CS124 Programming Assignment 2: Strassen's Algorithm

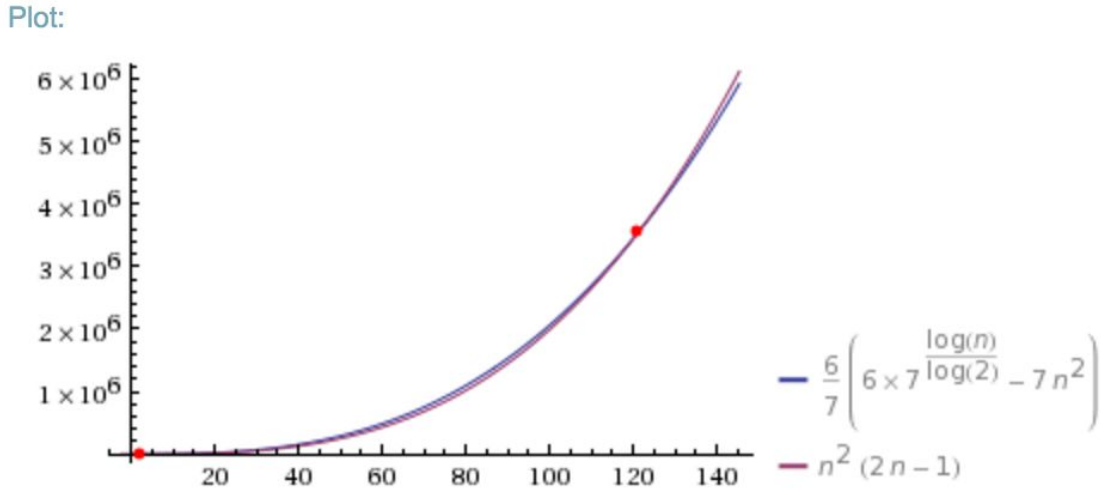
Abby Lyons and Timothy Tamm

Calculations

We used the following two equations to (analytically) approximate the actual runtime of Strassen's algorithm and conventional multiplication. The first equation was found by setting a base case for n = 2, and using Wolfram to solve the recurrence for Strassen's algorithm. The second equation came from analyzing the runtime of conventional matrix multiplication.

S(n) = 6/7 * (6 * 7^(log_2(n)) - 7n^2)
C(n) = n^2 * (2n - 1)

By graphing these equations, we found that the analytical crossover point is approximately 121.



Implementation

Before starting this assignment, we decided that one of the things we should optimize is the runtime of indexing into a two-dimensional array. Because it's actually stored one-dimensionally in memory, the computer requires multiplication to find the correct place in a two-dimensional array. To prevent this from happening, we created scanners for both "row-major" and

“column-major” matrices to expedite the matrix multiplication process, with the row-major matrices stored row-by-row in a one-dimensional array, and the column-major matrices stored column-by-column in a one-dimensional array. We made functions for the scanners to quickly move to the next item in the row or column, or move to the next row or column. This also helps with increasing the number of cache hits; if a matrix is the wrong majority, then it will result in more cache misses.

We implemented conventional matrix multiplication, matrix addition, and matrix subtraction without any issues. However, we initially forgot to use the scanners to their full capabilities, causing a lot of cache misses, so the experimental cutoff was lower than the analytical value. After fixing this, we got an experimental cutoff that was much closer to the analytical value.

When implementing Strassen’s, we initially made copies of each submatrix, which worked but was very space-inefficient. We then altered the code so the scanners kept an “original” dimension along with the “actual” dimension, so the scanners would work properly even if the array was a certain non-contiguous segment of memory (that is, if we didn’t want to copy the array). The implementation of the original dimension was buggy and consistently returned the wrong value when going to the next row or column. This same bug caused problems when stitching the quadrants back together into one matrix. Fortunately, we were able to find the source of the bug.

Next, we turned our attention toward fixing memory leaks. We were unable to multiply matrices greater than 512 in dimension, and a quick Valgrind run showed us just how much memory we were leaking. We implemented a linked list to store pointers to any allocated memory for new matrices (for example, the sum of two matrices), and unloaded them at the end. However, even that wasn’t good enough--Abby’s computer still ran out of memory for dimensions of greater than 1000. We fixed this by creating a linked list for each Strassen call and unloading as soon as possible, which helped immensely.

Finally, to make the algorithm work for non-powers of 2 dimensions, we implemented simple 0-padding before splitting up odd-sized matrices.

Given more time, we would have implemented free pooling to minimize malloc calls, by having an array of linked lists with each linked list corresponding to a power of 2 size.

Experimental Data

We found the ideal experimental cutoff through trial and error. We tested each power of 2 cutoff with dimensions of 256, 512, 1024, 2048, and 4096, and recorded the cutoff that gave the best runtime for each size. Our data is below:

Dim = 256

Crossover Point	Runtime (milliseconds)
2	1529.65
4	426.4
8	176.744
16	108.743
32	79.125
64	58.734
128	56.215

Dim = 512

Crossover Point	Runtime (milliseconds)
2	11494.3
4	3332.31
8	1317.08
16	753.084
32	544.55
64	447.026
128	404.148
256	406.128

Dim = 1024

Crossover Point	Runtime (milliseconds)
2	80681.9
4	22896.8
8	9659.17
16	5570.35
32	4300.05
64	3381.34
128	2968.34
256	3031.12
512	3272.36

Dim = 2048

Crossover Point	Runtime (milliseconds)
8	69293
16	39533.1
32	29970.4
64	24480.6
128	20729.2
256	21179.1
512	22263.8
1024	24543.1

Dim = 4096

Crossover Point	Runtime (milliseconds)
8	466428

16	260291
32	192506
64	154735
128	144740
256	151389
512	164726
1024	174999
2048	197310

Each dimension tested above gives a consistent result of 128 as the best cutoff point. This is entirely consistent with our analytical result of 121, and it is as close as we can get without overcomplicating the cutoff point by considering numbers other than powers of 2. With the cutoff now set at 128, we then found the runtime for different dimensions, taking an average of five trials.

Dimension	**Average Runtime (milliseconds)**	**dim^3 * 0.000006**	**dim^(log 7/log 2) * 0.00001**
2	0.0009	0.000048	----------
4	0.001	0.000384	----------
8	0.005	0.003072	----------
16	0.03	0.0246	----------
32	0.198	0.1966	----------
64	1.14	1.573	----------
128	7.92467	12.583	----------
256	61.864	----------	57.6
512	440.334	----------	403.5
1024	3318.94	----------	2824
2048	23023	----------	19773

4096	166153	----------	138412

For the dimensions that use traditional matrix multiplication, the runtime appears to be approximately n^3 * K, where K is approximately 0.000006. This matches the analytical worst-case runtime of O(n^3).

For the dimensions that use Strassen's algorithm, the runtime appears to be approximately n^(log_2 7) * K, where K = 0.00001. This matches the analytical worst-case runtime of O(n^(log_2 7)), which can be read as log base 2 of 7.