**WEEK 3-4 PROGRAMMING ASSIGNMENT**

**Task 1:**

TASK DESCRIPTION

This task implements the multiplication of 2 square, N x N matrices using the recursive divide-and conquer multiplication method discussed in class (Gongora, J. 2025). To simplify the recursive decomposition, matrix sizes are defined as powers of two, i.e. N = 2^m. This can be done by splitting the two input matrices, A and B, and the resultant matrix, C, into 4 smaller submatrices and recursively implement the block multiplication operation to compare recursion to the standard naïve multiplication approach.

METHOD USED

As discussed in lecture, the algorithm recursively divides matrices A and B into quadrants (lines 47-55) and computes sub-blocks of C through element-wise addition of partial products, or M blocks (59-72). A terminating condition is set for when the smallest submatrices are found, i.e. when the matrix size = 1x1 (lines 38-40).

The implementation consists of an adapted modular function (Stack Overflow, 2017); recursive\_mat\_mult – the core divide-and-conquer function that performs recursive calls until the terminating condition is met. I have chosen to store the matrices/submatrices as Numpy arrays, as early implementations required specific functions for element-wise addition/subtraction, but these are inbuilt functions in Numpy. This not only simplifies element-wise multiplications but also improves memory efficiency and readability compared to Python lists. Once all the sub-blocks of C have been found and the terminating condition is met, these blocks are appended to their respective rows and columns using the Numpy vstack/hstack functions (line 75).

To test this, I have reused my random\_matrix function from the previous week that generates a random matrix, and set these as A and B, as well as also using my iterative multiplication function from the previous week so I can compare the efficiency of each implementation. I also added a new function to format the output matrices for clearer visual comparison (Java2Blog, n.d.)

DISCUSSION

Although this method may have the same asymptotic complexity as the naïve multiplication, its implementation was slightly harder due to the logic involved. Instead of just iterative looping through each index in the matrices, it follows a more complex process of dividing matrices into smaller quadrants recursively until a terminating condition is met. Despite this, the benefit of this function is clear when matrices of higher dimensions are involved, as being able to split them into smaller quadrants will be faster than iterating through each element of the matrix – in a similar manner to a linear vs. binary search.

**Task 2:**

The task implements the Strassen multiplication as discussed in lecture (Gongora, J. 2025), which builds upon the recursive method discussed in task 1. The task is once again simplified by having a matrix size as a power of two (N = 2^m), and the process of multiplication is similar to that of the recursive method (recursion is still used in this method), but the arithmetic performed with the respective submatrices differs slightly.

METHOD USED

The implemented Strassen function (GeeksforGeeks, 2024) follows an almost identical initial structure to the previous recursive function; set a terminating condition to be met to end recursion, set a midpoint and divide input matrices into smaller submatrices. Unlike the previous approach, the Strassen method modifies the arithmetic of sub-block multiplications, replacing the eight products of the recursive method down to 7 through linear combinations of submatrices, thereby reducing asymptotic complexity. The resultant matrix is then constructed using these products and appended once again using the Numpy vstack/hstack functions.

DISCUSSION

The implementation closely mirrors the recursive method, differing primarily in the arithmetic used when calculating the respective sub-blocks of the resultant matrix, C. In comparison to the previous method, through the help of the slight differences in arithmetic, there is now only 7 matrix multiplications involved instead of the recursive methods hefty 8 multiplications. Using the master method (Cormen *et al.*, 2009, p. 101), the complexities of these methods can be compared via the recurrence (T(n) = 7T(n/2) + O(n^2)), and it is observable that the Strassen method yields an improved asymptotic complexity of O(n^2.81). This beats the complexities of both the naïve and recursive methods (O(n^3)). Once again, although the Strassen method may be harder to implement, it is superior with performing multiplication operations for matrices of higher dimensions. Despite this, for small matrix sizes, the implementation of extra arithmetic can outweigh the benefit, making Strassen slower in practice.

**Task 3:**

TASK DESCRIPTION

This task asks me to compare the performance, in this case the runtime, of all the explored matrix multiplication methods I have implemented, for varying values of m when the matrix size, N = 2^m.

METHOD USED

Firstly, I have separated all the functions involving any matrix operations into a separate file from the following runtime testing program, to ensure modularity and easier reading and understanding of the code, and imported the functions from “matrix\_functions.py” as mf before any function is called.

For the purpose of this task, instead of repeating multiple iterating loops to test each of the implemented methods, I have opted to initialise a dictionary of values, storing each method function with its respective method name (line 165). A similar idea is followed with the storing of the runtime results, in which a dictionary is made to store the runtime of each method iteration per iteration of matrix size as a function of 2^m (line 167-169). A number of iterations can also be implemented, in order to verify values and calculate an average runtime for each method.

The runtime calculations are then performed within the loop (line 173-191), for each iteration of matrix size N in the range of 2^1 to 2^9. At the start of each iteration, a new matrix A and B are created, and then for each function in the method dictionary, the timer is started, and the function is performed using matrix A and B, a set amount of times based on the input number of desired repetitions. The timer is then concluded, and an average runtime for each method. These runtime results and iteration number are appended to each methods respective array in the results dictionary.

From this, a log-log graph can then be plotted using Matplotlib with the average runtime of each method displayed against the iteration of matric size, NxN. The logarithmic graph is used to compare the polynomial runtime of each method, and how method fits linearly with the characteristic behaviour of a polynomial function.

RESULTS

Below are the results for matrix multiplications of each of the functions, naïve, recursive and Strassen, respectively, for a randomly generated matrix (figure 1.1) as well as the logarithmic plots of each functions runtime as a function of matrix size NxN (figure 1.2):  
A screenshot of a computer

AI-generated content may be incorrect.

Figure 1.1: results for each of the multiplication functions on random matrix, generated using the random\_matrix function from the previous assignment

A graph of a number of numbers

AI-generated content may be incorrect.

Figure 1.2: log-log plots of each respective multiplication function’s runtime against matrix size iteration

DISCUSSION

The runtime comparison (figure 1.2) clearly displays the differences in the scaling behaviour of the three separate algorithms. All the graphs display similar asymptotic complexities, consistent with their theoretical complexities of around O(n^3), as shown by the linear behaviour in the log-log graph (for cubics). Both the recursive approaches generally perform slower due to the additional overhead introduced by repeated function calls, creation of submatrices and arithmetic executed. However, although the Strassen algorithm originally exhibits a slower performance, the slope increases at a slower rate (shown around matrix sizes of 2^6), reflecting its improved theoretical complexity of O(n^2.81) for larger datasets. It is visible that as N increases, the gap between the methods narrows, portraying the asymptotic advantage of the Strassen approach for larger input matrix sizes. Overall, the plotted data results closely follow theoretical expectations, validation the predicted polynomial relationships of each implemented method.

REFERENCES:

Gongora, J. (2025) *Data, Algorithms and Numerical Optimisation – Week 2-3 Lecture Slides*. Available on Learn (accessed 12 October 2025).

Stack Overflow (2017) *Python divide and conquer recursive matrix multiplication*. Available at: <https://stackoverflow.com/questions/42453523/python-divide-and-conquer-recursive-matrix-multiplication> (Accessed: 12 October 2025).

Java2Blog (n.d.) *Print a matrix in Python*. Available at: <https://java2blog.com/print-a-matrix-in-python/#2_Basic_Looping_with_Indices> (Accessed: 13 October 2025).

GeeksforGeeks, 2024. *Strassen algorithm in Python*. [online] Available at: [https://www.geeksforgeeks.org/dsa/strassen-algorithm-in-python](https://www.geeksforgeeks.org/dsa/strassen-algorithm-in-python?utm_source=chatgpt.com) [Accessed 20 Oct. 2025].

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2009. *Introduction to Algorithms*. 3rd ed. Cambridge, MA: MIT Press. Available at: <https://ebookcentral.proquest.com/lib/lboro/reader.action?c=UERG&docID=6925615&ppg=125> [Accessed 20 October 2025].