ProgSet 2

CS 124: Data Structures and Algorithms

Due: Wednesday, March 27, 2024 **Denny Cao and Ossimi Ziv**

---

§1 Quantitative Results

§1.1 Analytical Crossover Calculation

The crossover point is the point at which `strassen` becomes faster than the naive algorithm. We can calculate the crossover point by setting the two algorithms equal to each other and solving for n.

The runtime of `standard` is

$$T(n) = n^2(2n-1)$$

assuming that all arithmetic operations have a cost of 1 by Task 1. This is because, for each of the resulting n^2 numbers in the resulting matrix, there are a total of n multiplications and $n-1$ additions.

For `strassen`, we only run "one layer" of the algorithm, with the subproblems using `standard` in order to calculate the resulting matrix. There are two cases:

1. n is even. This means that the runtime of `strassen` is

   $$T'(n) = 7T(n/2) + 18(n/2)^2$$

   as there are 7 subproblems and 18 matrix additions ($(n/2)^2$ elements in each) in the algorithm, since we can evenly split submatrices.

   We can now set the two algorithms equal to each other and solve for n:

   $$\begin{aligned} 2n^3 - n^2 &= 7(2(n/2)^3 - (n/2)^2) + 18(n/2)^2 \\ &= \frac{7}{4}n^3 - \frac{7}{4}n^2 + \frac{18}{4}n^2 \\ 0 &= -\frac{1}{4}n^3 + \frac{15}{4}n^2 \\ &= -\frac{1}{4}n^2(n-15) \\ n &= 15 \end{aligned}$$

   We can see that the crossover point is $n_0 = 15$.

2. n is odd. This means that the runtime of `strassen` is

   $$T'(n) = 7T((n+1)/2) + 18((n+1)/2)^2$$

   This is because the algorithm will pad the matrix by adding a column and row of zeros to the matrix, making the input size for subproblems $(n+1)/2$.

We can now set the two algorithms equal to each other and solve for n:

$$\begin{aligned} 2n^3 - n^2 &= 7(2((n+1)/2)^3 - ((n+1)/2)^2) + 18((n+1)/2)^2 \\ 0 &= \frac{7}{4}(n+1)^3 + \frac{11}{4}(n+1)^2 - 2n^3 + n^2 \\ &= \frac{7}{4}\left(n^3 + 3n^2 + 3n + 1\right) + \frac{11}{4}\left(n^2 + 2n + 1\right) - 2n^3 + n^2 \\ &= -\frac{1}{4}n^3 + 9n^2 + \frac{43}{4}n + \frac{18}{4} \\ n &= 37.17 \end{aligned}$$

Thus, the crossover point is around $n_0 = 37$.

We combine the two cases to get the crossover point for all n:

- For $n < 15$, `standard` is faster.
- For $15 \leq n \leq 37$, it is unclear which algorithm is faster.
- For $n > 37$, `strassen` is faster.

Thus, the theoretical crossover point is $n_0 = 37$.

§1.2 Empirical Crossover

We tested using a Macbook Pro M2 Pro 14', with a M2 Pro processor and 16GB of RAM.

We obtain the empirical crossover point by running the two algorithms on random matrices of size n with entires 0 and 1 and timing them. We then plot the results and find the point at which, for all n greater than the crossover point, `strassen` is faster. A subset of the results are shown below, taking the average of 5 runs for each matrix size between 1 and 50.

Matrix Size (n)	Average Time Strassen (ms)	Average Time Standard (ms)	Matrix Size (n)	Average Time Strassen (ms)	Average Time Standard (ms)
1	0.004005432	0.001764297	26	3.75418663	4.20546532
2	0.025367737	0.004434586	27	4.70714569	4.67915535
3	0.244951248	0.024557114	28	4.69846725	5.22465706
4	0.103759766	0.045967102	**29**	**5.78403473**	**5.77650070**
5	0.276184082	0.08940697	30	5.74755669	6.42280579
6	0.08349419	0.05903244	31	6.94327354	7.02953339
7	0.19207001	0.09231567	32	6.93907738	7.74512291
8	0.14777184	0.13208389	33	8.31937789	8.45537186
9	0.29582977	0.18420219	34	8.70699883	9.58261489
10	0.26364326	0.24600029	35	9.94524956	10.25118828
11	0.44384003	0.32444000	36	9.80730057	10.96653938
12	**0.41499138**	**0.42495727**	37	11.51275635	12.00428009
13	0.70180892	0.57215691	38	11.48490906	12.88061142
14	0.67152977	0.70323944	39	13.39626312	13.96603584
15	0.98776817	0.87027550	40	13.42787743	15.08932114
16	0.96721649	1.02620125	41	15.63568115	16.28928185
17	1.36899948	1.19962692	42	15.51976204	17.43607521
18	1.30710602	1.41773224	43	17.73638725	18.69397163
19	1.76682472	1.64866447	44	17.71345139	19.96340752
20	1.73182487	1.92041397	45	20.18704414	21.47617340
21	2.31013298	2.22172737	46	20.11113167	22.76115417
22	2.32915878	2.55317688	47	22.77207374	24.14793968
23	3.03268432	2.95033455	48	22.81498909	25.85563660
24	3.01985741	3.30181122	49	25.80103874	27.45056152
25	3.78847122	3.72204781	50	27.79173851	29.60662842

Table 1: Average runtimes of `strassen` and `standard` for matrix sizes n

We observe that, for $n > 29$, `strassen` is faster than `standard`. Thus, the empirical crossover point is $n_0 = 29$.

We also observe that, for $n < 12$, `standard` is faster than `strassen`, and for $12 \leq n \leq 29$, it is unclear which algorithm is faster.

§2 Counting Triangles

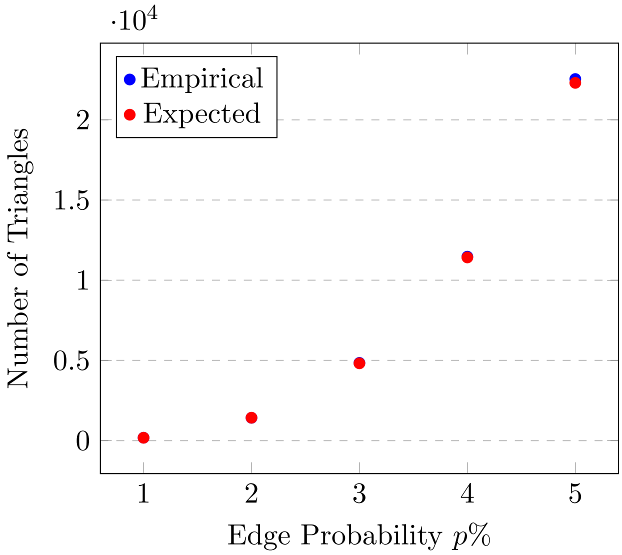


Figure 1: Empirical and expected number of triangles in triangles of random graphs with edge probability p

Trial	1%	2%	3%	4%	5%
1	193.0	1422.0	4738.0	11622.0	23007.0
2	183.0	1415.0	4836.0	11057.0	22146.0
3	168.0	1396.0	4933.0	11660.0	23159.0
4	184.0	1508.0	4776.0	11373.0	22129.0
5	168.0	1359.0	4951.0	11633.0	22284.0
Average	179.2	1420.0	4846.8	11469.0	22545.0
Expected	178.433024	1427.464192	4817.691648	11419.713536	22304.128

Table 2: Empirical and expected number of triangles in triangles of random graphs with edge probability p

The counts are similar, and the error can be due to the edge probabilities; in the actual trial, with more triangle counts, there may be more edges than the probability times 1024. With less, there may be less edges.

§3 Discussion

§3.1 Results: Analytical v. Empirical

We observe that n_0 is lower than the theoretical value when tested empirically $(29 < 37)$. This means that `Strassen` could handle matrices 8 sizes larger than predicted before it would be faster to swap over to the standard algorithm. This in turn implies one of two things; 1: `Strassen` was faster than we predicted, and better at handling the operations than we had calculated it to be, or 2: the standard algorithm was slower than we accounted for.

§3.1.1 Reasoning behind these results

Because we assumed that addition and multiplication come at a cost of $O(1)$ time it is nearly impossible that `Strassen` could be better than the mathematical estimate we gave it. Instead, what is likely is that our simplification of the standard algorithm and abbreviation of its operations to a constant time wasn't mirroring what happened in the system and how our computer processor handled those operations. CPUS are built in a way that optimizes their capability to perform operation on multiple points simultaneously, aka vectorization. The standard algorithm as it is simply adds and multiplies on an element by element basis, not taking advantage of any vectorization. The runtime is further harmed by Python's overhead, which is increased by each of these individual operations, causing longer delays.

Thus the standard algorithm, having to handle more operations than `Strassen` was slightly slower than we predicted, causing it to be more efficient to remain on `Strassen` for longer before crossing over. Hence the lower actual cross-over point.

Another potential reason for why the standard algorithm is slower than expected is the repeated indexing in the triple-for-loop. The culprit is the continuous calling for the indexes of the y array in this block of the code in the definition of the standard algorithm:

```
for i in range(x.shape[0]):
    for j in range(y.shape[1]):
        for k in range(y.shape[0]):
            result[i, j] += x[i, k] * y[k, j]
```

Python stores its arrays in contiguous blocks of memory, so in a double array as we have here, $arr[0,0]$ would be stored right next to $arr[0,1]$ which in turn is right next to $arr[0,2]$. This makes accessing indexes in sequential order by the second index very convenient and friendly for the machine. (accessing the `x` array by increasing k). However, when indexing the y array, since the SECOND index j is being held constant by the outer for-loop, while k (the first index) changes, we are not accessing memory sequentially and continuously as would be convenient, and instead jumping around rows of memory. The cache line that would store memory around $y[k,j]$ might not contain memory at $y[k+1,j]$ for larger matrices. As a result, we can encounter cache misses in our code, causing the program to potentially seek values from main memory or RAM, a significantly slower process, thus slowing the speed of our standard algorithm.

§3.1.2 Optimizations to Both Algorithms

To improve `standard` algorithm, we can change the order of the for loops (Almurayh 2022). In doing so, we improve the spacial locality of the algorithm to take advantage of cache more which will reduce the amount of "miss rate." We analyze the speed up below with low orders of n, as this results in speed ups within constants, which will not be seen with large orders of n:

Matrix Size n	Average Time Optimized Standard	Average Time Standard
1	1.7642974853515626e-06	3.62396240234375e-06
2	3.7670135498046877e-06	4.1961669921875e-06
3	9.012222290039062e-06	9.584426879882813e-06
4	1.9168853759765626e-05	1.9788742065429688e-05
5	3.561973571777344e-05	3.5762786865234375e-05
6	5.917549133300781e-05	5.941390991210937e-05
7	9.2315673828125e-05	9.260177612304687e-05
8	0.00013594627380371095	0.00013575553894042968
9	0.00018663406372070311	0.00018858909606933594
10	0.00025043487548828127	0.00025115013122558596

Table 3: Average runtimes of `optimized standard` and `standard` for matrix sizes n

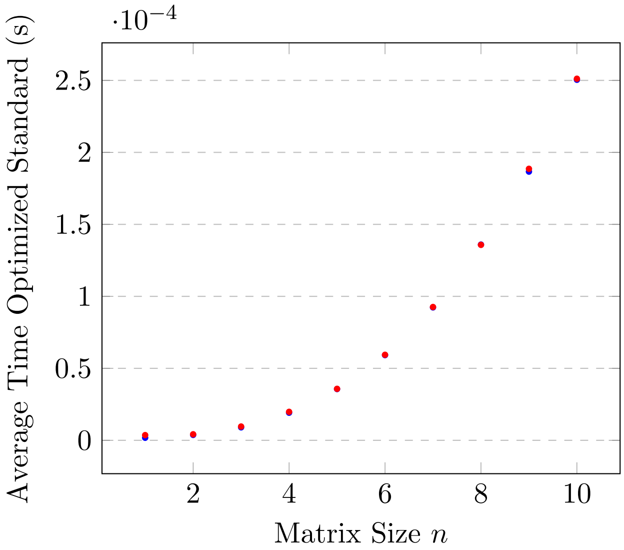


Figure 2: Average runtimes of `optimized standard` and `standard` for matrix sizes n

To optimize the `Strassen` implementation, we can implement the `Winograd` varient of the algorithm which reduces the number of addition/subtraction operations from 18 to 15, slightly cutting costs. We go over our implementation of this optimization later in the discussion. With regards to the computer, we can creatively pre-allocate and reuse memory so that `Strassen` doesn't unnecessarily continuously create new space that won't be referenced again.

The nature of a "cross-over" point in this algorithm when one becomes faster than the other is evidently a rather abstract measure that is heavily dependent on the hardware dependencies of whatever device is running the code (ie. processor speed, number of cores, cache size, memory bandwidth, etc). The cross-over point will vary based on what language it is implemented in, how each algorithm allocates its memory, and the CPU and RAM usage of the system running it. Interestingly, when reading the wikipedia article on `Strassen`, there was even a quote that "a 2010 study found that even a single step of Strassen's algorithm is often not beneficial on current architectures, compared to a highly optimized traditional multiplication, until matrix sizes exceed 1000 or more, and even for matrix sizes of several thousand the benefit is typically marginal at best" (D'Alberto et al.) Evidently, the "optimal cross-over point" in practical application is of a somewhat elusive nature very dependent on one's intention and execution.

§3.2 Implementation

§3.2.1 Padding: Handling Odd Sized Matrices

An interesting difficulty we encountered was how to efficiently implement padding into the algorithm. The initially intuitive solution we had was to pre-process the matrix with padding up to the nearest power of 2, and then post-process the result by only returning up to the original size after `strassen` algorithm had finished running to remove the pads. However, we wanted (yet struggled) to find an implementation that incorporated all the padding into the single `strassen` function. The various calls of recursion made it difficult to keep track of when un-padding should be done, as defining it within the function would cause issues with recursive calls.

Our next idea was to incorporate the padding as part of the `split` function call. We add a row and column of 0s for padding if the input matrix x has an odd size, which will result in an even sized matrix, and then call `split` again to split the matrix evenly. We now prove that this is correct.

> **Claim 3.1** — The result of multiplying the padded matrices is the same as the result of multiplying the original matrices.

Proof. Let A and B be odd matrices, and A' and B' be the padded matrices. We have that

$$A' = \begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix}, \quad B' = \begin{bmatrix} B & 0 \\ 0 & 0 \end{bmatrix}$$

We can now multiply the matrices using `strassen`, as they are now even-sized and can be split:

$$\begin{aligned} A'B' &= \begin{bmatrix} A & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} B & 0 \\ 0 & 0 \end{bmatrix} \\ &= \begin{bmatrix} AB & 0 \\ 0 & 0 \end{bmatrix} \end{aligned}$$

We can see that, after removing the padding, the result is the same as the result of multiplying the original matrices, and the proof is complete. We just have to ensure to trim the matrix to its original dimensions at the end so that the dimensions returned are the correct size and the 0's don't erroneously affect any other calculations. □

§3.3 Optimizations

A small optimization we decided to add after preliminary testing was implementing the `winograd` form of the algorithm discussed in Remark 5 of Lecture 9, maintaining the asymptotic runtime but reducing the number of additions/subtractions from 18 to 15. Mathematically, this changes the constant number of operations performed, and hypothetically reduces the cross-over point from 37 to 34

$$2n^3 - n^2 = 7T(\frac{n}{2}) + \mathbf{15}(\frac{n}{2})^2$$

$$2n^3 - n^2 = 7(2(\frac{n}{2})^3 - (\frac{n}{2})^2) + 15(\frac{n}{2})^2$$

Reducing the calculation gives $n_0 = 12$ for powers of two. Similarly, accounting for the padding on numbers not power of two with the updated number of operations, we have:

$$2n^3 - n^2 = 7(2(\frac{n+1}{2})^3 - (\frac{n+1}{2})^2) + \mathbf{15}(\frac{n+1}{2})^2$$

Which reduces to $n_0 \approx 34$, a minor reduction estimate. Thus we would expect that the empirical cross-over point of `Winograd` is a very small difference away from that of `Strassen`.

With empirical testing, we obtain the following data, averaging times from 5 trials:

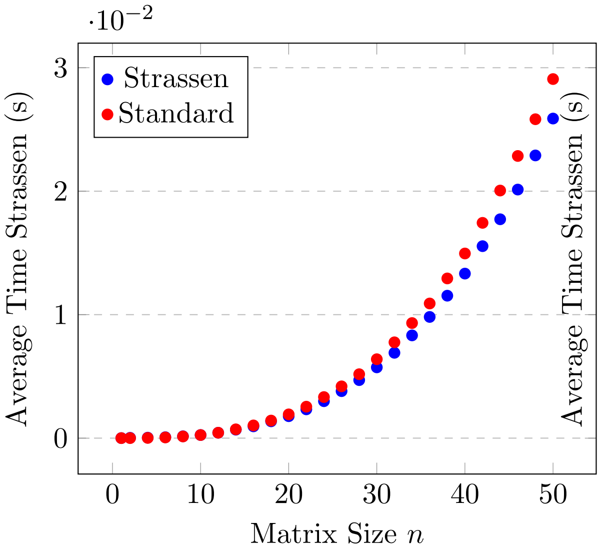
Matrix Size (n)	Average Time Winograd (s)	Average Time Standard (s)
1	3.814697265625e-06	1.71661376953125e-06
2	2.6178359985351562e-05	3.862380981445312e-06
3	8.440017700195312e-05	9.107589721679688e-06
4	4.191398620605469e-05	1.873970031738281e-05
5	9.965896606445312e-05	3.342628479003906e-05
6	8.044242858886718e-05	5.621910095214844e-05
7	0.00016927719116210938	8.921623229980468e-05
8	0.00015482902526855468	0.00013575553894042968
9	0.0002984523773193359	0.0001971721649169922
10	0.00027599334716796873	0.00026311874389648435
11	0.00046706199645996094	0.00033812522888183595
12	0.000435638427734375	0.0004309177398681641
13	0.000666666030883789	0.0005465984344482422
14	0.0006531238555908203	0.0006655693054199219
15	0.0009731292724609375	0.0008263587951660156
16	0.0009289741516113281	0.0009956836700439453
17	0.0013453960418701172	0.0011841297149658204
18	0.00130157470703125	0.0014186859130859374
19	0.001779794692993164	0.001665019989013672
20	0.0017538070678710938	0.0019189834594726563
21	0.0023659229278564452	0.002222633361816406
22	0.002311897277832031	0.0025493621826171873
23	0.0029861927032470703	0.002897500991821289
24	0.002963542938232422	0.0033051013946533204
25	0.0037668704986572265	0.003717947006225586
26	0.003761768341064453	0.004164743423461914
27	**0.004654836654663086**	**0.004653787612915039**
28	0.004636573791503906	0.005214595794677734
29	0.005778264999389648	0.005764532089233399
30	0.005742979049682617	0.006369829177856445
31	0.006974220275878906	0.00701904296875
32	0.006930398941040039	0.007732200622558594
33	0.00825643539428711	0.008376169204711913
34	0.008147287368774413	0.00911092758178711
35	0.009667158126831055	0.009928178787231446
36	0.009735441207885743	0.010892295837402343
37	0.011501312255859375	0.011954784393310547
38	0.011496591567993163	0.012907838821411133
39	0.01336979866027832	0.013849735260009766
40	0.013233804702758789	0.014958429336547851
41	0.015453815460205078	0.016249704360961913
42	0.015420484542846679	0.017423534393310548
43	0.01782994270324707	0.018680667877197264
44	0.01777782440185547	0.020090293884277344
45	0.020171403884887695	0.021331262588500977
46	0.020185375213623048	0.022751760482788087
47	0.022836875915527344	0.024216747283935545
48	0.02294459342956543	0.026280975341796874
49	0.026479005813598633	0.02758617401123047
50	0.025884246826171874	0.029093170166015626

By observing this data, we can see that the crossover point, (ie the final time that the `Winograd` algorithm is slower than the standard algorithm) is around $n_0 = 27$ This is a very slight improvement over the n_0 value of 29 of `Strassen`. A very slight improvement of the exact magnitude we would expect.

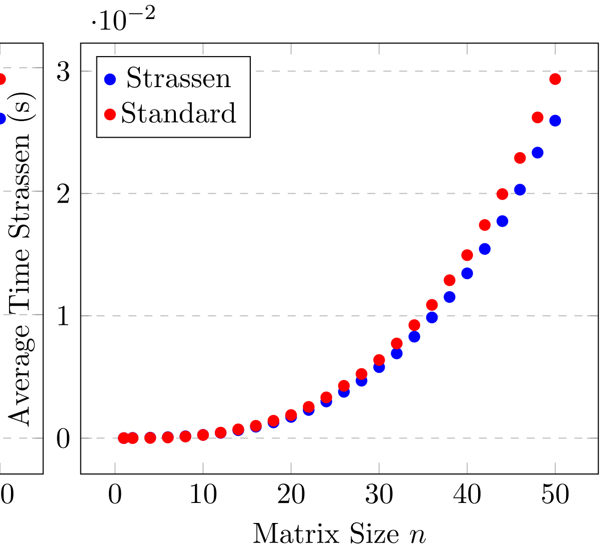
§3.4 Matrix Choice

We chose to multiply all matrices between 1-50: powers of two and non-powers of two. The matrix choice considerably varies the number of operations that need to be conducted as for non-powers of two, padding and un-padding has to applied, adding extra cost and changing the size of the matrix that is operated on. Thus, to ensure that our n_0 value was sound across all variations of the operations and we'd get general results, we decided to not limit which types of matrices we tested

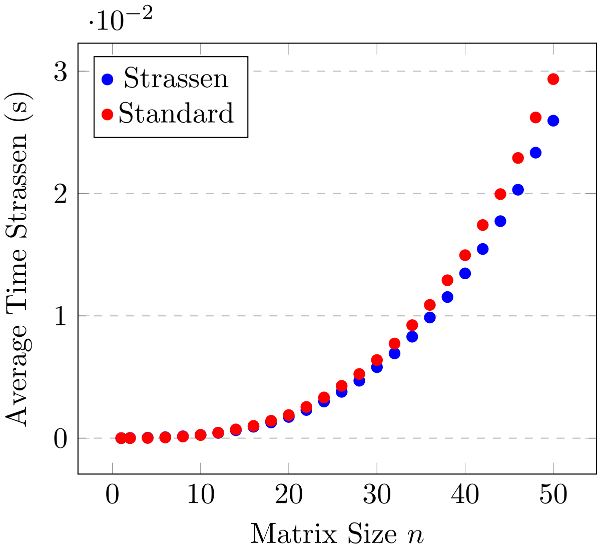
To ensure close to constant time arithmetic as possible, we chose to use random matrices with entries 0 and 1. With 0/1, -1/1, and 0/1/2 matrices, the runtime was roughly the same. We graph the results below:



(a) Average runtimes of `strassen` and `standard` for 0,1,2 matrices of size n



(b) Average runtimes of `strassen` and `standard` for -1,1 matrices of size n



(c) Average runtimes of `strassen` and `standard` for 2^{25} to 2^{26} matrices of size n