

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 efficiently 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 32-bit integer inputs.) You need not try all of these, but do test your algorithm adequately.

This assignment is partially autograded. The only languages which have been configured which work with autograder are **C11**, **C++ 17**, **Java 8**, and **Python 3.6**. Correspondingly, your code should be contained within a single file that is called one of: `strassen.c`, `strassen.cpp`, `strassen.java`,

or `strassen.py`.

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. To see this, you can see that (ij) th entry in the matrix A^2 counts the paths from i to j of length 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 a graph, we can simply add the entries in the diagonal, and divide by 6. This is because the j th 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 expectation.

Code setup:

Your code will be compiled, run, and tested by an autograding script. The way you will run the code will depend on the language, but it should take in three command line arguments: a flag, a dimension, and an input file. For example, if you did the assignment in C, we will run:

```
$ ./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 $a_{0,1}, a_{0,2}, \dots, a_{0,d-1}$; 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 (printf, cout, System.out, etc) a list of the values of the *diagonal entries* $c_{0,0}, c_{1,1}, \dots, c_{d-1,d-1}$, 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 (signed) integers, but you should make sure your matrix multiplication can deal with results that are up to 32 bits.

When you submit your code to Gradescope, some tests will run to make sure that your code compiles and to make sure that it is correct. These tests will make up a significant percentage of your grade. Note that depending on your choice of language and the server load on Gradescope at the time of submission, these tests may take several minutes to complete, so be sure to submit well before the deadline or to resubmit often.

What to hand in:

- A PDF file containing your project report
- A **single** code file that is one of: `strassen.c`, `strassen.cpp`, `strassen.py`, or `strassen.java`. Submissions which rely on external files or libraries may fail to compile.

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 .