Jacob Hoover

Programming Project 1: Analysis

Introduction

Matrix multiplication is a common problem in various Bioinformatics applications and requires algorithms with efficient time and space allocation. This analysis will demonstrate a naive matrix multiplication algorithm versus Strassen's algorithm which invokes a recursive strategy. The former takes two $n \; x \; n$ matrices as input and determines the product using three nested for loops. The latter partitions or splits the input matrices into small subproblems and then recursively solves the matrix product for each ultimately rejoining them to produce the output matrix. Further discussed in the analysis below, the benefit of the Strassen method is to reduce the time complexity on large datasets commonly seen in Bioinformatics. A caveat is that the submatrices created in the algorithm require additional temporary space and the functions of splitting and recurring can increase runtimes on smaller datasets. The discussion below will explore these complexities for both algorithms and analyze how the Strassen algorithm's overhead may be overcome as the size of data grows.

Data Structures, Design, Implementation

The program design employed the use of two data structures, 2D arrays to hold the matrices and lists to store the matrices. A 2D array is a common method in Java to represent matrix data and is easy to both implement and call operations upon. These structures are used as input for various functions throughout the program and subsequently used to produce the product result of the algorithms. A second data structure, lists, were used to store the matrices read in from the input file. In FileHandler, Matrix A and Matrix B are added to the matrix list as each order of matrix is read in from the file. This list of 2D arrays is then iterated over in the Main file to retrieve the matrices, perform the various functions, and write the results to an output file.

I designed the program to implement each algorithm, Naive and Strassen, in separate class files so the code was modular and organized. The implementation of each is supported with simple functions to reset and maintain the comparison counter for each order of matrix in the input file. The Naive algorithm is simple to execute and does not require supporting functions. The Strassen algorithm is supported with functions to split the matrices (divide), rejoin the matrices (conquer), and perform the addition and subtraction steps on the S-matrices. The last design choice with Strassen was to create the *multiply* function that wraps the Strassen implementation so that the call in Main was simplified and mimicked the same call for the Naive algorithm (A, B). The FileHandler class receives the input file, parses through the rows handling whitespace and empty lines, and then reads in the matrices as 2D arrays and inputs them in the matrices list. Input/Output and NumberFormat errors are incorporated for error handling between the *readFromFile* and *readMatrix* functions. The class then writes the results to the output file using the *writeMatrix* and *writeResultsToFile* functions. The modularity of this code within the class allows the Main class to be organized and simple to understand.

Time and Space Efficiency

The naive matrix multiplication algorithm consists of six lines of code with three nested loops. Each loop requires n iterations taking $O(n)$ time. Lines 3, 5, and 6 all operate in constant time $O(1)$. This creates a theoretical running time of $\Theta(n^3)$ for this algorithm. The Strassen algorithm attempts to achieve a lower time complexity by dividing the matrices into subproblems, recursively computing the matrix products, and rejoining the submatrices into the final matrix product. In doing so, the method cleverly eliminates a single multiplication from the recursion for an increase in constant time addition/subtraction operations leading to a theoretical reduction in time complexity. The recurrence relationships for Strassen is $T(n) = 7T(n/2) + \Theta(n^2)$ where 7 represents the multiplications, $n/2$ the subproblems, and $\Theta(n^2)$ for the creation of the submatrices and addition or subtraction calculations to derive the final matrix product. The recurrence has a base case when $n = 1$operating in constant time. Using the master method, the solution to the Strassen recurrence is $T = O(n^{lg7}) = O(n^{2.81})$ thus theoretically outperforming the Naive matrix algorithm in time efficiency.

Master Method - Case 1

$$n^{log_2 7} > n^2$$
$$= O(n^{lg7})$$

The Naive algorithm requires space for three matrices (A, B, C) of size n. Thus the space complexity for this approach is $Space\ Complexity = O(n^2) + O(n^2) + O(n^2) = O(n^2)$. Comparing this to the Strassen algorithm, this method does require additional space due to the creation of $n/2\ x\ n/2$ submatrices and subsequent creation of 10 S-matrices. Although, because these are temporarily stored and smaller than the storage of the input and output matrices the space complexity for Strassen remains the same as Naive at $O(n^2)$.

The theoretical efficiency of both algorithms are shown in Figure 2 where it is obvious that the Strassen algorithm should greatly outperform the Naive approach beginning at n near 50. Although, analyzing the result data in Table 1 and Figure 1 it is evident that the runtime performance of Strassen did not mirror the theoretical performance. The Naive approach greatly outperformed Strassen on the tested input matrices of size $n = 2$ to $n = 64$. Despite the matrices growing in size the Naive algorithm was faster by a factor of 14 even for the $64\ x\ 64$ matrices. This is likely due to the operations that the Strassen algorithm sacrifices in order to reduce the number of multiplications from 8 to 7. Albeit constant, the addition and subtraction steps invoke a greater runtime. Additionally the steps of dividing, recursively solving, and conquering the matrices creates the overhead of complexity and increased runtime of the Strassen algorithm. This indicates that to observe the theoretical efficiency seen in Figure 2 the algorithm requires a significant input size for n. The potential of Strassen surpassing the runtime of the Naive algorithm is evident in both Table 1 and Figure 3 due to the number of comparisons made by the Strassen algorithm being considerably less. There were less than half of the number of multiplications in the Strassen algorithm when performing matrix multiplication on the $64\ x\ 64$

matrices. As the dataset continues to grow the reduction in comparisons will benefit the time complexity and should show Strassen's efficiency over the Naive approach.

Size (n)	Naive Comparisons	Strassen Comparisons	Naive Time (ns)	Strassen Time (ns)
2	8	7	3600	4700
4	64	49	4700	58800
8	512	343	23500	442700
16	4096	2401	183100	2323600
32	32768	16807	1522500	8068800
64	262144	117649	3340900	46946300

Table 1. Input results for $n\ x\ n$ matrix products, number of comparisons, and runtime using Naive vs. Strassen.

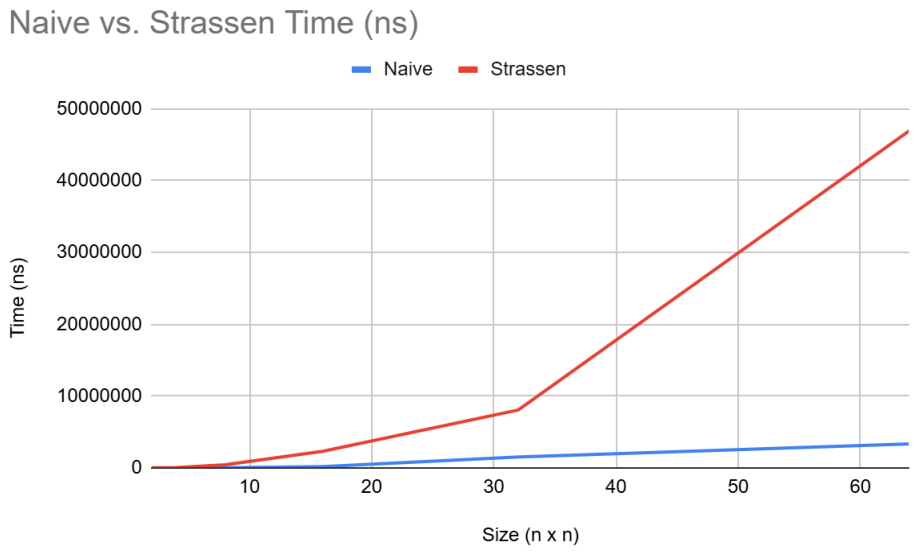

Figure 1. Runtime of Naive (blue) and Strassen (Red) on $n\ x\ n$ matrices.

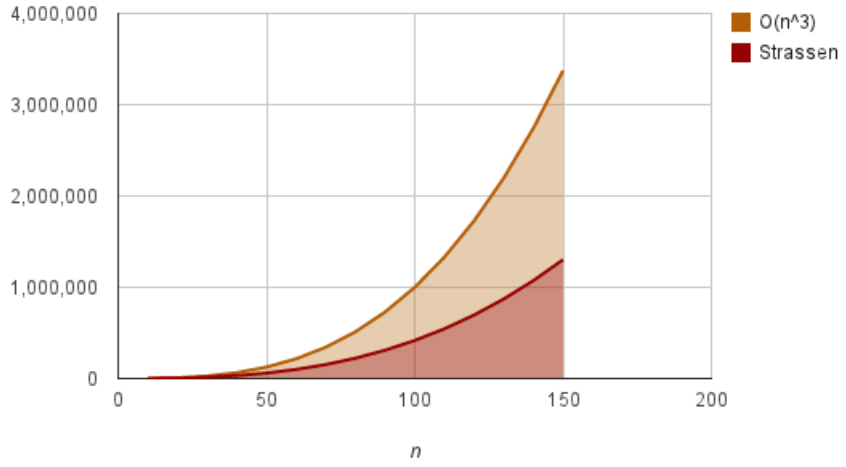

Figure 2. Theoretical runtime of Naive (orange) versus Strassen (red) on $n\ x\ n$ matrices (Stoimenov, 2012).

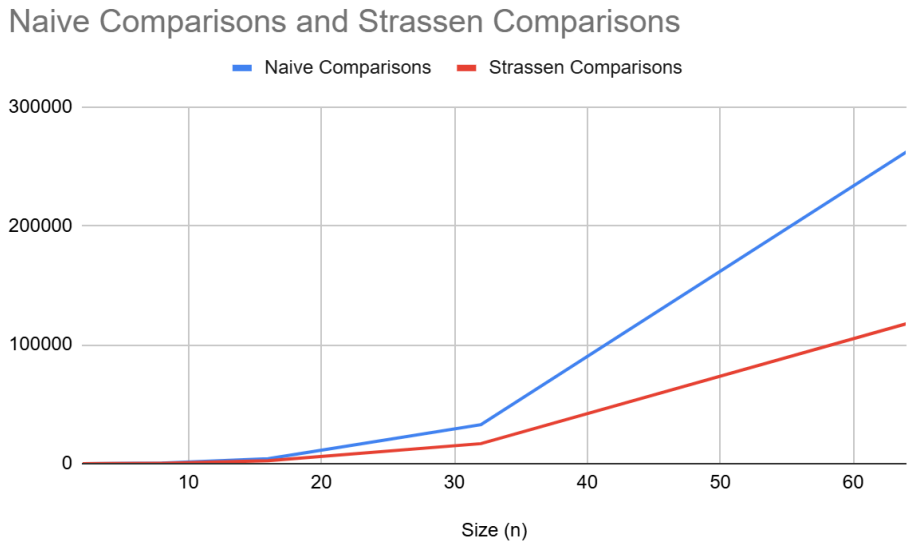

Figure 3. Number of comparisons (multiplications) for Naive (Blue) versus Strassen (red) on $n\ x\ n$ matrices.

Learning, Changes, Enhancements, and Applicability to Bioinformatics

Through working on this project I have learned a multitude of useful concepts and Java syntax that should greatly help me through the rest of the course. The biggest learning experience in this project is that just because the time complexity of Strassen should be better than the Naive algorithm theoretically, that doesn't mean it always will be in application. This key concept is applicable to other algorithms and highlights the importance for the programmer to identify and determine the correct application of an algorithm based on the use case. Additionally, I continued to learn of the importance and value of print statements for debugging as they greatly helped me identify a few errors in my program. A change I would make next time would be testing code more frequently as I develop the program. Two small items I learned about and had not used before was the System class for runtime

output and using bitwise operations as a method for verifying matrices are a power of 2. A few other ideas for items to implement would be code to generate a set of matrices for input testing to further build out the asymptotic costs of both algorithms. Also, building the program so that it can handle matrices of different sizes. Enhancements regarding the Strassen algorithm could consist of reducing the space and time of the additional operations so that the overhead cost of saving the single multiplication does not affect runtime on smaller datasets as drastically.

Matrix multiplication is used in many different Bioinformatics applications and commonly functions as a key operation in larger computational methods. In relation to my own experiences. I can assume it is extremely important in aspects of sequence alignments with regards to PAM and BLOSUM substitution scoring matrices. These matrices, used in algorithms like the Smith-Waterman and Needleman-Wunsch, use matrices to determine the impact of an amino acid substitution when building alignment scores. In my Protein Bioinformatics course we learned about the use of PDB files to create structure alignments, predict protein secondary structure, and many other structural and functional properties. This data stored in these files would use algorithmic operations like matrix multiplication for modeling aspects like molecular dynamics, homology modeling, folding, etc. A last example would be gene expression data which is often represented as a matrix and would apply efficient algorithms on big data to determine trends in gene expression across many samples.

A few enhancements I included in the program were error handling examples for; if the file didn't exist, skips empty lines, validates matrix size, identifies invalid integers with their location, not a power of 2, and if the matrix is missing rows or values. The command line produces statements for the matrices as they are read along with their size so that if there is an error it is simple to identify the location in the input file. Once the program finishes the command line outputs, "Results written to <outputfile.txt>".

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms*. The MIT Press. Publication date: April 5, 2022.ISBN-13: 978-0262046305

Stoimenov, S. (2012, November 26). *Computer algorithms: Strassen's matrix multiplication.* Stoimen.com. http://stoimen.com/2012/11/26/computer-algorithms-strassens-matrix-multiplication/