CS301 Assignment-1

Hasan Berkay Kürkçü - 27851

October 23, 2022

Problem 1 - Recurrences

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

Answer: $a = 2 \geq 1$, $b = 2 > 1$ and $f(n) = n^3$ is asymptotically positive function.
Hence, Master Theorem can be applied.
$n^{log_b a} = n^{log_2 2}$ Then, $f(n) = n^3 = \Omega(n^{log_2 2+\varepsilon})$ for some $\varepsilon > 0$ and $2f(n/2) = 2(n/2)^3 = n^3/4 \leq cn^3$ for some $c < 1$ and $c >= 1/4$ So, case 3 applies.
$T(n) = \Theta(f(n)) = \Theta(n^3)$

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

Answer: $a = 7 \geq 1$, $b = 2 > 1$ and $f(n) = n^2$ is asymptotically positive function.
Hence, Master Theorem can be applied.
$n^{log_b a} = n^{log_2 7}$ Then, $f(n) = n^2 = O(n^{log_2 7-\varepsilon})$ for some $\varepsilon > 0$ and case 1 applies. $T(n) = \Theta(n^{log_b a}) = \Theta(n^{log_2 7})$

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

Answer: $a = 2 \geq 1$, $b = 4 > 1$ and $f(n) = \sqrt{n}$ is asymptotically positive function.
Hence, Master Theorem can be applied.
$n^{log_b a} = n^{log_4 2}$ Then, $f(n) = \sqrt{n} = \Theta(n^{log_4 2})$ and case 2 applies. $T(n) = \Theta(f(n)) = \Theta(n^{log_b a} lgn) = \Theta(\sqrt{n} lgn)$

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

Answer: Here master theorem can not be used since b is not bigger than 1. That is why substitution method will be used.
First, Upper Bound: claiming that T(n) = O(n^2). Then, $\exists$ c, $n_0 \geq 0$ such that $\forall$ n $\geq n_0$: T(n) $\leq$ c n^2.
Base case: T(1) $\leq$ c1^2 $\leq$ c we can always pick c big enough.
Inductive step: Assume T(k) $\leq ck^2$ for all k < n and will show that T(n) $\leq cn^2$

T(n) = T(n - 1) + n $\leq$ (I.H) c$(n-1)^2 + n$ = cn^2 -2cn + c + n = cn^2 - (2cn - c - n)
We need to find term we need (cn^2) and minus something non-negative.
(2cn - c - n) is non-negative if n $\geq$ 1 and c $\geq$ 1. Thus, T(n) $\leq$ cn^2 and O(n^2)

Now Lower Bound: claiming that T(n) = $\Omega(n^2)$. Then, $\exists$ c, $n_0 \geq 0$ such that $\forall$ n $\geq n_0$: T(n) $\geq$ c n^2.
Base case: T(1) $\geq$ c1^2 $\geq$ c Again, we can always find c.
Inductive step: Assume T(k) $\geq$ ck^2 for all k < n and will show that T(n) $\geq$ cn^2
T(n) = T(n - 1) + n $\geq$ (I.H) c$(n-1)^2 + n$ = cn^2 -2cn + c + n = cn^2 + (c + n - 2cn)
We need to find term we need (cn^2) and plus something non-negative.
(c + n - 2cn) is non-negative if n $\geq$ 1 and 0 $\leq$ c $\leq$ 1/2. Thus, T(n) $\geq$ cn^2 and $\Omega(n^2)$
Now both upper and lower bounds are shown and by definition T(n) = $\Theta(n^2)$

Problem 2 - Longest Common Subsequence - Python

(a)(i) If we review naive recursion code, when there is a letter match function decrements both indexes by 1 and makes only 1 recursive call, whereas in else part it make 2 recursive calls. Then, worst case happens when there is no any common character or we can say common subsequence in given two strings and up until i == 0 or j == 0 code goes else part. Then, we can write recurrence relation as = $T(m,n) = T(m-1,n) + T(m,n-1) + \Theta(2)$ since finding max of two number is $\Theta(2)$. We can show that recurrence by recursion tree like that with calling $\Theta(2)$ as c to make computations easier (I prepared tree via powerpoint):

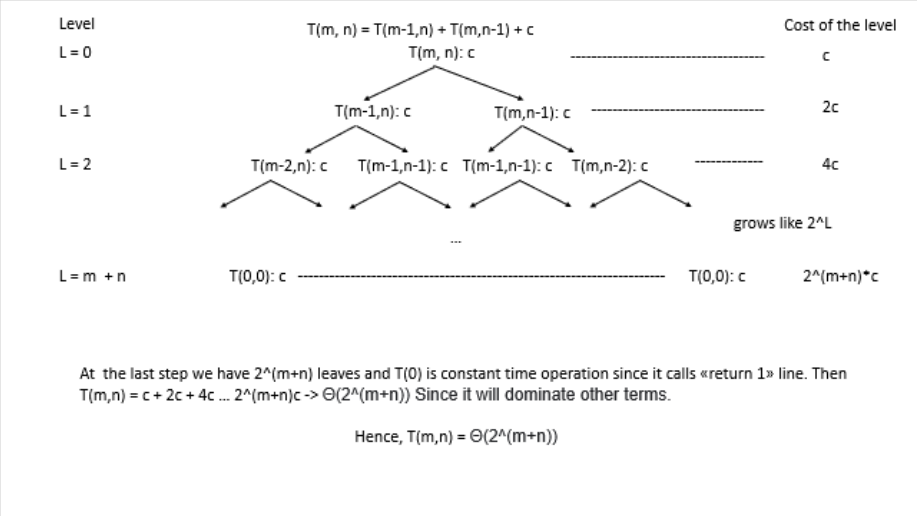

Figure 1: Recursion Tree

(a)(ii) *Worst-case running time of the recursive algorithm with memoization*
Explanation: Logically, similar to naive recursion worst case happens when given two strings do not have any single character in common or we can say they do not have common subsequence (such as abcde fghil etc.) because when this happens for every index code calls two recursive functions in the else part until it encounters 0th index in m or n. However, there is a major difference which is we keep record of previously computed solutions (by indexes). Think of a situation where program comes to i = 3, j = 7 for the first time and it computes the result of this subproblem by making recursive calls (during the process again smaller problems will be filled in the table starting from the time when one index becomes 0). Now, Whenever code comes to that indexes again it will just take O(1) (constant) time to return answer because it is in the table already. As the input sizes get larger memoization affect become more drastic since code will save a lot of time instead of calculating same sub-problem again and again. So, as a result we can say that time complexity will be equal to number of subproblems which is actually size of the table, then it can be said that T(n) = $O((m+1)*(n+1))$. So making that small addition to naive recursion will gain a lot of time.

(b)(i) Properties of the machine: Cpu with 3,80 GHZ, 16 GB Ram and OS as Win 10 Pro.
In all different length inputs, strings is selected to not include any common character to ensure it is worst case (program always goes to else part at the very below). For instance, for m = n = 5 abcde and opuyn are chosen etc.
Time table is below:

Important note: I have waited more than 3 hours and half for m = n = 20 case and program did not terminated. That is why input sizes 20 and 25 are unknown.

Algorithm	m = n = 5	m = n = 10	m = n = 15	m = n = 20	m = n = 25
Naive	0.001878738	0.125522851	103.982340574	?	?
Memoziation	0.000266790	0.000756263	0.001479148	0.002488613	0.003384828

Table 1: Time of different inputs

(b)(ii)

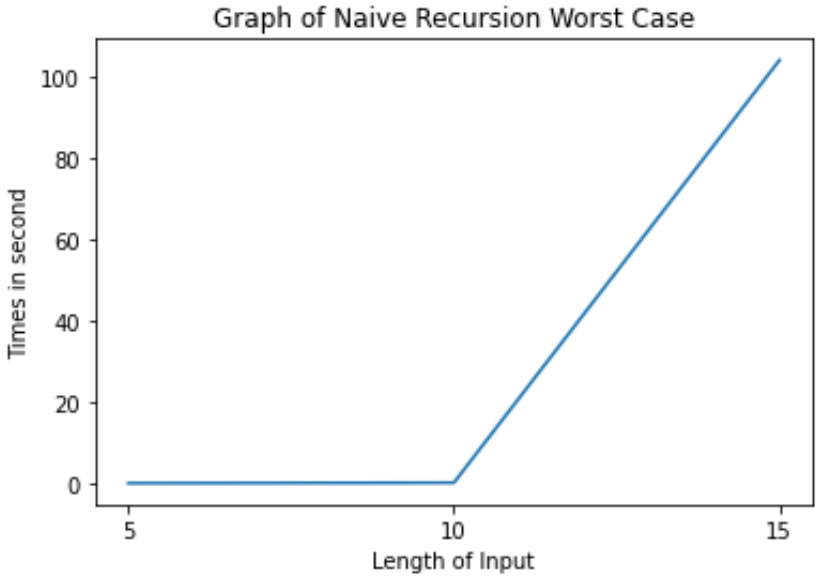

Figure 2: Naive Recursion Graph.

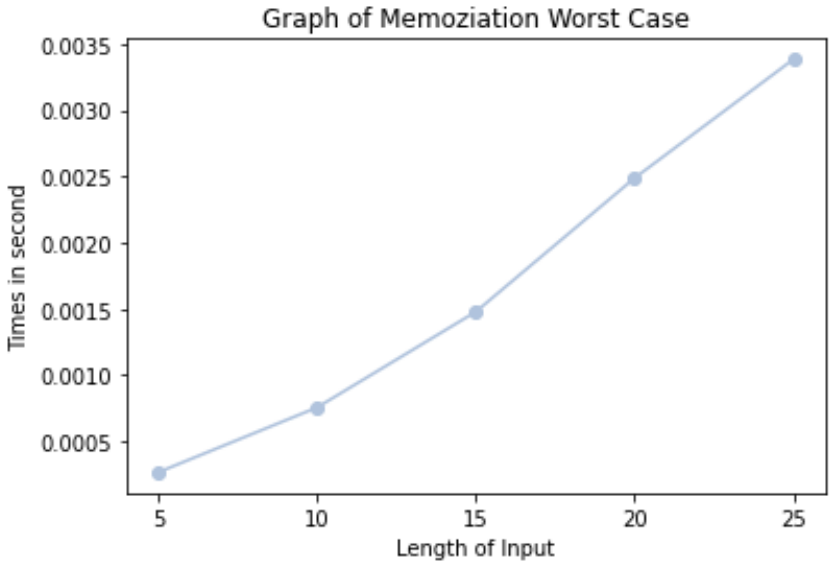

Figure 3: Memoziation Graph.

(b)(iii) So, let's define scalability of algorithm like ability of algorithm to maintain its efficiency while input sizes get larger. Then, we can say that naive recursion code is far from being scalable. If you review the graph above just by going from 10 to 15 we got drastic increase in time. Moreover, we were not even able to measure input sizes 20 (I waited more than 3 hours and 30 minutes) and sizes 25 which means even going from 15 to 20 increases time from 90 seconds to at least 12600 seconds which would go (considering exponential complexity) probably much beyond that number if i continued to measure. Also we can say that these are not huge numbers considering in real life, DNA sequences with much longer length may be the case. I tried size 10, size 11 and size 12. Results were 0.12 0.44 1.74 which actually demonstrates $\Theta(2^{m+n})$ complexity we increase both parameter by 1 which multiplies time by 4 that is parallel with theoretical part. When it comes to memoziation, we see much more scalable algorithm considering even difference between sizes 5 and sizes 25 is not dramatic (0.00026 to 0.0033). This stems from filling table as explained above and as sizes increase times do not scale rapidly because table saves from going through path again and it just takes O(1) time. Hence, it can be said that theoretical calculations comply with the measurements. For memoization we use same lengths then it should be look like second degree function graph (n^2) if we had more samples we would probably see that result, even with small input sizes we see something similar to (n^2) again parallel with theoretical part.

(c)(i) Sample Runs created with Sequence Manipulation Suite are used with the same computer in worst-case.

Note about average measures below: This time only size 25 of naive recursion is unknown. Considering size 20 took 30 minutes, size 25 will take extremely long time in naive part.

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	0.0000827	0.0000620	0.0052581	0.0077208	0.73849	1.14628	68.77036	78.84960	-	-
Memo	0.0000020	0.0000060	0.0000055	0.0000247	0.0000123	0.0000607	0.0000150	0.0000762	0.0000258	0.0001337

(c)(ii)

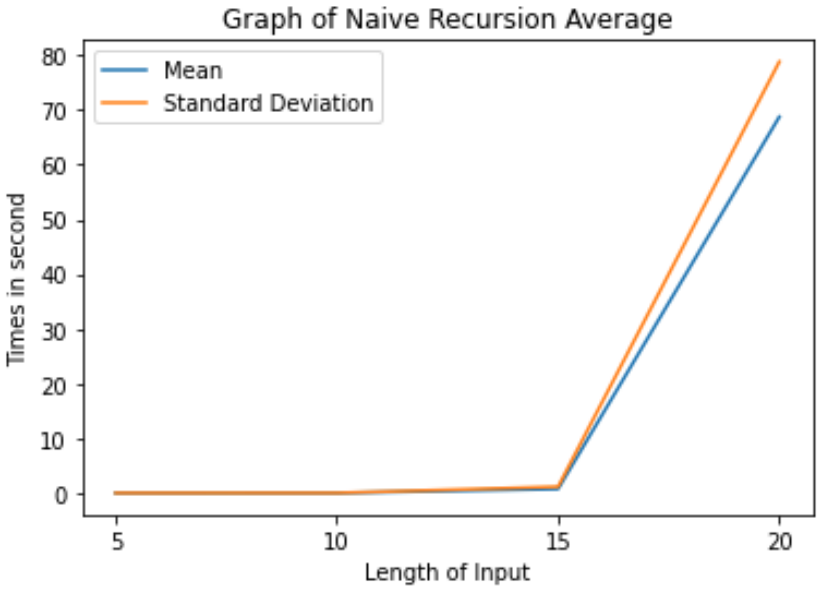

Figure 4: Naive Recursion Average

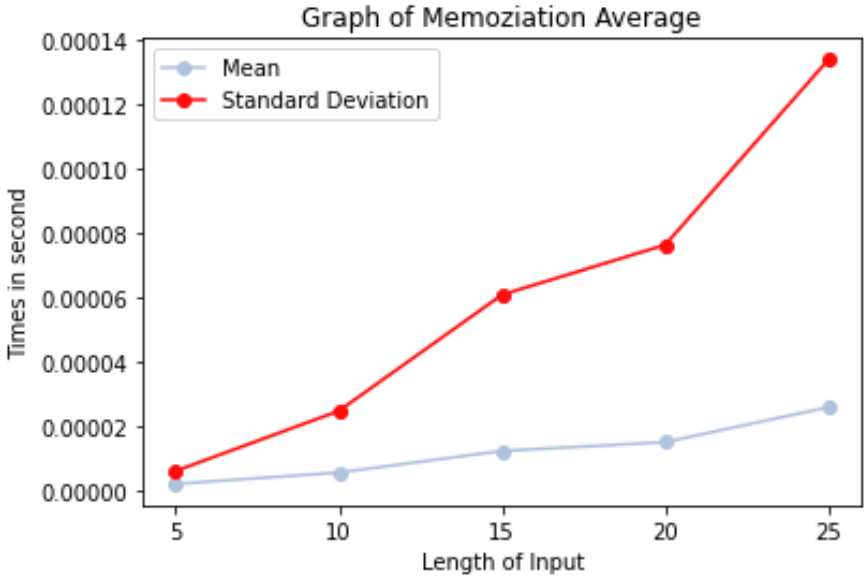

Figure 5: Memoization Average

(c)(iii) For the naive recursion, we see better scalability compared to worst case because first thing we notice is here, we have only four unique letters in the strings (a,t,c,g) and they are randomly generated. So, we will have some instances (may be many instances) where program encounters same letter and calls only one recursion before going into else part which will make time shorter. Also, in worst case for the input size 20, 3 hours and 30 minutes were not sufficient to get it only 1 times, here we can get the input for 30 times again for the same reason explained in first sentence. However, it is also important that again we don't have really good scalability because still size 25 is not measured in reasonable time (which is not high number for strings containing only 4 different characters). This is bad news because in real data sets we obviously do not get worst case always, average case is more realistic but again we do not get great results and still exponential. Think of a situation where strings with length more than 300 used, without high-performance computers it will last for days. For the memoization, same reason in naive case applies (having 4 unique chars and sometimes code calls one recursion at some instances) and we get better results compared to the worst case. Like worst case of memoization we have second degree function graph which is a bit ambiguous since sample is not large enough and we tried with small sizes but shape of the graph is similar to (n^2) and we can say that quadratic shape can be seen better here compared to worst case of memoization because as discussed in the lecture, considering central limit here we tested function 30 times for each size that gave us more realistic and consistent results.