Corey Lang

Index	Algorithm Used	n	Value	Timer (ns)
0	fibonacci_recursive	2	1	56800
1	fibonacci_recursive	7	13	35700
2	fibonacci_recursive	30	832040	1613055500
3	fibonacci_recursive	10	55	95000
4	fibonacci_recursive	0	0	900
5	fibonacci_recursive	8	21	36000
6	fibonacci_recursive	40	102334155	199890653000
7	fibonacci_top_down	2	1	20600
8	fibonacci_top_down	7	13	10800
9	fibonacci_top_down	30	832040	38400
10	fibonacci_top_down	10	55	900
11	fibonacci_top_down	0	0	1000
12	fibonacci_top_down	8	21	800
13	fibonacci_top_down	40	102334155	15700
14	fibonacci_bottom_up	2	1	22600
15	fibonacci_bottom_up	7	13	3900
16	fibonacci_bottom_up	30	832040	11800
17	fibonacci_bottom_up	10	55	4300
18	fibonacci_bottom_up	0	0	1100
19	fibonacci_bottom_up	8	21	3600
20	fibonacci_bottom_up	40	102334155	19600

The first table is the results from a run using inputs [2,7,30,10,0,8,40], on three different versions of solving the fibonacci sequence of a number.
(1) recursive version
(2) memorized top-down version
(3) bottom-up version

Index	Algorithm Used	Matrix Dimensions	Minimum number of multiplications	Optimal parenthesization	Timer (ns)
0	MatrixChainOrder_recursive	[30, 35, 15, 5, 10, 20, 25]	15125	((A1(A2A3))((A4A5)A6))	700500
1	MatrixChainOrder_memo	[30, 35, 15, 5, 10, 20, 25]	15125	((A1(A2A3))((A4A5)A6))	346200
2	MatrixChainOrder_bottom	[30, 35, 15, 5, 10, 20, 25]	15125	((A1(A2A3))((A4A5)A6))	250700
3	MatrixChainOrder_recursive	[2, 3, 10, 5, 4, 20]	360	((((A1A2)A3)A4)A5)	113200
4	MatrixChainOrder_memo	[2, 3, 10, 5, 4, 20]	360	((((A1A2)A3)A4)A5)	99800
5	MatrixChainOrder_bottom	[2, 3, 10, 5, 4, 20]	360	((((A1A2)A3)A4)A5)	60700

The second table is the results from a run using inputs [30, 35, 15, 5, 10, 20, 25] and [2, 3, 10, 5, 4, 20] as matrix dimensions for three different versions of the matrix multiplier order function.
(1) recursive version
(2) memorized top-down version
(3) bottom-up version

Corey Lang

Project 2 – Dynamic Programming

Implementation and Experimental Analysis

Dynamic programming is a powerful algorithmic technique used to solve complex problems efficiently. It is interesting and useful because it breaks down a problem into smaller subproblems, solves each subproblem only once, and stores the solutions to avoid redundant computations. In this paper, I will discuss the results of applying dynamic programming to solve two fundamental problems in computer science and mathematics: matrix multiplication and the Fibonacci sequence. I will also compare the run times of a recursive and bottom up approaches to these problems against the results of memoized dynamic programming approach. The variable used to test the performance of the functions is a timer function between each call of the functions.

The Fibonacci sequence is a sequence of numbers where each number is the sum of the previous two numbers. Dynamic programming can be applied to this problem by storing the solutions to smaller subproblems and using them to solve larger subproblems. The results showed that the top-down dynamic programming algorithm was the most efficient algorithm for solving the Fibonacci sequence for most values of n, while the bottom-up algorithm may be more suitable for some values of n. The recursive algorithm was the slowest of the three algorithms, with a very high time complexity. These results were expected, as the bottom-up dynamic programming algorithm solves each subproblem only once, while the top-down algorithm may solve some subproblems multiple times which will be quicker than recursive functions. However, in my results the memorized top-down approach performed better in terms of shortest runtime for most of the fibonacci problems but not all. The bottom up approach was faster in the cases where n was equal to seven and thirty. This was surprising to me and these results remained across different runs. An anomaly was observed in the timing results for the Fibonacci sequence, where the time complexity of the recursive algorithm was unexpectedly high for certain values of n. This can be explained by the fact that the recursive algorithm makes a large number of redundant calculations, which results in very high time complexity for larger values of n.

The matrix multiplication problem involves finding the most efficient way to multiply a chain of matrices. Dynamic programming is particularly useful for this problem because it avoids redundant calculations by solving each subproblem only once and storing the solutions. The results showed that the bottom-up dynamic programming algorithm was the most efficient algorithm for solving the matrix multiplication problem, with the lowest time complexity. The top-down dynamic programming with memoization algorithm was faster than the recursive algorithm but still slower than the bottom-up

dynamic programming algorithm. The recursive algorithm was the slowest of the three algorithms, with a very high time complexity. These results were expected, as the bottom-up dynamic programming algorithm builds up the solution from the bottom and solves each subproblem only once, while the top-down and recursive algorithms may solve the same subproblem multiple times.