CS 124 Programming Assignment 2: Spring 2023

Your name(s) (up to two): James Bardin, Sal Blanco

Collaborators: (You shouldn't have any collaborators but the up-to-two of you, but tell us if you did.)

No. of late days used on previous psets: James: [AMOUNT], Sal: 10
No. of late days used after including this pset: James: [AMOUNT], Sal: 10

Homework is due Wednesday 2023-03-29 at 11:59pm ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two** (college)/**four** (extension school).

Overview:

Strassen's divide and conquer matrix multiplication algorithm for n by n matrices is asymptotically faster than the conventional $O(n^3)$ algorithm. This means that for sufficiently large values of n, Strassen's algorithm will run faster than the conventional algorithm. For small values of n, however, the conventional algorithm may be faster. Indeed, the textbook Algorithms in C (1990 edition) suggests that n would have to be in the thousands before offering an improvement to standard multiplication, and "Thus the algorithm is a theoretical, not practical, contribution." Here we test this armchair analysis.

Here is a key point, though (for any recursive algorithm!). Since Strassen's algorithm is a recursive algorithm, at some point in the recursion, once the matrices are small enough, we may want to switch from recursively calling Strassen's algorithm and just do a conventional matrix multiplication. That is, the proper way to do Strassen's algorithm is to not recurse all the way down to a "base case" of a 1 by 1 matrix, but to switch earlier and use conventional matrix multiplication. That is, there's no reason to do a "base case" of a 1 by 1 matrix; it might be faster to use a larger-sized base case, as conventional matrix multiplication might be faster up to some reasonable size. Let us define the *cross-over point* between the two algorithms to be the value of n for which we want to stop using Strassen's algorithm and switch to conventional matrix multiplication. The goal of this assignment is to implement the conventional algorithm and Strassen's algorithm and to determine their cross-over point, both analytically and experimentally. One important factor our simple analysis will not take into account is memory management, which may significantly affect the speed of your implementation.

Tasks:

1. Assume that the cost of any single arithmetic operation (adding, subtracting, multiplying, or dividing two real numbers) is 1, and that all other operations are free. Consider the following variant of Strassen's algorithm: to multiply two n by n matrices, start using Strassen's algorithm, but stop the recursion at some size n_0, and use the conventional algorithm below that point. You have to find a suitable value for n_0 – the cross-over point. Analytically determine the value of n_0 that optimizes the running time of this algorithm in this model. (That is, solve the appropriate equations, somehow, numerically.) This gives a crude estimate for the cross-over point between Strassen's algorithm and the standard matrix multiplication algorithm.

2. Implement your variant of Strassen's algorithm and the standard matrix multiplication algorithm to find the cross-over point experimentally. Experimentally optimize for n_0 and compare the experimental results with your estimate from above. Make both implementations as efficient as possible. The actual cross-over point, which you would like to make as small as possible, will depend on how effi-

ciently you implement Strassen's algorithm. Your implementation should work for any size matrices, not just those whose dimensions are a power of 2.

To test your algorithm, you might try matrices where each entry is randomly selected to be 0 or 1; similarly, you might try matrices where each entry is randomly selected to be 0, 1 or 2, or instead 0, 1, or -1. We will test on integer matrices, possibly of this form. (You may assume integer inputs.) You need not try all of these, but do test your algorithm adequately.

3. Triangle in random graphs: Recall that you can represent the adjacency matrix of a graph by a matrix A. Consider an undirected graph. It turns out that A^3 can be used to determine the number of triangles in a graph: the (ij)th entry in the matrix A^2 counts the paths from i to j of lenth two, and the (ij)th entry in the matrix A^3 counts the path from i to j of length 3. To count the number of triangles in in graph, we can simply add the entries in the diagonal, and divide by 6. This is because the jth diagonal entry counts the number of paths of length 3 from j to j. Each such path is a triangle, and each triangle is counted 6 times (for each of the vertices in the triangle, it is counted once in each direction).

   Create a random graph on 1024 vertices where each edge is included with probability p for each of the following values of p: $p = 0.01, 0.02, 0.03, 0.04$, and 0.05. Use your (Strassen's) matrix multiplication code to count the number of triangles in each of these graphs, and compare it to the expected number of triangles, which is $\binom{1024}{3}p^3$. Create a chart showing your results compared to the epectation.

Code setup:

60% of the score for problem set 2 is determined by an autograder. You can submit code to the autograder on Gradescope repeatedly; only your latest submission will determine your final grade. We support the following programming languages: Python3, C++, C, Java, Go; if you want to use another language, please contact us about it.

Option 1: Single-source file:

In this option, you can submit a single source file. Please make sure to NOT submit a makefile/Makefile if you elect to use this option, as it confuses the autograder. Please ensure that you have exactly one of the following files in your directory if you choose to use this option:

1. strassen.py - for python. In this case, we will run

   python3 strassen.py <args>

2. strassen.c - for C. In this case, we will run

   gcc -std=c11 -O2 -Wall -Wextra strassen.c -o strassen -lm -lpthread

   ./strassen <args>

3. strassen.cpp - for C++. In this case, we will run

   g++ -std=c++17 -O2 -Wall -Wextra strassen.cpp -o strassen -lm -lpthread

   ./strassen <args>

4. strassen.java / Strassen.java - for Java. In this case, we will run

   javac strassen.java (javac Strassen.java)

   java -ea strassen <args> (java -ea Strassen <args>)

5. strassen.go - for Go. In this case, we will run

   go build strassen.go

   go run strassen.go <args>

Option 2: Makefile:

In this option, you submit a makefile (either Makefile or makefile). In this case, the autograder first runs make. Then, it identifies the language and runs the corresponding command above.

Your code should take three arguments: a flag, a dimension, and an input file:

$ ./strassen 0 dimension inputfile

The flag 0 is meant to provide you some flexibility; you may use other values for your own testing, debugging, or extensions. The dimension, which we refer to henceforth as d, is the dimension of the matrix you are multiplying, so that 32 means you are multiplying two 32 by 32 matrices together. The inputfile is an ASCII file with $2d^2$ integer numbers, one per line, representing two matrices A and B; you are to find the product $AB = C$. The first integer number is matrix entry $a_{0,0}$, followed by $a_{0,1}, a_{0,2}, \ldots, a_{0,d-1}$; next comes $a_{1,0}, a_{1,1}$, and so on, for the first d^2 numbers. The next d^2 numbers are similar for matrix B.

Your program should put on standard output (in C: printf, cout, System.out, etc.) a list of the values of the *diagonal entries* $c_{0,0}, c_{1,1}, \ldots, c_{d-1,d-1}$, one per line, including a trailing newline. The output will be checked by a script – add no clutter. (You should not output the whole matrix, although of course all entries should be computed.)

The inputs we present will be small integers, but you should make sure your matrix multiplication can deal with results that are up to 32 bits.

Do not turn in an executable.

What to hand in:

As before, you may work in pairs, or by yourself. Hand in a project report (on paper) describing your analytical and experimental work (for example, carefully describe optimizations you made in your implementations). Be sure to discuss the results you obtain, and try to give explanations for what you observe. How low was your cross-over point? What difficulties arose? What types of matrices did you multiply, and does this choice matter?

Your grade will be based primarily on the correctness of your program, the crossover point you find, your interpretation of the data, and your discussion of the experiment.

Hints:

It is hard to make the conventional algorithm inefficient; however, you may get better caching performance by looping through the variables in the right order (really, try it!). For Strassen's algorithm:

- Avoid excessive memory allocation and deallocation. This requires some thinking.
- Avoid copying large blocks of data unnecessarily. This requires some thinking.
- Your implementation of Strassen's algorithm should work even when n is odd! This requires some additional work, and thinking. (One option is to pad with 0's; how can this be done most effectively?) However, you may want to first get it to work when n is a power of 2 – this will get you most of the credit – and then refine it to work for more general values of n.

Solution Writeup:

Part 1:

In part one, we are tasked with finding a suitable value for n_0, or the cross-over point. The cross-over point is when the runtime of Strassen's no long has an advantage over the standard multiplication algorithm's runtime. So, in Strassen's as we split into smaller and smaller subproblems, we want to find the dimensions of the matrix at which it is faster to use the standard matrix multiplication algorithm.

In the regular algorithm where A and B are two square matrices with dimensions n by n, to calculate the n^2 entries $C[i][j]$ we need to do n^3 multiplications because there are a total of n^2 elements in each matrix that need to be multiplied. Next, we know that there are n summations, ($c_{i,k} = \sum a_{i,j} b_{j,k}$) so we have $n-1$ additions per element. So, this gives us a total of $n^3 + (n-1) * n^2$ steps.

In the implementation of Strassen's algorithm, we use padding (column/row of zeros) to make sure that the matrix has even dimensions at each step. We have that there are seven matrices, $M_i, M_2, ..., M_7$ that are used in Strassens. And to build these and combine them, there is a total of 18 addition/subtraction operations used. So, we have the following recurrence, $T(n) = 7T(n/2) + 18(n^2)$

When we are running Strassen's we have two cases, 1: the matrix has even dimensions ($n = 2k$), and 2: the matrix has odd dimensions ($n = 2k + 1$).

So, in the case that n is an even number, we have that the multiplication of two $2k$ by $2k$ matrices takes this many operations:

$$2(2k)^3 - (2k)^2 = 16k^3 - 4k^2$$

And running Strassen's for this takes,

$$(7(2k)^2/4)(2k-1) + 9(2k)^2/2$$
$$14k^3 + 11k^2$$

So, we have the following equality and we solve for 2k and find:

$$14k^3 + 11k^2 \leq 16k^3 - 4k^2$$
$$2k \geq 15$$

So, this gives us a cross over point of $n_0 = 15$

Next, in the case that n is an odd number, we have that the multiplication of two $2k+1$ by $2k+1$ matrices, using the normal mulitplication algorithm takes this many arithmetic operations:

$$2(2k+1)^3 - (2k+1)^2 = 16k^3 + 20k^2 + 8k + 1$$

And, when running Strassen, we ahve that the problem takes

$$7(2(k+1)^3 - (k+1)^2) + 19(k+1)^2 = 14k^3 + 53k^2 + 64k + 25$$

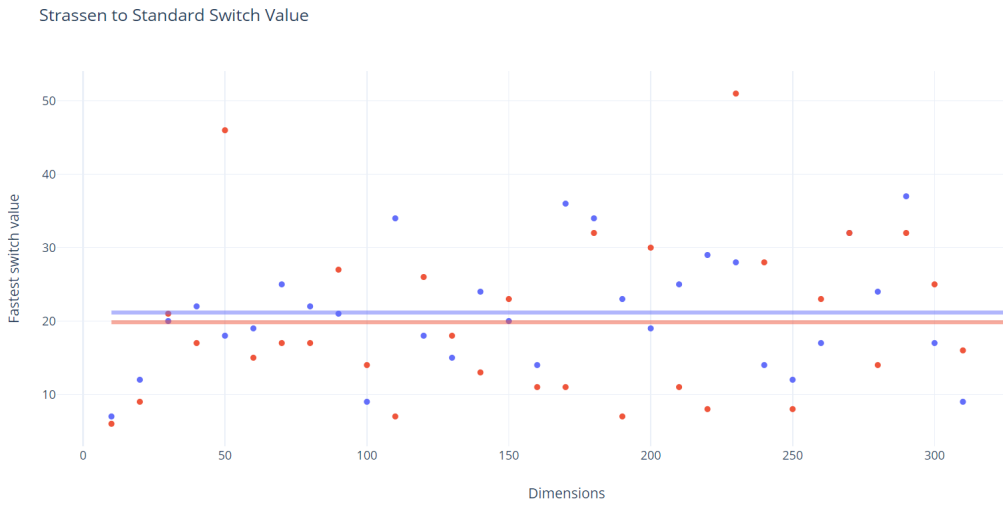
So, we set up the following inequality, this time solving for $2k+1$, and we get

$$14k^3 + 53k^2 + 64k + 25 \leq 16k^3 + 20k^2 + 8k + 1$$
$$2k + 1 \geq \approx 37.1698$$

So, the cross over point for a matrix with odd dimensions is approximately $n_0 = 37$.

Part 2:

In order to experimentally find a value for our optimal switch value, or n_0, we time the duration of each run for our modified Strassen algorithm. We decided to include a function named "switch test" that keeps track of the crossover value which produces the quickest calculation runtime. To make this switch test compatible with our Strassen implementation, we included an n_0 parameter which decides when we switch to our standard multiplication algorithm. Strassen effectively decreases the dimensionality of our matrix multiplication problem; if the maximum dimension of our current multiplication problem is less than or equal to our crossover value, we switch to standard multiplication. This ensures that we are not splitting our multiplication into smaller subproblems which do not positively affect our efficiency. Our optimal crossover values (for square matrices of given dimension) are as follows:



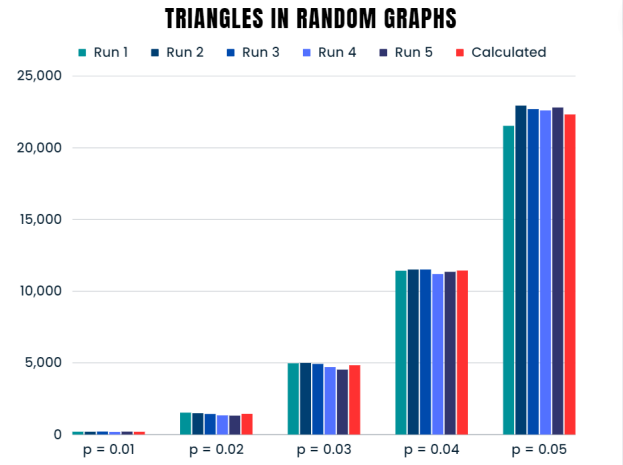
Key: Blue and Red dots signify two different runs; lines depict average values across all dimensions

As we can see, the majority of our n_0 values fall within the 10-30 range, which is consistent with our analytical derivation for optimal crossover values. However, we do see that our experimental n_0 is much better for matrices whose dimensions are not a power of 2; However, we do see that when we run our algorithm on matrices that aren't powers of 2, that the average n_0 is lower than the estimated n_0 for a power 2 dimension matrix; this is also true for matrices with odd dimension. This can be attributed to our implementation of a switch test and relating the current dimensionality of our multiplication problem to the previously stated crossover point.

Calculating the optimal crossover point theoretically disregards expensive operations such as storing data in matrices and manipulating them with padding. In addition to this, arithmetic operations do not have a constant cost $O(1)$; depending on the size of of the operands, computational cost varies. To optimize the multiplication algorithms, we could use parallelization or CUDA acceleration. Also, using a language such as C++ for both implementations would boost runtime. Then, doing work with the cache such as changing the order of loop and data access would boost speed. For different types of multiplicand matrices, such as ones with larger integer values rather than just zero and one, we saw that as n grew larger, they began to take more time to run, which makes sense because we are performing arithmetic operations with larger numbers which should effect our overall runtime based on memory caching.

Part 3:

Our resulting triangle counts (along with calculated values) are the following:



As we can see, the number of calculated and expected triangles are effectively the same; while numbers may differ across different runs (given probabilities of edges being formed), their correctness is verified by the expectation formula, $\binom{1024}{3}p^3$.