# PROGRAMMING PSET 2 - CS 124

KEVIN PATEL

## 1. Determining the Crossover point

This first felt as a very difficult thing to determine but following the instruction to count each arithmetic operation proved very helpful. At first, I thought the crossover point was simply to find when Strassen's would be better than conventional but that would of course be a very high value. I then realized the point was to find a point during your use of Strassen's, where you simply call the conventional method. This of course proved to yield a much lower crossover point. Each Strassens call calls Strassens 7 times with input n/2, and then there is other arithmetic that is done. I decided to look at those each separately as well as find out the cost of the other arithmetic operations done. I've listed each of the costs incurred in my code so it's easy to see but will also list here.
Strassens(n):
- 1 to find $n/2$
- 8 calls to newmatrix, each with cost $n/2$
- $n/2$ for first for loop
-$(n/2)^2$ for nested for loop
- 7 calls to Strassens with input $n/2$
-18 calls to add/subtract which come out to $(18*3)*(n/2)^2 + 18*(n/2)$
-$n/2$ cost for for for loop
- $(n/2)^2$ cost for nested for loop

Ultimately, the cost comes out to $50(n/2)^2 + 28(n/2) + 1$ for each strassens call + the cost of 7 Strassens calls at n/2 or calling the conventional function 7 times. In the chart below, we can see the cost of just using Strassens, just using conventional, and the the result of utilizing conventional when beneficial at lower n values.

| n | 7 Strassen calls at n/2: 7*Strassens only cost for n/2 | Other arithmetic: 50(n/2)^2 + 28(n/2) +1 | Strassens only Cost | Conventional only Cost | Utilizing Straseens at n and conventional at n/2 | Only Strassen vs Only Conventional |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 1 | 0 |
| 2 | 7 | 79 | 86 | 16 | 86 | -70 |
| 4 | 602 | 257 | 859 | 128 | 369 | -731 |
| 8 | 6013 | 913 | 6926 | 1024 | 1809 | -5902 |
| 16 | 48482 | 3425 | 51907 | 8192 | 10593 | -43715 |
| 32 | 363349 | 13249 | 376598 | 65536 | 70593 | -311062 |
| 64 | 2636186 | 52097 | 2688283 | 524288 | 510849 | -2163995 |
| 128 | 18817981 | 206593 | 19024574 | 4194304 | 3876609 | -14830270 |
| 256 | 133172018 | 822785 | 133994803 | 33554432 | 30182913 | -100440371 |
| 512 | 937963621 | 3283969 | 941247590 | 268435456 | 238164993 | -672812134 |
| 1024 | 6588733130 | 13121537 | 6601854667 | 2147483648 | 1892169729 | -4454371019 |
| 2048 | 46212982669 | 52457473 | 46265440142 | 17179869184 | 15084843009 | -29085570958 |
| 4096 | 323858080994 | 209772545 | 324067853539 | 137438953472 | 120468856833 | -186628900067 |
| 8192 | 2268474974773 | 838975489 | 2269313950262 | 1099511627776 | 962911649793 | -1169802322486 |
| 16384 | 15885197651834 | 3355672577 | 15888553324411 | 8796093022208 | 7699937067009 | -7092460302203 |
| 32768 | 111219873270877 | 13422231553 | 111233295502430 | 70368744177664 | 61586073387009 | -40864551324766 |
| 65536 | 778633068517010 | 53688008705 | 778686756525715 | 562949953421312 | 492634897252353 | -215736803104403 |
| 131072 | 5450807295680000 | 214750199809 | 5451022045879810 | 4503599627370500 | 3940864424148990 | -947422418509318 |
| 262144 | 38157154321158700 | 858997129217 | 38158013318287900 | 36028797018964000 | 31526056388722700 | -2129216299323940 |
| 524288 | 267106093228015000 | 3435981176833 | 267109529209192000 | 288230376151712000 | 252205015113925000 | 21120846942519500 |
| 1048576 | 1869766704464350000 | 13743910027265 | 1869780448374370000 | 2305843009213690000 | 2017626376972010000 | 436062560839321000 |

*Date*: March 22, 2021.

What we can see here is that purely using Strassens over conventional doesn't become cost effective until a very large n value, 524288. However, utilizing a crossover point, where we call Strassens at n and use conventional at n/2, we can see this becomes cost effective much sooner. At n=64, if we call Strassens, Strassens will make 7 calls to Conventional with input n/2 and also incur the cost of 52097 for other arithmetic operations. However this proves to be lower then the cost of simply calling conventional with input n. As a result, this analysis would indicate that the crossover point is n=32. As at n=64, we would want to start off using Strassens.

1.1. **Time trials.** I also wanted to test things out myself and see if the above analysis made sense when doing time trials (average taken over 3 trials per cell). After running time trials with different crossover points, I got the below results. These results indicate that having a crossover point at n=64 is actually slightly better then n=32. I believe the reason for this is that the above analysis only took into account arithmetic operations. However, there were other operations that we did such as creating and setting variables which take time. The analysis above indicates that n=32 is the crossover point but when we look at the chart, we can see it's pretty close where using Strassens at n=64 and switch to conventional at n=32 only slightly beats out using just conventional. Given that, and the time analysis, I ultimately believe that the best crossover point is n=64 but it's very close and could just be machine dependent.

| n | Conventional | Crossover at 16 | Crossover at 32 | Crossover at 64 | Crossover at 128 |
|---|---|---|---|---|---|
| 16 | 0.000018 | 0.000615 | 0.000019 | 0.000029 | 0.000024 |
| 32 | 0.000374 | 0.005099 | 0.000286 | 0.000265 | 0.00028 |
| 64 | 0.001362 | 0.0265097 | 0.001788 | 0.001219 | 0.001512 |
| 128 | 0.009129 | 0.183172 | 0.010128 | 0.008494 | 0.009178 |
| 256 | 0.078513 | 1.261605 | 0.069462 | 0.062417 | 0.071703 |
| 512 | 0.851047 | 9.190607 | 0.48534 | 0.434542 | 0.497781 |

1.2. **Auto-grader unknown issue.** I was experiencing an unknown issue with the autograder. When I set my crossover point at n=2, my code works and the auto-grader reports back correct results. It does report back that my code is running too slow for larger n values. This is understandable because the crossover point should be higher, (32 or 64). However, when I try to use a higher crossover point such as 32 or 64, the auto grader reports back that I have achieved the wrong results for dimensions 4x4, 5x5, 6x6, etc. This is rather odd because I've done multiple tests with different input files and from what I see they are reporting back the correct values and outputting the same output as when I have a crossover point of n=2. I tried this with negative values as well and still am getting the same results regardless of crossover point. In order to address this, I have two functions. One uses a pure version of Strassens that doesn't crossover at n=32. This is what will be used when the original input dimensions are 32 or lower. Anytime the input dimensions are higher, I utilize Strassens crossover which switches to conventional at n=32. This seems to work with the grading server however it is really odd as I could not find any bugs and the Strassens crossover function should be working on all inputs, as it seems to when I run locally.

## 2. Building code/optimizations/observations

2.1. **Passing matrices.** One observation I quickly learned is that C++ does not allow you to create functions that accept matrices as input if I need the function to accept matrices of different sizes. I would have to declare the number of columns or rows to allow for this. To account for this, I started using pointers. Each pointer would point to a list of size n and each of the n elements of the list would be an n sized list. This would achieve the same results as a matrice and I could point to each list with a pointer. This is also good because it uses less space. By passing a pointer to a matrice, I do not need to pass the entire matrice to be stored in memory. This would take up much more space as we got to large n values.

2.2. **Padding for odd sized matrices.** My method for dealing with odd sized matrices is to pad them with an additional column/row of 0's. During my very first implementation, I tried padding so that the original input matrices would be increased to a size of the next power of 2. For example, an input matrice of size 13 would be increased to size 16 which is a power of two. I realized that although this would work, it could nearly double the run time. For example, an input of n=257 would mean a padding of 255 rows/columns! This would take a lot longer to do. As a result, I switched to adding a row/column only when n was odd. As a result, I would add far fewer unnecessary rows/columns. As a result, I removed the below function as it was no longer needed which would find the next highest power of 2.

```
//Take an input and find the closest power of two. Used to determine necessary padding.
int closest_power(int v){
    v--;
    v |= v >> 1;
    v |= v >> 2;
    v |= v >> 4;
    v |= v >> 8;
    v |= v >> 16;
    v++;
    return v;
}
```

2.3. **Counting arithmetic operations.** The instructions recommended that we count each arithmetic operation in order to determine the cost of running strassens given n. One thing I realized is how many times I was doing arithmetic operations again that could instead be saved as variables. The ultimate cost savings here would be small but could become more substantial for larger n. Here is an example where I utilized creating variables instead of redoing operations. If the analysis hadn't called for counting arithmetic operations, I likely would have repeated them. Here's an example. Previously the calculations for m and n were being redone multiple times every time we looped.

```
int m;
int n;
for(int i = 0; i < new_dimmensions; i++){
    //Cost n/2
    m = new_dimmensions+i;
    for(int j = 0; j < new_dimmensions; j++){
        //Cost (n/2)^2
        n = new_dimmensions+j;
        a00[i][j] = a[i][j];
        a01[i][j] = a[i][n];
        a10[i][j] = a[m][j];
        a11[i][j] = a[m][n];
        b00[i][j] = b[i][j];
        b01[i][j] = b[i][n];
        b10[i][j] = b[m][j];
        b11[i][j] = b[m][n];
    }
}
```

## 3. Representing Graphs

In order to do this I created two functions called random and triangles. Random utilized the same random number generator I used from the first programming assignment to decide between 1 and 0. Triangles would create a new 1024x1024 matrice and then assign a value 1 or 0 based on the probability for each cell. I made sure to account for if an edge was already listed from i to j that it would also be marked the same for j to i. One mistake I made at first was assuming that the cell for a vertice to itself should be marked as 1. When I first did this, I was getting very large values that weren't close to the expected values. After I changed this so that each cell (x,x) was marked as 0, the number of triangles was in line with the expected values. I ran three trials for each and have the values listed below along with averages, rounding to the nearest integer .

| Probability | Trial 1 | Trial 2 | Trial 3 | Average of trials | Expected Value |
|---|---|---|---|---|---|
| .01 | 173 | 174 | 191 | 179 | 178 |
| .02 | 1430 | 1363 | 1344 | 1379 | 1427 |
| .03 | 4790 | 4778 | 4940 | 4836 | 4817 |
| .04 | 10848 | 11457 | 11699 | 11334 | 11420 |
| .05 | 22983 | 22754 | 21767 | 22501 | 22304 |