CS124 ProgAssign2 Report

Nibrass Fathi, Saketh Mynampati

March 2023

1 Strassen's algorithm

To analytically determine the optimal value of n_0 for the crossover point between Strassen's algorithm and the conventional matrix multiplication algorithm, we can analyze the running time of both algorithms.

For Strassen's algorithm, we know that its running time can be expressed as $T(n) = 7T(n/2) + O(n^2)$, where T(n) is the running time of multiplying two nxn matrices using Strassen's algorithm.

More specifically, we find that, combined, the 7 intermediate products in Strassen's algorithm involve 10 additions/subtractions between matrices of size $\frac{n}{2}$ (each taking $\frac{n^2}{4}$ steps because we add element-wise) and 7 products (each taking $T(\frac{n}{2}$ steps because we call the same function recursively.

Meanwhile, when recombining these subproducts, we have 8 additions/subtractions between matrices of size $\frac{n}{2}$ (each taking $\frac{n^2}{4}$ steps because we add element-wise).

In total, we get the more precise recurrence:

$$T(n) = 7 * T(n/2) + 18(n^2/4) = 7 * T(n/2) + 9n^2/2$$

For conventional matrix multiplication, the running time can be expressed as $T(n) = O(n^3)$, as we need to perform n^3 arithmetic operations to multiply two nxn matrices.

More specifically, noting that each addition and multiplication are of length 1 and indexing into various locations happens with negligible cost, we see that to compute each entry of the product (index i, j), we must compute the dot product of the ith row the first factor matrix and the jth column of the second. This involves, for square matrices, n distinct products and $n - 1$ sums, yielding $2n - 1$ steps per entry; this happens for each of n^2 entries.

In total

$$T_{naive}(n) = (2n - 1)(n^2)$$

To determine the optimal value of n_0, we need to find the largest value of n for which **an additional iteration of** Strassen's algorithm on the size-n matrix followed by conventional multiplication of each of the needed sub-matrix products is slower than the conventional algorithm.

We seek this quantity because the naive algorithm has smaller coefficients and a larger asymptotic growth rate; at any matrix size n we know that if the conventional algorithm is more efficient that an extra iteration of Strassen's, this is true for all smaller n.

We don't rigorously prove this, but the functions defining the runtimes of each make this evident and this also follows from the definition of each function's big-O notation, since the conventional algorithm grows faster, after the given crossing over point in each of the even- or odd- cases, conventional will always be slower than Strassen's followed by a conventional multiplication of the submatrices.

In other words, at each potential value of n_0, we must decide whether to run an additional iteration of Strassen's, and we do this by comparing the runtime of one iteration of Strassen's and the required conventional multiplications on matrices of size $n/2$ against the naive runtime of our new size$n$ by n matrix. The largest n_0 for which our conventional algorithm first proves faster is the desired n_0.

If $T_{naive}(n) \geq 7*T_{naive}(n/2)+18(n^2/4)$, then we go ahead and use Strassen's again; if not, then we set $T(n_{new}) = T_{naive}(n_{new})$ In other words, at any time, $T(n) = min(T_{naive}(n), 7T(n/2) + 9n^2/2)$ and since we're looking for the minimum then we know that this $T(n/2) \leq T_{naive}(n/2)$. Hence, we can further simplify our expression to $T(n) \leq min(2n^3 - n^2, 7(n^3/4 - n^2/4) + 9(n/2)^2/2)$. We can use this less precise expression as the question we are asking is simply "do we need **at least one iteration of Strassen's**?"

So basically finding this crossover point n_0 (for which we no longer want to do even one iteration of Strassen's) goes back to solving this inequality:

$$2n^3 - n^2 \leq 7(\frac{n^3}{4} - \frac{n^2}{4}) + \frac{9}{2}n^2$$

We shall solve this inequality in two cases

Case 1: For even n:
We just solve directly, as the matrix can be evenly split for Strassen's:

$$2n^3 - n^2 \leq 7(\frac{n^3}{4} - \frac{n^2}{4}) + \frac{9}{2}n^2$$

Our crossover point is $n_0 = 15$ (any even matrix size less than this should not undergo another iteration of Strassen's).

Case 2: For odd n:
Recall in this case we use padding in order to be able split the matrix into submatrices, resulting in a larger complexity for the Strassen's case:

In this case, we should solve for

$$2n^3 - n^2 \leq 7(\frac{(n+1)^3}{4} - \frac{(n+1)^2}{4}) + \frac{9}{2}(n+1)^2$$

We'll get our (largest integer) crossover point $n_0 = 37$.

2 Algorithm Implementation

We conducted experiments with the same philosophy,

In this section, we discuss optimizations and intermediate steps that led to the final results.

Optimizations:

1. Overall, using Numpy to represent matrices made many operations faster in both algorithms, compressing the data type used and providing helpful indexing features.

Naive Algorithm:

1. We began by iterating through the entry indices and computing the dot product of the desired quantities with np.dot()
2. This used C-code that caused the naive algorithm to far outperform the Strassen's implementation; this was then removed, but left in the code to demonstrate efficiency.
3. Index Optimization. In the final implementation of the naive algorithm, we have counters i, j, and k. Due to Python's optimizations, the outermost loop being the primary index in an array access makes the run time slightly faster, so we used the dot product incrementer, k as the 2nd nested loop.

Strassen:

1. We avoided memory allocation and reuse, because Python's memory management allows Python to perform the best. Using less memory led to slower run-times.

3 Minimizing n_0 Experimental Results

Below is a summary of the results we got experimentally using the Strassen and the conventional algorithm. As hinted by the star, the cutoff n_0 was 20 for even n and 35 when n was odd, which slightly different from our theoretical results which were 15 and 37 respectively. Notably, the even n case indicated that Strassen's was slower than expected, and the odd n had Strassen's as faster than expected. Due to the small difference in values, we believe that the theoretical values are mostly accurate, but here are a few possible causes for the discrepancy:

In our analytical calculation, we assumed that operations such as indexing and accession were negligible, but this may not be true. This could lead to overall slower times, and since Strassen's does more arbitrary indexing and accession, Python may not optimize away the complexity.

Also, our Strassen's variant performs a series of accessions and operations that each require Python to copy a sub-matrix before manipulating it, as Python

doesn't manipulate such objects in place. These indexing times, despite being efficient, could build up. They are at least $O(n^2)$, as this is the amount of data that must be copied over.

In odd n cases, Strassen's is faster than expected (lower cutoff) due to the fact that padding may not actually result in operations that need to be done. Ideally these operations would be optimized out, but even though we didn't explicitly do that, the multiplication goes much faster due to the fact that there are zeros involved, so Python can probably do these very quickly. This closes the gap between the even and odd n cases, and justifies our results.

Another important result to note is that **the values in the input matrices didn't make much of a difference in the relative run times!** This is interesting because if means that our analytical calculation ignoring this was right. This result makes sense because the same multiplications must be done either way, so if large values are present in the matrix, it will bloat the run time of both algorithms (naive and Strassen's). We tested matrices with values [0, 1], [-1, 0, 1], and [0, 1, 2, 3, 4].

Our approach to the analysis is justified by the same logic laid out in part one.

Also, there were a few reasons we chose to implement our algorithms the way they did, mostly to validate out theoretical claims in practice. Each of the two algorithms need to have comparable operations making them up. We can't use C vectorized code for one and not the other, or results will be meaningless when compared to theory, because there is another variable that far outweighs our actual algorithm implementation in terms of effect.

To make this analysis more effective, we removed optimizations such as np.dot() in the conventional algorithm and matrix adding in Strassen's to level the playing field, so that neither used C-operations. The overall slowdown is due to the time it takes for a Python instruction to run, and not because of the inefficiency of the techniques.

Data is shown below. NOTE: Asterisks indicated the found cross-over point, which is the largest n such that conventional is faster.

n	n_0	elements	time
4	*CONVENTIONAL*	[0, 1]	0.00017309188842773438
4	2	[0, 1]	7.009506225585938e-05
5	*CONVENTIONAL*	[0, 1]	5.030632019042969e-05
5	3	[0, 1]	0.0003829002380371094
6	*CONVENTIONAL*	[0, 1]	8.296966552734375e-05
6	3	[0, 1]	0.00013399124145507812
7	*CONVENTIONAL*	[0, 1]	0.00012493133544921875
7	4	[0, 1]	0.00028705596923828125
8	*CONVENTIONAL*	[0, 1]	0.00018501281738281250
8	4	[0, 1]	0.0002522468566894531
9	*CONVENTIONAL*	[0, 1]	0.00026106834411621094
9	5	[0, 1]	0.0004639625549316406
10	*CONVENTIONAL*	[0, 1]	0.00035500526428222656
10	5	[0, 1]	0.00043392181396484375
11	*CONVENTIONAL*	[0, 1]	0.00047016143798828125
11	6	[0, 1]	0.000718167602539062
12	*CONVENTIONAL*	[0, 1]	0.0006079673767089844
12	6	[0, 1]	0.0006921291351318359
13	*CONVENTIONAL*	[0, 1]	0.0007698535919189453
13	7	[0, 1]	0.0010647773742675781
14	*CONVENTIONAL*	[0, 1]	0.0009589195251464844
14	7	[0, 1]	0.001039743423461914
15	*CONVENTIONAL*	[0, 1]	0.00120375405883789
15	8	[0, 1]	0.0015537738800048828
16	*CONVENTIONAL*	[0, 1]	0.0014526844024658203
16	8	[0, 1]	0.0014920234680175781
17	*CONVENTIONAL*	[0, 1]	0.001708984375
17	9	[0, 1]	0.0020952224731445312
18	*CONVENTIONAL*	[0, 1]	0.002017974853515625
18	9	[0, 1]	0.0020599365234375
19	*CONVENTIONAL*	[0, 1]	0.002377033233642578
19	10	[0, 1]	0.002810955047607422
20*	*CONVENTIONAL*	[0, 1]	0.0027692317962646484
20*	10	[0, 1]	0.0027878284454345703
21	*CONVENTIONAL*	[0, 1]	0.0031991004943847656
21	11	[0, 1]	0.003628969192504883
22	*CONVENTIONAL*	[0, 1]	0.003668069839477539
22	11	[0, 1]	0.00359988212585449 2
23	*CONVENTIONAL*	[0, 1]	0.004207134246826172
23	12	[0, 1]	0.004641056060791016
24	*CONVENTIONAL*	[0, 1]	0.004765987396240234
24	12	[0, 1]	0.004608869552612305
25	*CONVENTIONAL*	[0, 1]	0.005373239517211914
25	13	[0, 1]	0.0057909488677978516

n	n_0	elements	time
26	*CONVENTIONAL*	[0, 1]	0.006064891815185547
26	13	[0, 1]	0.005948781967163086
27	*CONVENTIONAL*	[0, 1]	0.006757974624633789
27	14	[0, 1]	0.0072057247161865234
28	*CONVENTIONAL*	[0, 1]	0.0075571537017822266
28	14	[0, 1]	0.00715184211730957
29	*CONVENTIONAL*	[0, 1]	0.008369207382202148
29	15	[0, 1]	0.008713006973266602
30	*CONVENTIONAL*	[0, 1]	0.009277105331420898
30	15	[0, 1]	0.00867009162902832
31	*CONVENTIONAL*	[0, 1]	0.010215997695922852
31	16	[0, 1]	0.010482072830200195
32	*CONVENTIONAL*	[0, 1]	0.011220932006835938
32	16	[0, 1]	0.010435104370117188
33	*CONVENTIONAL*	[0, 1]	0.012306928634643555
33	17	[0, 1]	0.012537956237792969
34	*CONVENTIONAL*	[0, 1]	0.013469219207763672
34	17	[0, 1]	0.012498855590820312
35*	*CONVENTIONAL*	[0, 1]	0.014711856842041016
35*	18	[0, 1]	0.014747142791748047
36	*CONVENTIONAL*	[0, 1]	0.016010761260986328
36	18	[0, 1]	0.014664173126220703
37	*CONVENTIONAL*	[0, 1]	0.017364978790283203
37	19	[0, 1]	0.017216920852661133
38	*CONVENTIONAL*	[0, 1]	0.01886296272277832
38	19	[0, 1]	0.017138957977294922
39	*CONVENTIONAL*	[0, 1]	0.020331859588623047
39	20	[0, 1]	0.01991105079650879
40	*CONVENTIONAL*	[0, 1]	0.021878957748413086
40	20	[0, 1]	0.019908666610717773
41	*CONVENTIONAL*	[0, 1]	0.02355504035949707
41	21	[0, 1]	0.023024320602416992
42	*CONVENTIONAL*	[0, 1]	0.025325775146484375
42	21	[0, 1]	0.02287912368774414
43	*CONVENTIONAL*	[0, 1]	0.027181148529052734
43	22	[0, 1]	0.02626204490661621
44	*CONVENTIONAL*	[0, 1]	0.02909088134765625
44	22	[0, 1]	0.026179075241088867
45	*CONVENTIONAL*	[0, 1]	0.031147003173828125
45	23	[0, 1]	0.02994513511657715
46	*CONVENTIONAL*	[0, 1]	0.03322172164916992
46	23	[0, 1]	0.02986001968383789
47	*CONVENTIONAL*	[0, 1]	0.035394906997680664
47	24	[0, 1]	0.035040855407714844
48	*CONVENTIONAL*	[0, 1]	0.03842616081237793
48	24	[0, 1]	0.033831119537353516
49	*CONVENTIONAL*	[0, 1]	0.0402219295501709
49	25	[0, 1]	0.03835606575012207
50	*CONVENTIONAL*	[0, 1]	0.04279804229736328
50	25	[0, 1]	0.03837180137634277

n	n_0	elements	time
51	*CONVENTIONAL*	[0, 1]	0.045719146728515625
51	26	[0, 1]	0.04372906684875488
52	*CONVENTIONAL*	[0, 1]	0.04910421371459961
52	26	[0, 1]	0.04294919967651367
53	*CONVENTIONAL*	[0, 1]	0.05100297927856445
53	27	[0, 1]	0.04783296585083008
54	*CONVENTIONAL*	[0, 1]	0.053908348083496094
54	27	[0, 1]	0.04913830757141113
55	*CONVENTIONAL*	[0, 1]	0.057650089263916016
55	28	[0, 1]	0.05447816848754883
56	*CONVENTIONAL*	[0, 1]	0.0606389045715332
56	28	[0, 1]	0.0532679557800293
57	*CONVENTIONAL*	[0, 1]	0.06317901611328125
57	29	[0, 1]	0.05931711196899414
58	*CONVENTIONAL*	[0, 1]	0.0672459602355957
58	29	[0, 1]	0.05934786796569824
59	*CONVENTIONAL*	[0, 1]	0.07159209251403809
59	30	[0, 1]	0.06665706634521484
60	*CONVENTIONAL*	[0, 1]	0.07398509979248047
60	30	[0, 1]	0.06504178047180176
61	*CONVENTIONAL*	[0, 1]	0.07734203338623047
61	31	[0, 1]	0.0715792179107666
62	*CONVENTIONAL*	[0, 1]	0.08157896995544434
62	31	[0, 1]	0.07505512237548828
63	*CONVENTIONAL*	[0, 1]	0.0869297981262207
63	32	[0, 1]	0.08014988899230957
4	*CONVENTIONAL*	[-1, 0, 1]	2.7179718017578125e-05
4	2	[-1, 0, 1]	6.4849853515625e-05
5	*CONVENTIONAL*	[-1, 0, 1]	4.887580871582031e-05
5	3	[-1, 0, 1]	0.00021696090698242188
6	*CONVENTIONAL*	[-1, 0, 1]	8.392333984375e-05
6	3	[-1, 0, 1]	0.0001590251922607422
7	*CONVENTIONAL*	[-1, 0, 1]	0.00015401840209960938
7	4	[-1, 0, 1]	0.0002837181091308594
8	*CONVENTIONAL*	[-1, 0, 1]	0.00020503997802734375
8	4	[-1, 0, 1]	0.0002560615539550781
9	*CONVENTIONAL*	[-1, 0, 1]	0.000263214111328125
9	5	[-1, 0, 1]	0.00047397613525390625
10	*CONVENTIONAL*	[-1, 0, 1]	0.0003728866577148437
10	5	[-1, 0, 1]	0.00045680999755859375
11	*CONVENTIONAL*	[-1, 0, 1]	0.000418439483642578
11	6	[-1, 0, 1]	0.0007269382476806641
12	*CONVENTIONAL*	[-1, 0, 1]	0.0006201267242431641
12	6	[-1, 0, 1]	0.0006947517395019531
13	*CONVENTIONAL*	[-1, 0, 1]	0.000782012939453125
13	7	[-1, 0, 1]	0.0010781288146972656
14	*CONVENTIONAL*	[-1, 0, 1]	0.0010249614715576172
14	7	[-1, 0, 1]	0.001062154769897461
15	*CONVENTIONAL*	[-1, 0, 1]	0.0012197494506835938
15	8	[-1, 0, 1]	0.001528024673461914
16	*CONVENTIONAL*	[-1, 0, 1]	0.0014481544494628906
16	8	[-1, 0, 1]	0.0015060901641845703

n	n_0	elements	time
17	*CONVENTIONAL*	[-1, 0, 1]	0.0017862319946289062
17	9	[-1, 0, 1]	0.0021250247955322266
18	*CONVENTIONAL*	[-1, 0, 1]	0.002048015594482422
18	9	[-1, 0, 1]	0.002068042755126953
19	*CONVENTIONAL*	[-1, 0, 1]	0.002500772476196289
19	10	[-1, 0, 1]	0.0028700828552246094
20*	*CONVENTIONAL*	[-1, 0, 1]	0.002804994583129883
20*	10	[-1, 0, 1]	0.0028312206268310547
21	*CONVENTIONAL*	[-1, 0, 1]	0.003244161605834961
21	11	[-1, 0, 1]	0.003715038299560547
22	*CONVENTIONAL*	[-1, 0, 1]	0.003814220428466797
22	11	[-1, 0, 1]	0.0036520957946777344
23	*CONVENTIONAL*	[-1, 0, 1]	0.004302024841308594
23	12	[-1, 0, 1]	0.0047647953033447266
24	*CONVENTIONAL*	[-1, 0, 1]	0.004830121994018555
24	12	[-1, 0, 1]	0.004728078842163086
25	*CONVENTIONAL*	[-1, 0, 1]	0.005486011505126953
25	13	[-1, 0, 1]	0.005925178527832031
26	*CONVENTIONAL*	[-1, 0, 1]	0.006207942962646484
26	13	[-1, 0, 1]	0.005945920944213867
27	*CONVENTIONAL*	[-1, 0, 1]	0.006967067718505859
27	14	[-1, 0, 1]	0.007328987121582031
28	*CONVENTIONAL*	[-1, 0, 1]	0.007781982421875
28	14	[-1, 0, 1]	0.007281780242919922
29	*CONVENTIONAL*	[-1, 0, 1]	0.008600950241088867
29	15	[-1, 0, 1]	0.009155750274658203
30	*CONVENTIONAL*	[-1, 0, 1]	0.009602069854736328
30	15	[-1, 0, 1]	0.00881195068359375
31*	*CONVENTIONAL*	[-1, 0, 1]	0.010539054870605469
31*	16	[-1, 0, 1]	0.01085519790649414
32	*CONVENTIONAL*	[-1, 0, 1]	0.011664867401123047
32	16	[-1, 0, 1]	0.01076507568359375
33	*CONVENTIONAL*	[-1, 0, 1]	0.012839317321777344
33	17	[-1, 0, 1]	0.01272892951965332
34	*CONVENTIONAL*	[-1, 0, 1]	0.01376795768737793
34	17	[-1, 0, 1]	0.012511014938354492
35	*CONVENTIONAL*	[-1, 0, 1]	0.014940023422241211
35	18	[-1, 0, 1]	0.014811277389526367
36	*CONVENTIONAL*	[-1, 0, 1]	0.01628279685974121
36	18	[-1, 0, 1]	0.01473093032836914
37	*CONVENTIONAL*	[-1, 0, 1]	0.017806053161621094
37	19	[-1, 0, 1]	0.017409086227416992
38	*CONVENTIONAL*	[-1, 0, 1]	0.01931285858154297
38	19	[-1, 0, 1]	0.017328977584838867
39	*CONVENTIONAL*	[-1, 0, 1]	0.020904064178466797
39	20	[-1, 0, 1]	0.02043318748474121
40	*CONVENTIONAL*	[-1, 0, 1]	0.022683143615722656
40	20	[-1, 0, 1]	0.020589828491210938
41	*CONVENTIONAL*	[-1, 0, 1]	0.02439284324645996
41	21	[-1, 0, 1]	0.023476123809814453

n	n_0	elements	time
42	*CONVENTIONAL*	[-1, 0, 1]	0.026087045669555664
42	21	[-1, 0, 1]	0.023307085037231445
43	*CONVENTIONAL*	[-1, 0, 1]	0.0279080867767334
43	22	[-1, 0, 1]	0.02667522430419922
44	*CONVENTIONAL*	[-1, 0, 1]	0.029732227325439453
44	22	[-1, 0, 1]	0.02656412124633789
45	*CONVENTIONAL*	[-1, 0, 1]	0.031826019287109375
45	23	[-1, 0, 1]	0.030915021896362305
46	*CONVENTIONAL*	[-1, 0, 1]	0.03437972068786621
46	23	[-1, 0, 1]	0.0311739444732666
47	*CONVENTIONAL*	[-1, 0, 1]	0.037487030029296875
47	24	[-1, 0, 1]	0.03503274917602539
48	*CONVENTIONAL*	[-1, 0, 1]	0.03922295570373535
48	24	[-1, 0, 1]	0.034606218338012695
49	*CONVENTIONAL*	[-1, 0, 1]	0.041605234146118164
49	25	[-1, 0, 1]	0.03935813903808594
50	*CONVENTIONAL*	[-1, 0, 1]	0.044049978256225586
50	25	[-1, 0, 1]	0.039019107818603516
51	*CONVENTIONAL*	[-1, 0, 1]	0.047078847885131836
51	26	[-1, 0, 1]	0.04514312744140625
52	*CONVENTIONAL*	[-1, 0, 1]	0.06340885162353516
52	26	[-1, 0, 1]	0.04404282569885254
53	*CONVENTIONAL*	[-1, 0, 1]	0.05205583572387695
53	27	[-1, 0, 1]	0.04877614974975586
54	*CONVENTIONAL*	[-1, 0, 1]	0.05593276023864746
54	27	[-1, 0, 1]	0.049645185470581055
55	*CONVENTIONAL*	[-1, 0, 1]	0.05940890312194824
55	28	[-1, 0, 1]	0.05434417724609375
56	*CONVENTIONAL*	[-1, 0, 1]	0.06259512901306152
56	28	[-1, 0, 1]	0.05444693565368652
57	*CONVENTIONAL*	[-1, 0, 1]	0.06494283676147461
57	29	[-1, 0, 1]	0.0601038932800293
58	*CONVENTIONAL*	[-1, 0, 1]	0.0687251091003418
58	29	[-1, 0, 1]	0.060040950775146484
59	*CONVENTIONAL*	[-1, 0, 1]	0.07341289520263672
59	30	[-1, 0, 1]	0.0679631233215332
60	*CONVENTIONAL*	[-1, 0, 1]	0.07750988006591797
60	30	[-1, 0, 1]	0.0672760009765625
61	*CONVENTIONAL*	[-1, 0, 1]	0.08047986030578613
61	31	[-1, 0, 1]	0.07412314414978027
62	*CONVENTIONAL*	[-1, 0, 1]	0.0833582878112793
62	31	[-1, 0, 1]	0.07230901718139648
63	*CONVENTIONAL*	[-1, 0, 1]	0.08725190162658691
63	32	[-1, 0, 1]	0.08078193664550781
4	*CONVENTIONAL*	[0, 1, 2, 3, 4]	2.6702880859375e-05
4	2	[0, 1, 2, 3, 4]	6.413459777832031e-05

n	n_0	elements	time
5	*CONVENTIONAL*	[0, 1, 2, 3, 4]	4.887580871582031e-05
5	3	[0, 1, 2, 3, 4]	0.00023221969604492188
6	*CONVENTIONAL*	[0, 1, 2, 3, 4]	8.320808410644531e-05
6	3	[0, 1, 2, 3, 4]	0.00013303756713867188
7	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.0001251697540283203
7	4	[0, 1, 2, 3, 4]	0.00028705596923828125
8	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.00018405914306640625
8	4	[0, 1, 2, 3, 4]	0.0002510547637939453
9	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.00026202201843261 72
9	5	[0, 1, 2, 3, 4]	0.0004642009735107422
10	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.00035381317138671875
10	5	[0, 1, 2, 3, 4]	0.00043201446533203125
11	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.0004680156707763672
11	6	[0, 1, 2, 3, 4]	0.0007307529449462891
12	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.0006051063537597656
12	6	[0, 1, 2, 3, 4]	0.0006928443908691406
13	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.0007681846618652344
13	7	[0, 1, 2, 3, 4]	0.0010690689086914062
14	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.0009779930114746094
14	7	[0, 1, 2, 3, 4]	0.0010371208190917969
15	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.001171112060546875
15	8	[0, 1, 2, 3, 4]	0.001516103744506836
16	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.00142121315002441 4
16	8	[0, 1, 2, 3, 4]	0.0015101432800292969
17	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.0017139911651611328
17	9	[0, 1, 2, 3, 4]	0.0020830631256103516
18	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.0020389556884765625
18	9	[0, 1, 2, 3, 4]	0.0020737648010253906
19	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.0024030208587646484
19	10	[0, 1, 2, 3, 4]	0.002810955047607422
20*	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.00277829170227050 8
20*	10	[0, 1, 2, 3, 4]	0.0027818679809570312
21	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.0031909942626953125
21	11	[0, 1, 2, 3, 4]	0.0036351680755615234
22	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.0037381649017333984
22	11	[0, 1, 2, 3, 4]	0.0036230087280273438
23	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.004186868667602539
23	12	[0, 1, 2, 3, 4]	0.004621028900146484
24	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.0047800540924072266
24	12	[0, 1, 2, 3, 4]	0.0046558380126953125
25	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.005402803421020508
25	13	[0, 1, 2, 3, 4]	0.005933046340942383
26	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.006048917770385742
26	13	[0, 1, 2, 3, 4]	0.005788087844848633
27	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.006809234619140625
27	14	[0, 1, 2, 3, 4]	0.007202625274658203
28	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.007599830627441406
28	14	[0, 1, 2, 3, 4]	0.0071718692779541016

n	n_0	elements	time
29	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.008394002914428711
29	15	[0, 1, 2, 3, 4]	0.008954048156738281
30	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.00934600830078125
30	15	[0, 1, 2, 3, 4]	0.008738040924072266
31	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.01055002212524414
31	16	[0, 1, 2, 3, 4]	0.010797977447509766
32	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.011417865753173828
32	16	[0, 1, 2, 3, 4]	0.01091313362121582
33	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.012571096420288086
33	17	[0, 1, 2, 3, 4]	0.01275181770324707
34	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.013653993606567383
34	17	[0, 1, 2, 3, 4]	0.012836217880249023
35*	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.014992952346801758
35*	18	[0, 1, 2, 3, 4]	0.015079975128173828
36	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.016275882720947266
36	18	[0, 1, 2, 3, 4]	0.014979124069213867
37	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.017616748809814453
37	19	[0, 1, 2, 3, 4]	0.01765298843383789
38	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.019074678421020508
38	19	[0, 1, 2, 3, 4]	0.01729726791381836
39	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.020301103591918945
39	20	[0, 1, 2, 3, 4]	0.019931316375732422
40	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.02204108238220215
40	20	[0, 1, 2, 3, 4]	0.019979000091552734
41	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.023629188537597656
41	21	[0, 1, 2, 3, 4]	0.023099899291992188
42	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.025307178497314453
42	21	[0, 1, 2, 3, 4]	0.022883892059326172
43	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.02711009979248047
43	22	[0, 1, 2, 3, 4]	0.026233911514282227
44	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.029082775115966797
44	22	[0, 1, 2, 3, 4]	0.02618098258972168
45	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.031065702438354492
45	23	[0, 1, 2, 3, 4]	0.029950857162475586
46	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.033200740814208984
46	23	[0, 1, 2, 3, 4]	0.029884815216064453
47	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.03545498847961426
47	24	[0, 1, 2, 3, 4]	0.03414297103881836
48	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.037896156311035156
48	24	[0, 1, 2, 3, 4]	0.03411507606506348
49	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.04062390327453613
49	25	[0, 1, 2, 3, 4]	0.03887820243835449
50	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.043512821197509766
50	25	[0, 1, 2, 3, 4]	0.038971900939941406

n	n_0	elements	time
51	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.0456690788269043
51	26	[0, 1, 2, 3, 4]	0.04315614700317383
52	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.048191070556640625
52	26	[0, 1, 2, 3, 4]	0.042844295501708984
53	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.050673723220825195
53	27	[0, 1, 2, 3, 4]	0.04819202423095703
54	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.05369400978088379
54	27	[0, 1, 2, 3, 4]	0.0479578971862793
55	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.056648969650268555
55	28	[0, 1, 2, 3, 4]	0.05318593978881836
56	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.05972886085510254
56	28	[0, 1, 2, 3, 4]	0.05307412147521973
57	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.06311917304992676
57	29	[0, 1, 2, 3, 4]	0.05903005599975586
58	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.06640100479125977
58	29	[0, 1, 2, 3, 4]	0.05892133712768555
59	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.06996583938598633
59	30	[0, 1, 2, 3, 4]	0.06498908996582031
60	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.07394790649414062
60	30	[0, 1, 2, 3, 4]	0.06528687477111816
61	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.0796048641204834
61	31	[0, 1, 2, 3, 4]	0.07476282119750977
62	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.08245301246643066
62	31	[0, 1, 2, 3, 4]	0.07172393798828125
63	*CONVENTIONAL*	[0, 1, 2, 3, 4]	0.08536195755004883
63	32	[0, 1, 2, 3, 4]	0.07901597023010254

4 Triangles

As you can see below, We generated randomized graphs with 1024 vertices each and a probability p of an edge between two distinct vertices. 5 trials were run for each p value to find the number of triangles that can be found in such a graph, using our implementation of Strassen's algorithm as a subroutine.

n	p	triangles
1024	0.01	203
1024	0.01	171
1024	0.01	195
1024	0.01	166
1024	0.01	189
1024	0.02	1377
1024	0.02	1412
1024	0.02	1353
1024	0.02	1400
1024	0.02	1460
1024	0.03	4875
1024	0.03	4753
1024	0.03	4755
1024	0.03	5029
1024	0.03	4721
1024	0.04	11531
1024	0.04	11808
1024	0.04	11532
1024	0.04	11737
1024	0.04	11324
1024	0.05	22296
1024	0.05	22280
1024	0.05	23203
1024	0.05	22083
1024	0.05	22445

This can all be captured in the table below:

p	actual	expected	% error
0.01	184.8	178.43	3.6
0.02	1400.4	1427.46	1.9
0.03	4826.6	4817.69	0.18
0.04	11586.4	11419.71	1.46
0.05	22461.4	22304.128	0.7

All of the experimental values are within a few percentage points of the expected values. The actual values lie on a distribution due to the fact that we are generating random graphs, and due to the relatively small number of trials we ran, there are slight deviations from the expected value of $\binom{1024}{3}p^3$. There is no systematic difference, as some quantities are lower than expected and some are higher.

We confirmed that the multiplication is correct, and double checked to make sure no trend was readily apparent by creating higher probability graphs, up

until $p = 1$ and making sure that the number of triangles output was correct.