Yasemin Özkut
28363

CS 301 - Assignment 1

Problem 1: Give an asymptotic tight bound for T(n) in each of the following recurrences. Assume that T(n) is constant for n ≤ 2. No explanation is needed.

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

From Master Theorem:
a=2, b=2, a ≥ 0, b ≥ 0, f(n) = n^3
$n^{\lg_b a} = n^{\lg_2 2} = n$ → (Case 3)
$f(n) = n^3 = \Omega(n^{\lg_2 2+\varepsilon})$
$a * f(n/b) \leq^{?} cf(n)$ → $2 * f(n/2) \leq^{?} cf(n)$ → $2 * (n/3)^3 \leq^{?} c * n^3$
$n^3/4 \leq cn^3$ → $1/4 \leq c < 1$
$T(n) = \Theta(f(n)) = \Theta(n^3)$

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

From Master Theorem:
a=7, b=2, a ≥ 0, b ≥ 0, $f(n) = n^2$
$n^{lg_b a} = n^{lg_2 7}$ → $n^{lg_2 7} > 2$ → (Case 1)
$f(n) = O(n^{lg_2 2-\varepsilon})$
$T(n) = \Theta(n^{lg_2 7})$

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

From Master Theorem:
a=2, b=2, a ≥ 0, b ≥ 0, $f(n) = \sqrt{n}$
$n^{lg_b a} = n^{lg_4 2} = n^{1/2} = \sqrt{n}$ → (Case 2)
$f(n) = \sqrt{n} = \Theta(n^{lg_b a}) = \Theta(n^{lg_4 2})$
$T(n) = \Theta(n^{lg_4 2} lgn)$

d) $T(n) = T(n-1) + n$ i=1
i=2 = (T(n-2) + n -1) + n = T(n-2) + 2n -1
i=3 = (T(n-3) + n -2) + n-1 + n = T(n-3) + 3n -3
i=4 = (T(n-4) + n -3) +n -2 + n-1 + n = T(n-4) + 4n -6
.
.
.
$=T(n-k) + nk - \sum_{i=1}^{k-1} i$

When k = n+2 → $T(2) + n(n+2) - (n+1)n/2$
$= T(2) + n^2 + 2n - (n^2 + n)/2$ → since T(2) is constant because (n=2)≤2, n^2 is the leading term
$T(n) = \Theta(n^2)$

Problem 2:

Figure 1:

```
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 2:

```
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]
```

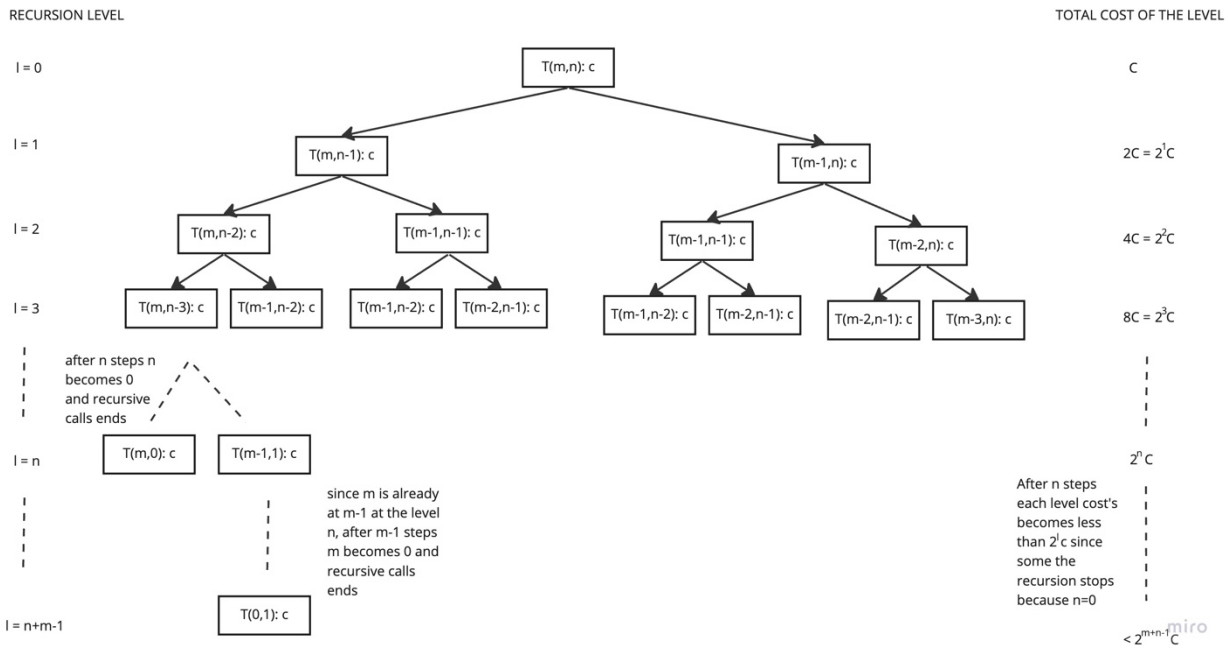
a) According to the cost model of Python, the cost of computing the length of a string using the function len is O(1), and the cost of finding the maximum of a list of k numbers using the function max is O(k). Based on this cost model:

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

If we look at the algorithm, we can see that there is only 2 if statements that includes recursion. First one is when the common letter found in both string and the second one is when the observed letters are not the same. When the common letter is found, the algorithm calls only one recursive call. When the 2 string of equal size has no common character, the worst-case running time occurs. When 2 of the string has no common character, the else part of the algorithm is called and it calls 2 recursions; one for keeping the letter stable in the first word and scan the other word by recursion and one for keeping the letter stable in the second word and scan the first word by recursion. If any of the letters are the same, only one recursion is called. So that every letter should be different for the worst-case. The lengths should also be equal for worst case because if one of them is less than the other, the amount of recursion calls for the shorter string will be less.

Time complexity → $T(m, n) = T(m, n-1) + T(m-1, n) + c$ → $T(m, n-1) + T(m-1, n)$ is coming from the recursive calls from the else part of the algorithm and constant c is coming from the checks that the algorithm make such as comparing the max of what 2

recursion returns which's complexity is given in the question as O(k)=O(2)=c or other if else statements which are also have constant complexity.



If we draw the recursion tree, we can see that after n^{th} level, n becomes 0 and the recursive calls end. Since m is already m-1 at the level n in the right node which can be seen in the above figure, after m-1 steps, m also becomes 0 and the recursive calls end. Until the level n, each level's cost is $2^{\ell}c$. However, after n^{th} level, each level's cost becomes less than is $2^{\ell}c$ since some of the nodes' recursive calls stops according to the algorithm. So, the tree grows to the middle and the nodes decreases at each level when the tree grows. So our total cost becomes $\leq c\sum_{i=0}^{m+n-1} 2^i$ which is equal to $(1-2^{m+n})/(1-2)$. So that our leading term becomes 2^{m+n}. So $T(m, n) = O(2^{m+n})$. In the worst-case n=m will be equal so $T(n,n) = O(2^{2n}) = O(4^n)$.

So, this will be our guess/claim for the substitution method.

Substitution method:
Guess: $T(m, n) = O(2^{m+n})$
Verification: We will use induction to prove our guess.

$$\exists c, n_0 \geq 0, m_0 \geq 0 \text{ such that } \forall n \geq n_0 \ \ \forall m \geq m_0: T(m, n) \leq c2^{m+n}$$

Induction Base: $T(1,1) \leq^{?} c*4$ → we can always find a bigger c to satisfy the equation.
Inductive Step: Assume that $T(k, r) \leq c2^{k+r}$ for all $k < m$, $r < n$ and $\max(m, n) = n$. (inductive hypothesis)

Since our max input size is m in our hypothesis, T(m-1, n) will take less time than T(m, n-1) in our equation right below. Since string size of m will be less than n

so there will be less recursive calls. So we do not need to calculate T(m-1, n) for our time complexity. So, our equation becomes;

$$T(m, n) = T(m, n-1) + c_1 \leq c*2^{m+n-1} + c_1$$
$$= c(2^{m+n})/2 + c1 \leq c*2^{m+n}$$
$$= c*2^{m+n} - (c(2^{m+n})/2 - c1) \leq c*2^{m+n}$$

In the equation our desired is $c*2^{m+n}$ and our residual is $(c(2^{m+n})/2 - c1)$. So we conclude that $T(n, n) = O(2^{n+m}) = O(2^{n+m})$.

→ In the worst case n=m. So, this makes our $T(n, n) = O(2^{n+n}) = O(4^n)$.

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

For Memoization Algorithm, also like in the Naive Algorithm, when the 2 string of equal size has no common character, the worst-case running time occurs because again the else part of the algorithm will be called and it will make 2 recursive calls; one for keeping the letter stable in the first word and scan the other word by recursion and one for keeping the letter stable in the second word and scan the first word by recursion. If any of the letters are the same, only one recursion is called. So that every letter should be different for the worst-case. The lengths should also be equal for worst case because if one of them is less than the other, the amount of recursion calls for the shorter string will be less.

However, different from the naive recursive algorithm, it's recursive calls would be less because this algorithm uses a matrix for not comparing the same indexes over and over again in the recursive calls. So, when the recursive step encounters the same index again, it will return the value from the matrix. This decreases the recursive calls. So actually, the recursive calls stop and comparisons of ifs and elif parts begins when the matrix is once full. The comparison operations have O(1) constant complexity. Because our matrix's size is (m+1)*(n+1), our $T(n) = O((m+1)*(n+1)) = O(mn+m+n+1) = O(mn)$ Since we said the worst case happen when the two strings are the same size, so m=n; $T(n, n) = O(n*n) = O(n^2)$.

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.

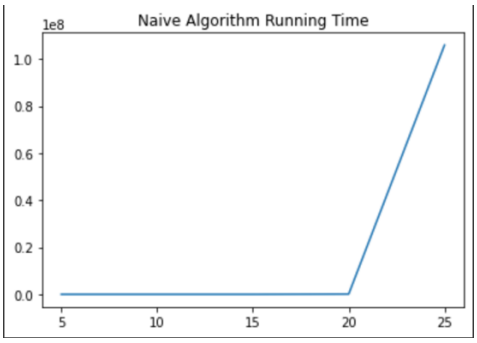
Algorithm	m=n= 5	m=n= 10	m=n= 15	m=n= 20	m=n= 25
Naive	0.00028s	0.122s	100.98s	Time out	Time out
Memoization	0.00012s	0.00025s	0.00045	0.00072s	0.00118s

CPU: M1 Pro 10-core
RAM: 16 GB
OS: MacOS
Compiled in Google Collaborate

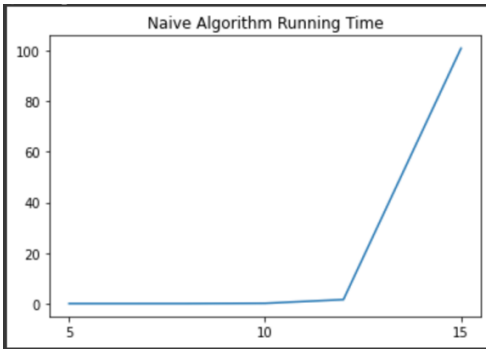
Since the time complexity is very big for m=n=20 and m=n=25 I got timeout. So, to draw the graph, I guessed the seconds for the m=n=20, and m=n=25. My guesses for m=n=20 is 103403.52s and m=n=25 is 105885204.48s.

(ii) Plot these experimental results in a graph.

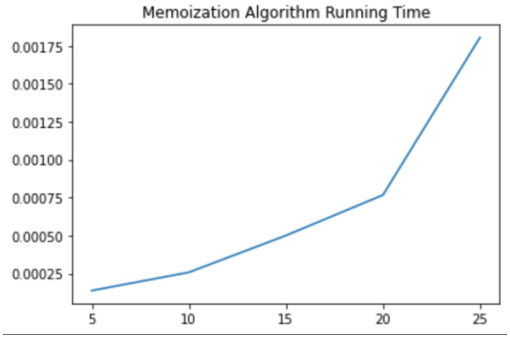
Graph for Naive Algorithm with n=m=5, n=m=10, n=m=15, and predictions for m=n=20 and m=n=25:



Graph for Naive Algorithm with m=n=5, m=n=8, m=n=10, m=n=12, m=n=15:



Graph for Memoization Algorithm:



(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)?

We can see that Naive Algorithm's graph is growing exponentially very fast after n=m=20 if we use predictictions. Because I could not find experimental results for n=m=30 and n=m=35 for Naive Algorithm, I wanted to draw a graph for less values than n=m=15 to see the growth, so I used m=n=5, m=n=8, m=n=10, m=n=12, m=n=15. It showed that the algorithm is growing exponentially and it grows very fast after n=m=12. In theory we have found out that the recursive function's time complexity as $O(4^n)$ in the worst-case which is also exponential.

We can see that Memoization Algorithm's graph is growing quadratically. In theory we have found out that the recursive function's time complexity as $O(n^2)$ in the worst case which is also exponential.

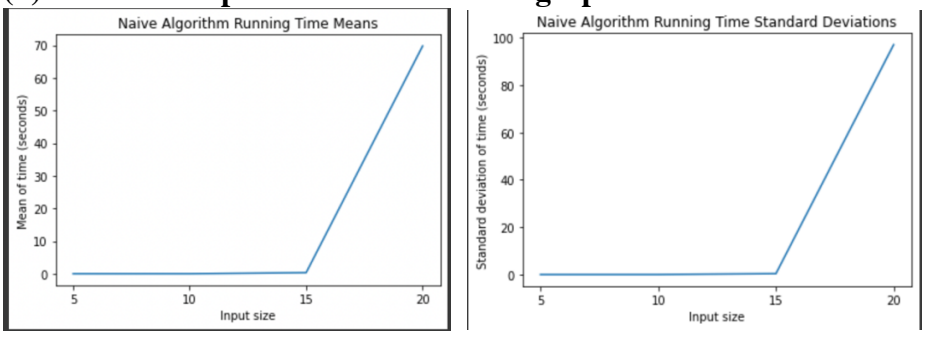
So, we can conclude that our theory is same as the experiment/practice.

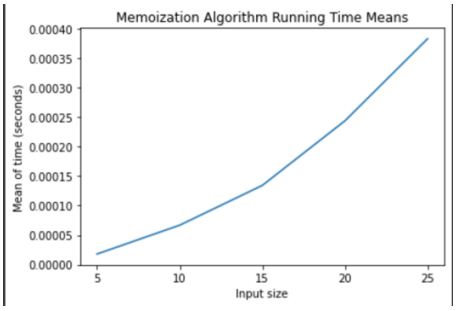
c) For each algorithm, determine its average running time experimentally by running it with randomly generated DNA sequences of length m = n. For each length 5, 10, 15, 20, 25, you can randomly generate 30 pairs of DNA sequences, using Sequence Manipulation Suite.

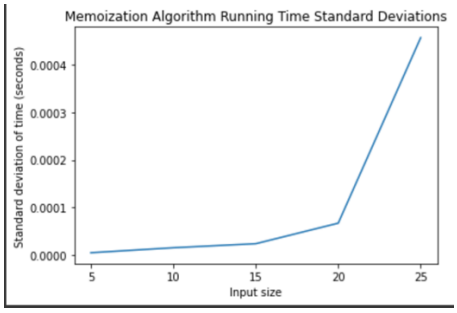
(i) Fill in following table with the average running times in seconds (μ), and the standard deviation (σ).

Algorithm	m=n= 5		m=n= 10		m=n= 15		m=n= 20		m=n= 25	
	μ	σ	μ	σ	μ	σ	μ	σ	μ	σ
Naive	5.37e-05	6.37e-05	0.0061	0.0073	0.330	0.365	69.74	97.08	Time out	Time out
Memoization	1.78e-05	4.8e-06	6.67-e05	1.52e-05	0.00013	2.37-e05	0.00024	6.70-e05	0.00038	0.00045

(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).

Similar to the worst case, Naive Algorithm still grows exponentially in average case and Memoization Algorithm still grows quadratically when we look at the graphs of for both mean and standard deviation. Even though I could not run n=25 for Naive algorithm because it took lots of time and caused time out, it still shows that the graphs grows exponentially very fast for Naive Algorithm.

However, if we look at both tables for average and worst cases, we can see that for the average case, running times are smaller than worst-case. While I could not run worst case scenario for n=20 in Naive Algorithm, I could run it at the average case scenario because it took less time than the worst case. I still could not run n=25 for Naive algorithm since the time complexity grows exponentially and it caused time out even for the average case.