CS 301 - Assignment 1

Muhammed Orhun Gale

03/20/2022

1 Problem 1

(a) $\Theta(n^3)$

(b) $\Theta(n^{\log_2 7})$

(c) $\Theta(\sqrt{n}log(n))$

(d) $\Theta(n^2)$

2 Problem 2

(a)

(i) For the naive algorithm, worst-case scenario occurs when input strings X and Y have no common sub-string which means they even do not have a common letter like $X = "aaaaa"$ and $Y = "bbbbb"$. In this case, same sub-problems will have to be computed more than once for any lcs function call. For instance, while computing $lcs(X, Y, i, j)$ even though the $lcs(X, Y, i-1, j)$ call computes the $lcs(X, Y, i-1, j-1)$ function, $lcs(X, Y, i, j-1)$ call must compute the same value again.

In naive algorithm, while there is overlap, problem is continued to be solved, like an iterative algorithm, without any extra calls. However, if current letters are not overlapped, the problem recursively creates two sub problems as $T(n+m) = T(n-1+m) + T(n+m-1) + \mathcal{O}(1)$.

Therefore the recurrence that defines the worst-case running time of this algorithm is $T(n+m) = 2T(n+m-1) + \mathcal{O}(1)$.

In order to show the tight-bound that defines the asymptotic behaviour of this algorithm, upper and lower bounds should be shown.

For upper bound, guess that it is $\mathcal{O}(2^{n+m})$.
Induction hypothesis: $T(k) \leq c2^{n+m}$ for all $k < n$ then,
$T(n+m) = 2c(2^{n+m-1}) + \mathcal{O}(1)$
$T(n+m) = 2(2^{n+m}) + \mathcal{O}(1)$
$T(n+m) = \mathcal{O}(2^{n+m})$

For Lower bound, guess that it is $\mathcal{O}(2^{n+m})$.
Induction hypothesis: $T(k) \geq c2^{n+m}$ for all $k < n$ then,
$T(n+m) = 2c(2^{n+m-1}) + \mathcal{O}(1)$
$T(n+m) = 2(2^{n+m}) + \mathcal{O}(1)$
$T(n+m) = \Omega(2^{n+m})$

Since $T(n+m) = \Omega(2^{n+m}) = \mathcal{O}(2^{n+m})$

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

(ii) Memoization helps to solve problem which is solving same sub-problem over and over again. It basically solves the bigger problem and caches the smaller solutions of sub-problems to a matrix. Therefore, in order to solving the whole problem requires to reach bottom (i.e. the escape statement which is i = 0 and j = 0 here) and use the same solutions along the way. To do that, going through the matrix once will be enough. In this sense asymptotic behaviour of the worst-case running time of the memoization algorithm is

$T(n+m) = \Theta(m.n)$

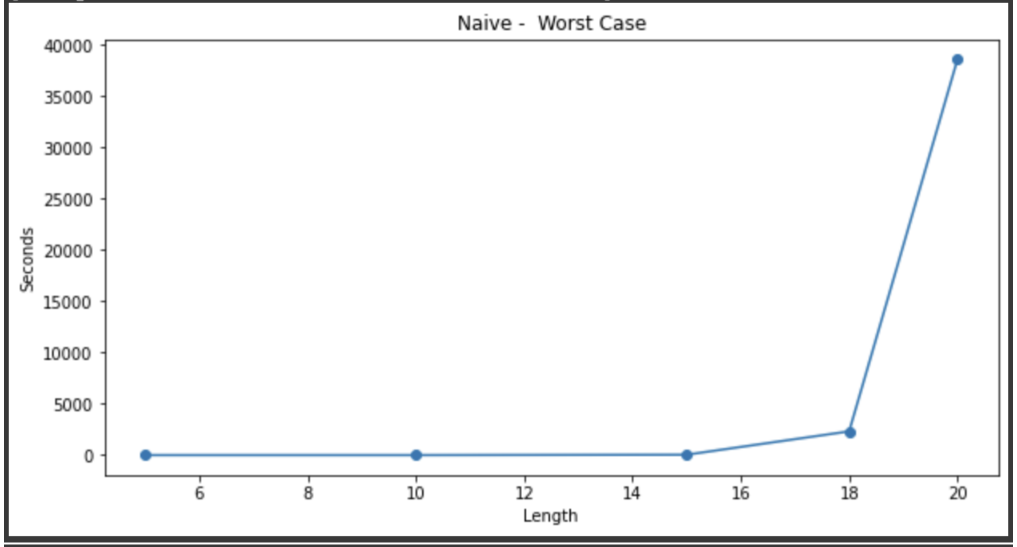
(b)

(i) macOS Monterey, Apple M1 3.2 GHz, 16GB RAM

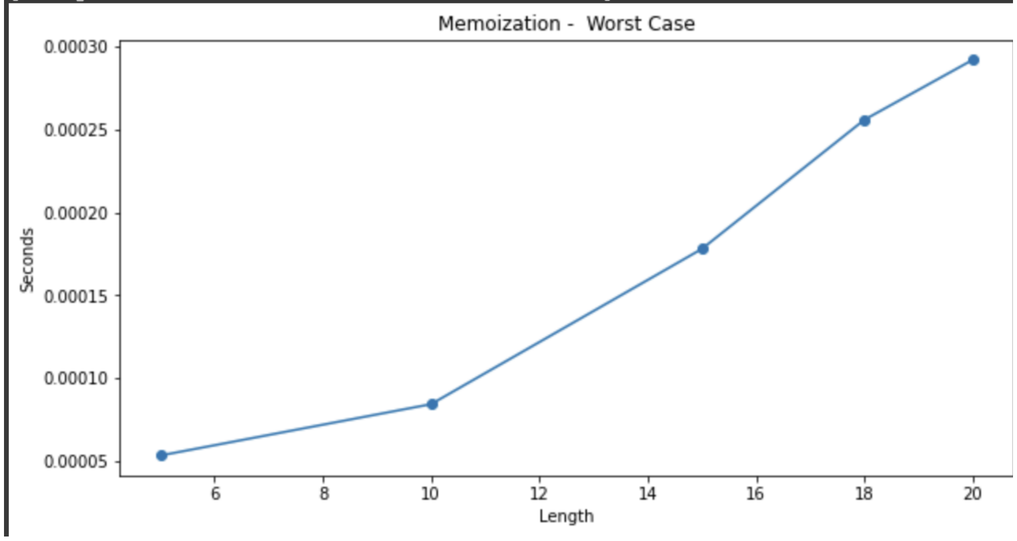
Algorithm	m = n = 5	m = n = 10	m = n = 15	m = n = 18	m = n =20
Naive	0.000127083	0.0613698329	39.502788916	2320.326040125	38613.8632319
Memoization	5.33749e-05	8.433294e-05	0.0001779169	0.000255749	0.000291875

(ii)

```
[<matplotlib.lines.Line2D at 0x7fda85b1a590>]
```


Naive - Worst Case
Seconds
Length
40000
35000
30000
25000
20000
15000
10000
5000
0
6
8
10
12
14
16
18
20

```
[<matplotlib.lines.Line2D at 0x7fda85a96d10>]
```


Memoization - Worst Case
Seconds
Length
0.00030
0.00025
0.00020
0.00015
0.00010
0.00005
6
8
10
12
14
16
18
20

(iii) As it can be clearly seen, Naive algorithm's run-time explodes when length goes beyond the 18. The growth is predicted to be $\Theta(2^{n+m})$ by asymptotic analysis and in the graph it seems not the case. However, it can be related with the exponential input growth. Therefore, may be in the log-scale the growth can be overlap.

For the memoization, even though it seems quadratic for initial inputs, it gets closer to a linear appearance as the inputs grows but it can be input/machine related. In the asymptotic analysis, it is found to be $\Theta(m.n)$ and since the m and n are equal, it is $\Theta(m^2)$ which means it is asymptotically quadratic. When this considered, it is probable that for very large inputs, it can behave as a quadratic function which means that the worst-case analysis is coherent to an extent.

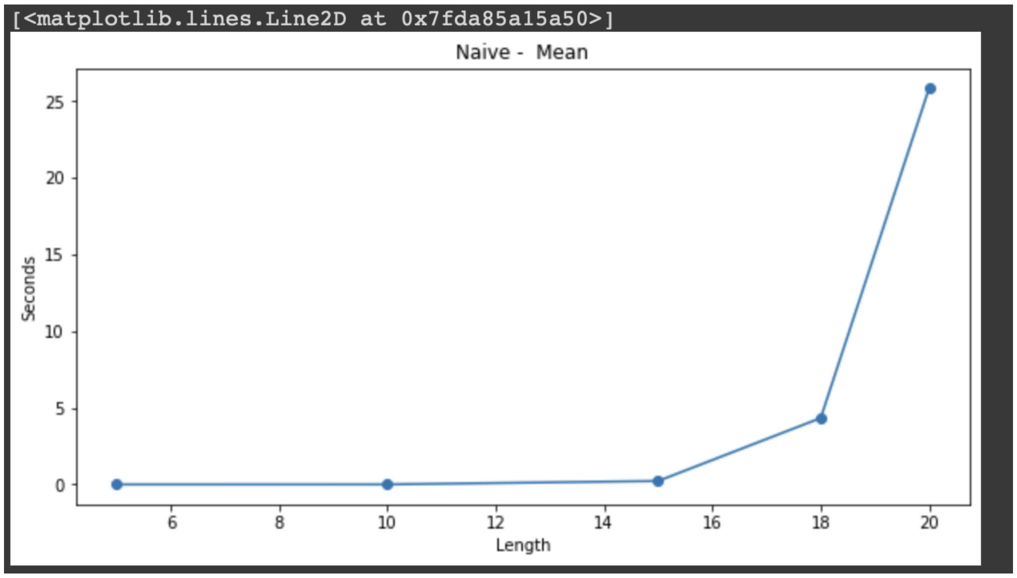
Also note that, after some observation for the naive algorithm, the input $n = m = 25$ is left out since the computer that is used to execute this algorithm's is executes around 29000000 instruction/sec (experimentally) and in this case $n = m = 25$ causes to take around 30000 days to execute.

(c)

(i)

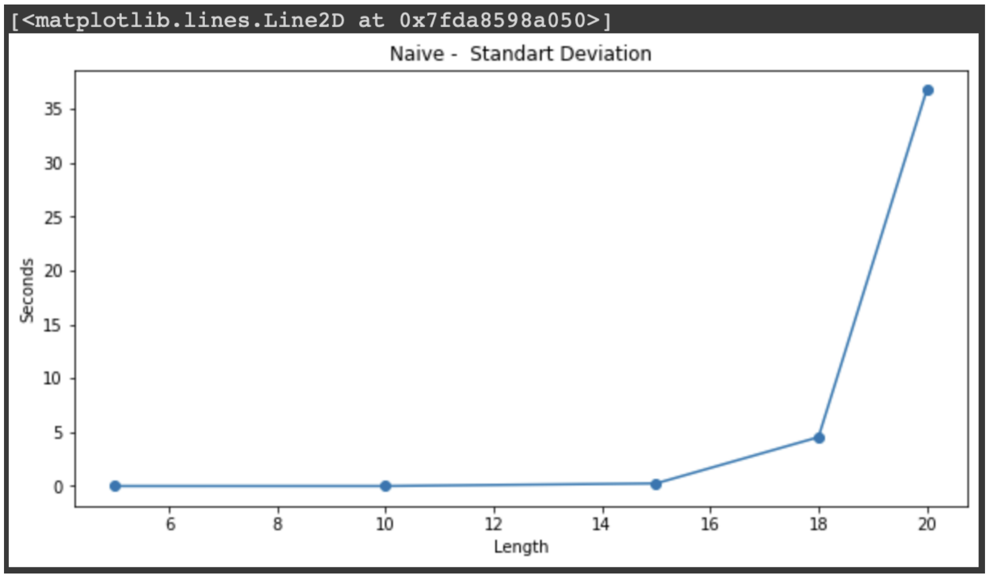
	m = n = 5		m = n = 10		m = n = 15		m = n = 18		m = n = 20	
	µ	σ	µ	σ	µ	σ	µ	σ	µ	σ
Naive	1e-05	1e-05	0.002	0.001	0.225	0.244	4.33	4.53	25.83	36.73
Memoization	1e-05	1e-06	3e-05	5e-06	7e-05	9e-06	9e-05	7e-06	7e-06	1e-05

(ii)
Naive



```
[<matplotlib.lines.Line2D at 0x7fda85a15a50>]
```

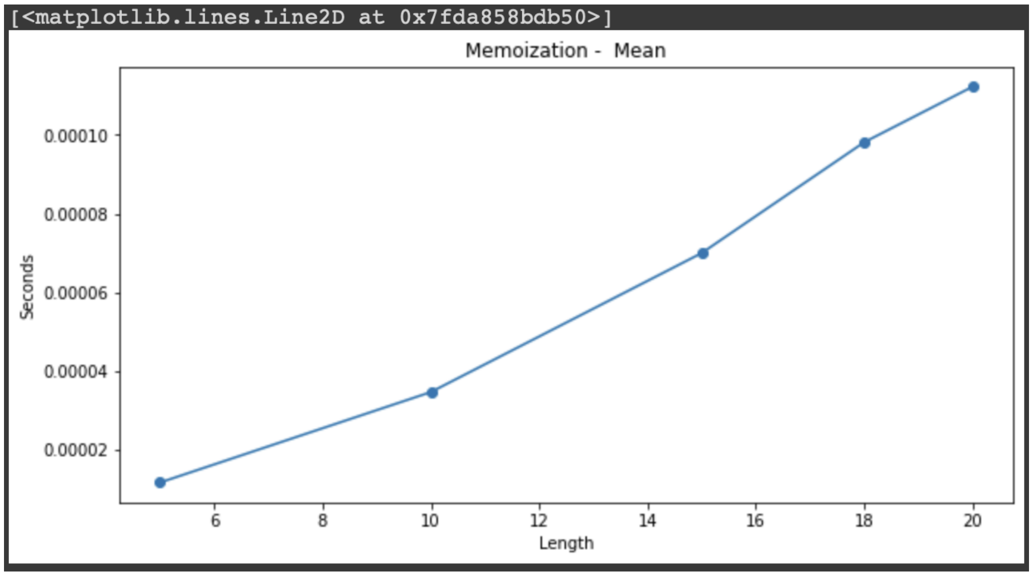
Naive - Mean
Seconds
Length
0
5
10
15
20
25
6
8
10
12
14
16
18
20



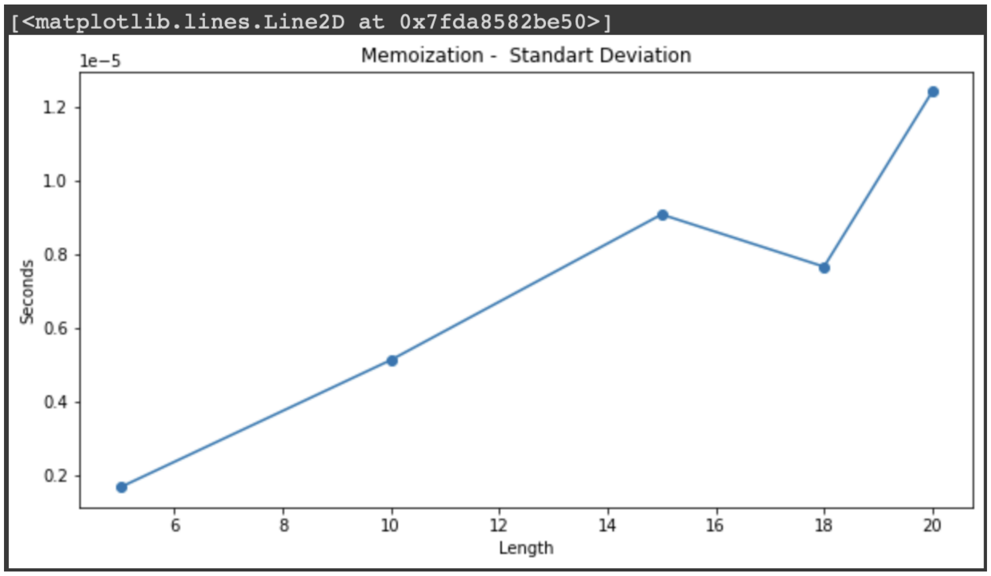
```
[<matplotlib.lines.Line2D at 0x7fda8598a050>]
```

Naive - Standart Deviation
Seconds
Length
0
5
10
15
20
25
30
35
6
8
10
12
14
16
18
20

Memoization



```
[<matplotlib.lines.Line2D at 0x7fda858bdb50>]
```



```
[<matplotlib.lines.Line2D at 0x7fda8582be50>]
```

(iii) For the naive algorithm, even though the graph seems similar as in the part b, it seems more likely to be a exponential function. It is because of the randomized inputs which have at least some common sub-sequences. Again, in a log-scale graph the exponential-like growth can be seen better. Even though, it performed better than in the worst case, it was still slow on the execution.

Memoization algorithm, behaves as a linear function which means that it matches with the worst-case experiment to an extent but strongly conflicts with theoretical analysis. During the execution, it is observed that the random DNA inputs did have around 10-12 lcs which means these runs were performed under the "average" conditions. Therefore, it can be argued that Memoization algorithm's average-case asymptotic behaviour is $\Theta(n)$.