**Project 2**

**Dynamic programming implementation and experimental analysis**

**Introduction:**

Dynamic Programming is an optimalization approach for solving a problem by recursively breaking it down into smaller sub-problems, where the optimal solution to the overall problem depends upon the optimal solution to its individual sub-problems.

**Why is dynamic programming interesting/useful?**

Dynamic Programming algorithm solves each sub-problem just once, and remembers its answer, thereby, avoiding re-computation of the answer for similar sub-problem every time. Thus, it reduces the time complexity.

The implementation of the three versions of the Fibonacci number and matrix multiplication algorithms will help us to understand the concepts of dynamic programming better, and allow us to experimentally verify them. By watching the algorithms in action will help us understand the benefits of using dynamic programming over straightforward recursive method. By visualizing and calculating the complexity and efficiency of every implementation allow us to evaluate the three approaches more clearly.

1. **Computation of the nth Fibonacci number:**

**Introduction:**

The Fibonacci algorithm finds nth integer number of the Fibonacci sequence, in which every number after the first two, 0 and 1, is the sum of the two preceding numbers. Three algorithms are implemented for the Fibonacci number problem. The three algorithms are:

1. Recursive Algorithm
2. Memoized Fibonacci Algorithm
3. Bottom-Up Fibonacci Algorithm

Solutions ii and iii are based on dynamic programming. The input requires to insert the nth number. The output of the program provides the Fibonacci result of the given nth number and time taken for each three approaches in nanoseconds.

**Expected performance:**

Expected/theoretical performance (e.g., Θ) for Fibonacci algorithms are as follows:

|  |  |
| --- | --- |
| **Fibonacci Algorithm** | **Expected/ theoretical performance** |
| Recursive Solution | O(2n) |
| Memoization Solution | Ο(n) |
| Bottom-up Solution | Ο(n) |

Table 1: The expected/theoretical performance of the three algorithms that we used to solve the matrix chain problem.

**Actual results and timing:**

This section provides the output of Fibonacci results of given nth number and the time taken in each approach in tabular form.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **nth** | **Fibonacci result** | **Recursive** | **Dynamic** | **Bottom up** |
| 2 | 1 | 5996 | 4425 | 5041 |
| 7 | 13 | 17196 | 7715 | 4781 |
| 30 | 832040 | 523138484 | 21818 | 9243 |
| 10 | 55 | 36307 | 6322 | 4190 |
| 0 | 0 | 2126 | 1526 | 2410 |
| 8 | 21 | 15223 | 5061 | 3263 |
| 50 | 12586269025 | 7724541415406 | 36228 | 13554 |

Table 2: The nth input number, the Fibonacci number, and the running time of the three algorithms.

**Graphical Representation:**

A graphical representation of the data for Fibonacci algorithms is provided in this section. These graphs below depict the time taken for completion of the Fibonacci problem for each three cases. The horizontal axis presents the nth number whereas the vertical axis represents the time in nanoseconds.

As the time difference varied a lot with recursive Fibonacci number 50 and other results, it is not possible to display all the results of time in nanoseconds in y axis. So, the results from table 2 of the time for each approach are converted into log scale (table 3) in order to display them into one graphical representation. The table shows the converted results of the original time provided into the table. The figure 1 provides the graphical representation of the three algorithms.

|  |  |  |  |
| --- | --- | --- | --- |
| **nth** | **Recursive** | **Memoized** | **Bottom-up** |
| 2 | 3.777861624 | 3.645913275 | 3.702516697 |
| 7 | 4.235427436 | 3.88733593 | 3.679518744 |
| 30 | 8.71861667 | 4.338814937 | 3.965812953 |
| 10 | 4.559990365 | 3.800854492 | 3.622214023 |
| 0 | 3.32756326 | 3.183554534 | 3.382017043 |
| 8 | 4.182500247 | 3.704236337 | 3.513617074 |
| 50 | 12.88787271 | 4.559044359 | 4.132067481 |

Table 3: Results after converting the data into log scale

Figure 1:Graphical representation of the output for the Fibonacci number

**Analysis, Anomalies, Performance, and Discussion:**

Theoretically, as seen from the table 1 above, the expected performance for the recursive Fibonacci algorithm is 2n, where it solves the same sub-problem repeatedly. For this reason, dynamic programming is used to solve this problem. The two methods to implement dynamic programming are the top-down with memoization and bottom-up approach. In the first approach, we write the program recursively, save the results of subproblems to use in the future. The second approach depends on the size of a sub-problem, which means solving a specific problem depends on solving smaller sub-problems. Both the top-down and bottom-up methods have the same asymptotic running time of O(n).

The theoretical and the actual performance results are consistent. Since the recursive Fibonacci approach is not using dynamic programming to solve the problem, the running time it takes is higher than the other two approaches. Also, the results of the actual performance are consistent with the expected performance. In the case of the two other methods, since we are applying the dynamic programming, the running time is less than the recursive method.

When we compare the top-down memoized and bottom-up approaches even though both have the same asymptotic running time of O(n), because of its doubly nested loop structure in most cases, the bottom-up method has a slightly higher running time. However, in a few input values, the top-down method has the higher running time. In addition, as seen in the graph, when the size of the input value increases the running time also becomes high, especially, in the case of recursive method. I was expecting that the bottom-up Fibonacci algorithm will always have the lowest running time, but the results vary in some cases which do not contradict the expected running time of O(n). From figure 1, the run time is higher in bottom-up approach compared to memorized approach for the input number 0 and 2. But for the other input numbers bottom-up approach has the lowest run time. Generally, the results of the actual performance are consistent with the expected performance.

**Bonus Fibonacci:**

1. **Straightforward Recursive Solution of Fibonacci number:**

To identify the largest n for the nth Fibonacci number in case of straightforward recursive solution, I ran the code several times and gathered the results which is shown in the following table 4.

|  |  |  |
| --- | --- | --- |
| nth number | Time taken for Recursive method in nanoseconds | Time in terms of hour/min/second |
| 45 | 638721058193 | 11min |
| 47 | 1695640231841 | 28min 18s |
| 48 | 2999810527558 | 50min 3s |
| 49 | 4899815664971 | 1hr 22 min 43 s |
| 50 | 7724541415406 | 2hr 8 min 48s |

Table 4: The nth input number, and the running time of the recursive approach.

The visualization is shown in the graphical view in figure 2

Figure 2: Graphical representation of the output for the Fibonacci number for recursive method

To me, in case of straightforward recursive approach larger nth number is 49 with a wait time of 4899815664971 ns which is about 1 hour 22 minute and 43 second. From the figure, it is visible that the wait time increased by almost one hour for 49 to 50. In case of 50 it took almost 2 hours 8 min 48s.

1. **Memoized top-down approach of Fibonacci number:**

To identify the largest n for the nth Fibonacci number in case of memorized top-down solution, I ran the code several times and gathered the results which is shown in the following table 5. I had to manually set the recursion depth to get higher run time as the recursion limit in python is usually 1000.

|  |  |  |
| --- | --- | --- |
| nth | Memorized runtime in nanosecond | Time in hour/min/sec |
| 20000 | 45231543 | 5s |
| 21000 | 46036025 | 7s |
| 21500 | 48985279 | 12s |
| 21599 | 50030081 | 29s |

Table 5: The nth input number, and the running time of the memoization approach.

The visualization is shown in the graphical view in figure 3

Figure 3: Graphical representation of the output for the Fibonacci number for memoization method

I couldn’t go over the nth number 21599 even though I set up the recursion limit. Each time I tried to so over this number my program crashed. So, according to the output I get for memoized approach 21599 is the largest n for this case.

1. **Bottom-up approach of Fibonacci number:**

To identify the largest n for the nth Fibonacci number in case of bottom-up approach, I ran the code several times and gathered the results which is shown in the following table 6. I had to manually set the recursion limit as I did for the memoized approach.

|  |  |  |
| --- | --- | --- |
| nth | Bottom-up runtime in nanosecond | Time in hour/min/sec |
| 450000 | 143146682045 | 2m32s |
| 480000 | 210587052344 | 3m 30s |
| 490000 | 324042956279 | 5m 28s |
| 491000 | 361222813102 | 6m 6s |

Table 6: The nth input number, and the running time of the bottom-up approach.

The visualization is shown in the graphical view in below figure 4

Figure 4: Graphical representation of the output for the Fibonacci number for the bottom-up approach

I couldn’t go over the nth number 491000 even though I set up the recursion limit. Each time I tried to so over this number my program crashed. So, according to the output I get for bottom-up approach 491000 is the largest n for this case.

**2. Identifying the optimal parenthesization for the matrix chain multiplication:**

**Introduction:**

Matrix chain multiplication is an optimization problem for finding the most efficient way to multiply a given sequence of matrices. Three algorithms are implemented for the matrix chain multiplication problem. The Dynamic programming implementation of the results also show the parenthesization for the most efficient method. The three algorithms are:

1. Recursive Algorithm
2. Memoized Matrix Chain Algorithm
3. Bottom-Up Matrix Chain Algorithm

Solutions ii and iii are based on dynamic programming. The input requires to insert the number of matrices n and the n+1 positive integer representing the dimensions P0, P1, P2….Pn. The output of the program is split into 8 lines, where memorized and bottom-up approach algorithm outputs 3 lines containing the optimal number of scalar multiplications, the optimal parenthesization, and the time taken in microseconds. The recursive algorithm does not give the ‘s’ table and thus outputs only 2 lines containing the optimal number of scalar multiplications and the time taken in microseconds.

**Expected performance:**

Expected/theoretical performance (e.g., Θ) for Matrix Chain Multiplication algorithms are as follows:

|  |  |
| --- | --- |
| **Matrix-Chain Multiplication Problem** | **Expected/ theoretical performance** |
| Recursive Solution | 𝛀 (2n) |
| Memoization Solution | Ο(n3) |
| Bottom-up Solution | Ο(n3) |

Table 7. Table showing the expected/theoretical performance of the three algorithms that we used to solve the matrix chain problem.

**Actual results and timing:**

This section provides the output results of the given input matrices into tabular form.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | **Optimal number** | **Optimal parenthesization** | **Recursive** | **Memoized** | **Bottom-up** |
| 5, 2, 3, 10, 5, 4, 20 | 560 | (A1((((A2A3)A4)A5)A6)) | 176.9065857 | 140.6669617 | 82.25440979 |
| 6, 30, 35, 15, 5, 10, 20, 25 | 13025 | ((A1(A2(A3A4)))((A5A6)A7)) | 391.7217255 | 201.4636993 | 124.4544983 |

Table 8: Table showing input matrices, the optimal number, optimal parenthesization, and the running time of the three algorithms.

The code was also tested on several large input sizes. The following table contains some of the input and output results of the large samples in which the code was tested.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | **Optimal number** | **Optimal parenthesization** | **Recursive** | **Memoized** | **Bottom-up** |
| 3, 6, 8, 11, 14, 19, 22, 26, 27, 31, 36, 39, 42, 46 | 25425 | ((((((((((((A1A2)A3)A4)A5)A6)A7)A8)A9)A10)A11)A12)A13) | 228256.702423095 | 498.056411743164 | 337.600708007812 |
| 10,10,10,15,17,19,25,25,27,28,35,37,37,39,40,50,53,58,56,64 | 245620 | ((((((((((((((((((A1A2)A3)A4)A5)A6)A7)A8)A9)A10)A11)A12)A13)A14)A15)A16)A17)A18)A19) | 165112773.656845 | 1207.35168457031 | 729.322433471679 |
| 2,5,23,21,4,5,6,9,33,11,4,30,4,22,6,9,44,5,3,22,5,7 | 5692 | ((((((((((((((((((((A1A2)A3)A4)A5)A6)A7)A8)A9)A10)A11)A12)A13)A14)A15)A16)A17)A18)A19)A20)A21) | 1481336627.48336 | 1687.04986572265 | 951.051712036132 |

Table 9: Table showing input matrices, the optimal number, optimal parenthesization, the running time of the three algorithms.

**Graphical Representation:**

A graphical representation of the data for Matrix Chain Multiplication algorithms is provided in this section. The graphs below depict the time taken for completion of the matrix multiplication problem for each three cases. The horizontal axis shows the size of input, whereas the vertical axis shows the time in microseconds. Figure 5 exhibits the graph for table 8 and figure 6 shows the graph for both table 8 and table 9 in order to visualize the change among small and large input size. As the minimum and maximum results varies a lot, the output was converted into log scale to make all the results visible. The table 10 shows the converted results.

Figure 5: Graphical representation of the output for the matrix chain multiplication solution

|  |  |  |  |
| --- | --- | --- | --- |
| **Input Size** | **Recursive** | **Memoized** | **Bottom-up** |
| 7 | 2.247744001 | 2.148192107 | 1.91515919 |
| 14 | 5.358423539 | 2.697278535 | 2.528403349 |
| 20 | 8.217780673 | 3.081833792 | 2.862919572 |
| 8 | 2.592977659 | 2.304196804 | 2.095010598 |
| 22 | 9.170653761 | 3.22712792 | 2.978204132 |

Table 10: Results after converting the data into log scale

Figure 6: Graphical representation of the output for the matrix chain multiplication solution for both given data and larger test data

**Analysis, Anomalies, Performance, and Discussion:**

From the table above, the theoretical performance for the recursive matrix-chain multiplication problem is 𝛀 (2n) as the recursive algorithm solves each subproblem every time it reappears repeatedly. The other two approaches are the top-down with memoization and bottom-up. Both the top-down and bottom-up approaches are more efficient because they take the advantage of the overlapping sub-problems property of dynamic programming. Both the top-down and bottom-up approaches have the same asymptotic running time of O(n3). But all sub-problems must be solved at least once the bottom-up approach outperforms the corresponding top-down approach, because the bottom-up algorithm has no overhead for recursion and less overhead for maintaining the table.

The theoretical and the actual performance results are consistent. As expected, the recursive solution takes the longest time to run than the other two dynamic programming solutions top-down and bottom-up approaches. Both outputs of the bottom-up method have the lowest running time since they did not recurse. In this problem, I did not find anything surprising since all the running time of the outputs are similar to the expected results.

**Conclusion:**

We can clearly infer from the graphs that the DP implementation is extremely efficient. Such drastic running time reduction in the DP implementations compared to recursive implementation is not something I had forecasted. As it stands, like with the theoretical time complexity of the recursive approach, the practical results had a similar story to tell. Operating in exponential time, a recursive algorithm is very inefficient. In contrast, memorized versions in dynamic programming reduced the execution time to a fraction of what it took for recursive.