CS 124 Programming Assignment 2 Writeup

Logistics

Your name(s) (up to two): Andrew Holmes, Lauren Cooke

Collaborators: None

No. of late days used on previous psets: Andrew: 3 Lauren: 1

No. of late days used after including this pset: Andrew: 4 Lauren: 2

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.

Discussion:

We decided to work in python and managed a working version of Strassen's algorithm reasonably quickly. However, we spent a huge amount of time trying to optimize things. As such, throughout this report we will refer to a few different versions of Strassen's algorithms that we implemented, each built on the previous version:

- Standard - the standard, $O(n^3)$ implementation of matrix multiplication, which we implemented first.
- Strassen - our first attempt at Strassen's algorithm with essentially no optimizations, which we implemented second.

  Note: We decided to pad to a power of 2 straight away, rather than lazily padding as needed. We decided to go for this for a few reasons: 1) it seemed easier to implement so that we could get a working implementation faster 2) it would make analysis and other optimizations easier, e.g. we knew early on that we might be wasting time by re-initializing the p and c matrices, so we knew we would need to work out how many of these matrices we would need and of what size - padding in this manner makes these calculations far easier, as we essentially treat any matrix as a matrix of size 2^k for some $k \in \mathbb{N}$, and every cutoff point is also going to be a power of 2, so we can work this out easily essentially by using 2 logs.
- Strassen_opt - our first serious change to our original algorithm, with the following optimizations:

  The matrices a, b, c, d, e, f, g, h are no longer stored, instead, Strassen_opt takes as inputs all the same inputs as our previous implementation, along with four new inputs: xa, ya, xb, yb, which, along with the input *size* which tells us how big the matrices we are multiplying at this step are, can be used to simply extract the correct values from the original matrices, rather than actually chopping each into 4 new matrices and storing these.
- Strassen_fin - our final version of Strassen's algorithm, built from Strasen_opt, in which we managed to implement all of the optimizations we were hoping for, including:

In this iteration, we aimed to eliminate redundant memory allocation and de-allocation by creating space for all the p_i and $c_{i,j}$ matrices that we would need first, such that we initialize these all to zero at the start, and then can re-use this space multiple times without needing to allocate memory for it again. This took the form of two functions, *create_P* and *create_C*, which, given the size of matrices we were multiplying (overall size at the start), and n0, the point we start using the standard multiplication algorithm, would create 7 p matrices (and one extra that we used to hold some of the subresults needed to calculate each p_i) and 4 c_{ij} matrices for each level of Strassen recursion that we would do, and then at each level of recursion we could workout, given the size of the matrices that we were working with at this level and which p or c we are currently trying to calculate, how to index in correctly to our P and C lists correctly (these lists just stored pointers to all the appropriate matrices), such that we only create these matrices once, update them whenever needed, and destroy them at the end of the algorithm.

Difficulties

The more we tried to optimize the algorithm, the more places there were for mistakes to creep in, and we ended up with functions taking a huge number of arguments. For example, the one line of code in our original version to calculate p_1 looked like this:

```
p1 = strassen(mat_sub(b, d, ns), mat_add(g, h, ns), ns, n0)
```

It's clear what is going on above and can directly be compared to the definition of Strassen's algorithm. However, we can see that we were storing submatrices such as b, d separately, and were creating p_1 at each recursive call. Comparing this to our final version, which takes the following three lines:

```
mat_sub_fin(matA, matA, bx, by, dx, dy, ns, P[index + 6])
mat_add_fin(matB, matB, gx, gy, hx, hy, ns, P[index + 7])
strassen_fin(P[index + 6], P[index + 7], ns, n0, 0, 0, 0, 0, P, C,
             newcount, P[index])
```

Here we perform the appropriate calculates by indexing into A and B using coordinates, and storing the first two subresults in p_7 and p_8 (the 8th matrix I mentioned earlier, we used these two to store temporary calculations when calculating earlier p_i's, and when we finally calculate p_7 we only need one extra matrix to store workings so we use p_8 and update p_7 correctly). We then update p_1 (which is always $P[index]$ by how I defined my index) using these two pieces of scratch work.

So we can see that in this version, these P matrices are initialized before we actually start the main algorithm, and we simply pass pointers around so that we store results in the matrices we created at the start, such that we should be updating values rather than repeatedly allocating and de-allocating memory.

However, after all of the effort we put in to optimize this, it turned out that our original version was actually faster than this final version! While we've hunted for mistakes extensively, it appears to be simply the case that our original version is actually faster, likely thanks to optimizations that python is implementing behind the scenes, and our own optimized version is actually not as well optimized as what the language does for us with our first version. Honestly, this is hugely frustrating, since we poured a huge amount of time into 'optimizing' our memory management, only for it to decrease performance. I am extremely curious (and hopeful) whether, if we transferred exactly what we did and adjusted it into C/C++, our final implementation would be significantly better than our original implementation.

Tasks:

Task 1: Analytically determining optimal value of n_0.

Solution: The conventional algorithm takes time $O(n^3)$, while Strassen's algorithm takes $O(n^{2.8074})$. To find n_0, we want to find the minimum value such that running Strassen's is faster than the conventional algorithm, as for small matrices the constant factors make the conventional algorithm faster.

n is even case

Strassen's recurrence:

$$T(n) = 7T\left(\frac{n}{2}\right) + \Theta(n^2)$$

To find the crossover point exactly we will need to find what this $\Theta(n^2)$ term is.

Strassen's algorithm uses:

- Strassen observed the following:

$$Z = \begin{Bmatrix} A & B \\ C & D \end{Bmatrix} \cdot \begin{Bmatrix} E & F \\ G & H \end{Bmatrix} = \begin{Bmatrix} (S_1 + S_2 - S_4 + S_6) & (S_4 + S_5) \\ (S_6 + S_7) & (S_2 + S_3 + S_5 - S_7) \end{Bmatrix}$$

where

$$\begin{aligned} S_1 &= (B - D) \cdot (G + H) \\ S_2 &= (A + D) \cdot (E + H) \\ S_3 &= (A - C) \cdot (E + F) \\ S_4 &= (A + B) \cdot H \\ S_5 &= A \cdot (F - H) \\ S_6 &= D \cdot (G - E) \\ S_7 &= (C + D) \cdot E \end{aligned}$$

When calculating the S_i's we do 10 additions/subtractions on matrices of size $(\frac{n}{2})^2$, while to build the final output matrix we do 8 additions/subtractions on matrices of the same size. Thus overall we do $18\frac{n^2}{4} = \frac{9n^2}{2}$ work for additions and subtractions (the multiplication term is dealt with by the recurrence term and we are assuming each of these operations has cost 1). Thus overall we have:

$$T(n) = 7T\left(\frac{n}{2}\right) + \frac{9n^2}{2}$$

Now for the conventional algorithm running time, we perform n multiplications and $n-1$ additions for each entry in our output matrix (e.g. $c_{1,1} = a_{1,1}b_{1,1} + a_{1,2}b_{2,1} + a_{1,3}b_{3,1}$). We do this n^2 times to generate the whole matrix, thus this takes

$$S(n) = n^2(n + n - 1) = 2n^3 - n^2$$

To find the crossover point, we want to find when it would be faster running the standard multiplication rather than another layer of Strassen's to solve our smaller subproblems. Thus in our Strassen's recurrence,

we use the standard algorithm time complexity as our $T(\frac{n}{2})$ term and will then check when this is faster than $S(n)$:

$$\begin{aligned}
T(n_0) &= 7T\left(\frac{n_0}{2}\right) + \frac{9n_0^2}{2} \\
&= 7\left(S\left(\frac{n_0}{2}\right)\right) + \frac{9n_0^2}{2} \\
&= 7\left(2(\frac{n_0}{2})^3 - (\frac{n_0}{2})^2\right) + \frac{9n_0^2}{2} \\
&= 14\frac{n_0^3}{8} - \frac{7n_0^2}{4} + \frac{9n_0^2}{2} \\
T(n_0) &= \frac{7}{4}n_0^3 + \frac{11}{4}n_0^2
\end{aligned}$$

Comparing to $S(n_0) = 2n_0^3 - n_0^2$:

$$\begin{aligned}
S(n_0) = T(n_0) \implies & \\
2n_0^3 - n_0^2 &= \frac{7}{4}n_0^3 + \frac{11}{4}n_0^2 \\
\frac{1}{4}n_0^3 &= \frac{15}{4}n_0^2 \\
n_0^3 &= 15n_0^2 \\
n_0 &= 15
\end{aligned}$$

n is odd case

If n is odd, we can't split our recurrence as previously since $\frac{n}{2}$ wouldn't be an integer. As such, we would need to pad to an even size to be able to even Stassen recurse. Thus, in this case, Strassen's recurrence is instead:

$$T(n) = 7T\left(\frac{n+1}{2}\right) + \frac{9(n+1)^2}{2}$$

However, the running time equation for the standard algorithm is unchanged for even or odd sized matrices, so we still have:

$$S(n) = n^2(n+n-1) = 2n^3 - n^2$$

Finding the crossover point as we did for the even matrices:

$$
\begin{aligned}
T(n_0) &= 7T\left(\frac{n_0+1}{2}\right) + \frac{9(n_0+1)^2}{2} \\
&= 7\left(S\left(\frac{n_0+1}{2}\right)\right) + \frac{9(n_0+1)^2}{2} \\
&= 7\left(2(\frac{n_0+1}{2})^3 - (\frac{n_0+1}{2})^2\right) + \frac{9(n_0+1)^2}{2} \\
&= 14\frac{(n_0+1)^3}{8} - \frac{7(n_0+1)^2}{4} + \frac{9(n_0+1)^2}{2} \\
T(n_0) &= \frac{7}{4}(n_0+1)^3 + \frac{11}{4}(n_0+1)^2
\end{aligned}
$$

Comparing to $S(n_0) = 2n_0^3 - n_0^2$:

$$
\begin{aligned}
S(n_0) = T(n_0) \implies & \\
2n_0^3 - n_0^2 &= \frac{7}{4}(n_0+1)^3 + \frac{11}{4}(n_0+1)^2 \\
2n_0^3 - n_0^2 &= \frac{7}{4}(n_0^3+3n_0^2+3n_0+1) + \frac{11}{4}(n_0^2+2n_0+1) \\
2n_0^3 - n_0^2 &= \frac{7}{4}n_0^3 + \frac{32}{4}n_0^2 + \frac{43}{4}n_0 + \frac{18}{4} \\
\frac{1}{4}n_0^3 &= 9n_0^2 + 10\frac{3}{4}n_0 + 4.5 \\
0 &= -\frac{1}{4}n_0^3 + 9n_0^2 + 10\frac{3}{4}n_0 + 4.5 \\
n_0 &\approx 37.17
\end{aligned}
$$

This seems surprising: the optimal break point for an odd sized matrix is more than double that for an even sized matrix! However, since the complexity difference between the two is reasonably significant ($n^{2.8}$ vs n^3), having to pad up one takes time, then all of our subsequent smaller matrix calculations are a little bit bigger since we had to pad at the start, and all of these contributions lead to requiring a significantly larger cutoff. *Also obviously we can't cutoff at 37.17, so 37 and 38 are both reasonably answers for the theoretical optimal cutoff point in this case.*

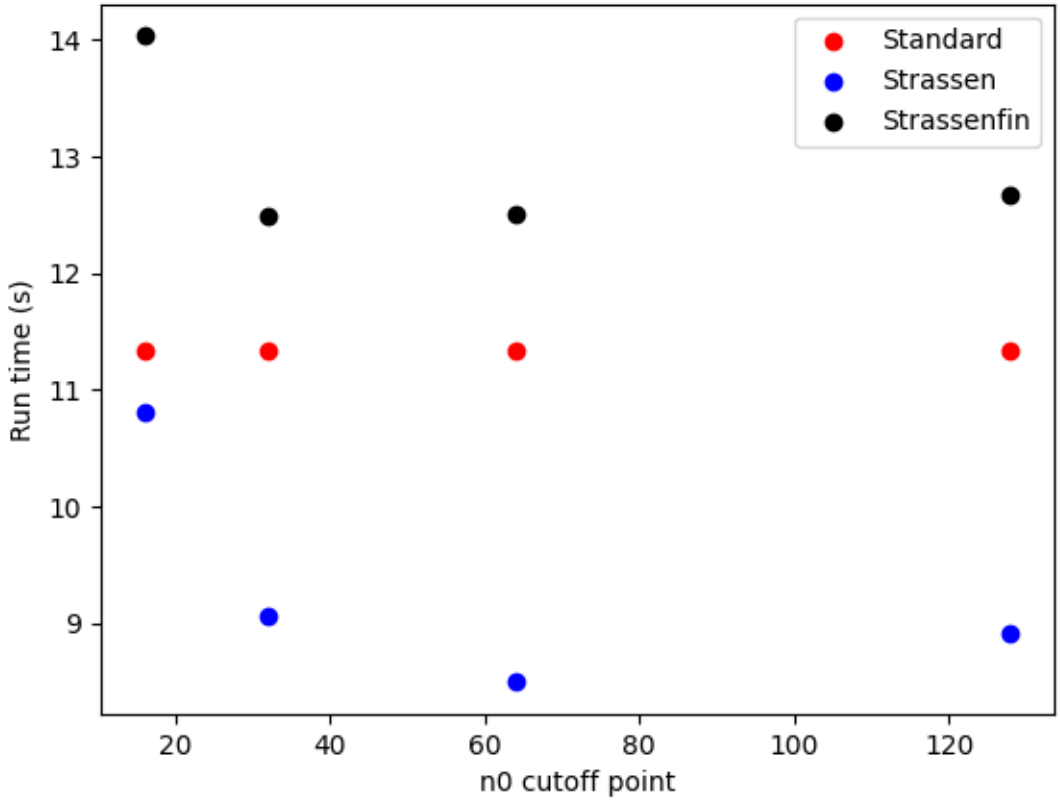
□

Task 2: Implement Strassen's and find the cross-over point experimentally

Solution:

To find the Strassen crossover point experimentally, we implemented Strassen's algorithm in python with a variable crossover point, $n0$, that defines at what size matrix we switch from running Strassen's algorithm to running standard matrix multiplication. As already discussed, we implemented several versions of Strassen, with our first version being called 'strassen', our next version 'strassen_opt' and our final

version 'strassen_fin'. Since the final and half optimized versions behaved very similar, we will just plot the final version, our original version, and the standard algorithm run times. Running these algorithms on different $n0$ values s.t. $n0 = 2^a$ where $a \in \{4, 5, 6, 7\}$, we get the following results (this data came from multiplying two randomly generated 512x512 matrices with each algorithm 5 times and taking an average):



The standard matrix multiplication algorithm is obviously unaffected by changing n_0. However, for our original Strassen algorithm, the best cutoff point was 64, while for our two improved versions, the best cutoff point was at 32 (although just barely! The difference was generally on the order of a second or two faster with 32 than 64 when multiplying 1024 or 2048 matrices, and a fraction of a second when using 512x512 matrices). This indicates that our optimizations were having an effect and bring us closer in line with the theoretical optimal cutoff point. However, even with our best effort at making optimizations, we couldn't reach the optimal value we calculated earlier, and as discussed in the first discussion section, our 'optimized versions' were actually slower than our original version!

There's many possible reasons for this. Firstly, we made a number of naive assumptions when calculating our theoretical value. We know that many of these operations do not take constant time - we've seen in class that multiplying numbers is generally more difficult than adding numbers, but there are some multiplications that might be cheap since we can simply perform a bit shift (e.g. multiplying by 2). It's also obviously not true that all other operations will be free - allocating, de-allocating and updating memory will cost significant amounts of time, and our theoretical analysis doesn't factor this in at all (nor do we really know how python is managing memory under the hood, making it even harder to quantify

how this impacts our run time). Thus our equations we had for the time for both the standard and Strassen implementations are not very reflective of realistic experimental conditions for actually running the algorithms.

We should also note that, since we padded to a power of 2, all of our multiplications should only be compared to the theoretical value we calculated for even size matrices since all of our matrices will be padded to become even matrices before we start any of our recursion (ignoring n = 1 as a trivial case), such that our 32 result is not actually lower than any theoretical result as the 37 result is essentially irrelevant to our implementation of the algorithm.

Data tables (512 x 512):

Standard algorithm:

Iteration	n0	Time (s)
1	N/A	11.358715599999982
2	N/A	11.343494599999985
3	N/A	11.31762789999999
4	N/A	11.365577700000017
5	N/A	11.29578939999999

Original Strassen:

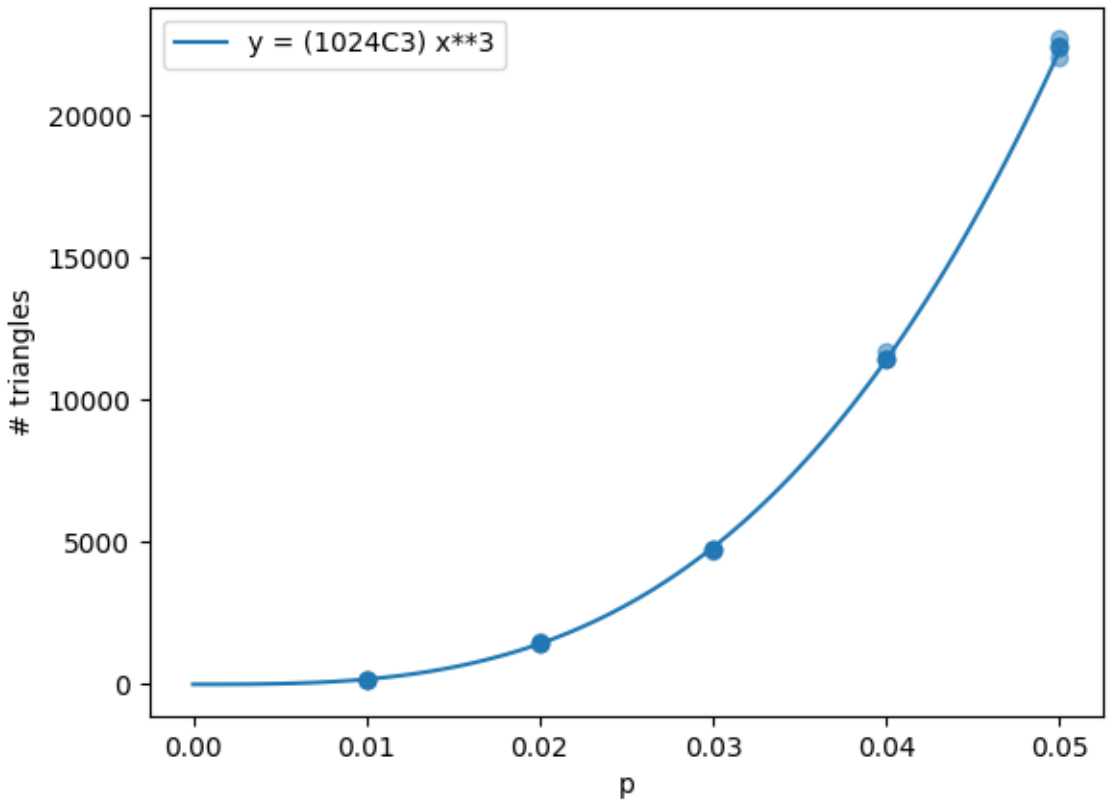
Iteration	n0	Time (s)
1	16	11.216115399999993
2	16	10.870206399999987
3	16	10.750079600000007
4	16	10.861177899999987
5	16	10.361114200000003
1	32	9.0384217
2	32	9.0848041
3	32	9.060236799999998
4	32	9.028240000000011
5	32	9.117033499999991
1	64	8.455562500000013
2	64	8.620280399999984
3	64	8.484205599999996
4	64	8.508401100000015
5	64	8.412863100000038
1	128	8.774544899999967
2	128	8.956711700000028
3	128	9.08377200000001
4	128	8.964981099999989
5	128	8.796846799999969

Optimized Strassen:

Iteration	n0	Time (s)
1	16	13.830352
2	16	13.634105000000002
3	16	14.500263
4	16	14.202159800000004
5	16	13.990113899999997
1	32	12.526326599999999
2	32	12.4449024
3	32	12.453965200000003
4	32	12.4973387
5	32	12.490191700000004
1	64	12.4472892
2	64	12.501840700000002
3	64	12.561747600000018
4	64	12.624206199999975
5	64	12.411495300000013
1	128	12.657394599999975
2	128	12.712954200000013
3	128	12.673712699999953
4	128	12.679424799999993
5	128	12.62395650000002

Task 3: Counting numbers of triangles in randomly generated graphs using Strassen's

Solution:
Here is our chart, where for each p value we have plotted 5 data points (although for the lower values of p the values are so close to each other that the results are indistinguishable on the graph) and also plotted the expected value curve (which we were told should be $triangles \approx \binom{1024}{3}p^3$), which clearly matches the data extremely well. It's also worth noting that when we set $p \geq 1$, our algorithm correctly returned 178433024 every time, since then we will always have a complete graph and thus our number of triangles is not just expected to be $\binom{1024}{3}$, but must be that as constant, as this is the number of triangles that there would be with all edges in the graph, and with $p \geq 1$ all edges must be included in the graph.


y = (1024C3) x**3
20000
15000
10000
5000
0
triangles
0.00
0.01
0.02
0.03
0.04
0.05
p

□