CS301-HW1

Ahmet Furkan Ün

October 15, 2023

Problem 1: Recurrences

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

By Master Theorem, $a = 2, b = 2, f(n) = n^3$.
So, $f(n) = \Theta(n^{\log_2 2+\epsilon})$ where $\epsilon = 2$ and $\epsilon > 0$
Therefore $T(n) = \Theta\ (f(n)) = \Theta(n^3)$

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

By Master Theorem, $a = 7, b = 2, f(n) = n^2$.
So, $f(n) = \Theta(n^{\log_2 7-\epsilon})$ where $\epsilon > 0$
Therefore $T(n) = \Theta(n^{\log_2 7})$

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

By Master Theorem, $a = 2, b = 4, f(n) = \sqrt{n}$.
So, $f(n) = \Theta(n^{\log_4 2}) = \sqrt{n}$
Therefore $T(n) = \Theta(n^{\log_4 2} * \log n) = \Theta(\sqrt{n} * \log n)$

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

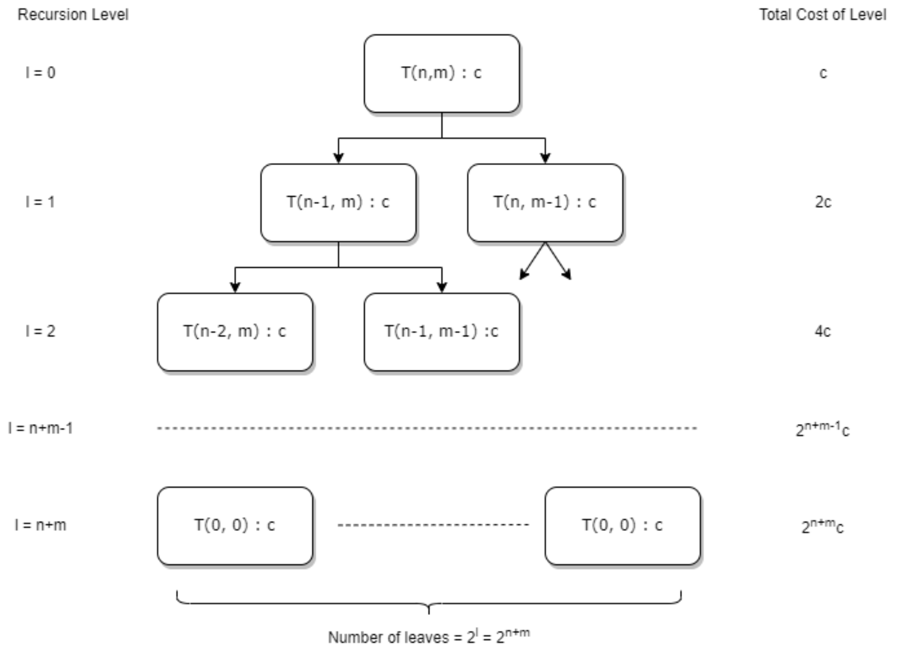
By recursion tree, which is a straight line, we can see that for every level the cost is n and the max level of the tree is n. So there are n levels each having n cost. Therefore $T(n) = n * n = n^2$
$T(n) = \Theta(n^2)$

Problem 2: Longest Common Subsequence - Python

(a)

(i)

The worst case occurs when two text are completely different. So the maximum number of function calling itself can be reached by this case. Let n and m be the length of the strings. The max operator at the end will have a constant time since it always compares 2 integers. Let the constant time for the max operator be c. For simplicity assume n = m
$T(n, m) = T(n-1, m) + T(n, m-1) + c$
$T(n, m) = c + 2c + 4c + ... + 2^{n+m+1}c + 2^{n+m}c$
$T(n, m) = \Theta(2^{n+m})$



For simplicity assume n = m
$T(n) = \Theta(2^{2n})$

(ii)

In the scenario where there is no common subsequence, the worst case occurs. At each iteration, the function examines the matrix array c to determine whether a position has already been checked. If a position has been checked, there is no need to recompute it. The size of the matrix is $(n+1)*(m+1)$ where n and m are the length of the strings similar to part i, and computation is required that many times, resulting in $nm+n+m+1$ distinct functions.

Expressing the time complexity as $T(n,m) = c(nm+n+m+1)$, the best asymptotic worst-case running time is $\Theta(nm)$. For simplicity if we assume $n = m$, resulting in a worst-case running time of $\Theta(n^2)$

(b)

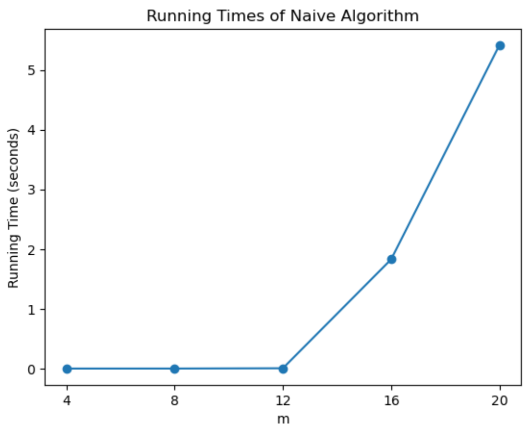
(i)

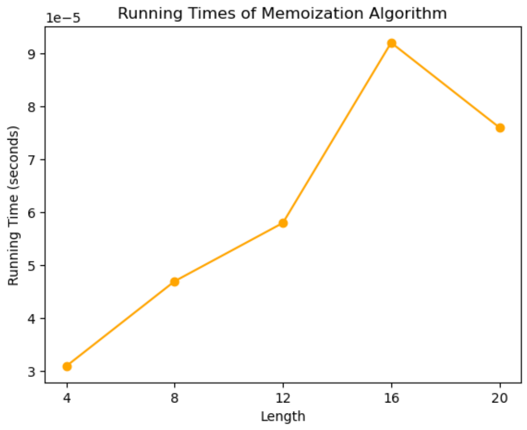
Computer Properties: Windows 11, i7-1165G7 2.80GHz, 16GB RAM

Table 1: Table of two algorithms in the worst case (running times in seconds)

Algorithm	m = 4	m = 8	m = 12	m = 16	m = 20
Naive	0.000020	0.000265	0.005002	1.827986	5.414081
Memoization	0.000031	0.000047	0.000058	0.000092	0.000076

(ii)





(iii)

The theoretical result for Naive Algorithm $T(n) = \Theta(2^{2n})$ indicates exponential growth, and this is consistent with the experimental results. The running time increases exponentially with the input size.

The theoretical result for Memoization Algorithm $T(n) = \Theta(n^2)$ suggests a quadratic growth rate. The experimental results confirm this trend, as the running time increases at a much slower rate compared to the naive algorithm. However there is an observation that the running time for the memoization algorithm is lower for $m = 20$ compared to $m = 16$. This behavior is unexpected given the typical growth pattern associated with the $\Theta(n^2)$ time complexity. This discrepancy might be attributed to specific characteristics of the input affecting the time.

(c)

(i)

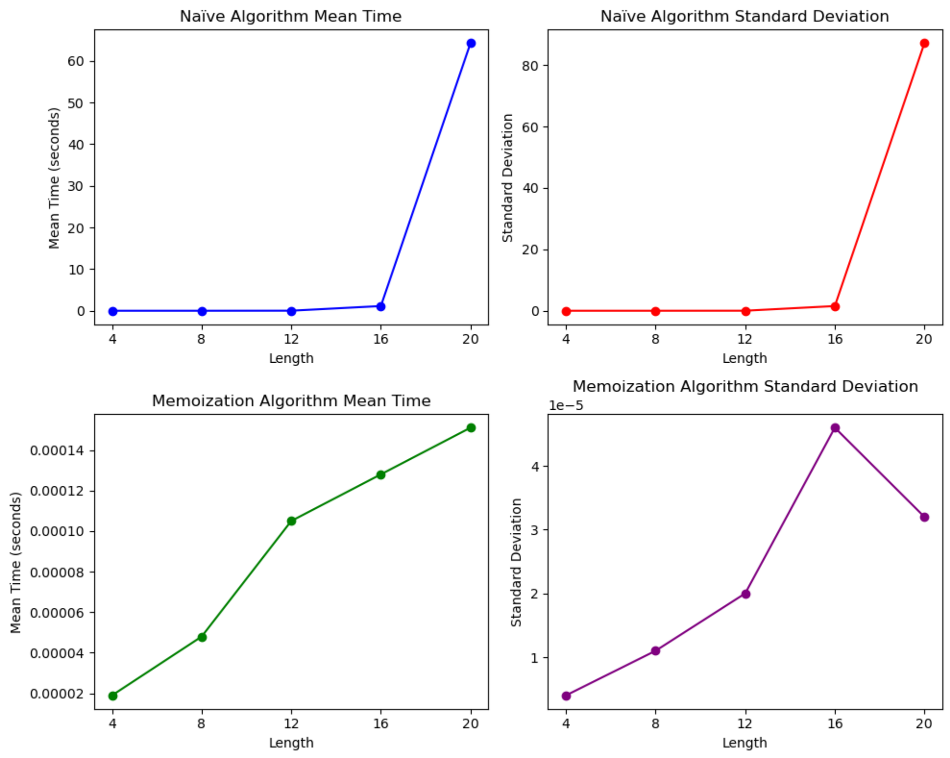
Table 2: Algorithm Running Times for Lengths 4, 8, 12

Algorithm	m = 4		m = 8		m = 12	
	μ	σ	μ	σ	μ	σ
Naive	0.000017	0.000011	0.000688	0.000852	0.015260	0.014498
Memoization	0.000019	0.000004	0.000048	0.000011	0.000105	0.000020

Table 3: Algorithm Running Times for Lengths 16, 20

Algorithm	m = 16		m = 20	
	μ	σ	μ	σ
Naive	1.134030	1.535854	64.294242	87.169168
Memoization	0.000128	0.000046	0.000151	0.000032

(ii)



(iii)

Experimental results provide valuable insight into the practical performance of naive and memorization algorithms for the given problem. In the worst case scenario, as noted in the first analysis (b), the Naive algorithm shows a clear exponential growth in running times as the input size increases. This is consistent with the theoretical expectation of $\Theta(2^{2n})$. The trend observed in the new results, where average running times increase exponentially with increasing input size, confirms the theoretical worst-case analysis.

Conversely, the Memoization algorithm with a better theoretical worst-case time complexity such as $\Theta(n^2)$ exhibits a more controlled growth in average running times as the input size increases. Test result suggests growth that appears to be very close to linear, demonstrating the efficiency gains of note taking over the Naive approach. This observation supports the theoretical analysis that the Memoization algorithm should perform better for larger input sizes, and the experimental results provide concrete evidence of this improvement.

The comparison between the two algorithms underscores the importance of algorithmic efficiency in addressing larger problem instances. The Memoization algorithm consistently outperforms the Naive algorithm across all input sizes tested, confirming theoretical expectations.