Student Name: Mert Ekici
Student ID: 26772

Algorithms (CS301)
Assignment 1

Sabancı Üniversitesi

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 use ***the Master Theorem*** here to find asymptotic tight bound for $T(n)$ in the following recurrences.

There are three cases that help us solve recurrences in the Master Theorem;

$$\text{Given } T(n) = aT(n/b) + f(n)$$
$$\text{such that } a \geq 1, b > 1, f(n) \text{ asymptotically positive}$$

- *Case 1*
  if $f(n) = O(n^{\log_b a - \varepsilon})$ for some $\varepsilon > 0$ $\implies T(n) = \Theta(n^{\log_b a})$

- *Case 2*
  if $f(n) = O(n^{\log_b a})$ $\implies T(n) = \Theta(n^{\log_b a} \log n)$

- *Case 3*
  if $f(n) = O(n^{\log_b a + \varepsilon})$ for some $\varepsilon > 0$
  and if $af(n/b) \leq cf(n)$ $\implies T(n) = \Theta(f(n))$
  for some $c > 1$ and for large n

(a) $T(n) = 2T(n/2) + n^3$
Case 3 holds. $n^3 = O(n^{1+\varepsilon})$ for some $\varepsilon > 0$ and $2(n/2)^3 \leq cn^3$ for some $c > 1$ and for large n

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

(b) $T(n) = 7T(n/2) + n^2$
Case 1 holds. $n^2 = O(n^{\log_2 7 - \varepsilon})$ for some $\varepsilon > 0$

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

(c) $T(n) = 2T(n/4) + \sqrt{n}$
Case 2 holds. $\sqrt{n} = O(n^{1/2})$

$$T(n) = \Theta(\sqrt{n} \log n)$$

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

For the last recurrence, **the Iteration Method** will be used since the form of $T(n-1)$ is not suitable for the Master Theorem.

$$\begin{aligned} T(n) &= T(n-1) + n \\ &= T(n-2) + 2n - 1 \\ &= T(n-3) + 3n - 3 \\ &= T(n-4) + 4n - 6 \end{aligned}$$

Since, $T(n)$ is constant for $n \leq 2$ and the recurrence expands with a pattern;

$$T(n) = T(n-i) + in - \frac{(i)(i-1)}{2} \text{ and } n - i = 2$$
$$T(n) = T(2) + n(n-2) - \frac{(n-2)(n-3)}{2}$$
$$T(n) = \Theta(n^2)$$

Problem 2 (Longest Common Subsequence - Python)

Consider the two algorithms for the Longest Common Subsequence problem, shown in Figures 1 and 2. Each algorithm takes two strings, X and Y, and two natural numbers, i and j, and returns the length of the longest common subsequence (LCS) between the prefixes $X[0..i]$ and $Y[0..j]$ of the given strings. The algorithm shown in Figure 1 is a naive recursive algorithm whereas the algorithm shown in Figure 2 is a slight variation with memoization. We can compute the length of the LCS of two given strings, using these two algorithms, as illustrated in Figure 3. In the following, m is the length of the string X, and n is the length of the string Y.

(a) 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 (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.

In the worst-case, all characters of the arrays X and Y mismatch. Accordingly, LCS becomes 0. The naive recursive algorithm's each recursive call ends up two recursive calls when there is no common subsequence in X and Y.

The worst-case complexity of the naive solution is $O(2^{(m+n)})$.

This LCS problem is called **overlapping subproblems** where the same subproblems are getting solved repeatedly. Overlapping substructure can be avoided by using the next algorithm: **Memoization**.

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

The recursive algorithm with memoization provides a top-down solution where it breaks the problem into subproblems and solves them recursively while skipping the ones that are already solved & stored (memoized). Since there is no repeated computation;

The worst-case running time of is $O(mn)$ with $O(mn)$ additional space requirement.

```
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

```
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

```
X = "acggacgggatctgggtccg"
Y = "tcccacatggtgcttccccg"
lX = len(X)
lY = len(Y)
#uncomment the next line to initialize c (for memoization)
#c = [[-1 for k in range(lY+1)] for l in range(lX+1)]
print ("Length of LCS is ", lcs(X,Y,lX,lY))
```

Figure 3: An example: Computing the length of the LCS of two given DNA sequences

(b) 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.

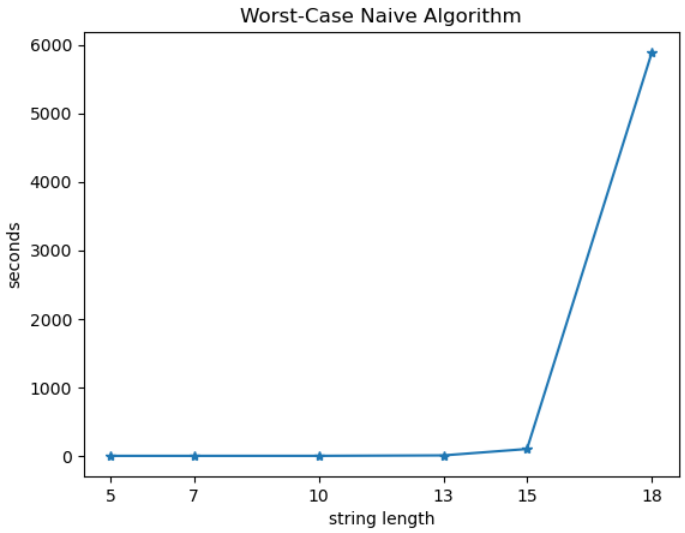
(i) Fill in following table with the running times in seconds.

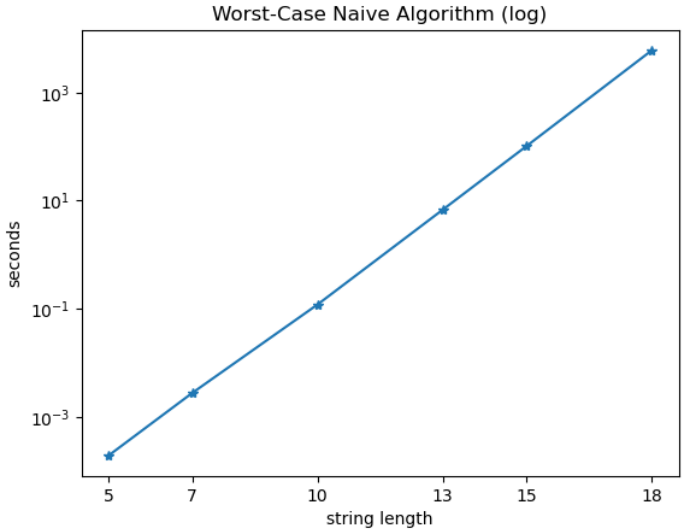
Algorithm	$m = n = 5$	$m = n = 7$	$m = n = 10$	$m = n = 13$	$m = n = 15$	$m = n = 18$
Naive	19e-4	28e-3	0.119	6.75	100.3	5889.5
Memoization	2.62e-5	7.46e-5	9.61e-5	1.71e-4	20.6e-4	29.4e-4

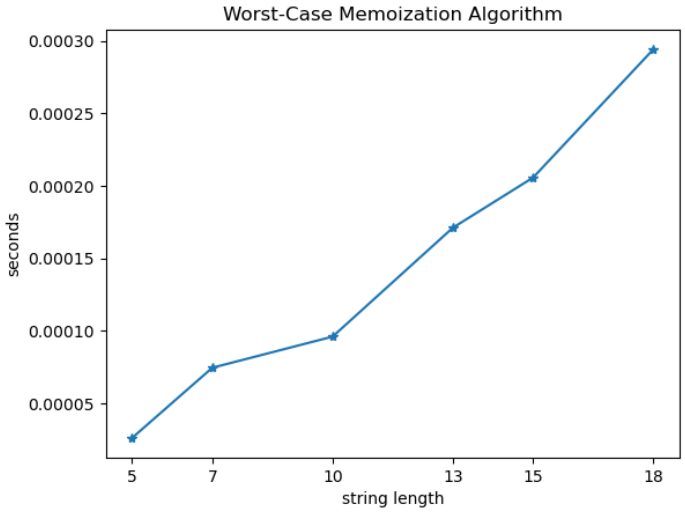
Specify the properties of your machine (e.g., CPU, RAM, OS) where you run your programs.

Processor:	Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz (12 CPUs), 2.6GHz
Memory:	32768MB RAM
Operating System:	Windows 10 Pro 64-bit
Graphics Card:	NVIDIA GeForce RTX 2070

(ii) Plot these experimental results in a graph.







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

The worst-case complexities are analyzed as $O(2^{m+n})$ for **the Naive Algorithm** and $O(mn)$ for **Memoization** in (a).

No-match string pairs are selected as the worst-case scenario for each given length. When results compared, the worst-case running time of Memoization increases almost linearly with the increasing string length whereas Naive Algorithm's running time increases exponentially where execution times with the lenghts of 13, 15 and 18 are 6.75, 100.3 and 5889.5 seconds respectively.

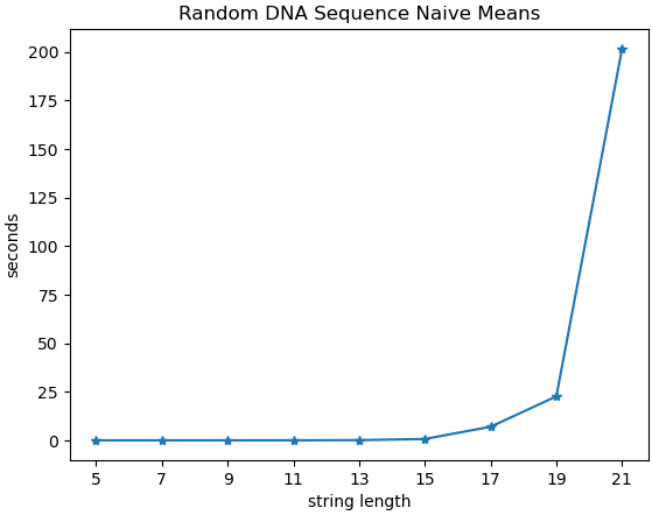
Difference between runtimes are understandable but the analysis conducted in (a) claims that both algorithms grow exponentially where the Naive Algorithm grows faster since $O(2^{m+n}) \geq O(mn)$. The results support the fact that Memoization Algorithm is usually faster than the Naive Approach for the worst-case scenario and the running time of the Naive Algorithm grows faster.

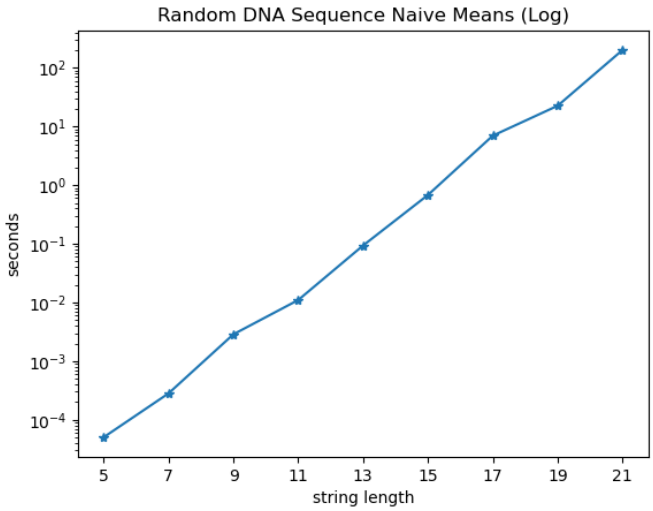
(c) For each algorithm, determine its average running time experimentally by running it with randomly generated DNA sequences of length $m = n$. For each length 5, 10, 15, 20, 25, you can randomly generate 30 pairs of DNA sequences, using Sequence Manipulation Suite.1

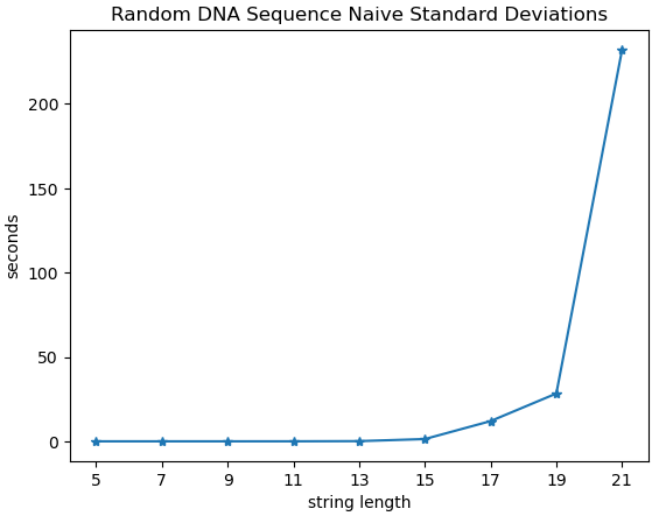
(i) Fill in following table with the average running times in seconds μ, and standard deviation σ.

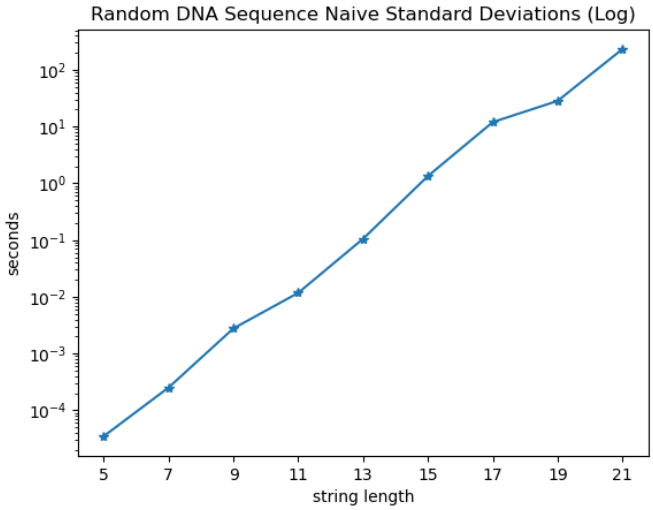
Length	**Statistics**	**Naive**	**Memoization**
$m = n = 5$	μ	5e-5	2e-5
	σ	3.5e-5	4e-6
$m = n = 7$	μ	2.8e-4	3.2e-5
	σ	2.5e-4	6.6e-6
$m = n = 9$	μ	2.8e-3	5.9e-5
	σ	2.8e-3	1.6e-5
$m = n = 11$	μ	0.011	8.1e-5
	σ	0.011	1.8e-5
$m = n = 13$	μ	0.09	1.1e-4
	σ	0.10	2.1e-5
$m = n = 15$	μ	0.68	1.4e-4
	σ	1.34	2.6e-5
$m = n = 17$	μ	6.98	1.8e-4
	σ	11.99	2.1e-5
$m = n = 19$	μ	22.61	2.2e-4
	σ	28.35	4.3e-5
$m = n = 27$	μ	201.6	2.7e-4
	σ	232.17	5.7e-5

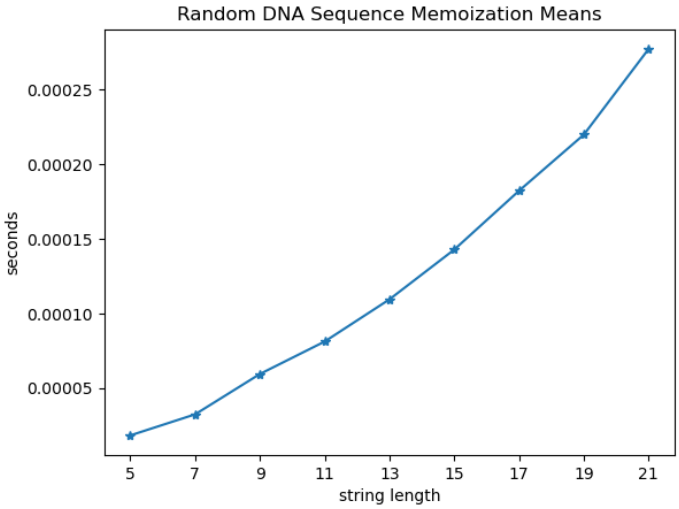
(ii) Plot these experimental results in a graph.

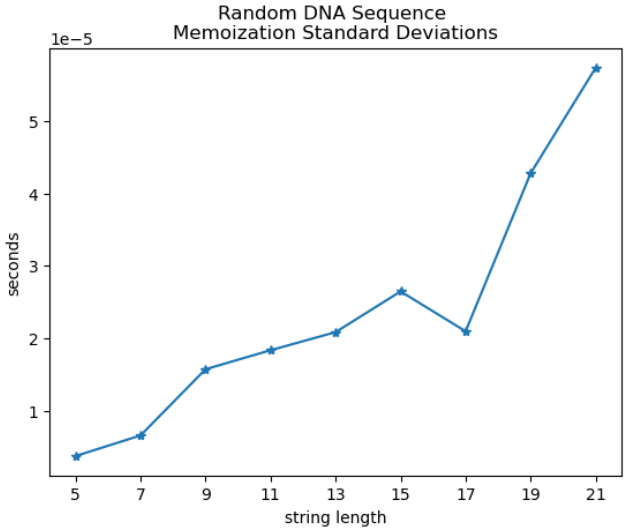


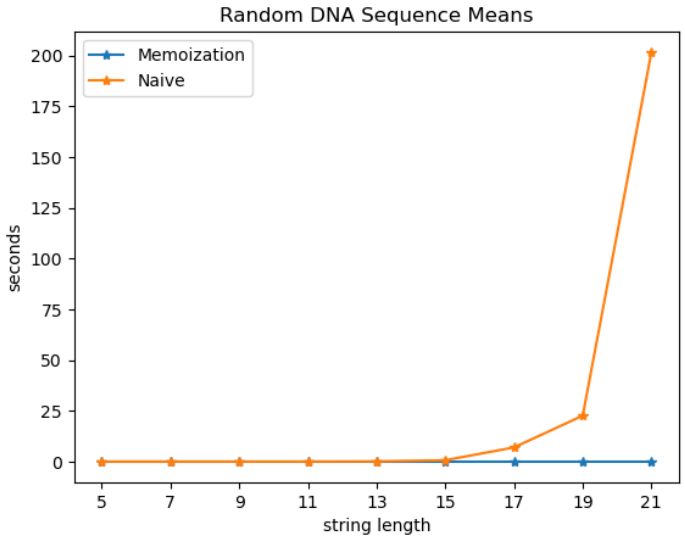












(iii) Discuss how the average running times observed in your experiments grow, compared to the worst case running times observed in (b).

The average running times of both algorithms show similar behavior compared to (b) when 30 randomly generated DNA sequences are used with given lenghts. Again, memoization's runtime seems to increase linearly if we ignore the slight convexity that can be observed from the *Random DNA Sequence Memoziation Means* plot. Reason for this behavior could be small string lengths. However, it is computationally difficult to run these algorithms for longer strings. For the Naive Algorithm, we can see that the runtime again increases exponentially which is also demonstrated in log-form plot of *Random DNA Sequence Naive Means (Log)*. Another interesting finding is the similar pattern followed by standard deviations where their values are pretty close to mean values and they grow almost identically.

In general, Memoization runs much faster than the Naive Algorithm with the random DNA sequences as well. It is important to understand that this case is far from the worst-case scenario since only 4 different characters are repeated in DNA sequences and accordingly, random sequences inevitably contain matching subsequences. Therefore, the running time results for DNA case is smaller than the results in the worst-case part for the same length strings.