CS 124 Programming Assignment 2: Spring 2021

Your name(s) (up to two): Yumi Yi

Collaborators:

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

Analytical:

I determined the value of n_0 by using the following equations and solving for n. The first is for matrices with even dimensions, and the second for ones with odd dimensions:

$2n^3 - n^2 = 7(2(\frac{n}{2})^3 - (\frac{n}{2})^2) + 18(\frac{n}{2})^2$

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

Solving for n yields $n = 15$ for the first equation, and approximately $n = 37$ for the second.

The left-hand side of the equations represents the number of operations used in the standard multiplication algorithm. Each entry requires computing n products and $n - 1$ additions to sum the products, and there are n^2 entries in total to fill.

Similarly, the right-hand side of the equations represents the number of operations used to make one call of Strassen's algorithm, then using the standard algorithm the rest of the way down. The recursion for the number of operation used in Strassen's algorithm is as follows:

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

since there are 7 multiplications of matrices of dimension $\frac{n}{2}$ and 18 additions of matrices of dimension $\frac{n}{2}$. Then we simply substitute in the equation for standard matrix multiplication for $T(\frac{n}{2})$ to get the first equation above.

Of course, this only works for matrices with even dimensions. For odd matrices, we can simply pad them with an additional row and column of 0s to create new matrices of dimension $n + 1$. We can then remove the padding from the output to obtain our final answer. This padding is accounted for in the second equation above by incrementing n by 1.

Experimental:

Using the theoretical values of $n_0 = 15$ and $n_0 = 37$ as a starting point, I experimented with different values of n_0 to find the greatest value for which the standard algorithm runs faster than Strassen's (so for values greater than n_0, Strassen's runs faster). I tested matrices whose entries are randomly selected to be 0 or 1. My results are as follows, where runtime is measured in CPU time:

dimension	n_0	standard	Strassen
512	15	0.729518	1.081658
513	16	0.710497	1.449018
767	11	2.470387	4.949227
768	11	2.489988	4.952738
1024	15	6.056280	7.570962
1025	16	5.973204	10.123757
1535	11	30.045291	35.918279
1536	11	30.584353	36.140175

So the smallest value of n_0 for which the standard algorithm is faster than Strassen's for every dimension tested is $n_0 = 11$, which is, surprisingly, lower than the theoretical value we found for the crossover point ($n = 15$). This may be because the actual cost of some arithmetic operations may be less than we assumed in our theoretical analysis, where we assumed each operation had a cost of 1.

I tested matrices with both even and odd dimensions, and dimensions close to and far from powers of 2. While our theoretical analysis suggested that the crossover point would be vastly different for even and odd dimensions ($n = 15$ and $n = 37$, respectively), we see that that is not the case, and the crossover point is similar for both types of matrices. However, the data also suggests that dimensions further from powers of 2 have noticeably lower crossover points than dimensions closer to powers of 2. This may have to do with the number of divisions involved.

Triangles:

The chart with my results is below. Results are averaged over 3 trials, and are close to the expected results:

p	expectation	results
0.01	178.43	171.33
0.02	1427.46	1411.67
0.03	4817.69	4805.67
0.04	11419.71	11400.00
0.05	22304.13	22358.00

Final Thoughts:

What I found most interesting was the difference between multiplying matrices with dimensions close to powers of 2 and far from powers of 2. Given our mathematical analysis, I expected to find a greater difference between even and odd dimensions, but in actuality, there didn't turn out to be much of a difference. I was also surprised to find such a low crossover point, and though I'm satisfied that it's lower than the theoretical crossover point, I can't help but wonder if I could have made it even lower by optimizing my code better. I tried to make my implementation as space-efficient as I could; for example, as a small optimization, rather than creating new 2D arrays to store the four Strassen sums ($AE + BG$, $AF + BH$, etc.), I overwrote the ones I used to store the original submatrices, since I didn't need them anymore at that point in the algorithm. However, I'm sure there are better optimizations to be made.