CS301 Algorithms-Assignment 1

Alper Kaan Odabaşoğlu - 28147

October 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 $\leq$ 2. No explanation is needed.

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

$\Rightarrow By\ Master\ Theorem : a = 2 \geq 1\ \ b = 2 > 1\ \ f(n) = n^3$
$\Rightarrow n^{log_b a} = n^{log_2 2} = n$
$\Rightarrow f(n) = n^3 = \Omega(n^{1+\epsilon})$
$\Rightarrow n^3 = \Omega(n^{1+\epsilon}) \longrightarrow n^3 \geq c \cdot n^{1+\epsilon}\ \ if\ c = 1\ and\ \ \forall n \geq 1$
$\Rightarrow Case\ 3\ of\ Master\ Theorem :$
$\Rightarrow\ a \cdot f(n/b) \leq c \cdot f(n)$
$\Rightarrow 2 \cdot f(n/2) \leq c \cdot f(n)$
$\Rightarrow\ 2 \cdot (n/2)^3 \leq c \cdot n^3$
$\Rightarrow\ n^3/4 \leq c \cdot n^3\ \ for\ some\ \ 1/4\ \ \leq c < 1$ and $\forall n$
$\Rightarrow Therefore\ T(n) = \Theta(f(n)) = \Theta(n^3)$

b) T(n) = 7T(n/2) + n^3

$\Rightarrow Master\ Theorem :\ a = 7 \geq 1\ \ b = 2 > 1\ and\ f(n) = n^2$
$\Rightarrow Lets\ Calculate\ n^{log_b a} = n^{log_2 7} \longrightarrow n^2 \leq n^{log_2 7} < n^3$
$\Rightarrow f(n) = n^2 = O(n^{log_2 7-\epsilon})\ for\ some\ \epsilon > 0$
$\Rightarrow\ Case\ 1\ of\ Master\ Method\ Applies :$
$\Rightarrow\ Therefore\ T(n) = \Theta(n^{log_b a}) = \Theta(n^{log_2 7})$

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

$\Rightarrow Master\ Theorem :\ a = 2 \geq 1\ \ b = 4 > 1\ and\ f(n) = \sqrt{n} = n^{1/2}$
$\Rightarrow n^{log_b a} = n^{log_4 2} = n^{1/2}$
$\Rightarrow f(n) = n^{1/2} = \Theta(n^{1/2}) = \Theta(n^{log_4 2})$
$\Rightarrow\ Case\ 2\ applies :\ T(n) = \Theta(n^{log_4 2} \cdot \log n) = \Theta(\sqrt{n} \cdot \log n)$

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

$\Rightarrow T(n) = [T(n-2)+n-1]+n = T(n-2)+2n-1$
$\Rightarrow \qquad = [[T(n-3)+n-2]+n-1]+n = T(n-3)+3n-3$
$\Rightarrow \qquad = [[[T(n-4)+n-3]+n-2]+n-1]+n = T(n-4)+4n-6$
$\Rightarrow \qquad .....$
$\Rightarrow \qquad ....$
$\Rightarrow \qquad ....$
$\Rightarrow \qquad ....$
$\Rightarrow \qquad = T(n-k)+k\cdot n-[\sum_{i=0}^{i=k-1} i]$
$\Rightarrow When\ \ k=n \longrightarrow T(0)+n^2-[\sum_{i=0}^{i=n-1} i]$
$\Rightarrow T(0)\ is\ constant\ because\ T(n)\ is\ constant\ for\ n\leq 2$
$\Rightarrow T(n) = T(0)+n^2-[0+1+2+3+...+(n-1)]$
$\Rightarrow \qquad = T(0)+n^2-[((n-1)\cdot n)/2]$
$\Rightarrow \qquad = T(0)+n^2-n^2/2+n/2$
$\Rightarrow \qquad = T(0)+n^2/2+n/2$
$\Rightarrow \qquad = c+n^2/2+n/2$ where c is a constant
$\Rightarrow$ So there are quadratic, linear and constant time functions exist. The dominant one is the quadratic ($n^2/2$).
$\Rightarrow$ Therefore $T(n) = \Theta(n^2/2) = \Theta(n^2)$

2 Question 2

2.1 What are the costs of naive and memoization 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))
```

When I looked at the recurrence sequence in naive algorithm, I found that the complexity is directly related with the size (m, n) of the strings that are given into the least common subsequence algorithm because the number of recursive calls depends on these sizes. The worst case happens when there is no common subsequence amongst the given strings and the sizes of these strings are equal (m = n). The main reason behind this conception is that when there is no subsequence exist, the algorithm enters the else condition and calls 2 different recursion up to the point that each recursion reach to the return statement where i or j (indices) equals 0. In other words, first lcs(X,Y,i,j-1) this recursion will be handled up to the point the j = 0 and after that other recursion will be called up to the point that i = 0. And this pattern will be tracked for each sub-recursive-function calls. So the main aspect that directly affects the time complexity is the else part of the algorithm since there is 2 recursive calls in worst case. So we can write the recursion as:

T(m, n) = T(m-1, n) + T(m, n-1) + c where c = O(1) i.e. c is constant

The constant c comes from the cost of each recursion, max() function's comparison and if-else checks. So the total complexity can be demonstrated with a recursion tree.

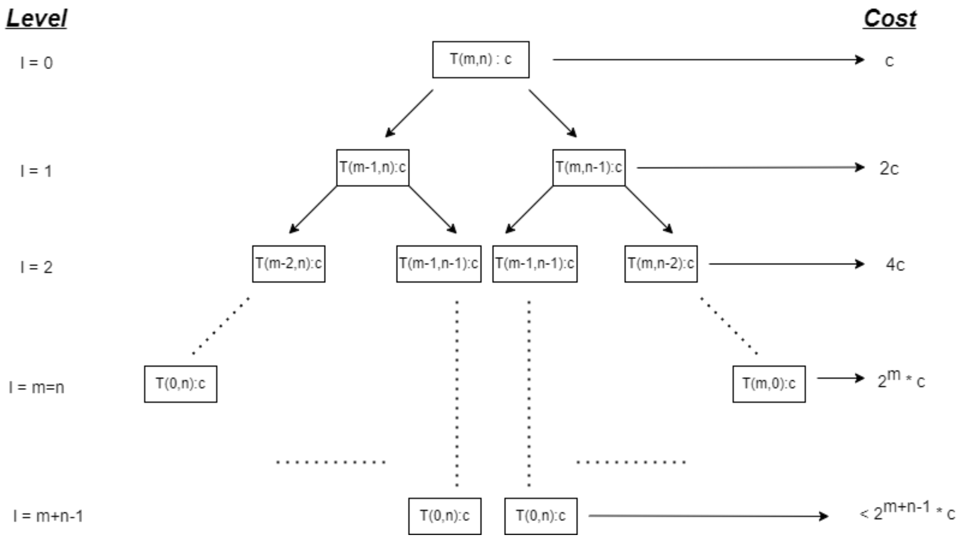


figure 1 (recursion tree)

The recursion in left and right sub-tree will be over earlier because just one of the m or n goes to 0. It just takes m = n steps in worst case. However the middle steps, take more time equal to m+n-1 steps which is directly related to the sum of the sizes. The reason is that one of m or n goes up to 1 and other goes to 0 to terminate the recursion. Thus, to make n = 0 and m = 1 or to make n=1 and m = 0, algorithm should cover m+n-1 steps which also represents the height of the tree.

It is not possible to calculate the exact total cost of the algorithm with this recursive tree since the leaves in the recursion tree is not distributed equally. We can see that the left and right most subtrees finish earlier. Even though the leaves in middle of tree go up to m+n-1 level, there will not be 2^{m+n-1} leaves.

That's why, it is demonstrated as $< 2^{m+n-1}$ at the bottom level cost (figure 1):

$$\begin{aligned} \textit{total cost} &< c + 2c + 4c + ... + (2^{m+n-1})c \\ &< c \cdot [\sum_{i=0}^{i=m+n-1} 2^i] \\ &< c \cdot ((2^{m+n} - 1)/(2-1)) \end{aligned}$$

So the total cost is lower than $(2^{m+n} - 1) \cdot c$ where the constants do not matters. So it can be evaluated as $O(2^{m+n})$. This result can be used as a guess for substitution method to found the best worst case complexity of algorithm.

Substitution Method:
T(m,n) = T(m-1, n) + T(m,n-1) + c. We need to substitute this recurrence to $O(2^{m+n})$. To do that we can look at $T(m,n) \leq 2^{m+n} \cdot (c_1 + c_2)$

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

Induction base: T(1,1) $\leq (c_1 + c_2) \cdot 2^{1+1} \leq c$ We can pick c big enough to ensure this equation in all permutations.

Induction hypothesis for variable m: $T(k,n) \leq c_1 \cdot 2^{k+n}$ where k < m $\exists c_1 > 0$
Induction hypothesis for variable n: $T(m,t) \leq c_2 \cdot 2^{m+t}$ where t < n $\exists c_2 > 0$

T(m-1, n) ensures the induction hypothesis for m because m-1 < m. So we can substitute it as:

$$\text{T(m-1,n)} \leq 2^{(m-1)+n} \cdot c_1$$

T(m, n-1) ensures the induction hypothesis for n because n-1 < n. So we can substitute it as:

$$\text{T(m,n-1)} \leq 2^{m+(n-1)} \cdot c_2$$

At the end we can sum these two equalities and just add the c from the main reccurence to reach the T(m,n):

$$\begin{aligned} T(m,n) &\leq 2^{(m-1)+n} \cdot c_1 + 2^{m+(n-1)} \cdot c_2 + c \\ &\leq 2^{m+n} \cdot (c_1 + c_2)/2 + c \\ &\leq 2^{m+n} \cdot (c_1 + c_2) - [2^{m+n} \cdot (c_1 + c_2)/2 - c] \\ &\leq 2^{m+n} \cdot (c_1 + c_2) \text{ for } c_1 = c_2 = c = 1 \text{ and } \forall m > 1 \text{ and } \forall n > 1 \end{aligned}$$

The desired part is $2^{m+n} \cdot (c_1 + c_2)$ and the residual part is $[2^{m+n} \cdot (c_1 + c_2)/2 - c]$. When we select for some $c_1 = c_2 = c = 1$ *and* $\forall m > 1$ *and* $\forall n > 1$ The residual part will always be positive. Thus, we can say that the final equation is smaller than $2^{m+n} \cdot (c_1 + c_2)$ in this condition.

Therefore the total complexity for this algorithm is $O(2^{m+n})$.

The worst case happens when m = n. So the best asymptotic worst-case running time of the naive recursive algorithm is $O(2^{2m}) = O(4^m)$

2.1.2 Memoization Algorithm

```
def lcs(X,Y,i,j):
    if c[i][j] $\geq$ 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]
```

Due to memoization mechanics, in each return statement, one of the index in c[i][j] 2D array become 0. For checking whether the array includes given index, O(1) cost is spent. So, in the worst case scenario, the array indices are checked except the i= 0 and j=0 indices. The reason why all of the slots in c[i][j] will be visit except the c[0][0] is that the design of the algorithm. Due to the if-else checks both i and j indices will not be equal to zero. After the algorithm memorizes the indices that were visited with matrix, the indices will be checked once more due to the recurrence mechanics. To check memorized indices, the algorithm will spend O(1) time. Also, max(lcs(X,Y,i,j-1),lcs(X,Y,i-1,j)) function costs O(1) (comparison of both recursive function calls' return statement) time to assign 0 to memorized indices. At the end, the dominant phenomena that shapes the complexity stems from the matrix size in the worst case since all array indices are sub-process with recurrence to ensure the memoization mechanic. Thus one index pair will just be checked at most 2 times. So the complexity becomes,

$\Theta((m+1)\cdot(n+1)+c) = \Theta(m\cdot n+m+n+c) = \Theta(m\cdot n)$ where c = O(1), m and n are size of strings that are given to algorithm

This algorithm can be understood with a recursion trees as well.

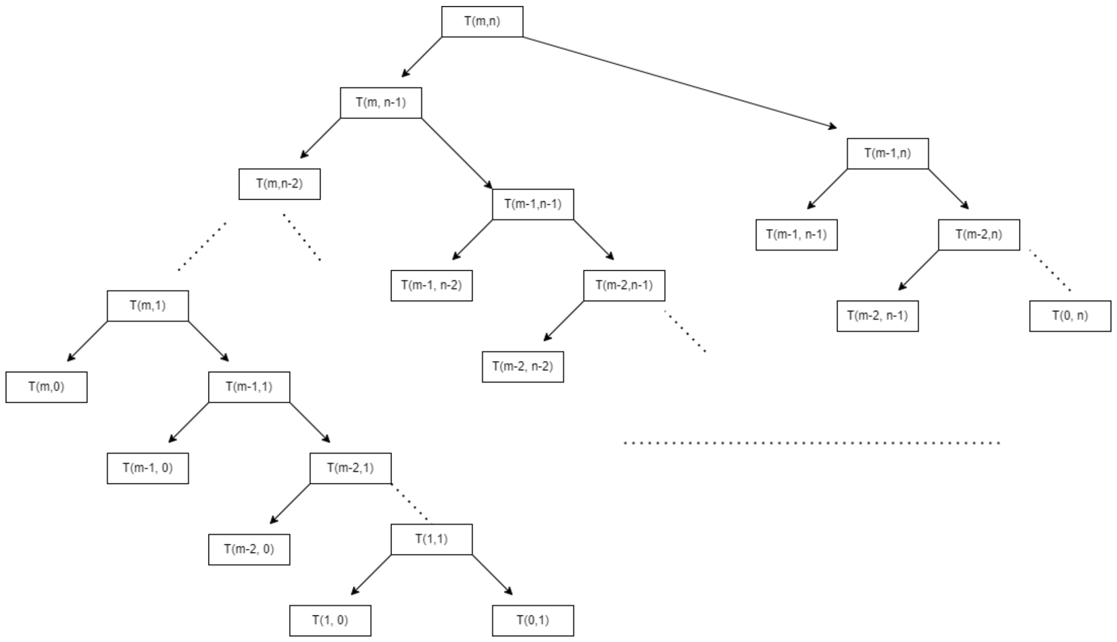


figure 2 (recursion tree of memoization)

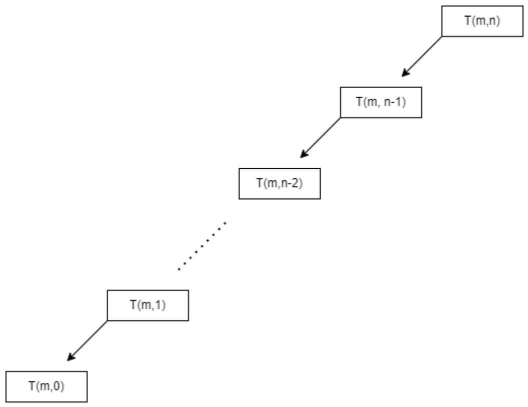


figure 3 (most-left sub-tree of memoization)

It can be observed that, except the most left subtree, which is represented in figure 3, all of the left subtrees in figure 2 bound to the right subtrees includes one leaf since all of the indices that these recursive function calls are stored in matrix in lower subtrees. Thus, if c[i][j] ≥ 0 block will be called and return c[i][j] statement executed, the recursive function do not continue to cover lower indices, like a domino. So each node in figure 3 includes $2 \cdot m - 1$ recursive function calls. Which result in $(2 \cdot m - 1) \cdot n$ leaves. This represents the costs coming from the right subtrees. In addition, the most left subtree led to the cost of (n+1) nodes. In total, there will be $\Theta(n \cdot (2m-1) + n + 1) = \Theta(2m \cdot n + 1) = \Theta(m \cdot n)$ cost.

It can be seen that, the mathematical manipulation is again directly related to the memoization mechanics in 2D array. Thus the array size which is $(m+1) \cdot (n+1)$ still dominantly affect the complexity.

In the worst case scenario, where m = n, the complexity of this algorithm is $\Theta(n \cdot n) = \Theta(n^2)$. This quadratic complexity is relatively better than the naive algorithm's exponential complexity.

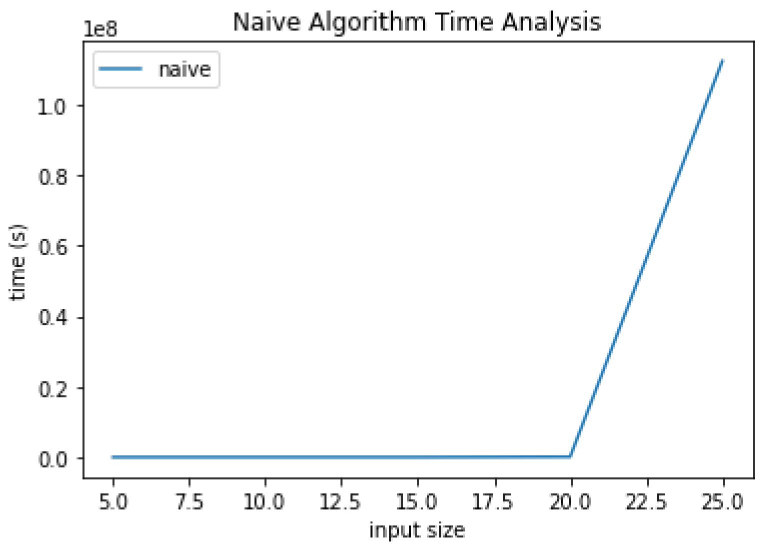
2.2 Worst Case Scenario Experiments

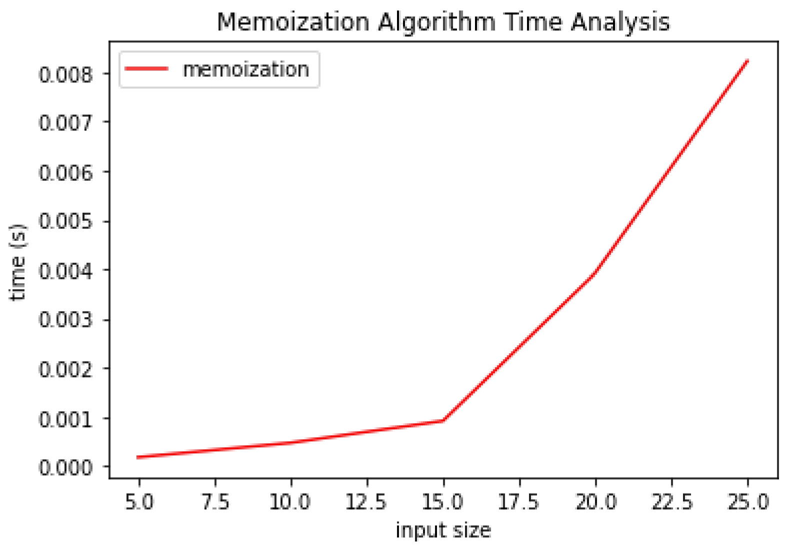
2.2.1 Experimental Result Representation on Table

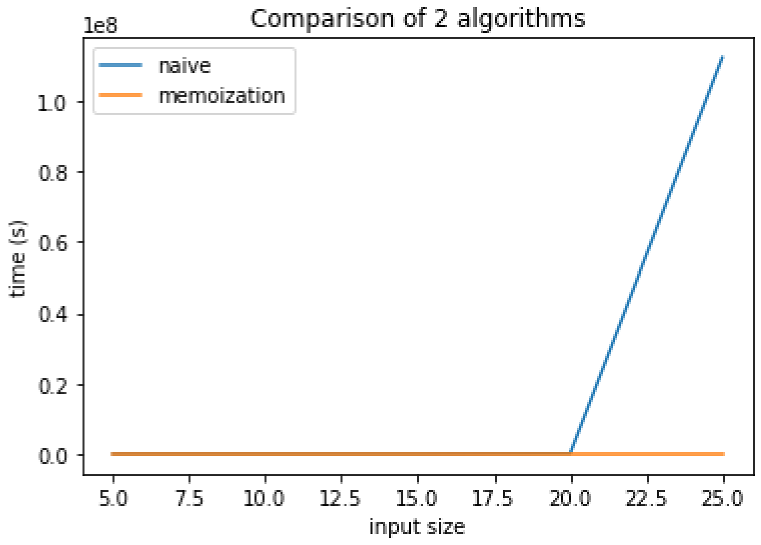
Algorithm	m=n=5	m=n=10	m=n=15	m=n=20	m=n=25
Naive	0.00037	0.1448	107.24845	109822.41	112458156.5
Memoization	0.000179	0.00047	0.00091	0.0049	0.0072

CPU: INTEL CORE I5 9300-H CPU @ 2.40 GHZ
RAM: 16GB
OS: Windows 11 Home Single Language 64bit
COMPILER: Google Colab

2.2.2 Vizualization of experimental Results







2.2.3 Scalability Of Algorithms

- Naive Algorithm

In terms of Scalability of Naive Algorithm, my experimental results almost matched with my theoretical results. For the bigger input sizes it is not suitable and scalable for compilation. In the worst case scenario, exponential growth is observed which can be also scene in graphs above. As it was found in theoretical worst case analysis, for each input size on table, there was nearly 4^x increase. Due to the exponential growth, I could not compile the results with m=n=20 (probably compiled in more than 30 hours) and m=n=25 (probably compiled in more than 3 years). Thus, I predict the values with the theoretical results I found in analysis. As I could not experiment the input sizes 20 and 25 I experiment another intermediate sizes like 12-13-14-17 and when I analyzed the results, again it can be observed that the successive steps are 4 times of each other. The experimental results observed in intermediate sizes:

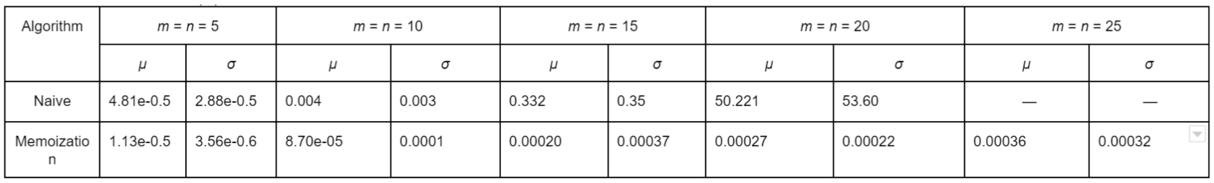
m=n=12: 1.7962124347686768 seconds
m=n=13: 6.964385986328125 seconds
m=n=14: 27.290936946868896 seconds
m=n=16: 399.00916028022766 seconds
m=n=17: 1568.313156604766 seconds

- Memoization Algorithm

In terms of scalability, memoization algorithm is much more suitable as well as scalable for bigger input sizes. Because the complexity $O(n^2)$ is much more faster than $O(4^n)$ in worst case where m=n. This conclusion can be observed by the experimental results and the comparison on graphs. As we can see, on m=n=25, there is a huge difference where naive algorithm behave at the level of 10^8 compared to the memoization algorithm which is still below 1. Again, the experimental results of memoization algorithm is almost matched with theoretical results where the increase in worst case (m=n) is quadratic (O(n^2)).

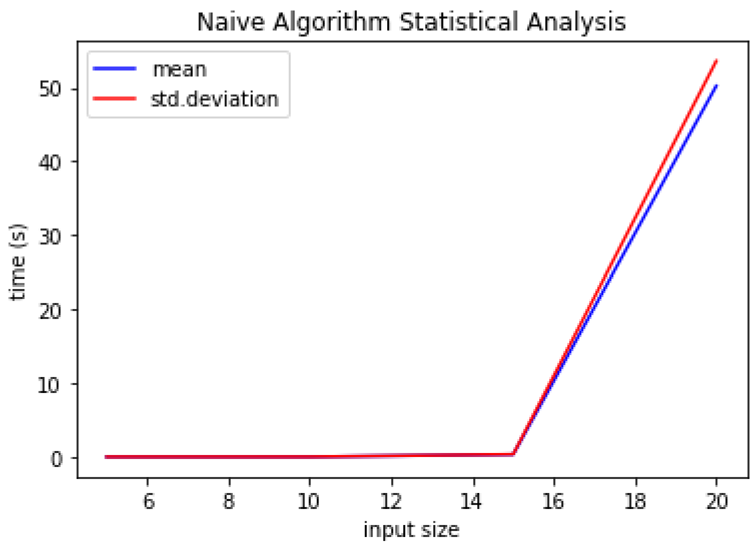
2.3 Statistical Analysis of Average Running Time

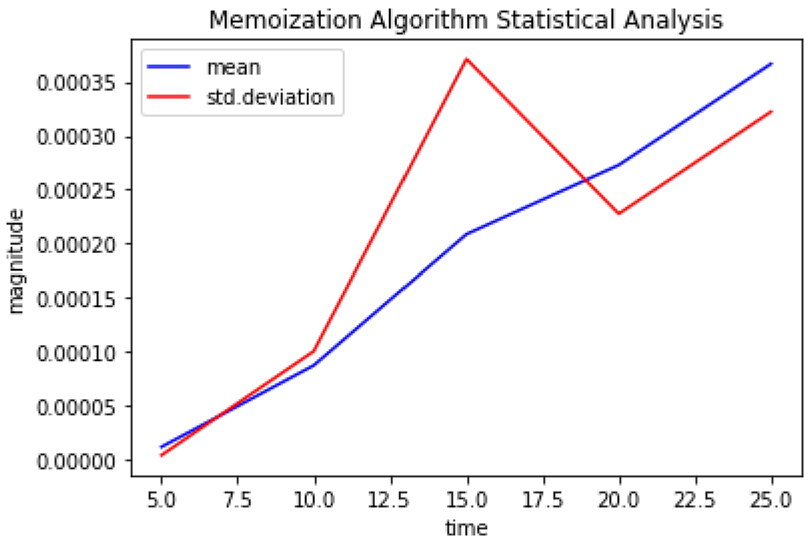
2.3.1 Statistical Result Representation on Table

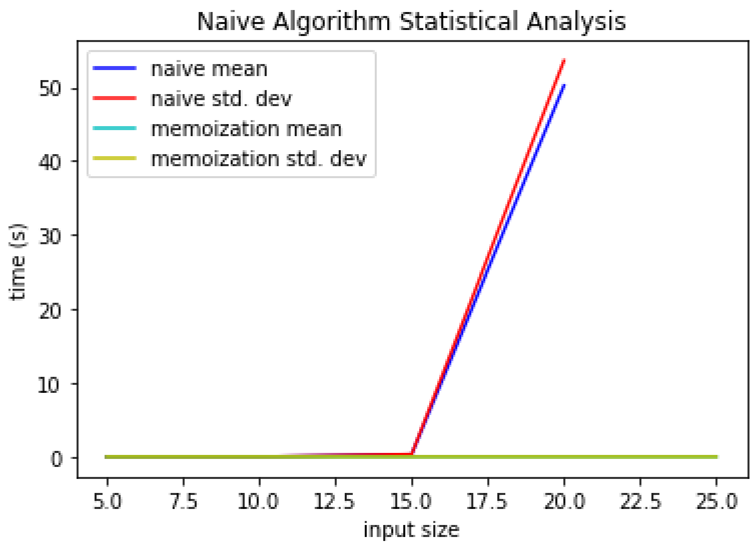


Algorithm	$m = n = 5$		$m = n = 10$		$m = n = 15$		$m = n = 20$		$m = n = 25$	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Naive	4.81e-0.5	2.88e-0.5	0.004	0.003	0.332	0.35	50.221	53.60	—	—
Memoizatio n	1.13e-0.5	3.56e-0.6	8.70e-05	0.0001	0.00020	0.00037	0.00027	0.00022	0.00036	0.00032

2.3.2 Vizualization of experimental Results







2.3.3 Comparison of average and worst case experimental running time

- Naive Algorithm

The average running time of naive is extremely faster than the worst case scenario even though the exponential behavior maintained because the successive 2 recursion call count decrease in our algorithm. Whenever there is a common subsequence, instead of 2 recursive calls, the algorithm execute 1 recursive call which crucially affect the running time. As we can see in m=n=20, our prediction was 109822.41 but in average case it go below to 50.221 seconds which is 2186.78 times faster. So in bigger input sizes the average case is way rapid than the worst case.

I could not measure the statistical values with m=n=25 because of the exponential growth behavior of this algorithm. It was still going on after 2+ hours.

- Memoization Algorithm

The average cases are faster in memoization algorithm as well. As we can compare the table values, the average case behave not like quadratic, instead more like a linearistic behavior can be observed, compare to exact quadratic behavior of naive algorithm. For instance, in worst case when m=n=25, the code compiled in 0.007 seconds. However, in memoization algorithm it takes 0.00032 seconds to compile which is 21.875 times faster. Again, the count of recursive function calls decrease as the common characters can be found in average case. The behaviors can be also observed on the graphs.