**Project 2: Dynamic Programming**

**Hassaan Naveed**

In this project, we were tasked with implementing different versions of popular algorithms, one with its standard implementation, and 2 with dynamic programming modifications to reduce the time and cost of running these algorithms. We implemented these 3 different algorithms with two popular algorithms, namely the Fibonacci series and the Matrix Chain Multiplication problem with optimal parenthesization.

**Disclaimer**: The machine I used to execute these programs is on a higher end in terms or its specifications. I have 6 CPU cores, 32 GBs of RAM, and a solid state M.2 NVMe SSD. The reason for this disclaimer is to emphasize that the performance in terms of time and limit to which I attained answers to these recursive algorithms will be different for different machines. With my machine I had a higher level I could reach which could not be replicated on machines with lower specifications such as my laptop. I also used Python to code these algorithms and monitor their performance. Python dynamically alters the length of numbers that can be computed and printed according to the resources it has access to, and some standard limitations such as the number of digits or levels of recursions that can be printed or called respectively can be heavily altered to achieve better sets of results.

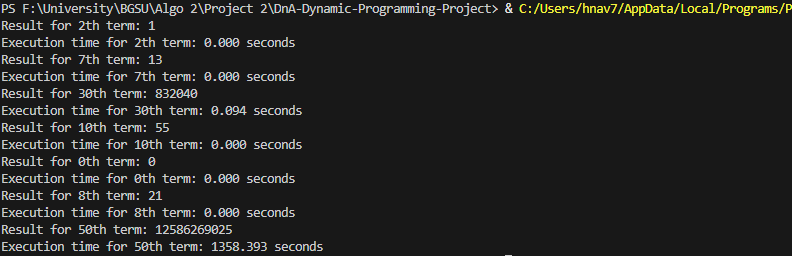
Why dynamic programming?

Dynamic programming is useful because it transforms problems that seem computationally unrealistic into ones that can be solved efficiently by breaking them down into simpler, overlapping subproblems and implementing ways to reduce and discard repeat computations and improve performance. This approach is valuable to solving optimization problems such as finding the shortest path, minimizing cost or maximizing profit. In the real world, efficiency is key, and dynamic programming helps to achieve this optimal result.

**Fibonacci Series:**

The standard recursive implementation of the Fibonacci algorithm has no way to store and reuse previous calculations and continues it’s recursive calls and repeats many of them multiple times. I used a sample list of “nth terms” to compute and record their execution time.

Standard algorithm results for different nth terms:

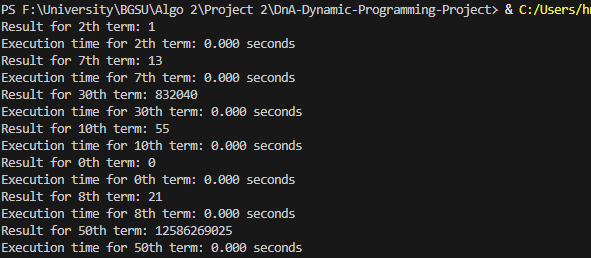


Result for the 53rd term:



The memoized version of the Fibonacci series helps the run time by implementing a sort of cache within the algorithm itself to check in on in case the value had been calculated before and can be used again without needing another set of recursive calls to complete. This pushed the algorithm to start giving results very fast.

Results for memoized algorithm for various nth terms:

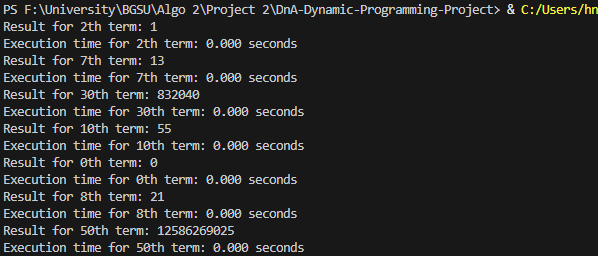


Result for the 1000000th term:



The last version is the bottom-up approach, it forgoes the recursion calls to just compute the terms of the Fibonacci series and opts to use a list, with the base cases already assigned which are 0 and 1, and then iteratively calculates the successive terms up until the term you would want.

Results for bottom-up algorithm for various nth terms:



Result for the 1000000th term:

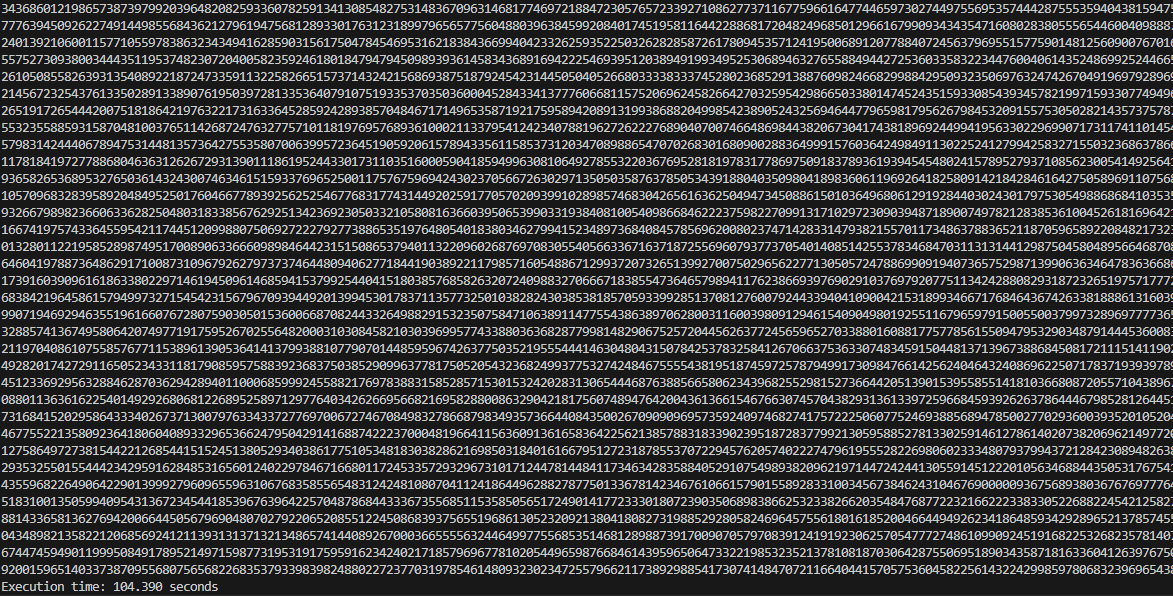


Table comparing the time taken for the 50th term for a consistent yet observable time taken:

|  |  |  |  |
| --- | --- | --- | --- |
| **Nth Term** | **Standard Recursive (s)** | **Memoized (s)** | **Bottom-Up (s)** |
| 50 | 1353.393 | 0.000009 | 0.000006 |

**Bonus:**

For lower terms such as the first 25-30 or so in the standard recursive approach, the results were very quick. As I approached the 40th term, time began to noticeably increase, but not so long that I felt I would not continue to test it’s limits. As I approached the 50th term, the time shot up considerably, to about 22 minutes. By about the 52nd term, the time shot up to close to an hour, and the time for the 53rd term ended up being just under 1 hour at 58 minutes. The 54th term took over 2 hours to compute, with the result itself a long result. At this stage I felt that the very noticeable exponential increase in time taken was not worth the time requirement. At this rate, I was not comfortable going past the 60th term, which is what I consider to be the absolute highest I would go given the time requirement.

Where I felt that in the standard version the 60th term would be the absolute highest, the memoized approach gave me an answer to the 60th term in mere microseconds. So I decided to push it’s limits. I started with the 100th term to test, then 1000, 10000, 100000 and finally 1 million-th term. At the 10000 term mark, the result was the thing that gave me my first error, as the number of terms to be printed exceeded the preset number of digits permissible (4300 digits) which I overrode to continue testing. The 1000000th term took around 2 minutes to execute and complete. At this point, time was not the issue I had an issue with, it was the extremely large result that I was getting. To me, it seemed like a collection of random numbers which I can never truly verify was the correct answer. At this stage, even though it was very interesting to watch this algorithm fly and compute incomprehensible numbers, the results really did not feel as if they were worth the effort to continue testing. At the rate the algorithm performed, to consume the prior algorithms 2 hour runtime, I would have to go into multi-million-th terms, at which rate I do not think my architecture would allow such a large number to be computed.

This approach helped with the timing issue a little bit, not by a whole lot, and performed very similarly to the memoized approach, with testing up to the 1000000th term took 100 seconds instead of 120 seconds. If gone further, it would more or less have very similar performances in greater terms that go into the multi-millions. My issue is the same as the memoized version that the result I got started to look like a huge jumble of numbers that were for the most part unverifiable to be absolutely correct.

Something to note about the memoized and bottom-up versions of these algorithms is that executing these algorithms put a sever strain on my machine on the higher terms, such as the millionth term. Even though I got a result very quick compared to the standard algorithm, my computer became extremely slow, so much so that I had issues even closing or opening my file explorer or even the start menu. With that in mind, it added another reason why I would not feel comfortable going much beyond the millionth term.

With all said and observed, even with the optimized approach taking a very reasonably less amount of time, I would consider the 10000th term to be the most reasonable to get in terms of calculating the nth term, so that the number achieved can at least be reasonably understood and somewhat verifiable.

**Matrix Chain Multiplication and Optimal Parenthesization:**

The matrix chain multiplication has the goal of finding the most efficient way to parenthesize a chain of matrix multiplications and to minimize the total number of scalar multiplications. It is to not, we do not multiply the matrices, but simply calculate the least amount of multiplications needed to have the most optimal way of actually calculating the result. The standard approach is simply recursively calling the function to calculate the value k which is the optimal spot for parenthesization and the number of scalar multiplications for a pair of matrices needed. We use the following formula:

The variations are similar to the Fibonacci series’ implementations of dynamic programming, with memoized using a sort of cache to reduce calculations, and the bottom-up abandoning the recursive calls in favor of iteratively getting the result.

Table of times taken by each algorithm using the sample data in the matrix chain txt file:

|  |  |
| --- | --- |
| **Algorithm** | **Time (s)** |
| Standard recursive | 0.000027 |
| Memoized | 2.9087066650390625e-05 |
| Bottom-up | 1.5974044799804688e-05 |

Results of standard, memoized and bottom-up respectively:







Observing these results, it is very clear which algorithm implementation achieves the best results in terms of time, and as you introduce larger and larger matrices, the memoized and bottom-up approach will be the most efficient to achieve the result the fastest.