Kaleb LuceWireman

CS4120

16th Apr. 2025

Report for Project 2

**Introduction**

I learned a lot doing this project. I am not the most proficient in python, but I know that data structures and algorithms are generally easier in it. Additionally, it is a very common language to use in development and so, I would like to get better at it. I didn’t know exactly how to code the matrix multiplication problem at first, but the textbook definitely gave me a good start, as the pseudocode is already very similar to python. The runs didn’t take me too long, as I didn’t get too ambitious with the test inputs. I mainly used the sample inputs provided and added a longer problem for the matrix-chain computation.

**Expected vs Actual Results**

The results for the recursive implementations for both algorithms went basically as expected. Due to the exponential time complexity, they became wildly inefficient with larger input sizes. It was actually impossible to show the larger input times on the graph while allowing you to see the difference between the memoized and iterative implementations. The difference between the observed and expected results came from the memoized and iterative implementations. I found that my memoized implementation, in matrix chain with the large set, was slightly faster than the iterative implementation. This is particularly interesting because for loops have a predetermined amount of iterations in python, which makes them more efficient than some other languages. Perhaps I didn’t use the most efficient implementation. Margin of error could have gone into effect, or maybe input sizes were too small to see a huge difference between them.

**Importance of Dynamic Programming**

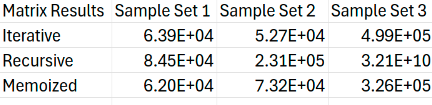
Dynamic programming is important because a lot of problems would be unsolvable without it. We learned in class that for certain problems, the greedy method simply doesn’t work. As shown in my results, it also provides a MASSIVE time complexity improvement compared to the naive implementation (recursive). The theoretical improvement is from O(2^n) with the naive implementation to only O(n^3) for matrix chain or O(n) for Fibonacci. The improvement is so large because the naive implementation wastes a lot of time recalculating the same problems over and over. While an iterative approach can be better compared to even the memoized approach (no stack overflow issues) the memoized approach is about as fast and can be more intuitive.

**Bonus**

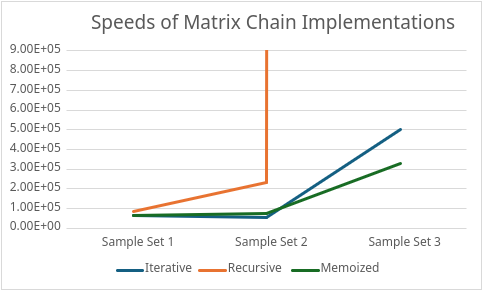
I decided to wait about 10 minutes for each algorithm as my time limit. For the recursive implementation, this was at about the 50th Fibonacci number. In the process of doing the bonus, I had to make some changes to my implementations. First, I added the python ‘sys’ module to increase the recursion limit for the memoized version (default limit is 1000). Then I had some errors about my memo array not being initialized for larger sizes. Finally, I had to improve my iterative algorithm to only store the last 2 values rather than all of them because it was immediately filling up all 32 GB of RAM I have in my computer and would crash. To get around the same time (594 seconds in my testing) the iterative algorithm calculated the 11millionth Fibonacci number. The memoized implementation simply took up too much ram on my system, and I couldn’t even get the 1 millionth Fibonacci to calculate without crashing. I did get the 700,000th number to calculate and that only took 7.57 seconds. Overall, the recursive method is only useful because of how intuitive it is, and it isn’t practical at all past n=10, when another method would do the same thing much faster. The iterative method is best because it requires much less memory and doesn’t require storing every value. The memoized method is similar in speed to the iterative method but simply needs too much memory to calculate the higher values, even with a large amount of memory (32 GB in my system).

**Results**

Matrix-Chain Results

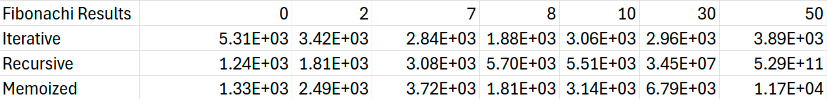


The times listed are in scientific notation in nanoseconds while the x-axis denotes the sample set tested (correlates to lines 1,2, and 3 respectively)

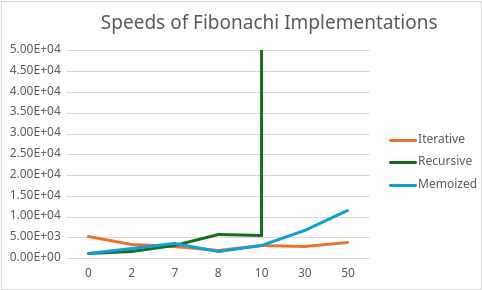


The results for matrix chain found that the iterative and memoized methods were similar speeds, while the recursive method was slower and became exponentially slower when provided with 20 matrices to multiply (sample set 3).

Fibonacci Results



The times listed are in scientific notation in nanoseconds while the x-axis denotes the n value being tested



The Fibonacci results showed that the recursive implementation was competitive with the others up to about n=8 but fell behind at n=10 and was eclipsed by both the other implementations at n=30 and n=50 as expected. Additionally, the iterative method showed a bit of a speed advantage over memoized for n=30 and n=50. In terms of RAM usage, both the recursive and iterative versions used very little, while the memoized version used much more by comparison.