Tamara Katic
TUID: 915749373

Operating Systems
- Lab 1 -

Part I: Memory Effects

- **Compose a program to multiply two NxN matrices in C.**
- **(2 pts) Create a script to generate six programs that will multiply the same matrices with identical data in six different (i,j,k) loop orders. Verify if the six output matrices are identical. Explain your findings.**

All six results from multiplying N*N matrices in six different orders are identical. I will explain the reason for that in the following example when N=3.
Assuming we have the following two matrices:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad B = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 3 & 4 \\ 3 & 3 & 3 \end{bmatrix}$$

And we want to multiply them in different orders.
First, I will take (i, j, k) order and then use (i, k, j) order.
Order (i, j, k):

```
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < N; k++)
            C[i][j] += A[i][k] * B[k][j];
```

$$Step\ 1 : C = \begin{bmatrix} 1*1+2*2+3*3 & - & - \\ - & - & - \\ - & - & - \end{bmatrix}$$

$$Step\ 2 : C = \begin{bmatrix} 1*1+2*2+3*3 & 1*1+2*3+3*3 & - \\ - & - & - \\ - & - & - \end{bmatrix}$$

$$Step\ 3 : C = \begin{bmatrix} 1*1+2*2+3*3 & 1*1+2*3+3*3 & 1*1+2*4+3*3 \\ - & - & - \\ - & - & - \end{bmatrix}$$

$$Step\ 4 : C = \begin{bmatrix} 14 & 16 & 18 \\ 32 & - & - \\ - & - & - \end{bmatrix}$$

$$Step\ 5 : C = \begin{bmatrix} 14 & 16 & 18 \\ 32 & 37 & - \\ - & - & - \end{bmatrix}$$

. . .

$$Step\ 9 : C = \begin{bmatrix} 14 & 16 & 18 \\ 32 & 37 & 42 \\ 50 & 58 & 66 \end{bmatrix}$$

We could see that every iteration we fill the matrix row by row. Now, we will see **Order (i, k, j):**

```
for (i = 0; i < N; i++)
    for (k = 0; k < N; k++)
        for (j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```

$$Step\ 1 : C = \begin{bmatrix} 1*1 & 1*1 & 1*1 \\ - & - & - \\ - & - & - \end{bmatrix}$$

$$Step\ 2 : C = \begin{bmatrix} 1*1+2*2 & 1*1+2*3 & 1*1+2*4 \\ - & - & - \\ - & - & - \end{bmatrix}$$

$$Step\ 3 : C = \begin{bmatrix} 1*1+2*2+3*3 & 1*1+2*3+3*3 & 1*1+2*4+3*3 \\ - & - & - \\ - & - & - \end{bmatrix}$$

$$Step\ 4 : C = \begin{bmatrix} 1*1+2*2+3*3 & 1*1+2*3+3*3 & 1*1+2*4+3*3 \\ 4*1 & 4*1 & 4*1 \\ - & - & - \end{bmatrix}$$

$$Step\ 5 : C = \begin{bmatrix} 1*1+2*2+3*3 & 1*1+2*3+3*3 & 1*1+2*4+3*3 \\ 4*1+5*2 & 4*1+5*3 & 4*1+5*4 \\ - & - & - \end{bmatrix}$$

$$Step\ 6 : C = \begin{bmatrix} 1*1+2*2+3*3 & 1*1+2*3+3*3 & 1*1+2*4+3*3 \\ 4*1+5*2+6*3 & 4*1+5*3+6*3 & 4*1+5*4+6*3 \\ - & - & - \end{bmatrix}$$

. . .

$$Step\ 9 : C = \begin{bmatrix} 14 & 16 & 18 \\ 32 & 37 & 42 \\ 50 & 58 & 66 \end{bmatrix}$$

From the image above we could see that result is the same but there is one difference in the way how is matrix C calculated. The matrix C is filled row by row but it is different from (i, j, k) order as it updates gradually values per row.

The following orders are (j, i, k) and (j, k, i):

Order (j, i, k):

```
for (j = 0; j < N; j ++)
    for (i = 0; i < N; i ++)
        for (k = 0; k < N; k ++)
            C[i][j] += A[i][k] * B[k][j];
```

$$Step\ 1 : C = \begin{bmatrix} 1*1+2*2+3*3 & - & - \\ - & - & - \\ - & - & - \end{bmatrix}$$

$$Step\ 2 : C = \begin{bmatrix} 1*1+2*2+3*3 & - & - \\ 4*1+5*2+6*3 & - & - \\ - & - & - \end{bmatrix}$$

$$Step\ 3 : C = \begin{bmatrix} 1*1+2*2+3*3 & - & - \\ 4*1+5*2+6*3 & - & - \\ 7*1+8*2+9*3 & - & - \end{bmatrix}$$

$$Step\ 4 : C = \begin{bmatrix} 1*1+2*2+3*3 & 1*1+2*3+3*3 & - \\ 4*1+5*2+6*3 & - & - \\ 7*1+8*2+9*3 & - & - \end{bmatrix}$$

$$Step\ 5 : C = \begin{bmatrix} 1*1+2*2+3*3 & 1*1+2*3+3*3 & - \\ 4*1+5*2+6*3 & 4*1+5*3+6*3 & - \\ 7*1+8*2+9*3 & - & - \end{bmatrix}$$

$$Step\ 6 : C = \begin{bmatrix} 1*1+2*2+3*3 & 1*1+2*3+3*3 & - \\ 4*1+5*2+6*3 & 4*1+5*3+6*3 & - \\ 7*1+8*2+9*3 & 7*1+8*3+9*3 & - \end{bmatrix}$$

. . .

$$Step\ 9 : C = \begin{bmatrix} 14 & 16 & 18 \\ 32 & 37 & 42 \\ 50 & 58 & 66 \end{bmatrix}$$

Order (j, k, i):

```
for (j = 0; j < N; j ++)
    for (k = 0; k < N; k ++)
        for (i = 0; i < N; i ++)
            C[i][j] += A[i][k] * B[k][j];
```

$$Step\ 1 : C = \begin{bmatrix} 1*1 & - & - \\ 4*1 & - & - \\ 7*1 & - & - \end{bmatrix}$$

$$Step\ 2 : C = \begin{bmatrix} 1*1+2*2 & - & - \\ 4*1+5*2 & - & - \\ 7*1+8*2 & - & - \end{bmatrix}$$

$$Step\ 3 : C = \begin{bmatrix} 1*1+2*2+3*3 & - & - \\ 4*1+5*2+6*3 & - & - \\ 7*1+8*2+9*3 & - & - \end{bmatrix}$$

$$Step\ 4 : C = \begin{bmatrix} 1*1+2*2+3*3 & 1*1 & - \\ 4*1+5*2+6*3 & 4*1 & - \\ 7*1+8*2+9*3 & 7*1 & - \end{bmatrix}$$

$$Step\ 5 : C = \begin{bmatrix} 1*1+2*2+3*3 & 1*1+2*3 & - \\ 4*1+5*2+6*3 & 4*1+5*3 & - \\ 7*1+8*2+9*3 & 7*1+8*3 & - \end{bmatrix}$$

$$Step\ 6 : C = \begin{bmatrix} 1*1+2*2+3*3 & 1*1+2*3+3*3 & - \\ 4*1+5*2+6*3 & 4*1+5*3+6*3 & - \\ 7*1+8*2+9*3 & 7*1+8*3+9*3 & - \end{bmatrix}$$

. . .

$$Step\ 9 : C = \begin{bmatrix} 14 & 16 & 18 \\ 32 & 37 & 42 \\ 50 & 58 & 66 \end{bmatrix}$$

The last two orders (k, i, j) and (k, j, i).

Order (k, i, j):

```
for (k = 0; k < N; k++)
    for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
            C[i][j] += A[i][k] * B[k][j];
```

$$Step\ 1 : C = \begin{bmatrix} 1*1 & 1*1 & 1*1 \\ - & - & - \\ - & - & - \end{bmatrix}$$

$$Step\ 2 : C = \begin{bmatrix} 1*1 & 1*1 & 1*1 \\ 4*1 & 4*1 & 4*1 \\ - & - & - \end{bmatrix}$$

$$Step\ 3 : C = \begin{bmatrix} 1*1 & 1*1 & 1*1 \\ 4*1 & 4*1 & 4*1 \\ 7*1 & 7*1 & 7*1 \end{bmatrix}$$

$$Step\ 4 : C = \begin{bmatrix} 1*1+2*2 & 1*1+2*3 & 1*1+2*4 \\ 4*1 & 4*1 & 4*1 \\ 7*1 & 7*1 & 7*1 \end{bmatrix}$$

$$Step\ 5 : C = \begin{bmatrix} 1*1+2*2 & 1*1+2*3 & 1*1+2*4 \\ 4*1+5*2 & 4*1+5*3 & 4*1+5*4 \\ 7*1 & 7*1 & 7*1 \end{bmatrix}$$

$$Step\ 6 : C = \begin{bmatrix} 1*1+2*2 & 1*1+2*3 & 1*1+2*4 \\ 4*1+5*2 & 4*1+5*3 & 4*1+5*4 \\ 7*1+8*2 & 7*1+8*3 & 7*1+8*4 \end{bmatrix}$$

. . .

$$Step\ 9 : C = \begin{bmatrix} 14 & 16 & 18 \\ 32 & 37 & 42 \\ 50 & 58 & 66 \end{bmatrix}$$

Order (k, j, i):

```
for (k = 0; k < N; k++)
    for (j = 0; j < N; j++)
```

$$for\ (i\ =\ 0;\ i\ <\ N;\ i++)$$
$$C[i][j]\ +=\ A[i][k]\ *\ B[k][j];$$

$$Step\ 1 : C = \begin{bmatrix} 1*1 & - & - \\ 4*1 & - & - \\ 7*1 & - & - \end{bmatrix}$$

$$Step\ 2 : C = \begin{bmatrix} 1*1 & 1*1 & - \\ 4*1 & 4*1 & - \\ 7*1 & 7*1 & - \end{bmatrix}$$

$$Step\ 3 : C = \begin{bmatrix} 1*1 & 1*1 & 1*1 \\ 4*1 & 4*1 & 4*1 \\ 7*1 & 7*1 & 7*1 \end{bmatrix}$$

$$Step\ 4 : C = \begin{bmatrix} 1*1+2*2 & 1*1 & 1*1 \\ 4*1+5*2 & 4*1 & 4*1 \\ 7*1+8*2 & 7*1 & 7*1 \end{bmatrix}$$

$$Step\ 5 : C = \begin{bmatrix} 1*1+2*2 & 1*1+2*3 & 1*1 \\ 4*1+5*2 & 4*1+5*3 & 4*1 \\ 7*1+8*2 & 7*1+8*3 & 7*1 \end{bmatrix}$$

$$Step\ 6 : C = \begin{bmatrix} 1*1+2*2 & 1*1+2*3 & 1*1+2*4 \\ 4*1+5*2 & 4*1+5*3 & 4*1+5*4 \\ 7*1+8*2 & 7*1+8*3 & 7*1+8*4 \end{bmatrix}$$

$$\cdot \quad \cdot \quad \cdot$$

$$Step\ 9 : C = \begin{bmatrix} 14 & 16 & 18 \\ 32 & 37 & 42 \\ 50 & 58 & 66 \end{bmatrix}$$

❖ (2 pts) Compose a script to harvest the running times of N=1000, 2000, 5000 running six different loop orders.

The below table presents results that I got when N=1000, 2000, 5000. Running times are presented in seconds.

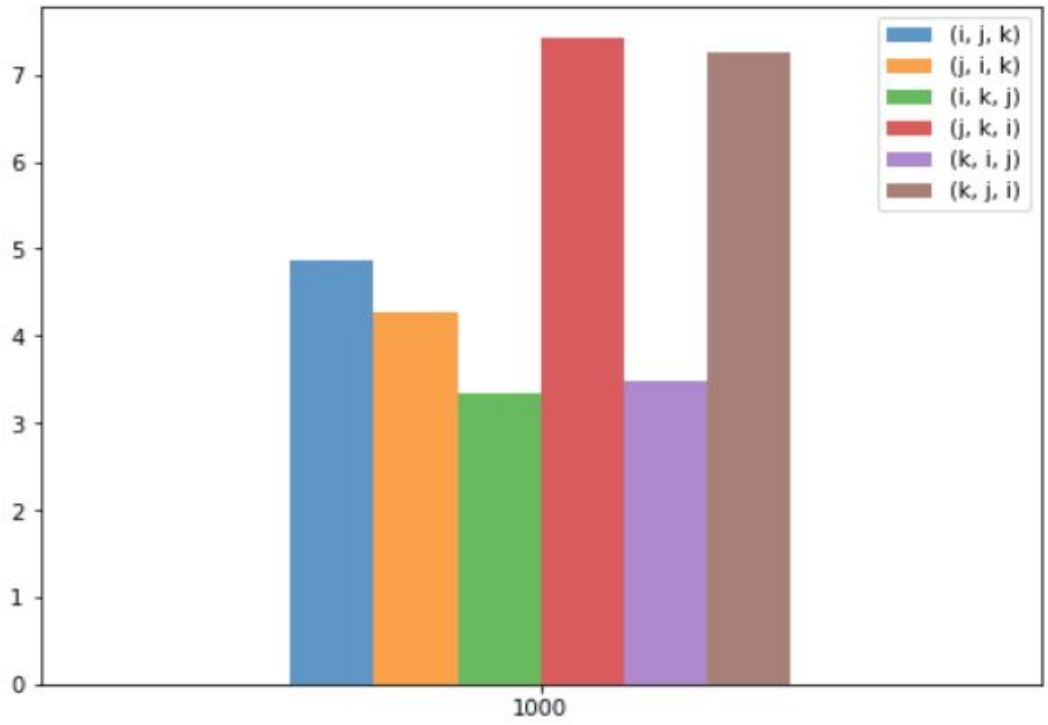
N	(i, j, k)	(j, i, k)	(i, k, j)	(j, k, i)	(k, i, j)	(k, j, i)
1000	4.866718	4.277955	3.332042	7.417254	3.472387	7.245092
2000	63.641975	50.59465	26.38158	76.0185	26.71803	76.41906
5000	1113.91732	902.7091	415.0124	1347.94	415.2246	1356.065

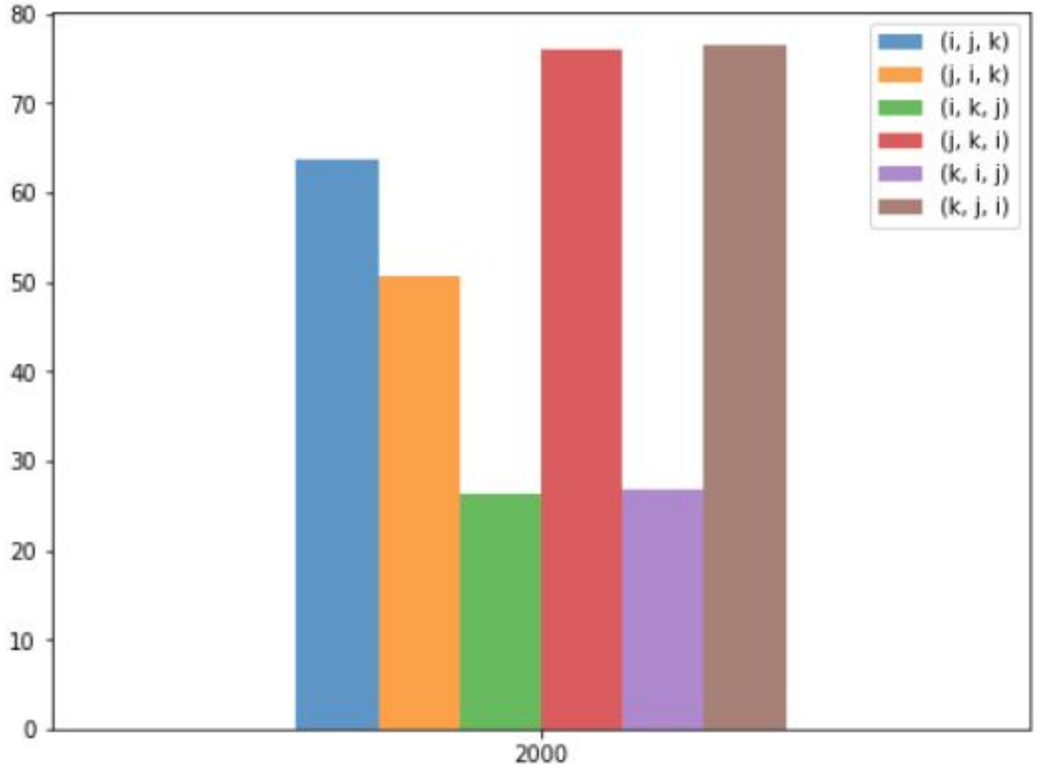
Blue color presents the fastest running time per each N while red color presents the longest running time per each N.

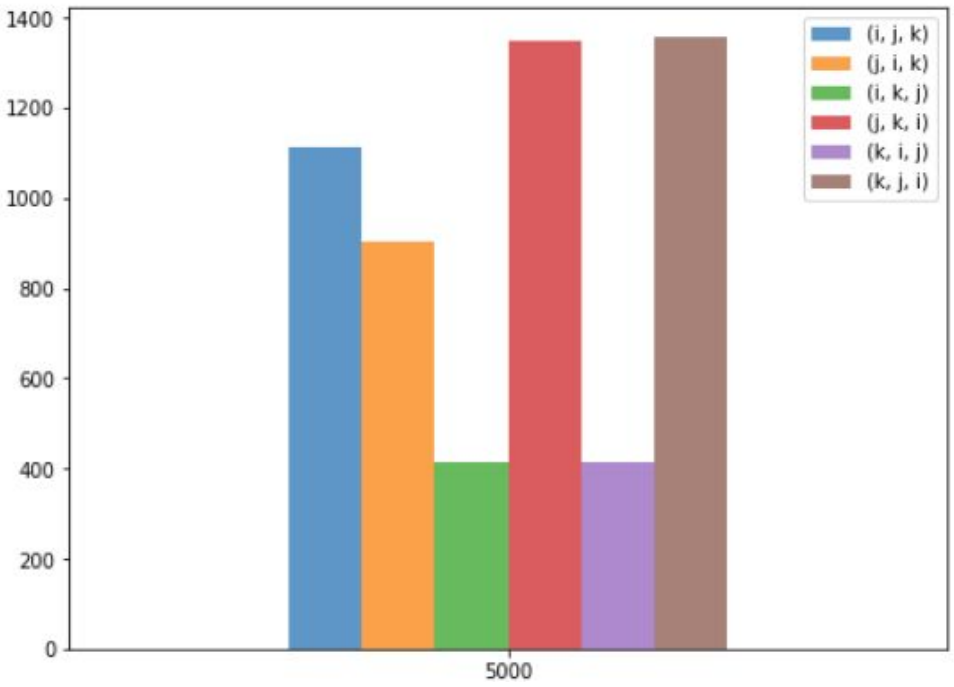
From the results above, we could see that (i, k, j) order is the fastest because of memory locality. Results of this order filled matrix C gradually row by row (filled the first row, then the second row, etc). As memory access first checks the memory caches and then checks the main memory. Arrays here will be stored in memory row by row. Looking at for loop of (i, k, j) order we could see that there are **no cache misses**. The value of matrix C[i][j] will be cached because of the value from previous iteration C[i][j] that are cached and it is ready to be read for the next iteration. A[i][k] result will be probably in the cache as well while B[k][j] won't be in the cache as results are changing faster. Therefore, in each iteration of the inner loop probably there are no cache misses. Because of that (i, k, j) order performs fastest.

❖ (1 pt) Plot your results and explain why the performance differences?

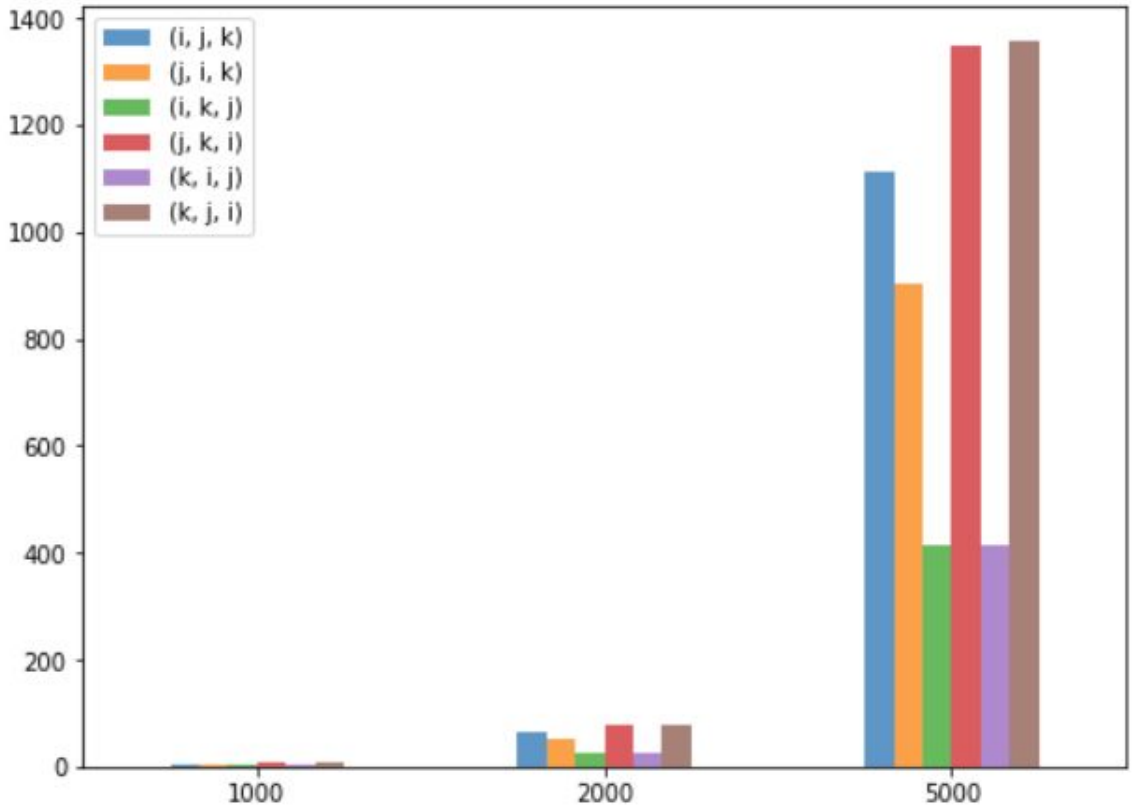
The following images represent plots of results for N=1000, 2000, and 5000:


(i, j, k)
(j, i, k)
(i, k, j)
(j, k, i)
(k, i, j)
(k, j, i)
0
1
2
3
4
5
6
7
1000


(i, j, k)
(j, i, k)
(i, k, j)
(j, k, i)
(k, i, j)
(k, j, i)
0
10
20
30
40
50
60
70
80
2000



From the images above we could see that (i, k, j) order is the fastest while (j, k, i) order is slowest when N=1000 and (k, j, i) is slowest when N=2000 and N=5000.



From the results above we could see when the matrix size is bigger there is huge variability between different orders due to the O(n^3) time complexity. The reason for the different times between different orders is due to the memory hierarchy. While the most inefficient order will read or write the data from the disk every time, the most efficient order will do the same from the cache. To obtain a good performance of memory hierarchy we should optimize a program's memory access patterns.

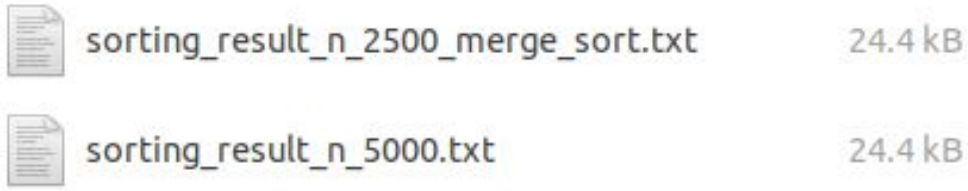
Part II: Program Structure Persistence

- **(1 pt) Compose an Insertion Sort program to sort N (5000) randomly generated integers in C. Insert time measures to report the total elapsed time.**
- **(2 pts) Revise the Insertion Sort program, to sort two N/2 integers and then merge-sort them onto a single output file. Insert time measures to report the total elapsed time.**
- **(1 pt) Verify that the two output files are identical.**
- **(1 pt) Verify the elapsed times of two programs. Explain why the timing differences?**

Results of elapsed time (seconds) are presentend below:

N	BubbleSort	BubbleSort and Merge (Sort arr1 + sort arr2 + Merge arr1+arr2)
5000	0.049672	-
First 2500 (arr1)	0.010951	0.021755
Second 2500 (arr2)	0.010769	

We could see from the above picture that the two output files are identical (they have the same size and same output results).



As we know the complexity of BubbleSort is O(N^2). Sorting half of the array takes O(N^2)/4 time. Merging two sorted arrays will take O(n1+n2), whereas n1 is the size of array 1 and n2 is the size of array2 or O(N) which is the same. Therefore total complexity is 2 * O(N^2)/4 + O(N). We could also see as we divide the array into more pieces the algorithm performs faster.

I also show this on a bigger number of N as it is more relevant.

N	**BubbleSort**	**BubbleSort and Merge (Sort arr1 + sort arr2 + Merge arr1+arr2)**
100000	26.878327	-
First 50000 (arr1)	6.559619	13.104844
Second 50000 (arr2)	6.544720	

We could see from table above that splitting the array on two equal half and sorting them separately using BubbleSort and mergin them we could achieve better time (13.104844 seconds) than if we are doing BubbleSort on original array (26.878327 seconds).
Also, we could see from picture above that their results are identical.