Computer Science 124 March 25, 2016

Programming Assignment 2

Javier Cuan-Martinez & Arianna Benson

Introduction

For this assignment, we chose to use Python to implement our algorithms. We chose this language because of its automatic memory allocation and relative syntactical simplicity. Nevertheless, we understand that because we do not have as much control over our memory allocation as other languages like C provide, our fixed memory management limits us in optimizing how much memory is being used at a certain time. This would prevent our program from optimally running our algorithms.

We do believe that the reduced implementation time that Python provides offsets the advantages that manual memory allocation can pose when it comes to optimization. Furthermore, because the assignment revolves mostly around figuring out the cross-over point between Strassen's algorithm and the typical algorithm, we are only concerned with finding the cross-over point in relation to Python rather than trying to reach certain time benchmarks.

We chose to represent our matrices as two-dimensional Python lists (lists of lists). This implementation allows us to execute matrix operations with the conventional methods that we have been taught. However, we realize that by using this implementation, we sacrifice caching power by scattering the memory.

To run the code, this is what you should input into the command line:

```
python strassen.py 0 dimension inputfile
```

Analysis

As mentioned in class, the runtime of the typical algorithm is $O(n^3)$. For each cell of the new matrix, you do n multiplications and $n-1$ additions, meaning $2n+1$ operations. Since there are n^2 cells, the exact runtime of the typical algorithm is $\boldsymbol{f(n) = (2n+1)n^2 = 2n^3 + n^2}$.

Then, we can get the crossover point of Strassen's algorithm by assuming a crossover point of n, and finding $T(n)$ using $f(\lceil \frac{n}{2} \rceil)$.

We get the following general recurrence:

$$\boldsymbol{T(n) = 7f(\lceil \frac{n}{2} \rceil) + 18\lceil \frac{n}{2} \rceil^2}$$

Now, we have two cases for $T(n)$, where n is assumed to be the crossover point:

Case: n is even.

When n is even, $\lceil \frac{n}{2} \rceil = \frac{n}{2}$. As such, we have:

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

$= 7(n-1)(\frac{n}{2})^2 + 18(\frac{n}{2})^2$

$= \frac{7}{4}n^3 + \frac{11}{4}n^2$

Case: n is odd.

Since n is odd, $\lceil \frac{n}{2} \rceil$ is equal to $\frac{n+1}{2}$. Here, we have:

$$\begin{aligned} T(n) &= 7f(\tfrac{n+1}{2}) + 18(\tfrac{n+1}{2})^2 \\ &= 7(n(\tfrac{n+1}{2})^2) + 4.5(n+1)^2 \\ &= \tfrac{7}{4}n(n+1)^2 + 4.5(n+1)^2 \end{aligned}$$

Crossovers

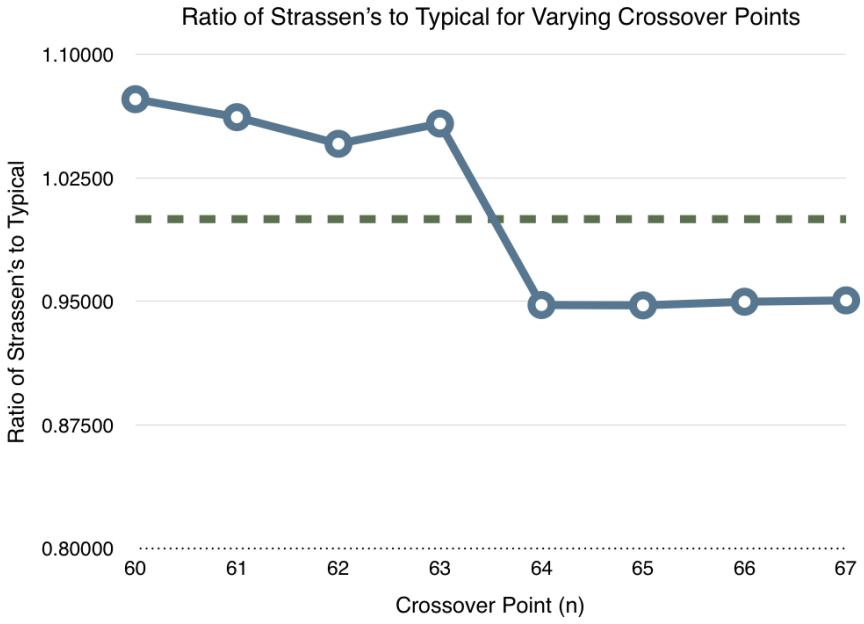
For even n, when you set $T(n) = f(n)$, you get $n = 15$. But since this only matters for n even, we have a crossover point $n_0 = 16$.

For odd n, when you set $T(n) = f(n)$, you get $n = 37.170$. Since we need to round up, the proper n_0 is 39. This makes sense because for odd n_0, you need to pad, which takes additional time (and space). As such, for odd numbers between 16 and 39, we will switch to the typical algorithm; but for even numbers, we will continue using Strassen's.

As such, the crossover from T to f occurs for even n when $n_0 = 16$, and for odd n when $n_0 = 39$.

Results

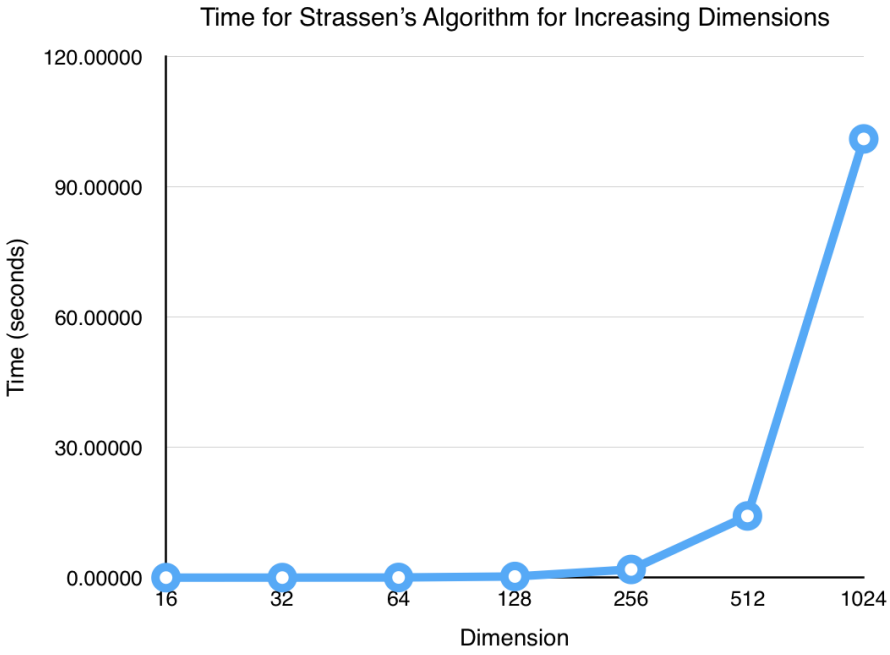
We found, experimentally, a cross-over point of 64. We did this by running Strassen's with varying crossover points at different times, using our testing flag as the input for the crossover point. As can be seen in the graph below, the optimal crossover point is approximately 64.



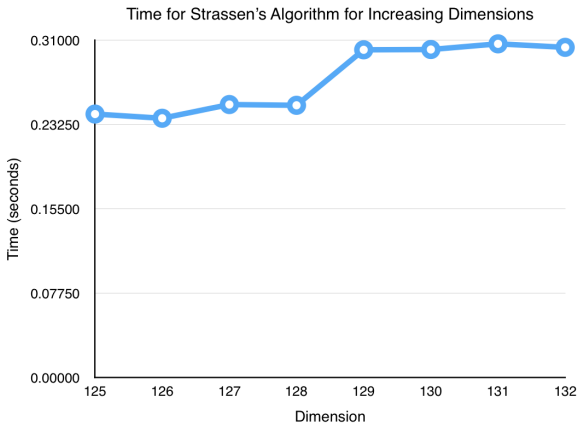
The data we obtained, here, also illustrates this:

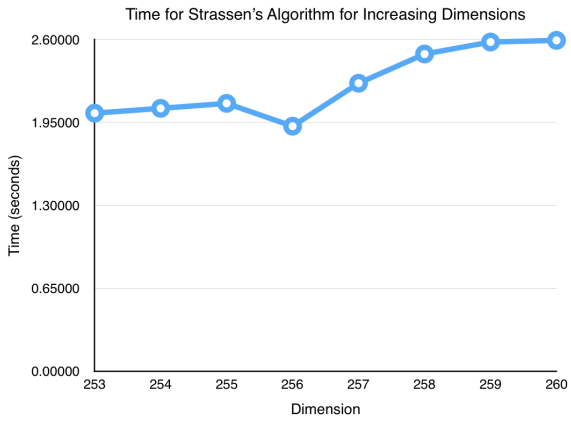
Dimension	60	61	62	63	64	65	66	67
Strassen (s)	2.13811	2.11721	2.09828	2.10413	1.88019	1.88171	1.88805	1.88889
Regular (s)	1.99291	1.99369	2.00644	1.98863	1.98371	1.98572	1.98781	1.98696
Ratio	1.07285	1.06195	1.04577	1.05808	0.94781	0.94762	0.94982	0.95064

The runtimes for our two algorithms (typical and Strassen's with crossover at 64), testing on matrices of differing dimensions (of the format 2^k, where k is an integer) filled with random values chosen from the closed interval $[0, 2]$, can be seen in the graph below.



Yet, when you do not restrict the dimensions to the format 2^k, you begin to see an interesting pattern:





The jagged increases between dimensions 2^k and 2^k+1 in the graph are likely due to the impact that padding has on the runtime. So, while a matrix of size 128 does not require you to pad, a matrix of size 129 requires you to pad on practically every iteration.

Evaluation

Our experimental crossover point of 64 was higher than both of the theoretical crossover points of 16 and 39. This is likely because we did not fully make use of the space available to us – because we could not pad in-place if we simply kept our subdivided matrices (a through h) in the main matrix, we decided to

create copies. This added approximately $2n^2$ additional cells of each iteration of Strassen's, which – in total – summed to up to an additional $2n^2 \log_2 n$ cells total. All of this array allocation likely added up!

Since we structured our Strassen's function so that we always multiply square matrices, we always ensure that we pad before we pass a matrix onto our next iteration of Strassen's (or the typical algorithm). Yet, while we did remove the inefficiency of padding to the next power of 2 (for example, you almost quadruple the space used if you pad a matrix of size 129 by 129 to 256 by 256), our algorithm still requires us to add up to one full row and one full column of zeros on each iteration of Strassen's. We considered representing these virtually (for instance, by setting a flag that indicated there were empty rows/columns), but we ran into difficulties when it came to subdividing/adding/subtracting that matrix, since you could not rely on the properties of the array to give you the actual dimensions of the matrix.

For the traditional algorithm, you generally get a speedup by looping through the variables i, j, and k in different orders. We experimentally tried all $3! = 6$ different orders (ijk, ikj, jik, jki, kij, and kji), and found that looping first through i, then through j, and finally through k gives us the best performance for the typical algorithm, because it allows us to still take advantage of the Python append() function. This was the only optimization we undertook for our typical algorithm, since regardless of what you do, you will still have to perform $O(n^3)$ operations.

Since correctness is the most important factor of our algorithm, to ensure the robustness of our algorithm, we wrote a testing function. This function asserted the equality of our typical multiplication algorithm with Strassen's algorithm on randomly generated matrices of different sizes. By using this throughout the coding process, we were able to pinpoint exactly for which values our program worked and for which values it failed.

Additionally, since Python is interpreted, rather than compiled, it does not have the full capabilities for caching and optimization that lower-level languages like C and Java do. The ability to manually allocate memory, for example, would give us far more control over how we stored our matrices. As such, were we to redo the assignment, we would consider switching to a language suited for these manipulations.