Analysis
Programming Assignment 1: Strassen's Algorithm
Kevin Feng Yong Chen
605.620 - Algorithm for Bioinformatics

Description of the program

This program uses 2d arrays to store the squared matrices. In particular, numpy.array was used to store each matrix. NumPy was used because of its many built-in functions. For example, it allows for vectorized addition and subtraction which makes the computations of p1 to p7 within the Strassen's algorithm much easier. It also allows for concatenation of submatrices. This is another useful feature that makes coding very efficient when combining submatrices. Further, a lot of the code in NumPy was written in C which makes it faster than using a Python list. Thus, NumPy is an appropriate and efficient tool for this assignment.

A total of 4 functions are written in this program. The main function was used to handle the input/output stream as well as invalid inputs from the input file. This program stops when it encounters an invalid input. An alternative approach to handling error would be to skip the invalid matrix and continue with the calculation of the next matrix in the input file. However, since there are numerous ways for an input to be invalid, I believe that it is simpler and more efficient to stop the program and print a message to ask the user to provide the appropriate inputs as opposed to trying to code for the numerous possible cases in which an input could go wrong.

Strassen's algorithm and the ordinary matrix multiplication were each implemented as a separate function. This design is appropriate because it allows me to test each function separately and make sure that they work properly before moving on to the other parts of the program. Keeping these methods modular also makes my code more readable to reviewers and easier to understand as each module includes a detailed description of the module. Strassen's algorithm had to be implemented using recursion due to the nature of the algorithm. The ordinary matrix multiplication was implemented using an iterative approach because this approach saves more space compared to a recursive approach. More on the runtime and space complexity of these 2 methods will be discussed in the next section.

A smaller functions,log2, was implemented so that my main function won't be too long. By keeping my main function short and concise, my code will be easier to understand to reviewers. Another reason why I want to write separate functions for different features is because I want to reuse my code throughout the program without having to write them every time.

Efficiency

The number of multiplications were compared between Strassen's algorithm and the ordinary matrix multiplication. We will first discuss the runtime complexity of these 2 methods. Before we look at the data from this lab, let's first discuss their asymptotic complexity. An ordinary matrix multiplication implemented using an iterative approach will give you $\theta(n^3)$ run time. This is because an iterative approach would require a triply-nested for loop. As shown in section 4.2 of the textbook, given 2 square matrices A and B both with dimensions n x n, each entry in the product matrix, C, is defined as:

$$\sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

To get entry C_{ij}, we need to compute the dot product of the ith row and the jth column from A and B, respectively. When calculating C, the outermost for loop specifies the ith row of C, the 2nd outermost for loop specifies the jth column of C and the innermost for loop computes the entry at C_{ij} by calculating the

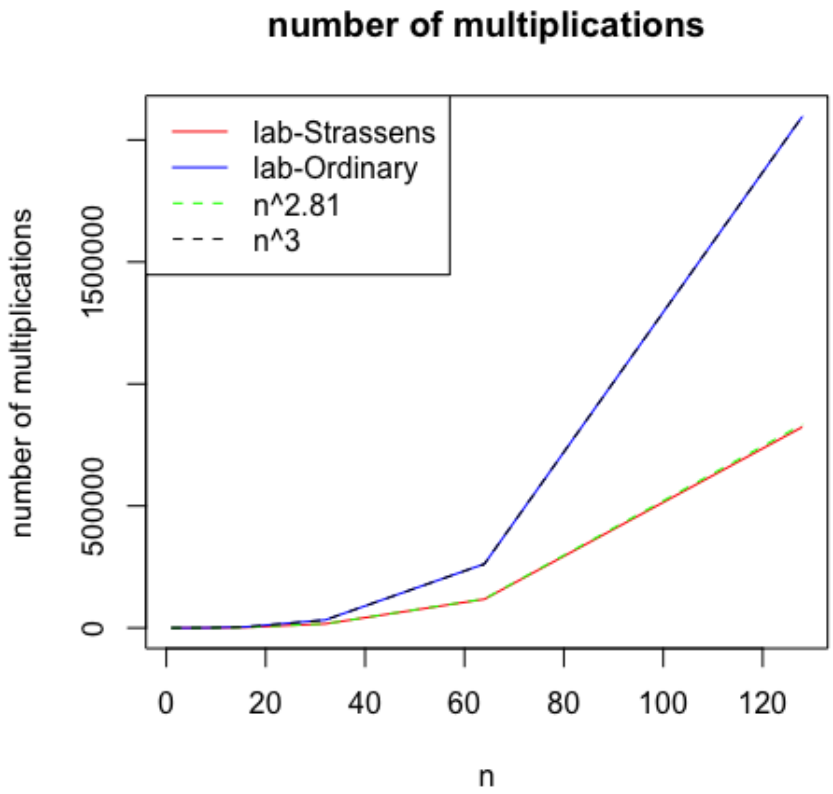
dot product of vector i and j from matrix A and B, respectively. Each of the for loop iterates n times and so the runtime of this algorithm is $\theta(n^3)$.

The runtime complexity of Strassen's algorithm could be understood using the Master's theorem. As stated in section 4.2 of the text book, the Strassen's algorithm makes 7 recursive calls at each level of the recursive tree. $2\frac{n}{2} * \frac{n}{2}$ matrices were passed on to each recursive call. Within each recursion, a constant number of additions and subtractions of $\frac{n}{2} * \frac{n}{2}$ matrices are performed. Thus Strassen's algorithm can be represented by the following recurrence:

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

Based on the Master's theorem, a=7, b=2 and f(n)=n^2. Thus, $n^{\log_2 7} > n^2$ and this belongs to case 1. The runtime complexity should therefore be $n^{\log_2 7} = n^{\sim 2.81}$. As we shall see from the data in this lab, the number of multiplications is very consistent with the theoretical runtime of the 2 methods.

n	Strassen - lab	Ordinary - lab	n^2.81	n^3
1	1	1	1	1
2	7	8	7.01284577052828	8
4	49	64	49.1800058012164	64
8	343	512	344.891795677617	512
16	2401	4096	2418.67297060768	4096
32	16807	32768	16961.7805122171	32768
64	117649	262144	118950.350725731	262144
128	823543	2097152	834180.463989798	2097152



The picture above shows the number of multiplications recorded from the lab (in solid lines) and the theoretical runtime for each method (in dash). As indicated, the number of multiplications are very consistent with their theoretical speed.

Next we will look at the space complexity of the Strassen's algorithm. In each recursion, computing s1 to s10, p1 to p7 and the final product C requires $O(n^2)$ space. In each of its subsequent recursive calls, we allocate enough space for a constant number of $^{n}/_{2} \times {}^{n}/_{2}$ matrices. Thus, we can use the same recurrence we used for time complexity and apply master's theorem to get a solution of $n^{2.81}$. On the other hand, the iterative approach of ordinary matrix multiplication has a better space complexity. Because it does not use recursion, it only allocates an extra n^2 space for the result and some additional space to store a constant number of integers and pointers. Thus it's space complexity should be $O(n^2)$.

The space complexity of the iterative approach is therefore better than the Strassen's algorithm. However, space and storage are relatively cheap in modern days, especially with the advancement in cloud computing and high performance computer clusters. It is relatively easy to allocate a large amount of memory and storage using cloud or a computer cluster. Hence, one could argue that the runtime complexity of an algorithm is more important compared to its space requirement. Another advantage that the ordinary iterative approach may have over Strassen's algorithm is that it requires relatively fewer lines of code. The iterative approach took a total of 10 lines to implement. On the other hand, the Strassen's algorithm took about 44 lines of code to implement. But if the goal is to optimize speed then the Strassen's algorithm should be chosen over the ordinary approach.

What I learned

Overall I find this to be a very meaningful lab as it helped me solidify the concept of recursion and it improved my understanding of the Strassen's algorithm. This assignment helped me better understand

recurrence and each of its components because I had to implement each component of the equation including the base case, the recursive calls and the work that is done within the recursion (i.e. combining the solutions from the sub problems into the solution for the original problem). Writing this assignment in python has improved my understanding of this programming language. In particular, I learned how to handle input/output in python using the argparse library. I also learned how to store and process 2D matrices using NumPy and its functions. NumPy is an important tool in the field of data science and so it is important for a bioinformatician to know how to use it. Through email communication with Dr. Chlan, I have also learned about programming best practices such as avoid using global variables in general unless if it is well justified. Another important thing that I learned from this assignment was how to handle errors when reading in and processing matrices. For example, I learned to use the try-except clause to cast inputs into integers while also handling potential value errors when I am only expecting a specific type of input such as integers.

Enhancement

In addition to the given inputs for the assignment, I have generated 30 additional matrices with orders ranging from 2^0 to 2^7 and values within the matrices range from -300 to 300. I have also generated 7 test cases to test different kinds of errors. I believe extensive testing will allow me to confidently conclude that my script will work regardless of the type of erroneous input. Further, since the results from the iterative approach and the results from the Strassen's algorithm are the same, I could be confident that my implementation of the Strassen's algorithm is correct. Only results from the Strassen's algorithm were shown in the output, results from the iterative approach were not shown. Lastly, I consider the use of NumPy as a form of enhancement since it processes 2D arrays faster than an ordinary 2D list in python.

Room for improvement

Since the Strassen's algorithm was implemented using recursion, it naturally requires more computational resources compared to an iterative approach. This is apparent when comparing the space complexity between the Strassen's algorithm and the iterative approach. Although not done in this assignment, the space requirement of Strassen's algorithm could be reduced by adding a few more features. For example, after computing p1 to p7, s1 to s10 are no longer needed and so we could delete these objects to save space. Similarly, after computing c11 – c22, p1 – p7 are no longer needed and they could be deleted. By routinely deleting objects that are not needed, the amount of memory required to run the recursion could be reduced.

Applicability to Bioinformatics

Matrix multiplication can be applied to many fields including bioinformatics. One example where matrix multiplication is applied is in the construction of the point accepted mutation (PAM) matrix. PAM indicates the replacement of a single amino acid in the protein sequence with another amino acid that is accepted by natural selection. PAM tables are often used in bioinformatics as substitution matrices to score the similarity between 2 sequences. The similarity between 2 sequences depends on the their point of divergence in evolution. 2 genes that diverged 10 thousand years ago should be measure using a different standard compared to genes that diverged more recently. PAM tables are constructed by taking sequence that are closely related to each other and measure the rate of substitution for each amino acid in the sequence. The resulting matrix gives you the probability that an amino acid is substituted by another amino acid. You can then extrapolate the rate of substitution of amino acids between more distant sequences by multiplying the matrix by itself. For example, a PAM 250 takes the original PAM matrix (PAM1) and multiply it by itself 250 times so that more distant sequences can be compared during sequence alignment.