$CS301_h w1$

Mustafa Yağız Kılıçarslan

October 2022

1 Recurrences (10 points)

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.

Note that according to Master Method:
if recurrence is in the form of $T(n) = aT(n/b) + f(n)$ where $a, b \geq 1$ and $f(n)$ is asymptotically positive, then:

Case 1: *if* $f(n) = O(n^{\lg_b a - \epsilon})$ for some $\epsilon > 0$ *then* $T(n) = \Theta(n^{\lg_b a})$

Case 2: *if* $f(n) = \Theta(n^{\lg_b a})$ *then* $T(n) = \Theta(n^{\lg_b a} \lg n)$

Case 3: *if* $f(n) = \Omega(n^{\lg_b a + \epsilon})$ for some $\epsilon > 0$ and if $cf(n) \geq af(n/b)$ for some $c < 1$ and for large n *then* $T(n) = \Theta(f(n))$

a. $T(n) = 2T(n/2) + n^3$
   Using Master Theorem **Case 3**:
   $a = b = 2 \geq 1$, $f(n) = n^3$ asymptotically positive
   $f(n) = n^3 = \Omega(n^{\lg_2 2 + \epsilon}) = \Omega(n^{\epsilon})$, $\epsilon > 3$
   Check: $2(n/2)^3 = n^3/4 \leq cn^3$ for $c = 1/4 < 1$ and for large n
   In this case we can apply Case 3:
   $T(n) = \Theta(f(n)) = \Theta(n^3)$

b. $T(n) = 7T(n/2) + n^2$ Using Master Theorem **Case 1**:
   $a = 7 \geq 1$, $b = 2 \geq 1$, $f(n) = n^2$ asymptotically positive
   $f(n) = n^2 = O(n^{\lg_2 7 - \epsilon})$, $\epsilon > 0$
   We can apply Case 1:
   $T(n) = \Theta(n^{\lg_b a}) = \Theta(n^{\lg_2 7})$

c. $T(n) = 2T(n/4) + \sqrt{n}$
   Using Master Theorem **Case 2**:
   $a = 2 \geq 1$, $b = 4 \geq 1$, $f(n) = \sqrt{n}$ asymptotically positive

$n^{\lg_b a} = n^{\lg_4 2} = \sqrt{n}$
$f(n) = \sqrt{n} = \Theta(n^{\lg_4 2}) = \Theta(\sqrt{n})$
We can apply Case 2:
$T(n) = \Theta(n^{\lg_b a} \lg n) = \Theta(\sqrt{n} \lg n)$

d. $T(n) = T(n-1) + n$
Prove that $T(n) = \Theta(n^2)$

d.1 Prove $T(n) = O(n^2)$ by induction:
Assume that $T(k) \leq ck^2$ for some $c > 0$, for all $k < n$ holds.
Inductive Step:

$$T(n) \leq c(n-1)^2 + n$$

$$T(n) \leq cn^2 - 2cn + c + n$$

$$T(n) \leq cn^2 - (2cn - c - n)$$

$(2cn - c - n) > 0$ for $c > 1/2$ and $n_0 < \frac{c}{2c-1}$
Therefore, $T(n) \leq cn^2$
$T(n) = O(n^2)$ for mentioned c and n_0 values.

d.2 Prove $T(n) = \Omega(n^2)$ by induction:
Assume that $T(k) \geq ck^2$ for some $c > 0$, for all $k < n$ holds.
Inductive Step:

$$T(n) \geq c(n-1)^2 + n$$

$$T(n) \geq cn^2 - 2cn + c + n$$

$$T(n) \geq cn^2 + (n + c - 2cn)$$

$(2cn - c - n) > 0$ for $c < 1/2$ and $n_0 > \frac{c}{2c-1}$
Therefore, $T(n) \geq cn^2$
$T(n) = \Omega(n^2)$ for mentioned c and n_0 values.

Since we can show that: $T(n) = O(n^2)$ and $T(n) = \Omega(n^2)$
$T(n) = \Theta(n^2)$

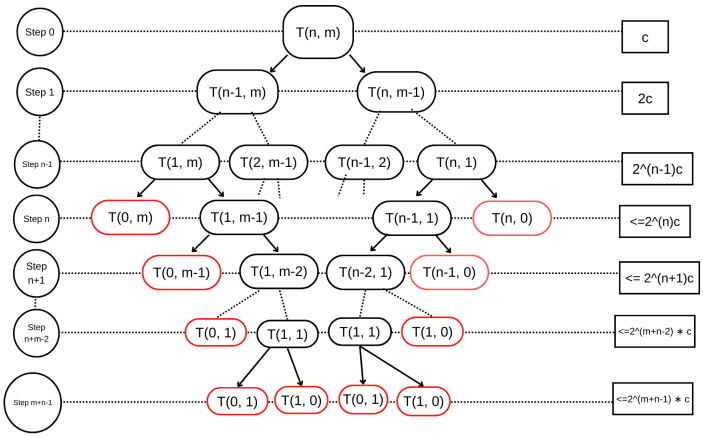
2 (Longest Common Subsequence - Python)

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.

In naive algorithm, worst-case happens when there are no common subsequence between two strings, for example *abcd* and *efgh*. In that case, recursion becomes $T(n,m) = T(n-1,m) + T(n,m-1) + \Theta(1)$ MAX is $\Theta(1)$ since it is $O(k)$ where $k = 2$, thus it is $\Theta(1)$.

If we the draw the recursion tree, we will see something similar to this:



If we sum the costs denoted in the right column we get:

$$T(n,m) \leq \sum_{k=0}^{n+m-1} 2^k + \Theta(1)$$

from geometric sum,

$$T(n,m) \leq \frac{2^{n+m} - 1}{2 - 1} + \Theta(1)$$

$$T(n,m) \leq \frac{1}{2} 2^{n+m} + \Theta(1)$$

From the insight of recursion tree, we can use $T(n) = O(2^{n+m})$ as a guess.

To prove $T(n,m) = O(2^{n+m})$, substitute $l = m + n$
So that, $T(n,m) = T(n-1,m) + T(n,m-1) + \Theta(1)$ becomes:
$T(l) = 2T(l-1) + \Theta(1)$.
Now we will prove that $T(l) = O(2^l)$

Prove by induction:
Inductive Step: Assume that $T(k) \leq c_1 2^k - c_2 k$ for all some $c_1, c_2 > 0$

and all $k < l$ holds.

$$T(l) \leq 2(c_1 2^{l-1} - c_2(l-1)) + \Theta(1)$$

$$T(l) \leq c_1 2^l - c_2 l - [c_2 l - 2c_2 - \Theta(1)]$$

For sufficiently large l, $[c_2 l - 2c_2 - \Theta(1)] > 0$,
Since we have $T(l) \leq$ (*what we want - something non-negative*)
we can say that $T(l) = O(2^l)$

Thus, if we substitute (n,m) back, $T(n, m) = O(2^{n+m})$.

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, worst-case happens when there is no common subseqence between two strings. Similarly to Naive algorithm, it enter two recursions because there is no common subseqence, however, two-d array (c in the implementation) stores values new calculations and when the a previously known calculation is to be executed again, algorithm uses the value previous calculation.
From the same recursion tree provided above, we can see that we calculate the same subproblems over and over. Since we have n and m decreasing by at most 1 for each recursive call, we can say that the size of possible subproblems are $(n+1)(m+1)$. Other calls basically ask the calculation of same subproblems. When the same subproblem is to be calculated, algorithm simply looks the value of it from the table. Look-up takes $\Theta(1)$ or constant time.

Thus, worst-case time complexity is $\Theta((n+1) * (m+1)) = \Theta(n * m)$.

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.

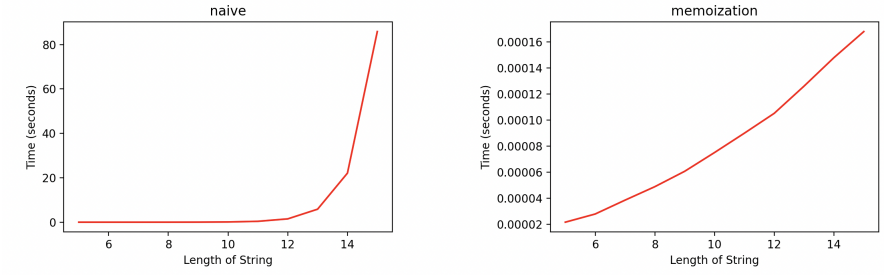
Machine Properties:
- Apple M1
- 16GB RAM
- MacOS Monterey version 12.5.1

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	0.000167s	0.10206s	85.712s	87777, 28s	89883934, 72s
Memoization	$2.204e-05$s	$7.4208e-05$s	0.0001718s	0, 005497s	0, 1759s

Since running the program for inputs $n = m = 20$ and $n = m = 25$ took infeasible amount of time, values of them are guessed based on the previous results. I ran the algorithm for inputs of 1 to 15. For each incremented input size, naive run time quadrupled (4^n) and memoization run time looks like n^2, therefore I guessed the non-found values according to those results.

ii. Plot these experimental results in a graph.



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)?

Results of the implementation suggest that:
- For naive algorithm, with each increment in input size, run-time became by 4^n.
- For memoization algorithm, with each increment in input size, run-time became by n^2.
Therefore, memoization is much more scalable.

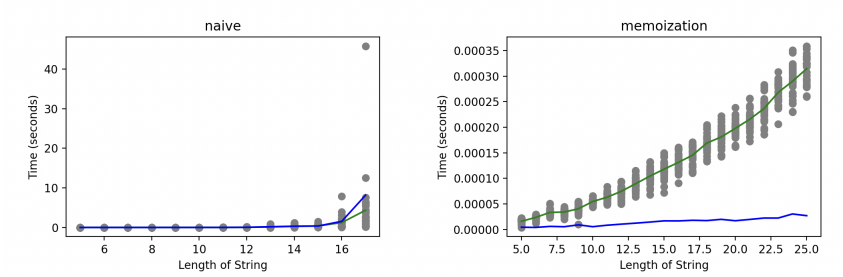
Experiments confirm the theoretical results because:
- For Naive time complexity we suggested: $\Theta(2^{n+m})$, if we assume $n = m$, then it becomes $\Theta(2^{2n}) = \Theta(4^n)$ which is supported by our experimental results.
- For Memoization time complexity we suggested: $\Theta(m * n)$, if we assume $n = m$, then it becomes $\Theta(n * n) = \Theta(n^2)$ which is supported by our experimental results.

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).

2*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	$4.47e-05$s	$2.99e-05$s	0.00477s	0.0033s	0.402s	0.356s	-	-	-	-
Memoization	$1.6e-05$s	$4.5e-06$s	$5.48e-05$s	$5.5e-06$s	0.00011s	$1.69e-05$s	0.00019s	$1.721e-5$s	0.000314s	$2.7119e-5$s

ii. Plot these experimental results in a graph.



Legend:
- Grey dots are the individual cases.
- Green line is the mean
- Blue line is the standard deviation.

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

- Naive: According to results, growth is still exponential, but since the average test case does not always generate 2 recursive calls, the overall running time is drastically smaller compared to the worst-case.
- Memoization: Due to the same reasoning mentioned in Naive algorithm, the running time is also much smaller compared to worst-case scenario, but the growth rate is still quadratic.