SE2205: Algorithms and Data Structures for Object-Oriented Design
Lab Assignment 1
Assigned: Sept 30, 2019; Due: Oct 21, 2019 @ 10:00 p.m.

If you are working in a group of two, then indicate the associated student IDs and numbers in the `Test.java` file as a comment in the header.

1 Objectives

A dense matrix is a matrix in which almost all matrix entries are non-zero. On the other hand, a sparse matrix contains entries that are mostly zero. In this assignment, you will implement the Strassen's algorithm for multiplying two dense square matrices. With regular matrix multiplication, you can use three nested loops and directly compute the product. The complexity of regular matrix multiplication is $O(n^3)$ where n is the size of the square matrices that are being multiplied. As the size of the matrices grow, the processing power required for computations also increases in a cubic manner. Amongst all arithmetic operations, multiplication is the most intensive and expensive operation. The Strassen's algorithm was proposed by Volker Strassen in the 1960s in an attempt to make dense matrix multiplication more efficient (i.e. entails less multiplication operations). This algorithm has been proven to be more efficient than regular multiplication and has an efficiency of approximately $O(n^{2.8})$. Although many more efficient algorithms for dense matrix multiplication have been proposed at recent times, we will focus on implementing the Strassen's algorithm for this assignment.

2 Strassen's Algorithm

Following are assumptions made about the two matrices (A and B) which are operands of the dense matrix multiplication operation:

- Both are square matrices with size 2^n x 2^n (i.e. $A \in \mathbb{R}^{2^n \text{x} 2^n}$, $B \in \mathbb{R}^{2^n \text{x} 2^n}$)
- Almost all entries in both matrices are non-zero

Since both matrices A and B are square matrices with size 2^n x 2^n, these can be divided into four equal blocks as follows:

$$A = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} B = \begin{bmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \end{bmatrix}$$

Suppose that the size of A is 4 x 4. Then all sub-matrices $A_{0,0}, A_{0,1}, A_{1,0}, A_{1,1}$, are 2 by 2 equally sized blocks as illustrated in the following:

$$A = \begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$$

$$A_{0,0} = \begin{bmatrix} a_{0,0} & a_{0,1} \\ a_{1,0} & a_{1,1} \end{bmatrix}, A_{0,1} = \begin{bmatrix} a_{0,2} & a_{0,3} \\ a_{1,2} & a_{1,3} \end{bmatrix}, A_{1,0} = \begin{bmatrix} a_{2,0} & a_{2,1} \\ a_{3,0} & a_{3,1} \end{bmatrix}, A_{1,1} = \begin{bmatrix} a_{2,2} & a_{2,3} \\ a_{3,2} & a_{3,3} \end{bmatrix}$$

Sub-matrices of B which are $B_{0,0}, B_{0,1}, B_{1,0}, B_{1,1}$, can also be computed in a similar manner. Multiplying matrices A and B results in matrix C.

$$A * B = C$$

$$\begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix} * \begin{bmatrix} B_{0,0} & B_{0,1} \\ B_{1,0} & B_{1,1} \end{bmatrix} = \begin{bmatrix} C_{0,0} & C_{0,1} \\ C_{1,0} & C_{1,1} \end{bmatrix}$$

Following are the set of computations performed when regular matrix multiplication is used to compute C:

$$C_{0,0} = A_{0,0}B_{0,0} + A_{0,1}B_{1,0}$$

$$C_{0,1} = A_{0,0}B_{0,1} + A_{0,1}B_{1,1}$$
$$C_{1,0} = A_{1,0}B_{0,0} + A_{1,1}B_{1,0}$$
$$C_{1,1} = A_{1,0}B_{0,1} + A_{1,1}B_{1,1}$$

As you can observe from the above, the computation of C via regular multiplication has required 8 multiplications for each matrix block. With the Strassen's algorithm, only 7 multiplications will be required. In an implementation of the Strassen's algorithm, it is necessary to first compute 7 matrices M_0 to M_6 as follows:

$$M_0 = (A_{0,0} + A_{1,1})(B_{0,0} + B_{1,1})$$
$$M_1 = (A_{1,0} + A_{1,1})B_{0,0}$$
$$M_2 = A_{0,0}(B_{0,1} - B_{1,1})$$
$$M_3 = A_{1,1}(B_{1,0} - B_{0,0})$$
$$M_4 = (A_{0,0} + A_{0,1})B_{1,1}$$
$$M_5 = (A_{1,0} - A_{0,0})(B_{0,0} + B_{0,1})$$
$$M_6 = (A_{0,1} - A_{1,1})(B_{1,0} + B_{1,1})$$

As you can observe from the above, there are a total of 7 matrix multiplication operations performed to compute the above set of matrices. These matrices can be combined to obtain the resultant matrix C via addition and subtraction operations as follows:

$$C_{0,0} = M_0 + M_3 - M_4 + M_6$$
$$C_{0,1} = M_2 + M_4$$
$$C_{1,0} = M_1 + M_3$$
$$C_{1,1} = M_0 - M_1 + M_2 + M_5$$

It is clear from the above that although fewer multiplication operations are required for the Strassen's algorithm in comparison to the regular multiplication method, more addition and subtraction operations are necessary. However, since multiplication operations are more expensive than addition or subtraction, the Strassen's algorithm results in being more efficient than regular matrix multiplication.

The Strassen's algorithm outlined above can be implemented recursively. Multiplications that are required to compute matrices $M_0 \dots M_6$ are also dense matrix multiplications. In order to compute these matrices, the dense multiplication function can be called recursively. At each recursive call, the size of the matrices that are multiplied is halved. For instance, suppose that the size of A and B is n x n. Matrix M_0 is computed by multiplying matrices that are half the size of the original set of matrices passed into the recursive function (i.e. size is $\frac{n}{2}$ x $\frac{n}{2}$). The original problem is therefore halved and this simpler sub-problem is of the same kind as the original multiplication problem. The base case of this multiplication problem occurs when $n = 1$. At this point, two numbers (not matrices) are multiplied. Hence, the Strassen's algorithm is a perfect candidate for recursive implementation!

3 Implementation of the Strassen's Algorithm

In order to implement the Strassen's algorithm, you are required to expand and implement the following functions:

- `public int[][] denseMatrixMult(int[][] A, int[][] B, int size)`
- `public int[][] sum(int[][] A, int[][] B, int x1, int y1, int x2, int y2, int n)`
- `public int[][] sub(int[][] A, int[][] B, int x1, int y1, int x2, int y2, int n)`
- `public void printMatrix(int n, int[][] A)`

- `public int[][] readMatrix(String filename, int n) throws Exception`

The first function implements the Strassen's algorithm in a recursive manner (i.e. `public int[][] denseMatrixMult(int[][] A, int[][] B, int size)`). All other functions serve as helper functions for the recursive function. The first argument to `denseMatrixMult` is `A` which is a two-dimensional array representing one dense matrix. `B` is a two-dimensional array representing the second dense matrix. This function will return the two-dimensional integer resultant matrix `C` of the product of `A` and `B` via the Strassen's algorithm. `n` is the size of matrices A and B.

In order to compute matrices $M_0 \dots M_6$, you will use the `sum` and `sub` functions. For instance, to compute M_0, it is necessary to compute the sum of sub-matrices $A_{0,0} + A_{1,1}$ and $B_{0,0} + B_{1,1}$. To compute the first term $A_{0,0} + A_{1,1}$, you can call the function `sum(A, A, 0, 0, n/2, n/2, n/2)` where n is the size of A. Similarly to compute the second term $B_{0,0} + B_{1,1}$, you can call the function `sum(B, B, 0, 0, n/2, n/2, n/2)`. Results from these two summations are multiplied via a recursive call to the original function. All remaining matrices $M_1 \dots M_6$ can be computed in a similar manner.

After matrices $M_0 \dots M_6$ are computed, these can be combined to obtain $C_{0,0}, C_{0,1}, C_{1,0}, C_{1,1}$. This requires two nested loops (to add columns and rows of matrices).

4 Materials Provided

You will download content in the folder SE2205A-LabAssignment1.zip which contains two folders (code and expOutput) onto your local workspace. Folder code contains a skeleton of function implementations and declarations. Your task is to expand these functions appropriately in `Assignment1.java` as required for implementation. `Test.java` evokes all the functions you will have implemented and is similar to the file that will be used to test your implementations. Use `Test.java` file to test all your implementations. `matrix1.txt` and `matrix2.txt` are sample files containing data that will be read by your program. Folder `expOutput` contains outputs expected for the supplied data files `matrix1.txt` and `matrix2.txt`. Note that we will **NOT** use the same sample files for grading your assignment. Do **NOT** change the name of these files or functions.

5 Division of Assignment into Parts

Part 1: Initializing and Printing Matrices

In this part, function implementations of `initMatrix` and `printMatrix` in `Assignment1.java` file will be tested. In the `initMatrix` function, a 2D array of size `n` will be initialized. `printMatrix` prints the content of the 2D array represented by the reference `A`. This part will be tested by executing the program with the commands:

- `javac Assignment1.java Test.java` and
- `java Test 1`

The output resulting from this part must match the content of file `expOutput/part1.txt`.

Part 2: Reading Matrix from Data File and Printing to the Console

In this part, function implementation of `readMatrix` in `Assignment1.java` file will be tested. The `readMatrix` function will read the content of a data file called `filename` and store the content of this file in a 2D array of size `n`. This array will then be returned by the function. This part will be tested by executing the program with the commands:

- `javac Assignment1.java Test.java`
- `java Test 2`

The output resulting from this part must match the contents of the file `Part2.txt`.

Part 3: Adding and Subtracting Sub-Matrices

Here you will complete the implementation of two functions: `sum` and `sub`. Suppose that there are two square matrices A and B of the same size. Suppose that these matrices are of size 8 (this can change). The function `sum` will perform a matrix addition of two sub-matrices (one from matrix A and the other from matrix B) of size n and return the reference to the added matrix. The sub-matrices are identified by coordinates (x_a, y_a) and (x_b, y_b) which denote the indices at which a sub-matrix begins in the main matrix. The arguments passed to these functions are `A, B,` x_a, y_a, x_b, y_b, `n` where `A` and `B` are references to the main matrices. This part of the assignment will be tested via the following commands:

- `javac Assignment1.java Test.java`
- `java Test 3 1`
- `javac Assignment1.java Test.java`
- `java Test 3 2`

Outputs from these tests must match contents in `Part3_1.txt` and `Part3_2.txt` files resulting from the provided `matrix1.txt` and `matrix2.txt` files.

Part 4: Recursive Dense Matrix Multiplication

In this part, you will implement the Strassen's algorithm introduced in Section 2 by expanding the function `denseMatrixMult`. To this function, two two-dimensional arrays `matrix1` and `matrix2` representing two dense matrices and `n` which is the size of these square matrices are passed as arguments. This function will return the resultant matrix. This part of the assignment will be tested via the following commands:

- `javac Assignment1.java Test.java`
- `java Test 4`

Outputs from these tests must match the contents of `Part4.txt` which is the result of multiplying matrices obtained from the provided `matrix1.txt` and `matrix2.txt` files.

6 Grading: Final Mark Composition

It is **IMPORTANT** that you follow all instructions provided in this assignment very closely. Otherwise, you will lose a *significant* amount of marks as this assignment is auto-marked and relies heavily on you accurately following the provided instructions. Following is the mark composition for this assignment (total of 40 points):

- Successful compilation of all program files i.e. the following command results in no errors (3 points):
  `javac Assignment1.java Test.java`
- Successful execution of the following commands: (5 points)
  ```
  java Test 1
  java Test 2
  java Test 3 1
  java Test 3 2
  java Test 4
  ```
- Output from Part 1, 2, 3a and 3b exactly matches expected output (2 points each for a total of 8 points)
- Output from Part 4 exactly matches expected output (14 points)
- Code content (10 points)

Sample expected outputs are provided in folder `expOutput`. We will test your program with a set of completely different data files. There is a 25% penalty for each late day up to 3 days. All submissions after that will be assigned a grade of 0/40.

7 Code Submission

- Upload your codes "SE2205A-LabAssignment1.zip" to OWL in Assignment1 section.
- If you work in a group of two then indicate the associated student IDs and numbers in the Test.java as a comment in header.

ENSURE that your work satisfies the following checklist:

- You submit before the deadline
- All files and functions retain the same original names
- Your code compiles without error (if it does not compile then your maximum grade will be 3/40)