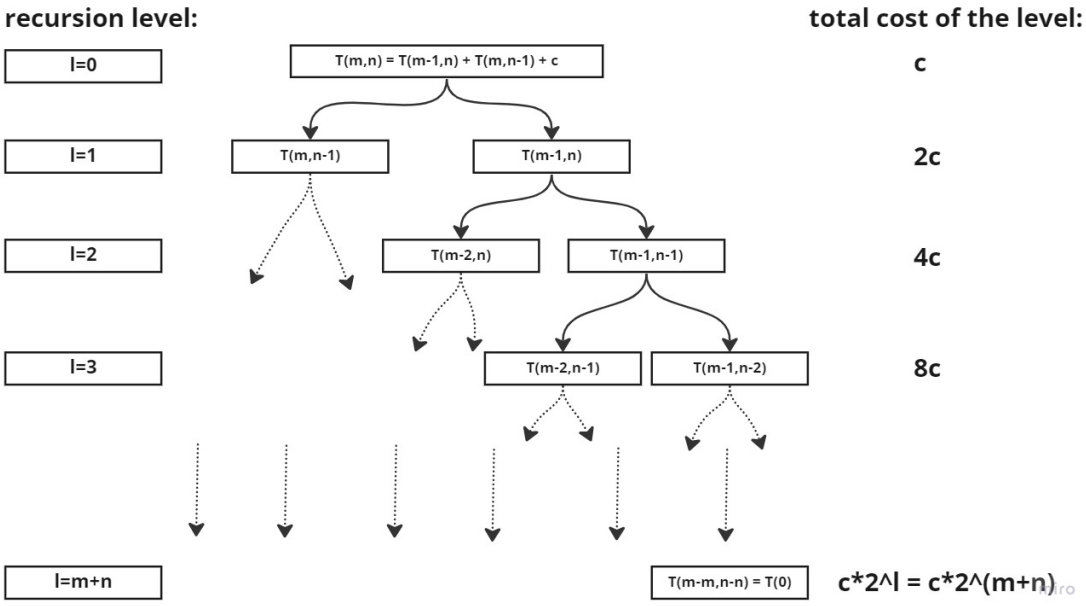


Figure 1: Recursion tree of naive algorithm

One of the best ways to represent such algorithms is drawing a recursion tree. As can be seen in the recursion tree in figure 1 above. In each recursion, a constant amount of work done such as conditional checks. Since its constant, it is denoted in figure as "c". In each recursion level on tree, the leaves are doubling which represents recursions. In correlation, total cost of each level is doubled with the leaves. the recursions will continue until i or j (Can also be called as n or m) is 0. So the total recursion level is m+n. Similarly, since leaves are doubling every recursion level, total recursive iteration is 2 to the power of l which is m+n. Since T(0) also takes constant time:

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

The worst case running time is when both strings are in equal length which means m=n. So:

$T(n,n) = \Theta(2^{n+n})$
$T(n,n) = \Theta(2^{2n})$
$T(n,n) = \Theta(2^n * 2^n)$
$T(n,n) = \Theta(4^n)$

2.1.2 (ii) Recursive Algorithm with memoization

```
def lcs(X,Y,i,j):
    if c[i][j] ¿= 0:
        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]
```

Solution:
Similar to previous case, since it has to make 2 recursions instead of 1 when the characters to compare are not same, worst case is when two strings have no common sequence.

Unlike the Naive LCS Algorithm, it doesn't make the same recursions again and again. Instead, if already computed before, it takes the needed recursion result from the matrix it stores. So, it doesn't solve the subproblems again and again. In this way, it saves a lot of time and space, and it is more efficient. Multiple recursion of same process is deposited in one place of matrix and used when needed. The size of the matrix is (m+1)*(n+1) since we need to calculate base step too. Since each subproblem is deposited in matrix, the total amount of problems and subproblems are the total amount of elements in matrix, which is (m+1)*(n+1). Each recursion takes a constant amount of time O(1) because of condition checks and assign operations. Each subproblem is solved easily with the previous subproblem. The algorithmic complexity of this algorithm is $\Theta((m+1)*(n+1))$ which is $\Theta((mn+m+n+1))$ The asymtotic worst-case running time of the memoizatino algorithm is $\Theta(mn)$.

2.2 (b) - Implementation and testing in worst case

2.2.1 (i) Tables

I waited around 30000+ seconds for m=n=20, then I computed the time each recursion it takes and calculated approximately 51847 for the m=n=20. I also tested many other values between 2 and 20.

Worst-Case Running Times					
Algorithm	m = n = 4	m = n = 8	m = n = 12	m = n = 16	m = n = 20
Naive	2e-4	0.08	1.17	295.18	51847
Memoization	0.0007	0.0009	0.001	0.001	0.0011

My Specs:
CPU: Intel® Tiger Lake Core™ i7-11800H 8C/16T; 24MB L3; 8GT/s; 2.3GHz ¿ 4.6GHz; 45W; 10nm SuperFin
GPU: Nvidia RTX3070 Max-Performance 8GB GDDR6 256-Bit DX12
RAMs:16GB (2x8GB) DDR4 3200MHz
OS: Windows 11 Pro 64-bit (10.0, 22000)

2.2.2 (ii) Graphs

From figure 2 below, you can see comparison of running times of 2 algorithms in terms of sec/length. The first graph represents Memoization, the second graph represents naive and the third graph is a comparison of both. Here, in memoization graph, I used some other n values I tested between 2-20 for a more precise graph.

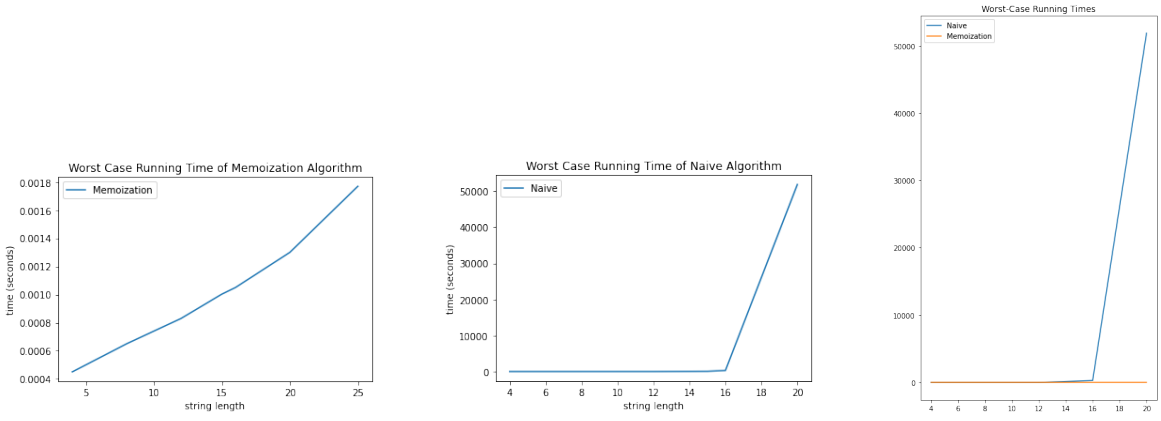


Figure 2: Graphs for Worst-Case Running Times

2.2.3 (iii) Discussion

From my experiments of different various lengths varying from 2-25, I observed that my results are accurate in terms of worst-case complexities of algorithms. When n=m, the naive algorithm's exponential growth is very fast with increasing length. In memoization algorithm, it has a growth similar to quadratic. In graph 2, comparing two algorithms show how important complexity is as the memoization algorithm seems like almost not increasing at all compared to naive algorithm although it is also exponential (When m=n, $\Theta(m * n)$ is $\Theta(n^2)$. On the other hand, $\Theta(2^{m+n})$ which is $\Theta(4^n)$ when m=n is clearly grows a lot faster than $\Theta(n^2)$. My experimentations match with my results on part (a) considering growth of time requirement of execution. Since naive algorithm generates a lot more recursive functions and have a higher complexity, it takes more time as length of string increases. Furthermore, it is a lot less scalable than the memoization algorithm.

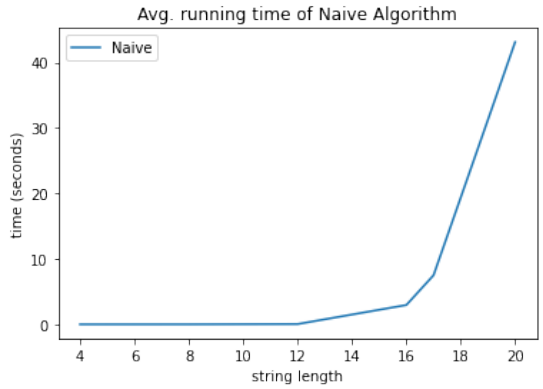
2.3 (c) - Average running times

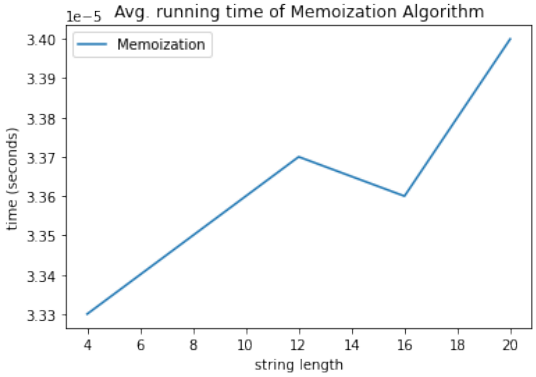
2.3.1 (i) Tables

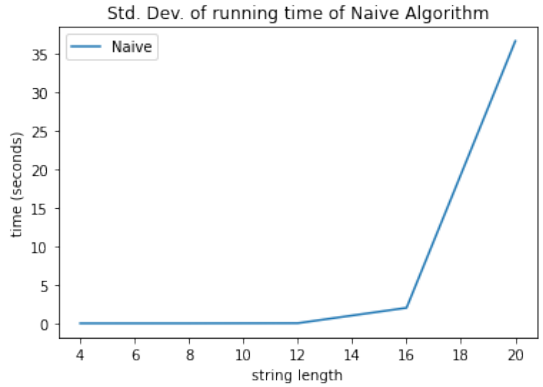
Algorithm	m = n = 4		m = n = 8		m = n = 12		m = n = 16		m = n = 20	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Naive	6.66e-5	2.4e-4	2.76e-3	2.23e-3	3.94e-2	2.14e-2	2.95	2.016	41.11	36.69
Memoization	3.33e-5	1.7e-4	3.33e-5	1.7e-4	3.34e-5	1.8e-4	3.33e-5	1.7e-4	3.34e-5	1.82e-4

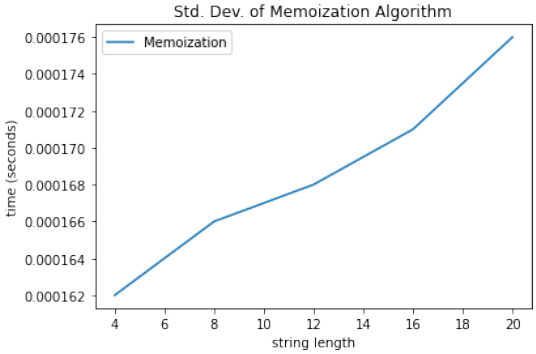
2.3.2 (ii) Graphs

From figure 3 below, you can see comparison of averages and standard deviations of running times of 2 algorithms in terms of sec/length. Last 2 graphs are comparisons of two algorithms in terms of avg. running time and std. deviation. I also included some other tests i conducted with values to increase accuracy in average running time graphs.









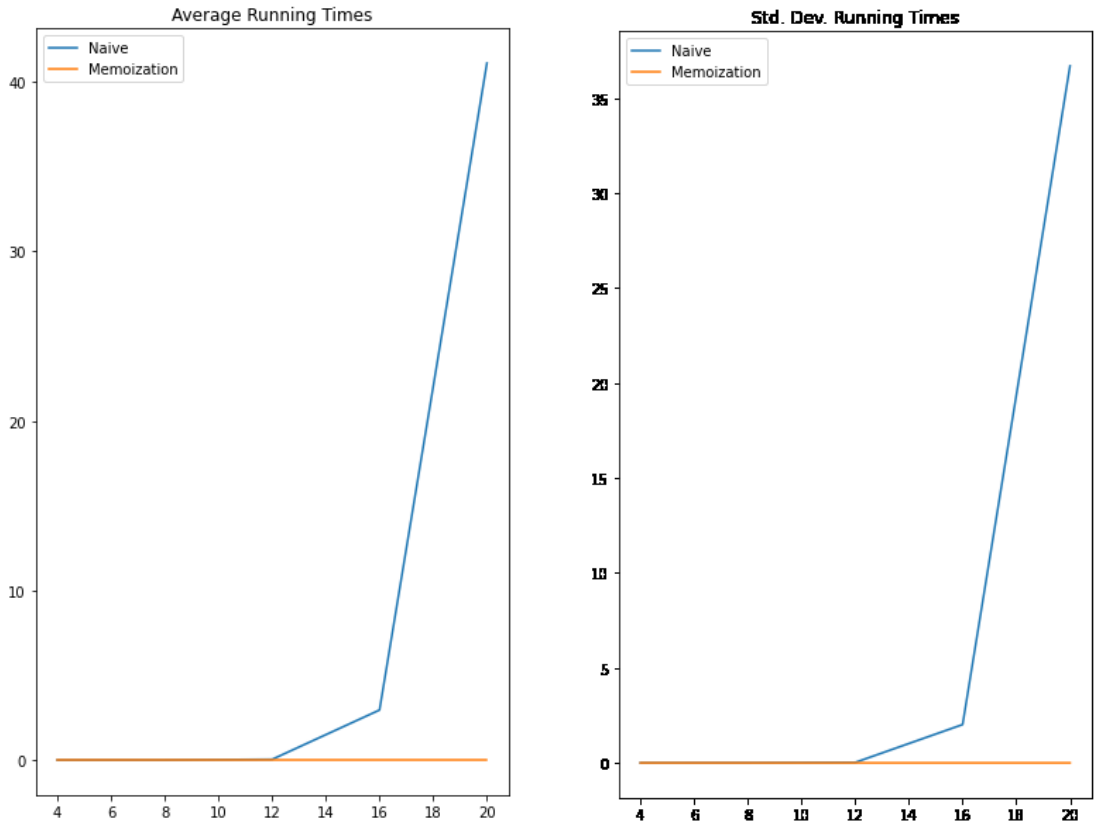


Figure 3: Graphs for Avg and Std. Dev. of Running Times

2.3.3 (iii) Discussion

Growth of naive algorithm might still be exponential considering the graphs. Although some growth is evident in memoization graph, from my experiments, because of my computer's hardware, I couldn't get much information. It may be polynomial or quadratic. Obviously, the average running times are faster than the worst-case running times. I experienced that running 30 pairs of DNA sequences with both algorithms came to a result a lot faster than worst cases. Also, the growth is significantly slower compared to the the worst case. This is because the recursive functions didn't have to make 2 recursions when there was a matching sequence. Instead, they only had to make 1 recursive call. The difference in growth between 2 algorithms are less but still very clear in terms of running time as can be seen in last 2 graphs. In addition, I saw a correlation between recursion amount with the string length, and increase in the mean and standard deviation. By considering worst cases, It can be said that although an algorithm can be used in avg. cases, we must also consider the worst cases. Because, the average time it takes for naive algorithm is 41 seconds when m=n=20 but the worst case takes 15+ hours. It is not sustainable or viable to use considering how long it takes when bad inputs are evident. The standard deviation is a lot less and its growth is also slower in the optimal memoization algorithm compared to the naive algorithm. We can say that by optimizing an algorithm considering worst-cases, we can significantly reduce the growth of mean and standard deviation of execution time too. When optimized, the algorithms can cope with different inputs that take distinct time to execute. In correlation, the growth of std. dev. and mean are also greatly reduced. The difference of growth can be observed easier between optimal and un-optimal algorithms in worst-cases.