NAME: Dilara Nur MEMIS

Solution for Problem 1 (Recurrences) Give an asymptotic tight bound for $T(n)$ in each of the following recurrences. Assume that $T(n)$ is constant for $n \leq 2$. No explanation is needed.

We can apply Master Theorem for this question (for the first 3 parts).We shall write recurrence relations in the form of $T(n) = aT(n/b) + f(n)$ where $a \geq 1$ and $b > 1$ and $f(n)$ is a given function.

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

a = 2, b = 2, $f(n) = n^3$
$n^{log_b a} = n^{log_2 2} = n$
$f(n) = \Omega(n^{log_2 2+e})$ for some constant $e > 0$ and $2f(n/2) \leq c * f(n)$ for $c < 1$.
Therefore, $T(n) = \Theta(f(n)) = \Theta(n^3)$.

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

a = 7, b = 2, $f(n) = n^2$
$n^{log_b a} = n^{log_2 7}$
$n^2 = O(n^{log_2 7-e})$ for some constant $e > 0$. In other words, $n^{log_b a}$ grows polynomially larger than $f(n)$. Therefore, $T(n) = \Theta(n^{\log_2 7})$.

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

a = 2, b = 4, $f(n) = \sqrt{n}$
$n^{log_b a} = n^{log_4 2} = \sqrt{n}$
Then $f(n) = \Theta(n^{\log_b a})$.
Therefore $T(n) = \Theta(n^{log_b a} * logn) = \Theta(\sqrt{n} * logn)$.

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

Since $b = 1$ we can not apply Master Theorem for this equation.
Instead, we can expand the equation as follows:

$T(n) = T(n-1) + n$
$= T(n-2) + (n-1) + n$
$= T(n-3) + (n-2) + (n-1) + n$

$= T(0) + n + (n-1) + (n-2) + (n-3) + ... + 1$

Provided that $T(0)$ is constant, total is:
$1 + 2 + 3 + ... + (n-2) + (n-1) + n = (n * (n-1))/2 = \Theta(n^2)$

QUESTION 2:

(a) **(20 points)** According to the cost model of Python, the cost of computing the length of a string using the function `len` is $O(1)$, and the cost of finding the maximum of a list of k numbers using the function `max` is $O(k)$. Based on this cost model:

(i) What is the best asymptotic worst-case running time of the naive recursive algorithm shown in Figure 1? Please explain.

```
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))
```

Figure 1: A recursive algorithm to compute the LCS of two strings

ANSWER: Let $c[i,j]$ be length of LCS of $X[0:i]$ and $Y[0:j]$. Then recurrence equation of the algorithm is as follows:

```
c[i,j] =
    0                          if i = 0 or j = 0
    c[i-1,j-1]+1               if i,j > 0 and x[i] = y[j]
    max(c[i,j-1],c[i-1,j])     if i,j > 0 and x[i] != y[j]
```

Worst-case happens when the two strings do not have any common characters. In such a case, following part of the function will be executed in every step except the cases $i = 0$ or $j = 0$:

```
else:
  return max(lcs(X,Y,i,j-1),lcs(X,Y,i-1,j))
```

Then recursive equation of worst-case:

$T(n, m) = T(n-1, m) + T(n, m-1) + \Theta(1)$.

We will check every possible subsequences of one of the strings and check whether it is also a subsequence of the other one. However, we will compute the same T(i,j) values for many times since we do not keep the results. This involves a lot of redundant work.

Application of Substitution Method:

1. Guess of Complexity: Since Y has 2^n different subsequences, we will check X for each of them. Each check will take $\Theta(m)$ time (using one of fast string matching algorithms). So, our guess is $T(n, m)$ is $\Theta(2^n * m)$.

2. Proof by Induction: We need to show that $\exists c, n_0 \geq 0$ such that $\forall n \geq n_0, T(n, m) \leq c * (2^n * m)$.

Base case: $T(0, i)$ or $T(j, 0)$. In all such cases$T(i, j) = 0$, so we can choose c large enough to satisfy required inequality.

Induction step: Assume $T(a, b) \leq 2^a * b$ for all cases except the one $a = n \;\; \& \;\; b = m$.

- $T(n, m) = T(n-1, m) + T(n, m-1) + \Theta(1)$.
- $T(n, m) \leq c * 2^{n-1} * m + c * 2^n * (m-1) + \Theta(1)$
- $T(n, m) \leq c * 2^{n-1}(m + 2 * (m-1)) + \Theta(1)$
- $T(n, m) \leq c * 2^{n-1} * m * (3 - 2/m) + \Theta(1)$
- $T(n, m) \leq c*2^n*m*(3-2/m) - c*2^{n-1}*m*(3-2/m) + \Theta(1)$

Therefore, $T(n, m) \leq c*2^n*m$. The best asymptotic worst-case running time of the naive recursive algorithm is: $T(n, m) = \Theta(2^n * m)$ (assuming $n > m$).

(ii) What is the best asymptotic worst-case running time of the recursive algorithm with memoization, shown in Figure 2? Please explain.

ANSWER:

Application of Substitution Method:

```
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]
```

Figure 2: A recursive algorithm to compute the LCS of two strings, with memoization

1. Guess of Complexity: In this algorithm, we will compute $T(i, j)$ for all possible values of i and j again. But we will compute each $T(i, j)$ only once since we keep the results in a table c.

We know that $0 \leq i \leq m$ and $0 \leq j \leq n$.

Therefore, there are $m * n$ combinations for $T(i, j)$s in total. Each $T(i, j)$ will take constant time to compute since we basically compare previously found two numbers in the worst-case.Then, complexity of the algorithm should be:

$$T(n, m) = \Theta(n * m).$$

2. Proof by Induction: We need to show that $\exists c, n_0 \geq 0$ such that $\forall n \geq n_0$, $T(n, m) \leq c * (m * n)$.

Base case: $T(0, i)$ or $T(j, 0)$. In all such cases both sides of the equation will be 0. So, base case holds.

Induction step: Assume $T(a, b) \ \leq \ c * a * b$ for all cases except the one $a = n \ \& \ b = m$.
- $T(n, m) = T(n - 1, m) + T(n, m - 1) + \Theta(1)$.
- $T(n, m) \leq c * (n - 1) * m + c * n * (m - 1) + \Theta(1)$
- $T(n, m) \leq c * n * m - m - n + \Theta(1)$
- $T(n, m) \leq c * n * m - (m + n) + \Theta(1)$

$m + n > 0$. Therefore, $T(n, m) \leq c * m * n$. The best asymptotic worst-case running time of the algorithm with memoization is: $T(n, m) = \Theta(n * m)$.

(b) **(30 points)** Implement these two algorithms using Python. For each algorithm, determine its scalability experimentally by running it with different lengths of strings, in the worst case.

(i) Fill in following table with the running times in seconds.

Algorithm	$m = n = 5$	$m = n = 10$	$m = n = 12$	$m = n = 15$	$m = n = 16$
Naive	0.0011	0.4264	5.9327	350.53	1354.04
Memoziation	6.43e-5	0.00024	0.00035	0.00054	0.00066

Specify the properties of your machine (e.g., CPU, RAM, OS) where you run your programs.

Machine Properties:
Storage: 256 GB SS
Processor: Intel Core i5 (7th Gen) Processor
Ram 8 GB DDR4 RAM
Operating System: Windows 10

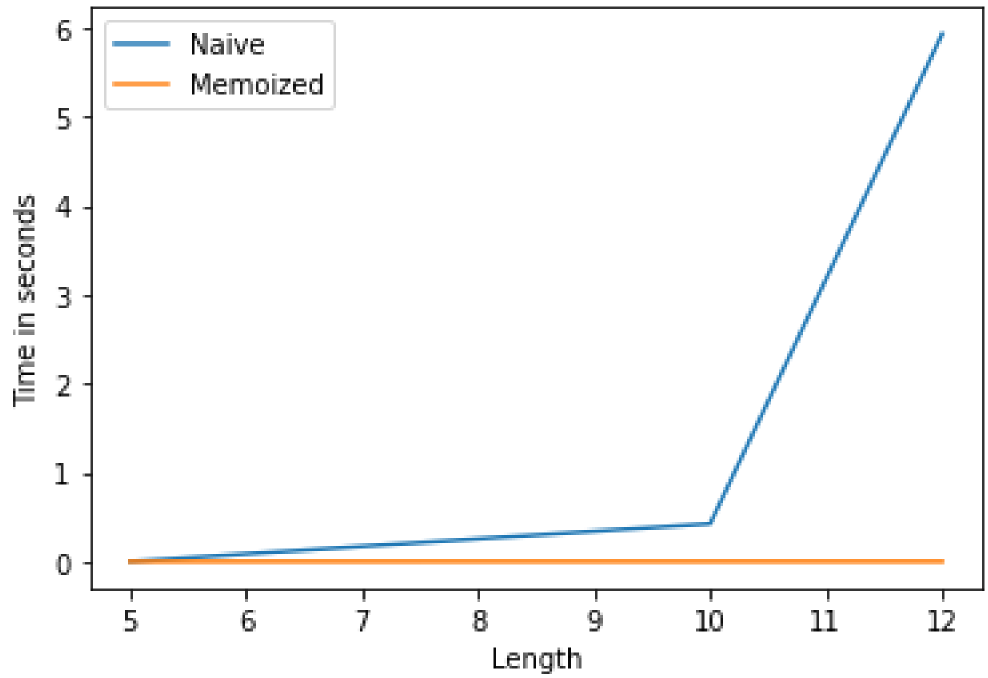
(ii) Plot these experimental results in a graph.

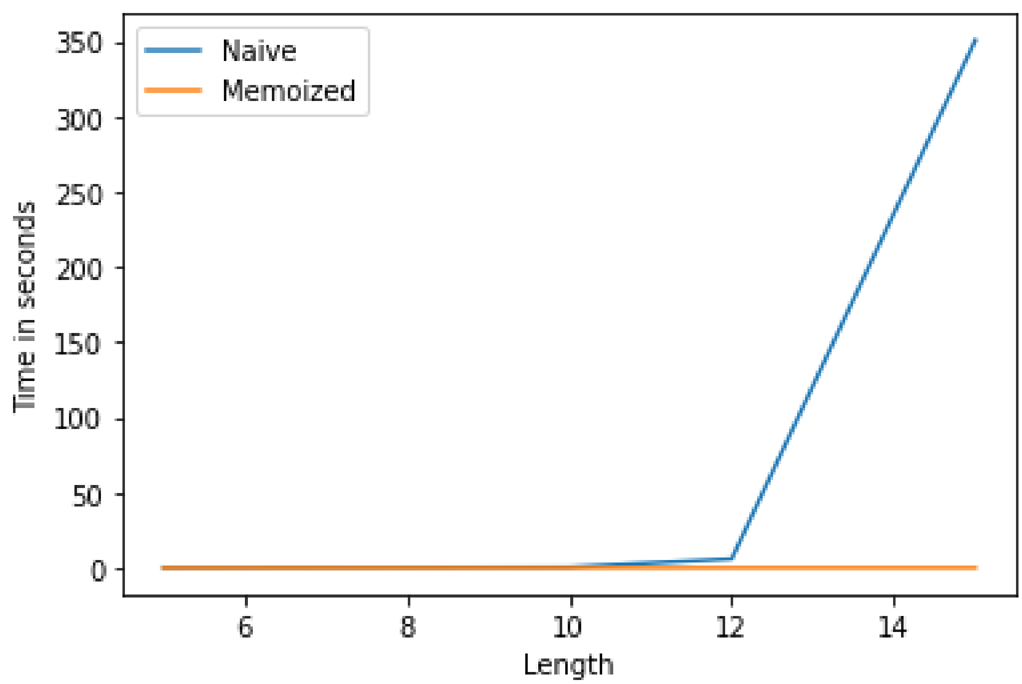
You can see 3 different graphs below:

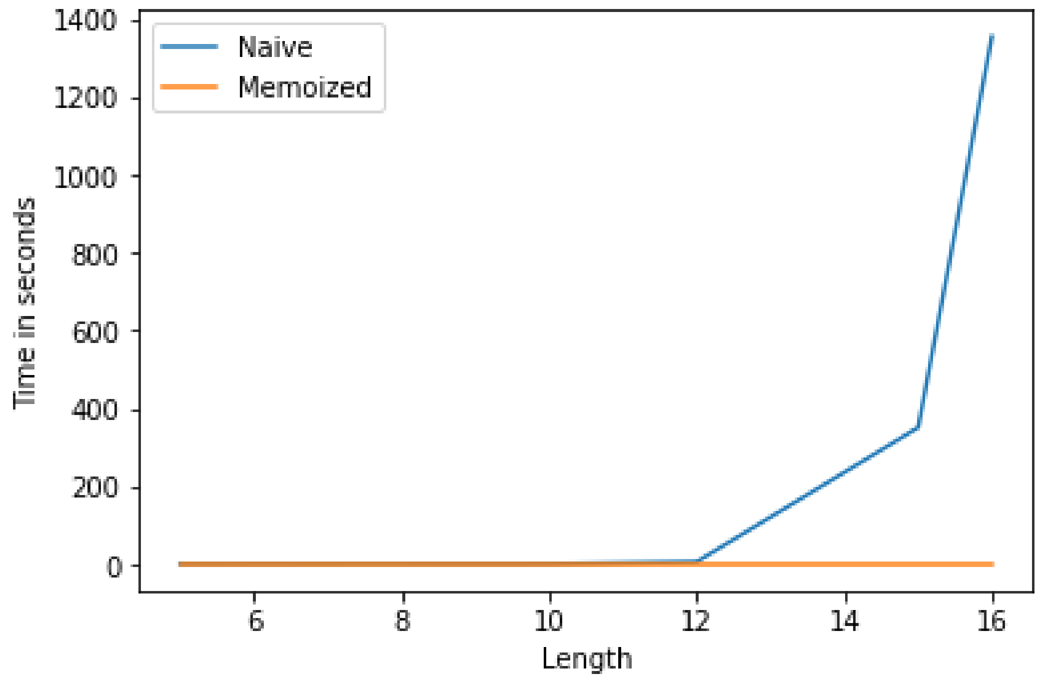
1. The first one shows the time each algorithm requires for n = 5,10,12. As it is seen, difference between required times of two algorithms is much smaller for n values smaller than 10.

2. Second graph shows times for n = 5,10,12 and 15.

3. Third graph shows times for n = 5,10,12,15,16. As it can be seen, difference is increasing with a rate which is also increasing, This implies that the naive algorithm grows much faster than the other one.


Naive
Memoized
Time in seconds
Length
0
1
2
3
4
5
6
5
6
7
8
9
10
11
12


Naive
Memoized
Time in seconds
Length
0
50
100
150
200
250
300
350
6
8
10
12
14

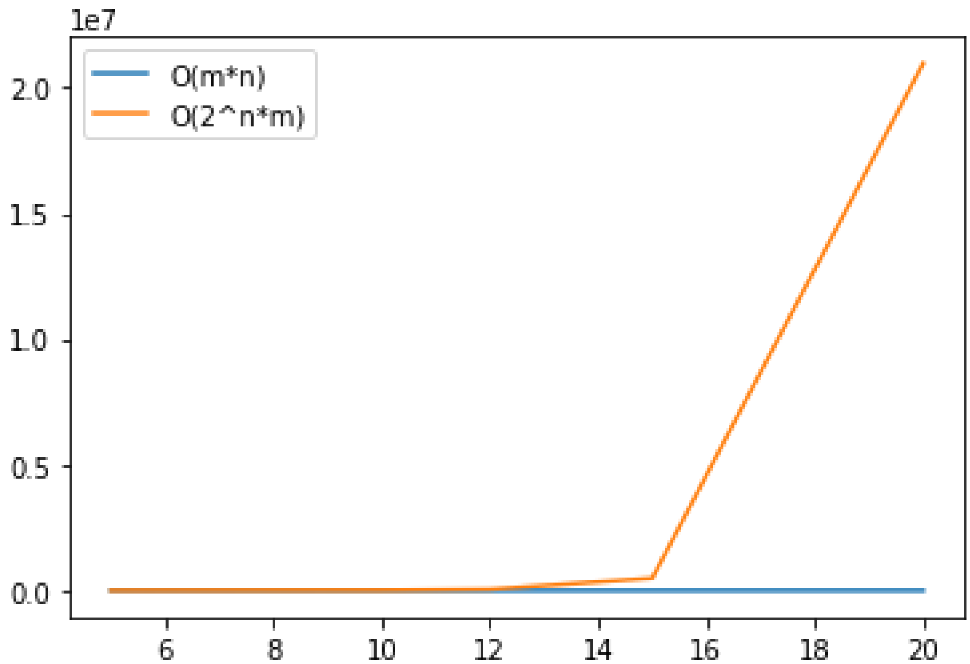


(iii) Discuss the scalability of the algorithms with respect to these experimental results. Do the experimental results confirm the theoretical results you found in (a)?

Answer: Experimental results confirm the theoretical result I've found.

For instance; when n = m = 5, the time memoized algorithm requires is approximately 0.0000643. Since the complexity is $\Theta(n * m)$, we expect the time for n = m = 10 to be 4 times of this value. $(10^2/5^2 = 4)$ $0.0000643 * 4 = 0.0002572 \approx 0.00024$ (the time I've found is 0.00024).

You can see the graph of functions $m*n$ and 2^n*m for values 5,10,12,15,20 and realize how similar they grow with plot of our results below.



(c) **(40 points)** For each algorithm, determine its average running time experimentally by running it with randomly generated DNA sequences of length $m = n$. For each length $5, 10, 15, 20, 25$, you can randomly generate 30 pairs of DNA sequences, using Sequence Manipulation Suite.[1]

(i) Fill in following table with the average running times in seconds (μ), and the standard deviation (σ).

Algorithm	$m = n = 5$		$m = n = 10$		$m = n = 12$		$m = n = 15$		$m = n = 20$	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Naive	3.20e-5	2.17e-5	0.0046	0.0046	0.0235	0.0222	0.5138	0.6105	46.531	57.814
Memoization	1.48e-5	4.359e-6	5.84e-5	9.328e-6	8.703e-5	3.4110e-5	0.00013	1.434e-5	0.00022	2.687e-5

(ii) Plot these experimental results in a graph.

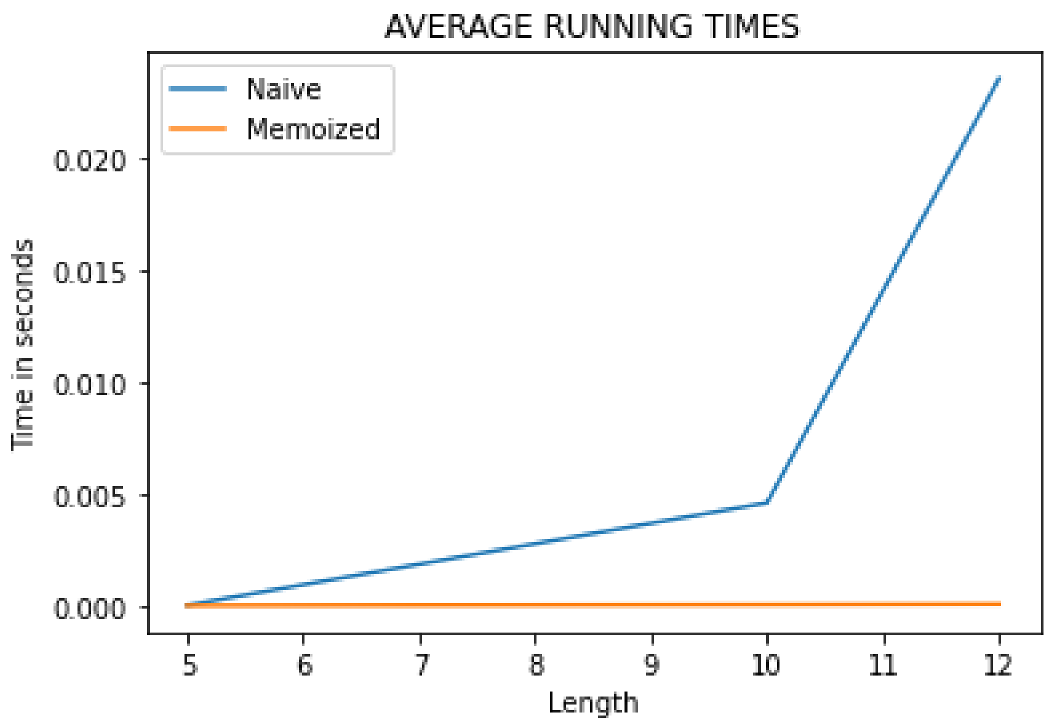
You can see 3 different graphs below:

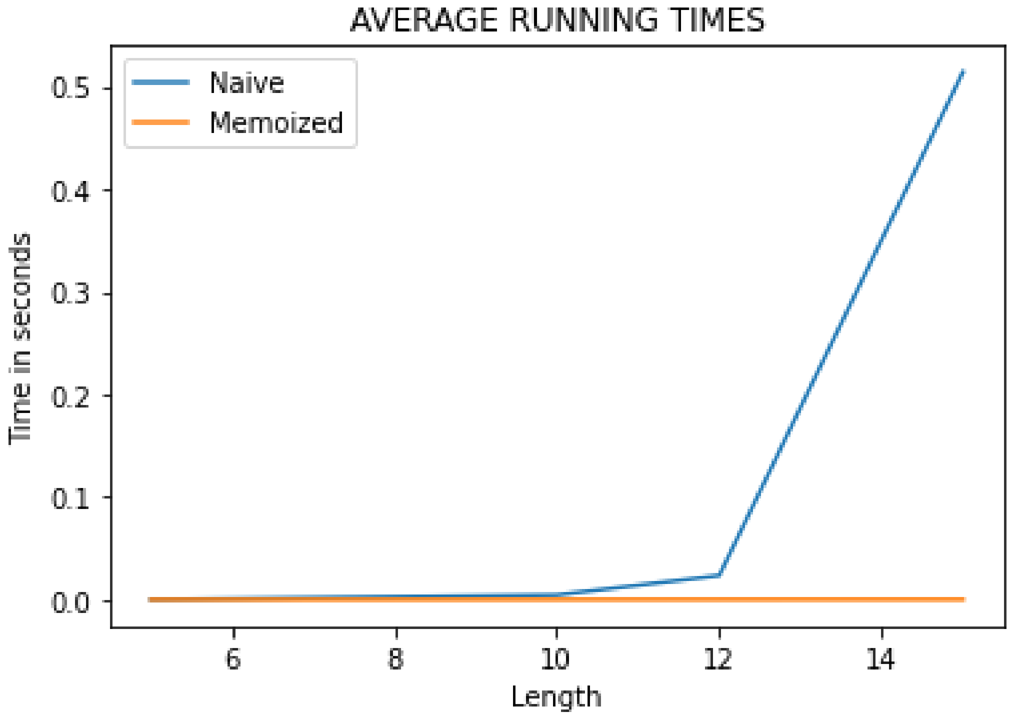
1. The first one shows the time each algorithm requires for n = 5,10,12. As it is seen, difference between required times of two algorithms is much smaller for n values smaller than 10.

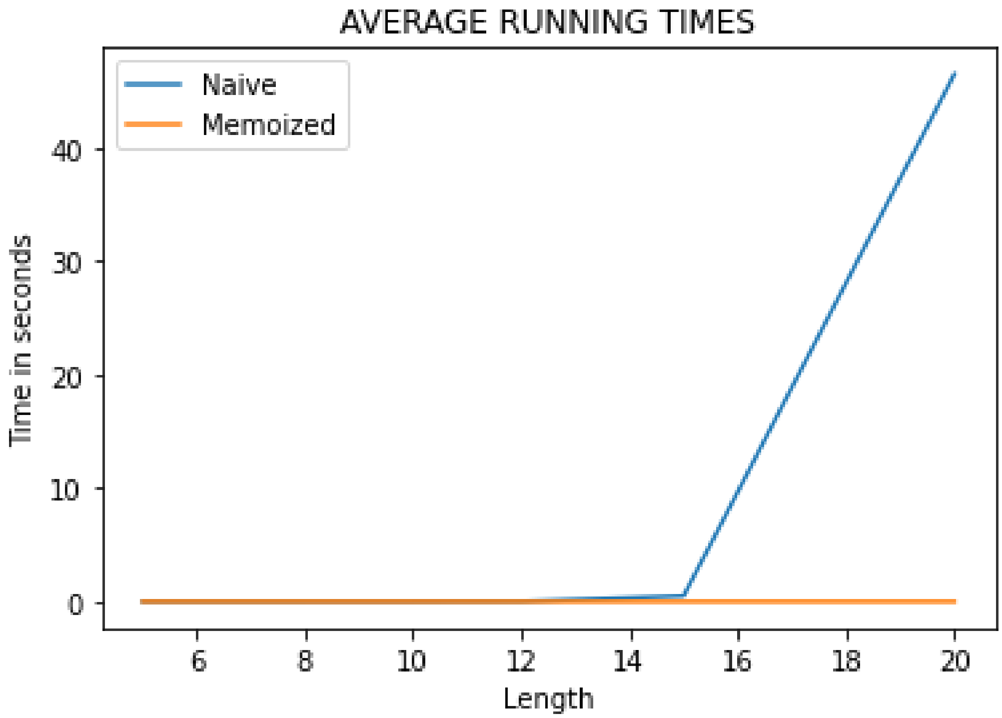
[1] `https://www.bioinformatics.org/sms2/random_dna.html`.

2. Second graph shows times for n = 5,10,12 and 15.

3. Third graph shows times for n = 5,10,12,15,20. As it can be seen, difference is increasing with an increasing rate. This implies that the naive algorithm grows much faster than the other one.




AVERAGE RUNNING TIMES
Naive
Memoized
Time in seconds
0.0
0.1
0.2
0.3
0.4
0.5
6
8
10
12
14
Length



(iii) Discuss how the average running times observed in your experiments grow, compared to the worst case running times observed in (b).

ANSWER: Average running times grows much slower than the worst case running times observed in (b). For example; n = m = 15 requires 0.022 seconds in average with naive algorithm whereas it requires 350 seconds in the worst-case. Similarly, m = n = 10 requires 0.00024 seconds in the worst case with memoization, but it requires only 0.0000584 seconds in average case. There is a big difference between worst-case and average case. However, we can still say that the naive algorithm grows much faster than the memoized algorithm both in worst-case and average case.We can also say that the naive algorithm still grows exponentially and memoized algorithm still grows polynomially in average case.