Hazelyn Cates
9/19/22
EN.605.620.81.FA22

Project 1

Implementation and Design

This program consists of three functions: divide_matrix, Strassen_method, and matrix_multiplication. As description of each function follows:

divide_matrix(matrix)

Takes a single matrix as an argument. The shape of the matrix (i.e. its order) is determined and using this value, the matrix is divided by two to create submatrices of size order/2, which get returned. This function is called within the Strassen_method function.

Strassen_method(matrixA, matrixB)

Takes two square matrices of the same size whose order are a power of 2 as arguments. If one matrix is already of size 1x1 (which in the case of square matrices means they both are), they will be multiplied, and the product returned. If not, the divide_matrix function gets called twice (once for each matrix) to assign four values (a, b, c, d) containing size order/2 submatrices to matrixA and four values (e, f, g, h) containing size order/2 submatrices to matrixB. The p1-p7 calculations require recursive calls to the Strassen_method function, where the a, b, c, d, e, f, g, and h values become the arguments. For example:

p1 = Strassen_method(a, f-h), where "a" is acting as matrixA, and "f-h" is acting as matrixB. These variables are storing order/2 matrices at the first call.

This first recursive call to generate the value for p1 sends the program to the top of the function, where it once again checks if the matrices are 1x1 (i.e. the base case). If not, the divide_matrix function gets called again to generate order/4 size submatrices. The program once again reaches the recursive call for Strassen_method for p1, and the arguments, which are still "a" and "f-h", are now submatrices of size order/4. With this second recursive call, the program once again goes to the top of the function to check if the matrix size is the base case.

This recursive process continues for p1 until a size of 1x1 for each matrix is reached, in which their multiplied product gets returned and stored in the p1 variable.

This process repeats for variables p2-p7 in the exact same manner. Once all the p-values have been calculated, the C-values must be calculated via various arithmetic manipulations of the previously calculated p-values. The C-values must then be combined together to generate the matrix of the products, and the final matrix C is returned.

matrix_multiplication(matrix1, matrix2)

Takes two matrices as arguments, which in this case are square matrices of the same size whose order is a power of 2. A matrix of the proper size is initialized to all zeros, which will contain the final

product. The function then enters three nested for loops that traverse through the rows of the first matrix and then then columns and rows of the second matrix and sums the products at each index, before finally returning the resulting matrix product.

No output is printed to the terminal. Instead, the results of each multiplication, number of multiplications, and execution times for each function are written to a text file named "products".

Efficiency and Analysis

The recurrence relation to represent the algorithm implementing Strassen's method of matrix multiplication is as follows:

$$T(n) = \begin{cases} \theta(1) & if\ n = 1 \\ 7T\left(\frac{n}{2}\right) + \theta(n^2) & if\ n > 1 \end{cases}$$

According to case 1 of Masters Theorem, the solution to the recurrence relation is:

$$T(n) = \ \theta(n^{lg7})$$

Therefore, Strassen's algorithm has a time complexity of $O(n^{lg7})$.

(pp. 79-80, Cormen et al, 2009; "Divide and Conquer", 2022)

By comparison, the naïve matrix multiplication function has a time complexity of $O(n^3)$ since there are triply nested for loops that iterate by a constant amount, which run in linear time (i.e. O(n) time) (pg. 76, Cormen et al, 2009; "How to Analyse", 2022).

Regarding space complexity, the naïve method of matrix multiplication is $O(n^2)$ due to the requirement of creating a third matrix to hold the multiplication product, which is n x n in size ("Time Complexity", 2022; "Python program", 2022).

For Strassen's method, since it is a recursive function, the depth of the recursion is how many calls are required until the base case of the matrix is reached (More, n.d.). For the specifics of this project, only square matrices whose order are a power of 2 get multiplied, making the depth of the recursion $\log_2 n$, where n is a power of 2. For example, the recursion depth of multiplying two 8x8 matrices would be $\log_2 8$, which equals 3, which means three recursive calls must be implemented to reach the base case. Therefore, the space complexity for Strassen's method would be O(lgn). Using the above recurrence relation for Strassen's method, the following proof can be generated to show this asymptotic bound:

Prove: $T(n) = \ O(lgn)$

$$T(n) \leq 7\lg\left(\frac{n}{2}\right) + cn^2$$

$$= 7lgn - 7lg2 + cn^2$$

$$= 7lgn - 7 + cn^2$$

Where: $lgn \leq 7lgn - 7 + cn^2$

If: $7lgn - 7 + cn^2 \geq 0$

Where: $c \geq \frac{7(1-lgn)}{n^2}$

$\therefore T(n) = O(lgn)$ and n > 1

Given the above time complexities, Strassen's method has a theoretically better efficiency, and therefore it is expected that the greater the matrix size, the more efficient Strassen's method becomes, with its efficiency versus the naïve method negligible on matrices of small size. To test this, a text input file containing 12 square matrix couples was generated, with nine of theses matrix couples having a size of a power of two, and three of them not, therefore the latter were excluded from multiplication. For each matrix multiplication, the run time and number of multiplications of each method was recorded and can be seen in the following table:

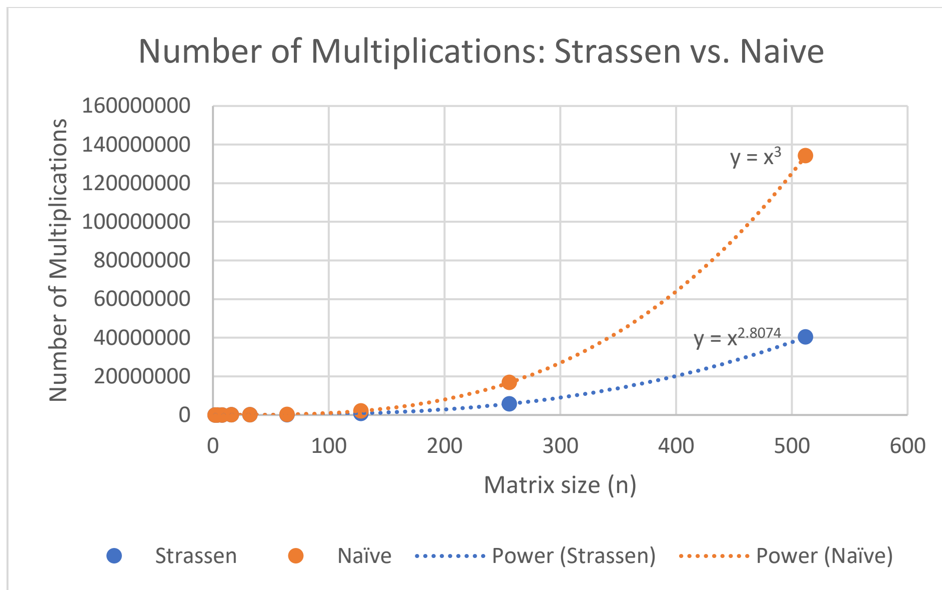
Matrices size (n)	**Observed Strassen's method run time (sec)**	**Observed naïve method run time (sec)**	**# multiplications for Strassen's method**	**# multiplications for naïve method**
2	6.81e-05	1.86e-05	7	8
4	4.72e-04	9.26e-05	49	64
8	2.56e-03	6.36e-04	343	512
16	2.04e-02	4.20e-03	2401	4096
32	1.10e-01	3.05e-02	16807	32768
64	1.20e+00	4.84e-01	117649	262144
128	7.82e+00	2.15e+00	823543	2097152
256	7.46e+01	2.65e+01	5764801	16777216
512	4.56e+02	1.79e+02	40353607	134217728

Table 1: Shows run time of Strassen's method and naïve matrix multiplication as well as number of multiplications conducted by each method given the size (n) of two square matrices as input.

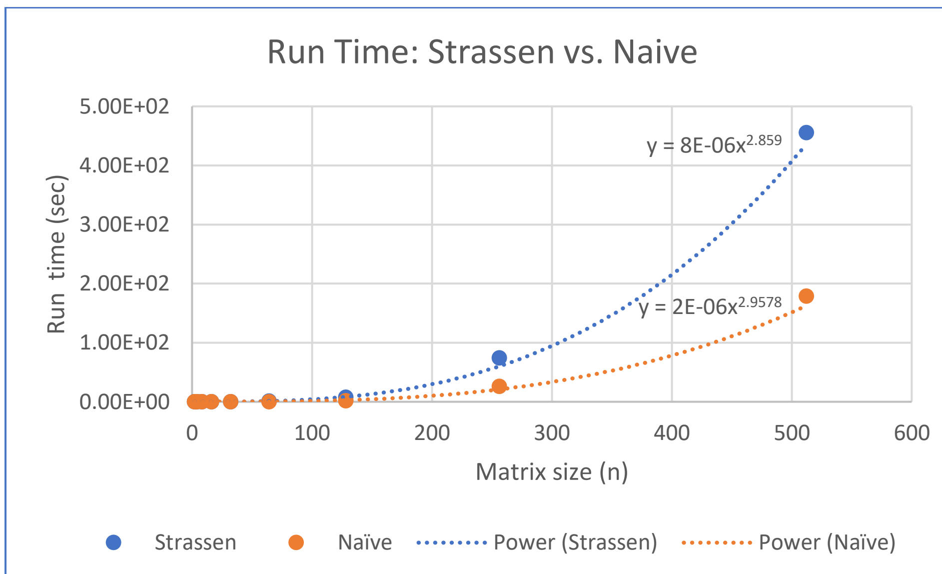
As can be observed in table 1, a significant difference between observed Strassen method and naïve method run times can be seen. Theoretically it was assumed that as the size of the matrix increased, Strassen's method would become more efficient. However, as can be seen in table 1, this was not observed, with the naïve method retaining its lead in efficiency for all cases tested. The reason for this is likely due to the recursion calls that Strassen's method implements. As the size of the matrices increases, so does the number of recursion calls, thus increasing the recursion depth, as described above. Even though the time complexity for Strassen's method is theoretically smaller than the naïve matrix multiplication method, the experimental run times seen in table 1 do not support this claim. Additionally, it can be observed that Strassen's method requires fewer multiplications than the naïve method, as the resulting number of multiplications follows the asymptotic upper bounds of each, that is

n^{lg7} and n^3, respectively. However, Strassen's method still runs slower, again likely attributed to the number of recursion calls that must be performed compared to the iterative nature of the naïve method.

These findings are also represented in the following graphs:



Graph 1: Number of multiplications required for Strassen's method (blue line) and naïve matrix multiplication (orange line) for a matrix size of size n.



Graph 2: Run time of Strassen's method (blue line) and naïve matrix multiplication (orange line) given in seconds for a matrix of size n.

Above, graph 1 is a visual representation of the observed number of multiplications performed to implement Strassen's method of matrix multiplication and the number of multiplications performed to implement naïve matrix multiplication for two square matrices of size n (as recorded in table 1). Notice in graph 1 that the equation for the trendline of Strassen's method in (blue) and of the naïve method (orange) match the asymptotic upper bounds given above. Graph 1 would suggest that Strassen's method is favorable in that it performs fewer multiplications.

Next, graph 2 is a visualization of the observed run times of Strassen's method and naïve matrix multiplication on two square matrices of size n. Observe that in graph 2, Strassen's method (blue) has a notably increased run time for all values of n tested (see table 1) compared to that the naïve method (orange). Given this, graph 2 would suggest that the naïve method is more efficient and therefore more favorable compared to Strassen's method. Also note that the associated equations for each trendline, constants aside, are very similar to their respective asymptotic time complexities (see above). Again, this observed discrepancy in comparison to the theoretical outcome is likely attributed to the time required for recursion, even though Strassen's method performs fewer multiplications.

Bioinformatics Application

Common examples of matrix usage in bioinformatics include protein sequence analysis and alignment and phylogenetic studies. In protein sequence analysis, 20 x 20 amino acid substitution matrices can be employed to determine the rate of substitution of a specific amino acid in a protein over time (Trivedi & Nagarajaram, 2020). The nature of the protein (i.e. a transmembrane protein or cytosolic protein, etc.) dictates which amino acids are more likely to be substituted and which are more likely to remain unchanged due to some evolutionary functional role (Trivedi & Nagarajaram, 2020). Additionally, using a substitution matrix for protein sequence analysis can give insight into specific amino acid enrichment of certain protein families (Trivedi & Nagarajaram, 2020).

Given this, substitution matrices used should vary to correlate with the family of protein being studied, if possible (Trivedi & Nagarajaram, 2020). There are various types of amino acid substitution matrices, where certain matrices, as stated above, are better suited for specific protein types, like G protein-coupled receptor transmembrane substitution matrix (GPCRtm) or beta-barrel Transmembrane Matrices (bbTMs) (Trivedi & Nagarajaram, 2020). However, the two most commonly used are: BLOcks SUbstitution Matrix (BLOSUM) and Point Accepted Mutation (PAM) (Trivedi & Nagarajaram, 2020).

The following is an example of BLOSUM62 amino acid substitution matrix:

Hazelyn Cates
9/19/22
EN.605.620.81.FA22

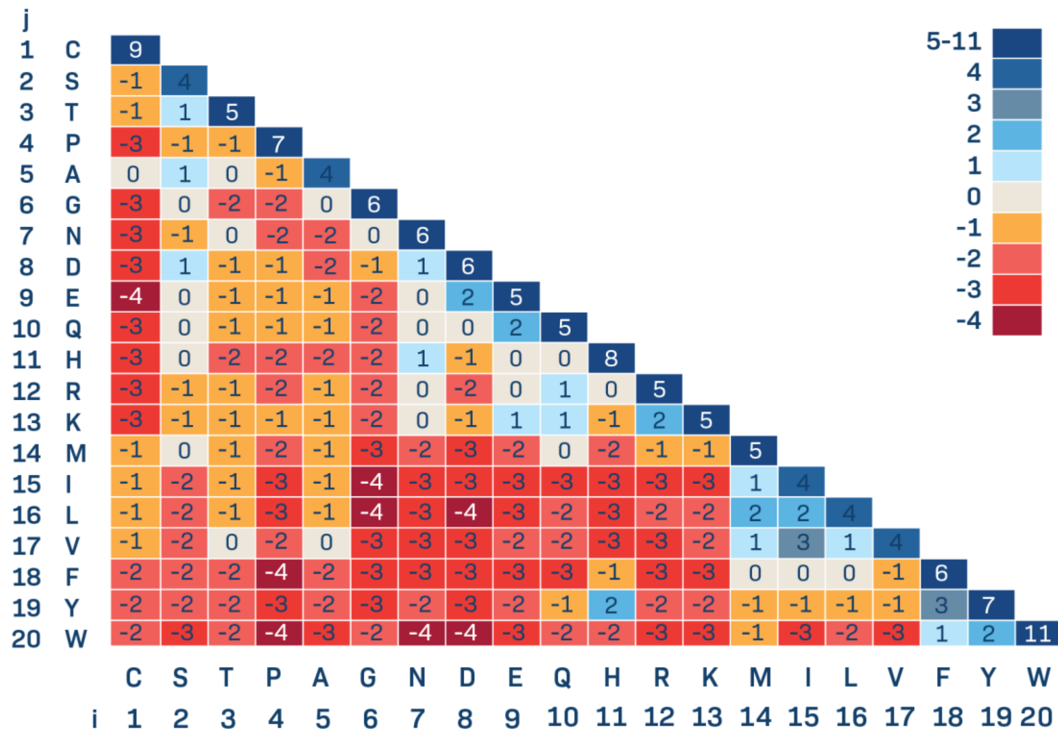


Fig. 1: BLOSUM62 matrix ("BLOSUM62", 2021)

In figure 1, the values ranging from red to blue represent the BLOSUM scores for the likelihood of an amino acid from row i being substituted with an amino acid from column j ("BLOSUM62", 2021). The higher the value, the more conserved that amino acid is and therefore less likely that it will be substituted, which is demonstrated by the fact that the values in dark blue diagonal represent the amino acids from row i being substituted with itself in column j ("BLOSUM62", 2021). In figure 1, this is referred to as a BLOSUM62 matrix due to the fact that, based on sequence alignments, no alignment pairs share identity over 62% ("BLOSUM62", 2021). Other BLOSUM matrices include BLOSUM45 and BLOSUM50 (utilized when sequences are distantly related), and BLOSUM80 and BLOSUM90, which are used for more closely related sequences (Pearson, 2013).

PAM matrices, compared to BLOSUM, are used on very closely related protein sequences, and give insight into the likelihood that an amino acid in a given protein sequence(s) was substituted via a point-accepted mutation ("Amino Acid Substitutions", n.d.). There are also various types of PAM matrices: PAM40, PAM60, and most commonly used, PAM250 ("Amino Acid Substitutions", n.d.). The number associated with each type of PAM matrix refers to the similarity of the sequences being aligned; higher value means farther apart evolutionarily and vice versa ("Amino Acid Substitutions", n.d.). Both PAM and BLOSUM can be used as a scoring matrix when performing a protein BLAST.

Beyond protein phylogenetic analysis and alignment, matrices are also utilized in DNA microarray analysis. A paper from 2010 described five methods of matrix factorization (i.e. matrix decomposition) in order to determine genes that are regulated together in multiple biological processes

(Kossenkov & Ochs, 2010). Such methods could allow for further elucidation of biological markers and further understanding of gene regulation in a given organism (Kossenkov & Ochs, 2010).

Additionally, a paper from 2004 presented a Monte-Carlo type randomized algorithm for identifying Boolean networks, which is a way to model dynamic and complex gene regulatory interactions in a qualitative manner (Leifeld et al, 2018; Akutsu et al, 2004). The algorithm described in the paper utilizes fast matrix multiplication plus string matching via randomized fingerprint function, the latter which maps large data to shorter byte sequences and is commonly used for extracting unique information from large amounts of data (Akutsu et al, 2004; "Fingerprinting Algorithms", 2020). This approach aids in profiling gene expression from microarray data, as mentioned above (Akutsu et al, 2004). The ability to profile gene expression in a high-throughput way has many implications, for example in cancer research, where the gene expression profile of a given patient can prove vital in diagnosis, understanding the etiology of the disease and provide a personalized view of that patient's specific case which could aid in treatment decisions.

Reflection

This project aided in my understanding of matrix multiplication and in the use and implementation of recursive functions. Additionally, this project helped to enhance my handling of file input and output and utilizing different libraries like numpy, re, time, and math. It also enhanced my problem-solving skills and made me think critically about how to solve the given problem, because as I've come to find, a lot of success in programming comes from knowing what to look for. Additionally, this project helped my understanding of the time complexities that have been presented in the course so far, as seeing the programs implemented and comparing observed results versus those expected made a rather abstract concept more tangible.

Given the end result, I would not do anything differently. Strassen's method was successfully implemented as was the naïve multiplication method. However, one of the bigger difficulties in this project was the file input. The main issue was figuring out how to read in all the matrices from a single file and making sure that matrices within the file met all requirements prior to implementation (that is, the matrices were square matrices of a power of two). And if any did not, to skip them but continue reading through the file. Ultimately, this task was completed successfully after much trial and error. Additionally, making sure the recursion was executed successfully was a challenge and making sense of what the program was actually doing took time and a lot of handwritten notes.

An interesting aspect to this project that I was not initially familiar with was recording the time it takes for a program to run. To accomplish this, the time library and the time.perf_counter() function was utilized. This project also gave an opportunity to use regex, which I became familiar with in a prior class. Overall, this project was challenging but ultimately, I believe it improved my skills as a programmer and helped drive home how important it is to understand how an algorithm runs, beyond just the code in the IDE.

Hazelyn Cates
9/19/22
EN.605.620.81.FA22

References:

Akutsu, Tatsuya et al. (2004). Algorithms for Identifying Boolean Networks and Related Biological Networks Based on Matrix Multiplication and Fingerprint Function. *Journal of Computational Biology, 7*(3-4), 331-343. http://doi.org/10.1089/106652700750050817

Amino Acid Substitutions & Replacement Matrices. (n.d.). Proteinstructures.com. https://proteinstructures.com/sequence/amino-acid-substitutions/

BLOSUM62 Substitution Matrix. (2021). LabXchange. https://www.labxchange.org/library/items/lb:LabXchange:24d0ec21:lx_image:1?source=%2Flibrary%2Fclusters%2Flx-cluster%3AIntroBio

Cormen, H. Thomas et al. (2009). Introduction to Algorithms. 3rd ed. The MIT Press.

Divide and Conquer | Set 5 (Strassen's Matrix Multiplication). (2022). GeeksforGeeks. https://www.geeksforgeeks.org/strassens-matrix-multiplication/

Fingerprinting Algorithms. (2020). Devopedia. https://devopedia.org/fingerprinting-algorithms

How to Analyse Loops for Complexity Analysis of Algorithms. (2022). GeeksforGeeks. https://www.geeksforgeeks.org/analysis-of-algorithms-set-4-analysis-of-loops/#:~:text=(Log%20n)%3A-,The%20time%20Complexity%20of%20a%20loop%20is%20considered%20as%20O,considered%20as%20O(Logn)

Kossenkov, Andrew V & Michael F Ochs. (2010). Matrix factorisation methods applied in microarray data analysis. International Journal of Data Mining and Bioinformatics, 4(1), 72-90. 10.1504/ijdmb.2010.030968

Leifeld, Thomas et al. (2018). Identification of Boolean Network Models From Time Series Data Incorporating Prior Knowledge. Frontiers in Physiology, 9. 10.3389/fphys.2018.00695

More, Nilesh. (n.d.). Time and Space Complexity of Recursive Algorithms. IDeserve. https://www.ideserve.co.in/learn/time-and-space-complexity-of-recursive-algorithms#:~:text=To%20conclude%2C%20space%20complexity%20of,would%20be%20O(nm)

Pearson, William R. (2013). Selecting the Right Similarity-Scoring Matrix. Current Protocols in Bioinformatics, 43, 3.5.1-3.5.9. 10.1002/0471250953.bi0305s43

Python program to multiply two matrices. (2022). GeeksforGeeks. https://www.geeksforgeeks.org/python-program-multiply-two-matrices/

Time Complexity and Space Complexity. (2022). GeeksforGeeks. https://www.geeksforgeeks.org/time-complexity-and-space-complexity/

Trivedi, Rakesh & Hampapathalu Adimurthy Nagarajaram. (2020). Substitution scoring matrices for proteins - An overview. Protein Science, 29(11), 2150-2163. 10.1002/pro.3954