John Goza
EN.605.620.81.SP23 – Algorithms for Bioinformatics
Lab 1 – Strassen Matrix Multiplication
Due 21 Feb 2023

Abstract

In Lab 1 we were tasked with implementing and analyzing the efficiency of matrix multiplication using two different algorithms. My implementations were written in Python. Implementation posed many challenges other than the inherent complexity of the multiplication step, as we were required to parse a given input file with three different matrix multiplications, handle errors, and define and perform math on the given matrices without using certain "convenience" classes (e.g. NumPy). My program handles input and output errors as well as errors in specific matrix inputs. It performs both types of multiplication for all valid matrices given in an input file, and outputs user friendly results to a file. Upon running the given input file, I found that ordinary multiplication was $\Theta(n^3)$, while recursive Strassen multiplication was $\Theta(n^{\lg(7)})$, which matched my expectations.

Data Structures and Design

I chose to ingest the provided data and parse it into an array of arrays (see Figure 1 for an example) I made this choice for two reasons: 1) by using an array of arrays, I was able to write cleaner code, as I could use the length of a given array rather than having to pass the order of the matrix to each function, and 2) an array of arrays closely resembles a matrix when accessing an individual value by index (i.e. matrix_as_array[i][j] is similar to matrix(i, j)). Alternative data structures were considered; storing the data in any string format was rejected due to the potential for errors between string and integer conversion, and a flat array structure was rejected due to the loss of row / column length context.

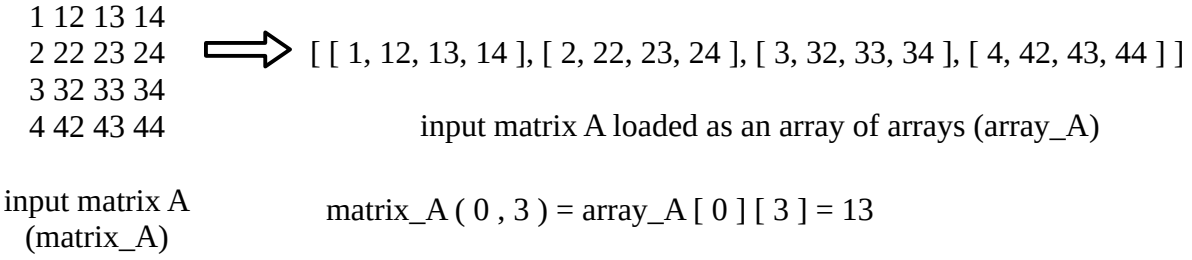

Figure 1. Example of data transformation and storage format. Input was ingested in row major form as seen in matrix A, which was then transformed into an array of arrays as seen in array A. An example of matrix and array access for the same element is provided to demonstrate the similarity between the two.

To parse the data into the required format, I created a module called "matrix_parser.py". This module has two functions: 1) parse the data, and 2) handle errors. This module contains validation logic as well as parsing and formatting logic, and flags matrix inputs that fail validation due to incorrect row / column sizes, orders that are not exact powers of two, etc. The entry function in this module (parse_matrix_file()) receives the input filename and returns a list of key value maps. Each key value map contains either the parsed and formatted input for a given set of matrices, or a string representing the error encountered while parsing that specific set of matrices. This allows processing of other

matrices contained in the same file to continue, ensuring that one error will not cause a catastrophic failure.

The file output is handled by the "output_handler.py" module; it contains a formatter for converting arrays of arrays into nicely printed matrix output, as well as methods for writing both successful and failed results to an output file. IOErrors were allowed to bubble out of both the matrix_parser and output_handler files; since input and output are both required for successful completion of the problem outlined in this lab, IOErrors were deemed to be critical and non-ignorable.

The matrix math that is not part of the inherent question posed by this lab can be found in the "matrix_functions.py" file. This file includes the functions to split a given matrix into four quadrants (required for recursion), to merge rows back together, and to add or subtract two matrices. I am particularly proud of the "do_math" function. It takes the math operand as a parameter to the function, thus preventing the need for separate addition / subtraction functions.

All of the design decisions were made with the intent to reduce complexity. I chose to push the error handling to the edges as described above for this reason, and to be aggressive in modularity for this reason. Aggressive modularity allowed me to keep only the two multiplication algorithms and a small amount of driver code in the "Lab1.py" file. Finally, additional test cases were defined in the "test_input.txt" file; test cases included valid matrices and invalid matrices.

Results

I expected ordinary multiplication to have a computational complexity of $\Theta(n^3)$, as it required a triple nested for loop. I expected recursive Strassen multiplication to have a computational complexity of $\Theta(n^{lg(7)})$, as proved on page 96 of our textbook (Cormen et. al). To confirm, I ran the provided input file (Appendix A) through my program, which returned correctly multiplied matrices for each of the three individual problems contained in the input. The number of multiplication steps performed by each algorithm were recorded by incrementing an integer when multiplication occurred, and produced the results found in Table 1.

Matrix Size	2x2	4x4	8x8	Complexity
Ordinary	8	64	512	$\Theta(n^3)$
Strassen	7	49	343	$\Theta(n^{lg(7)})$

Table 1. Multiplications performed by ordinary and Strassen algorithms for required input matrices of different sizes.

While these results certainly match our expected complexities, they were not enough in number or size to provide a conclusive overall result. To establish that this relationship held as the matrices grew larger, I designed more input and reran the application. As seen in Table 2 and Figures 2a and 2b, the complexity for each algorithm matched their expected complexity as matrix size increased. Matrix size was capped at 1024 due to hardware constraints, but I would expect these relationships to continue for all values of n.

Space complexity was assumed to be measured similarly as the time complexity, i.e. only for the multiplication portions of the code, and not the ingesting / formatting portions. Space complexity was not directly measured, but can be estimated from the structure of the different algorithms. For linear multiplication, I chose to append the multiplied values to a placeholder array, which would require space for three matrices of size m x m (two input and one output), which we chose to represent as n. One could

write this as Θ (3n), which it then simplified as Θ (n). For Strassen's algorithm, space increases at a rate of 7T(n / 4) per iteration (as the matrix is partitioned into fourths and passed to seven recursive calls to the same function). We can then represent the space complexity by $T(n)=7T(\frac{n}{4})+n$, which can be solved using case 1 of the Master Theorem. Hence, the space complexity for Strassen's algorithm is determined to be $\Theta(n^{lg4(7)})$.

Matrix Order (n)	2	4	8	16	32	64	128	256	512	1024
Ordinary	8	64	512	4096	32,768	262,144	2,097,152	16,777,216	134,217,728	1,073,741,824
n^3	8	64	512	4096	32,768	262,144	2,097,152	16,777,216	134,217,728	1,073,741,824
Strassen	7	49	343	2401	16,807	117,649	823,543	5,764,801	40,353,607	282,475,249
$n^{lg(7)}$	7	49	343	2401	16,807	117,649	823,543	5,764,801	40,353,607	282,475,249

Table 2. Multiplications for ordinary and recursive Strassen algorithms compared to expected values. Note that as n grows, the number of operations for ordinary multiplication remains n^3. As n grows, the number of operations required for Strassen remains $n^{lg(7)}$.

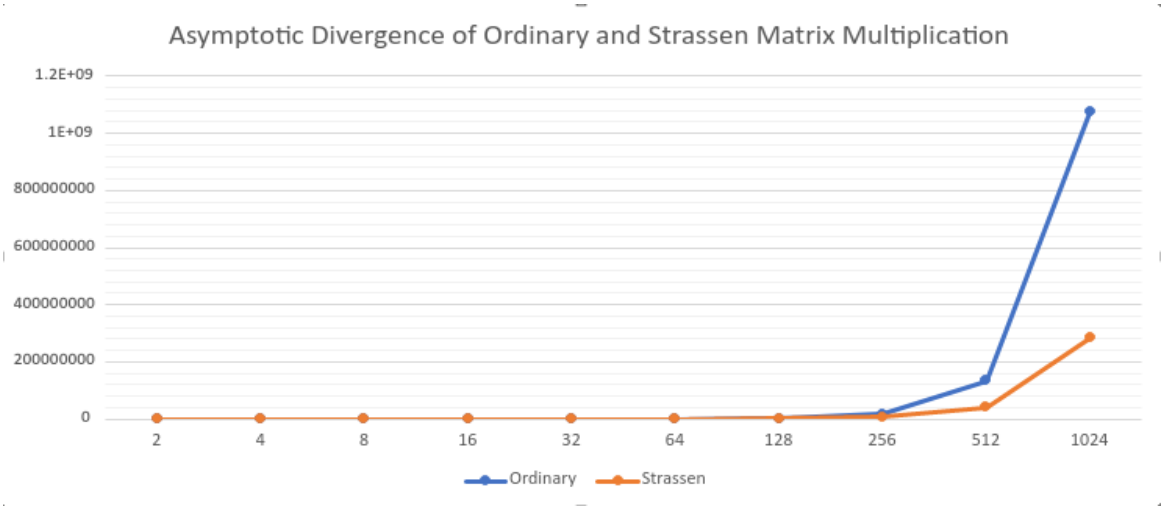

Figure 2a. Computational complexities of ordinary and recursive Strassen matrix multiplication for different input matrix sizes (linear scale).

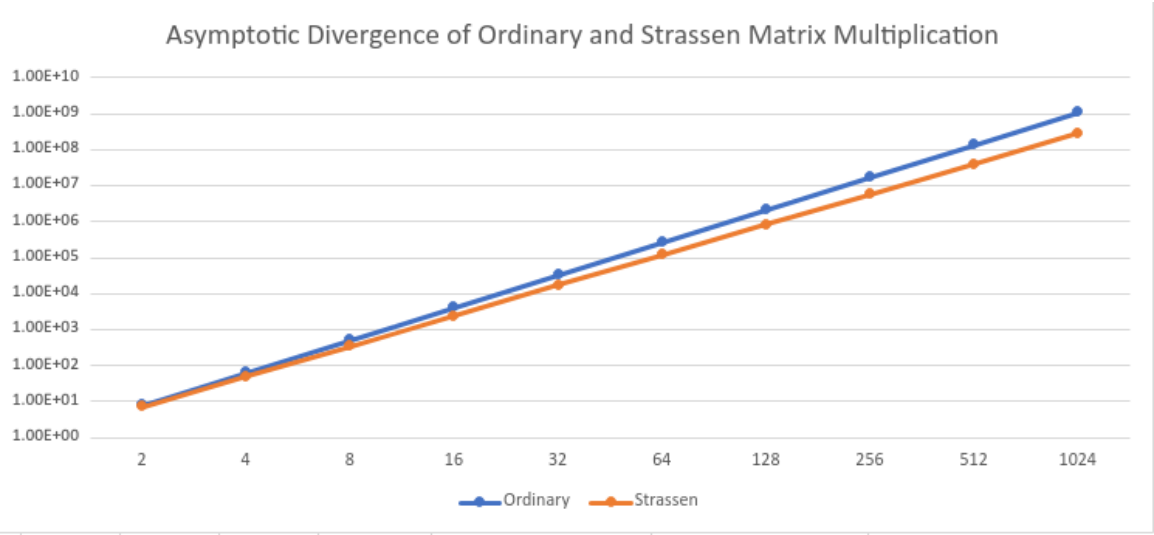

Figure 2b. Computational complexities of ordinary and recursive Strassen matrix multiplication for different input matrix sizes (logarithmic scale).

Conclusions and Reflections

The results outlined above establish conclusively that Stratten's algorithm outperforms ordinary multiplication at all matrix orders that are powers of two. This is an important finding, especially in the context of Bioinformatics. There are many classification and annotation algorithms that use matrix multiplication behind the scenes (e.g. Oxford Nanopore's basecalling software), and since genetic data tends to be quite large, efficient matrix multiplication implementations are critical to their performance. Additionally, many machine learning models require matrix comparison, projection, and other computations. Faster computation typically leads to faster models, which is always desirable.

I learned many things while completing this lab. In the context of the assignment, I learned that even the most efficient algorithm is only as good as its surroundings. The ingestion and formatting of the matrices was detrimental to the performance of the program as a whole, since the process added a significant amount of computational complexity. Given time and the usage of other libraries, this complexity could be reduced, but under the constraints provided it was a necessary evil. Personally, I learned a lot regarding the struggles of implementation. It had been years since I had implemented a recursive algorithm, and the implementation of Strassen's method was a great refresher. I had to re-learn the importance of selecting the correct data structure, the difficulty of flat file handling, and a significant amount of matrix math operations.

These lessons were reinforced by a swath of mistakes I made while implementing the lab. If I had to redo implementation, I would more carefully study the input file and the lab requirements before starting to code. I made the mistake of assuming each matrix would be one line rather than in row major format, which required a significant correction in my parsing logic. I also misread the lab assignment at first and implemented recursive ordinary multiplication rather than linear ordinary multiplication. While this was extremely helpful for implementation of Strassen's algorithm, I would have preferred to implement only the two multiplication algorithms required, rather than three. Overall, this lab gave me a much deeper knowledge of matrix multiplication algorithms and the measuring of computational complexity, which are two topics that will continue to be relevant to my work.

References:

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. Mit Press.

Element-wise addition of 2 lists. Stack Overflow. (1960, August 1). Retrieved February 21, 2023, from https://stackoverflow.com/questions/18713321/element-wise-addition-of-2-lists