CS 124 Programming Assignment 2: Spring 2023

Your name(s) (up to two): Luke Richey

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: 6
No. of late days used after including this pset: 6

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.

Writeup

Task 1:

To complete task 1, we want to mathematically approximate the crossover point, n_0, at which we should switch from Strassen's algorithm to the naive algorithm when multiplying n x n matrices. We can do so by equating the expression for the runtime of the naive algorithm to the runtime of Strassen's algorithm utilizing the naive algorithm for its submatrices. Through some algebra, we can estimate n_0. First, let's determine the closed-form expression of the naive approach.

In the naive approach, we perform $2n-1$ operations for each of the n^2 entries (n multiplications and $n-1$ additions occur in the dot product of the rows/columns). Thus, the expression for the naive approach is:

$$n^2(2n-1)$$

Now, let's find the expression for Strassen's algorithm, using the naive algorithm to solve its submatrices. In Strassen's algorithm, we divide the matrices into 7 sub-matrices of size $\frac{n}{2}$ x $\frac{n}{2}$. Then, we perform 18 additions/subtractions on each of the submatrices of size $\frac{n}{2}$ x $\frac{n}{2}$. This yields the following expression:

$$7T(\frac{n}{2}) + 18(\frac{n}{2})^2$$

On the subproblems of Strassen's, we use the naive algorithm. Now, to find the crossover point we can just equate these two expressions and find the n_0 where they are approximately equal. We will first handle the case where n is even.

$$n^2(2n-1) = 7T(\frac{n}{2}) + 18(\frac{n}{2})^2$$
$$2n^3 - n^2 = 7(\frac{n}{2})^2 \cdot (2(\frac{n}{2}) - 1) + \frac{9n^2}{2}$$
$$2n^3 - n^2 = \frac{7n^2}{4} \cdot (n-1) + \frac{9n^2}{2}$$
$$8n^3 - 4n^2 = 7n^3 - 7n^2 + 18n^2$$
$$n^3 - 15n^2 = 0$$
$$n^2(n-15) = 0$$
$$n = 15$$

Thus, $n = 15$ is a solution for when n is even. When n is odd, we have:

$$2n^3 - n^2 = 7(\frac{n+1}{2})^2 \cdot (2(\frac{n+1}{2}) - 1) + \frac{9(n+1)^2}{2}$$
$$2n^3 - n^2 = \frac{7n(n+1)^2}{4} + \frac{9(n+1)^2}{2}$$
$$8n^3 - 4n^2 = 7n(n+1)^2 + 18(n+1)^2$$

$$n^3 - 36n^2 - 43n - 18 = 0$$
$$n \approx 37$$

$n = 37$ is an approximate solution for when n is odd.

Task 2:

In Task 2, we experimentally try to find the optimal crossover point between Strassen's and the naive algorithm.

My strategy for this was to use a sort of binary search method. I decided to use an initial cutoff of 64, as it seemed like a reasonable number that was higher than our theoretical estimation. I then tried a much higher cutoff (128) and a much lower cutoff (32). Whichever had a lower average runtime, I then did a similar process to test cutoffs between 64 and the other cutoff. For my tests, I used the same n and ran 5 experiments at each cutoff. While different n's could have been more accurate, I thought that if I did not use some methodical way of generating multiple n's that the difference in padding between different values of n could affect the results. I thought one methodical way could be to experiment on n's that are a power of 2, but then my results seemed to gravitate toward a power of 2 as well (i.e. 64 was a much better cutoff than 70). Thus, I decided to use a seemingly random number, that would have approximately similar paddings. For an even sized matrix, I used $n = 1000$. My cutoff results were as follows:

n	n_0	Runtime
1000	64	125719 ms
1000	32	140658ms 1000
128	135377 ms	
1000	80	128205 ms
1000	72	123544 ms
1000	67	124896 ms

Table 1: Runtimes for cutoffs on $n = 1000$

My experimental cutoff for even-sized matrices is around $n_0 = 72$.

Then, for odd-sized matrices I used $n = 1001$, to minimize any differences. Note that entries are in the order I performed the experiment. I got the following results:

n	n_0	Runtime
1001	64	128719 ms
1001	32	143506 ms
1001	128	134021 ms
1001	80	123375 ms
1001	100	132139 ms
1001	90	126020 ms
1001	85	120111 ms

Table 2: Runtimes for cutoffs on $n = 1001$

For odd-sized matrices, I get an approximate cutoff of $n_0 = 85$.

Discussion of Results:

We observe that both of the cross-over points are significantly higher than that of those found in Task 1. There are a couple reasons that I think can explain this:

1. The analytical approach fails to account for memory utilization. In my implementation of Strassen's, we allocate quite a bit of memory having to store all the submatrices. This memory usage (as well as cache misses and other similar inefficiencies) are not accounted for in the idealistic analytical approach.
2. In the analytical approach, we also assume that addition/multiplication operations take O(1) time. This is not the case, as when we are multiplying matrices there are some larger integers that reach the upper bound of 32-bit wide results. These will take longer than simple arithmetic and on large enough matrices, this difference becomes nontrivial.

In my experiments, I also tried 0/1/2 matrices and -1/0/1 matrices, but did not seem any significant changes in my results.

Overall, I am not sure how effective a cross-over point is. I think it would probably be more beneficial in practice to choose one algorithm and optimize it as much as possible. There are a vast amount of optimizations you can make in matrix multiplications, and so dividing your attention among two different algorithms seem pretty inefficient.

Optimizations:

In my implementation, my main optimization to Strassen's was threading. Threading is a pretty simple optimization in this case, as when we compute the submatrices these are perfect use cases for threading. The differences in runtime was pretty marginal, mostly because I think I failed to optimize my memory well.

In my naive implementation, I used Eigen's library for its matrix representation as well as its .row(), .col(), and its dot product functions. From what I researched, these operations seemed to be relatively optimized for the matrix representation I had chosen.

Task 3:

I generated random graphs with $p = 0.01, 0.02, 0.03, 0.04, 0.05$. I ran ten trials at each probability, and averaged the number of triangles I found. The results are in the table below:

# of triangles	expected # of triangles
179.6	178.43
1411.6	1427.46
4866.5	4817.69
11347.2	11419.71
22242.7	22304.13