****

**Omar Alshahrani Abdulrahman Alshathry Team**

**201416900 201427580 #22**

project report

KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DEPARTMENT OF MANAGEMENT AND MARKETING

ICS353: Design and Analysis of Algorithms (181)

Table of Contents

1. How to run the code 3
2. Comparative table 5

* Table 1**5**
* Table 2 **5**

1. **Experiments Environment** 5

* Operating System and System type / version**5**
* System Manufacturer**5**
* Processor type and speed**5**
* Memory**5**
* Available OS Memory**5**
* DirectX Version**5**
* DxDiag Version**5**
* User DPI Setting**5**
* System DPI Setting**5**

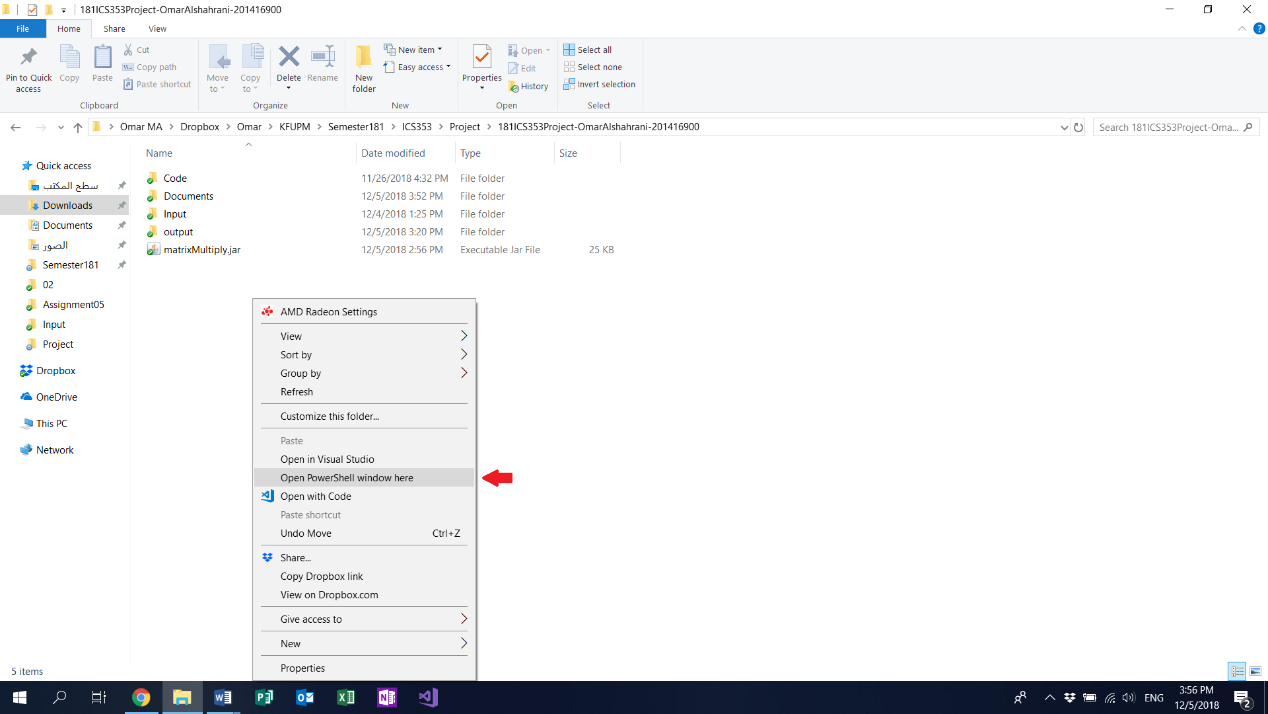
1. Results analysis6
2. Comments and thoughts7

**1) How to run the code**

**First, four folders in a zipped file named (*181ICS353Project-OmarAlshahrani-201416900*) will be provided in addition to a jar file named (*matrixMultiply.jar*). These folders are as follows:**

* (*Code*): contains the source code.
* (*Documents*): contains all documentation-related files.
* (*Input*): contains the input files.
* (*Output*): contains the output files.

To run the program, and while inside the **(*181ICS353Project-OmarAlshahrani-201416900*) file. Press SHIFT + RIGHT-CLICK and open the command line as shown below.**

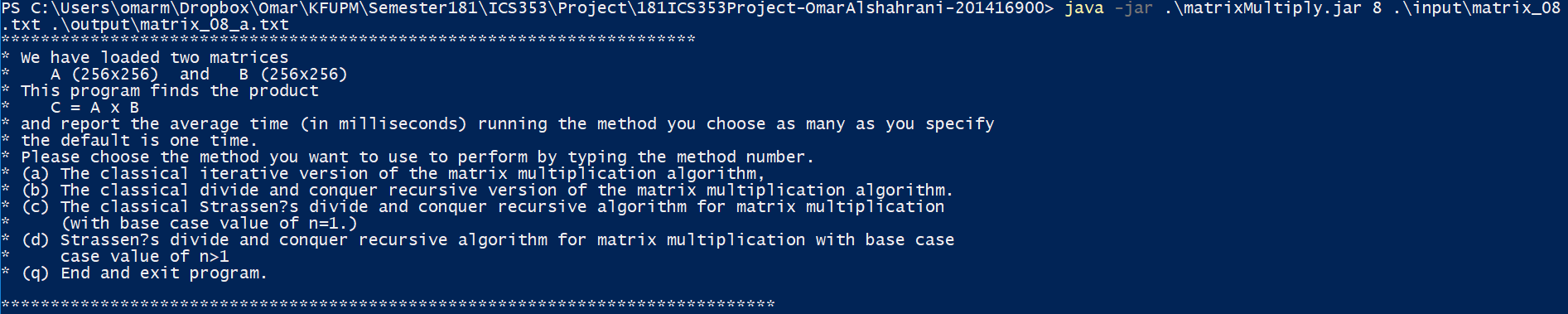
****

[***Note***: you need JDK8 or newer versions installed to run the program.]

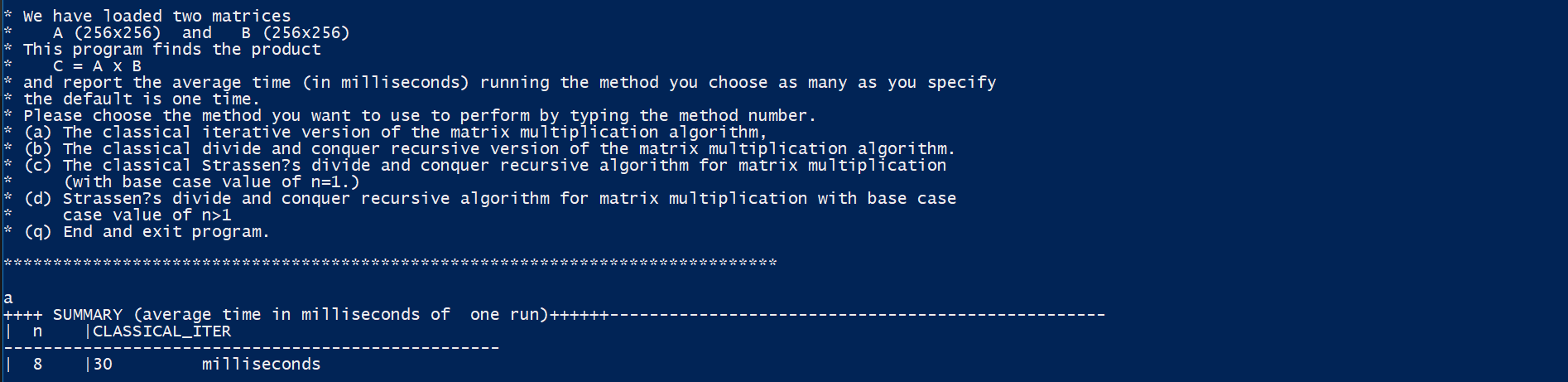
Then, write the command line (java -jar .\matrixMultiply.jar **n** .\input\matrix\_0**n**.txt .\output\matrix\_0**n**\_x.txt) where **n** is the size of the matrix and **x** is the method’s character. For example



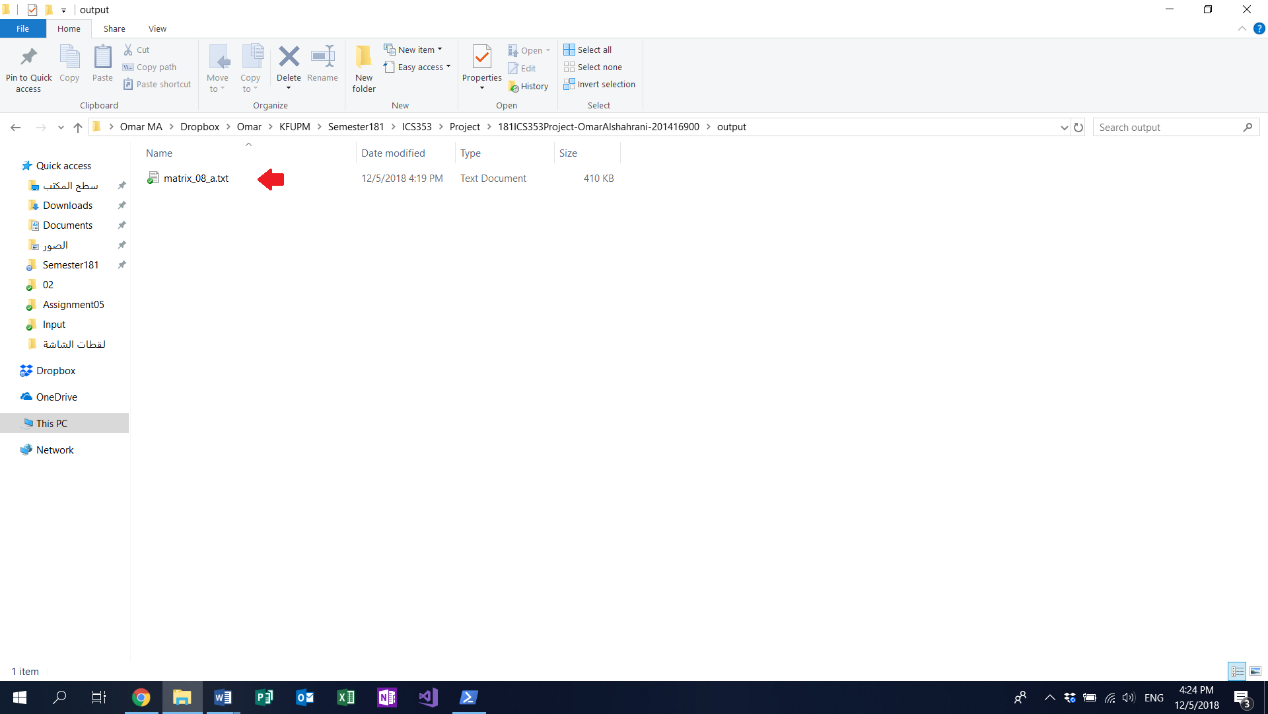
After that, a menu will prompt to the user to choose one method by its character.



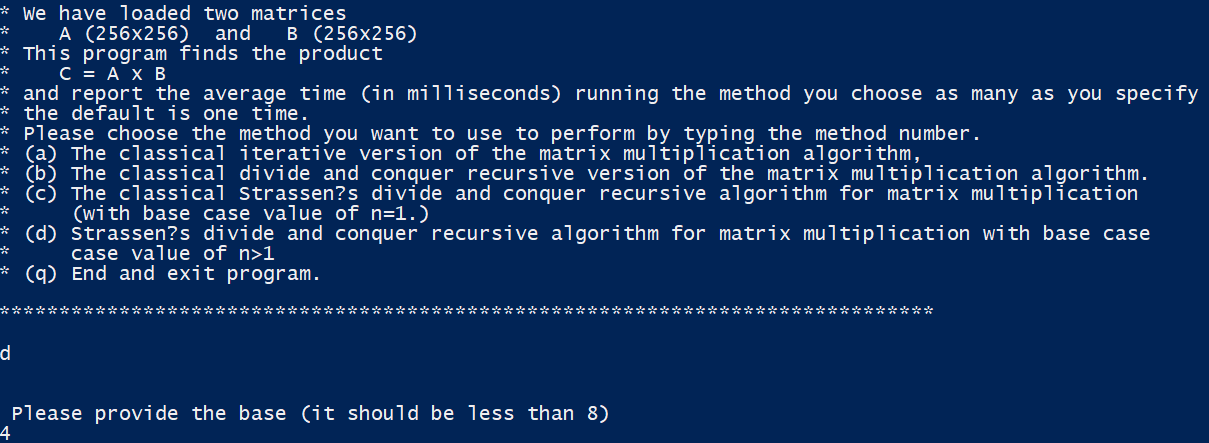
Then, enter “**a**”. For example.



The output file will be found in the (*output*) folder.



[Note: for option “**d**”, you need to enter the base case value.]



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| File size (n)  methods | 4 | 6 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| The classical iterative | 0 | 0 | 12 | 350 | 4113 | 67281 | 647155 | JHSE | JHSE |
| The classical divide and conquer | 5 | 109 | 5831 | 46108 | 369380 | 2938320 | \*\*\* | JHSE | JHSE |
| Strassen’s divide and conquer recursive with base d=1 | 2 | 29 | 922 | 5760 | 39318 | 274515 | 1914759 | JHSE | JHSE |
| Strassen’s divide and conquer recursive with base (d)>1 | 1 (2) | 4 (4) | 44 (4) | 113 (6) | 1042 (8) | 7350 (8) | 52750 (8) | JHSE | JHSE |

**2) Comparative table**

Table 1:

[***Note***:

* The “d” values in this table for the last method are fixed.
* The time units are in milliseconds.
* \*\*\*: the running time took three to four hours before we decided to terminate.
* JHSE: java heap space error.

**]**

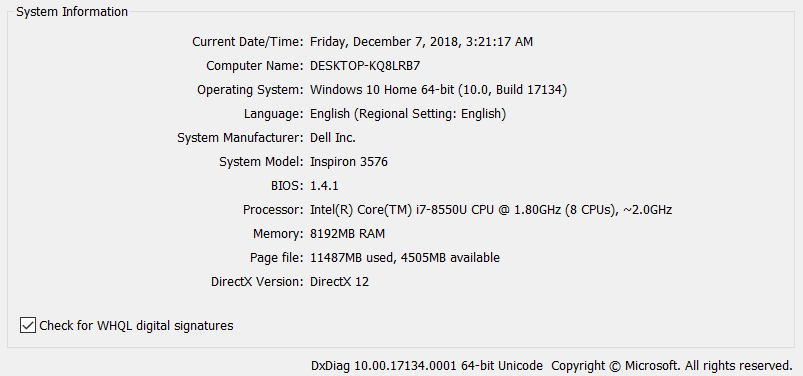
Table 2:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| d  n | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 11 | 11280 | 6873 | 6345 | 4941 | 7350 | 11278 | 38957 |

[***Note***: The matrix size “**n**” in this table is fixed.]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**3) Experiments Environment**

* **Operating System and System type / version**: Windows 10 Home 64-bit (10.0, Build 17134) (17134.rs4\_release.180410-1804).
* **System Manufacturer**: Dell Inc.
* **Processor type and speed** Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz (8 CPUs), ~2.0GHz.
* **Memory: 8192MB RAM.**
* **Available OS Memory: 8058MB RAM.**
* **DirectX Version: DirectX 12.**
* **DxDiag Version: 10.00.17134.0001 64bit Unicode.**
* **User DPI Setting: 120 DPI (125 percent).**
* **System DPI Setting: 120 DPI (125 percent).**

**4) Results analysis**

One of the objectives of the program was to calculate the execution time complexity of each method to be compared. Therefore, it is important to implement a function for obtaining the execution time of the algorithms. Thankfully, the Java has a built-in method (System.nanoTime()) to obtain the current run time of the execution.

As shown in ***Table 1***, we ran each algorithm on different sizes varying from very small like 2^4 and 2^6 up to bigger sizes 2^8, 2^9, . . ., 2^13 and 2^14. We immediately noticed, that the classical iterative (**a**) is way faster than the classic divide and conquer recursive (**b**) and the Strassen’s divide and conquer recursive with base n=1 (**c**) throughout the different matrices sizes. However, compared to the Strassen’s divide and conquer recursive with base n>1 (**d**), (**a**) gave similar yet better results for the first 8 matrices. After that, (**d**) performed better than all of them. From the results, one can only conclude that (**a**) is the superior version for the smaller matrices while (**d**) is more suitable for large matrices.

For ***Table******2***, you can clearly see that there is a mid-point of the base cases, where the (d) algorithm performs at its best. Before and after that point, the (d) algorithm is slower. Hence, we conclude that there should be a balance between the Strassen’s divide and conquer and the classical iterative to achieve optimal results.

Unfortunately, whenever we try to load the matrices 2^13 and 2^14 or above we get the error (Java.lang.OutOfMemory: Java heap space.) which is most likely due to the experiments’ environment used for running the program.

One last thing to note, is that every time the program runs an algorithm, for example 2^11, it takes 2^8 matrix from that same matrix and run the algorithm on it for five iterations before starting with the whole matrix. The reason for this, is to warm up in which we noticed gave more accurate and consistent results throughout all the experiments.

**5) Comments and thoughts**

The complex testing between conventional matrix multiplication and Strassen’s algorithm was successful. However, there are still some changes can be made for further investigation and evaluation. Firstly, from the testing, it can be observed that the Strassen’s algorithm was not efficient when applied to small size matrices. In that, preceding the recursion all the way down to one level would result in significant recursion overhead. Nevertheless, it would still be interesting to know at which point Strassen’s algorithm can continue beat the conventional matrix multiplication. However, this was not carried out in the testing for 2^16 and above due to the huge requirement of memory storage and long and ongoing execution time.