Lab 1, Divide-And-Conquer

Due date: Monday, October 3, by the beginning of lab period.

Lab Assignment

Assignment Preparation

Because this lab involves preparing an analytical report, this is a *pair programming lab.*

Most of your follow up labs will ask for similarly structured analytical reports. For the first lab, you can prepare and submit one report per pair. Future labs will be individual and you will have to work on this task by yourself.

The Task

Implement three algorithms:

- Iterative Matrix Multiplication algorithm.
- Divide-and-Conquer Matrix Multiplication algorithm.
- Strassen's Algorithm.

You can assume that all matrix multiplication operations will take two **square** $n \times n$ matrices as input.

In implementing these algorithms you are only allowed to use the following language features:

- `Integer` variables

- `Integer` arrays of any dimensionality[1].
- Addition and subtraction of `integers`.
- Multiplication of `integers`.
- Control structures of your programming language (loops and conditionals) as well as function calls and user-defined functions.

You are **not allowed** to use **an matrix arithmetics** operations[2] or any vector arithmetics operations.

Note: You are implementing the algorithms to test them side-by-side in their form that most resembles our pseudo-code. The restrictions above are designed to achieve just that. You goal will be to time your implementations, not Java's or Python's implementations of these operations.

In addition, implement a **testing harness** that allows you to collect the information about the run time of your implementation. The testing harness, at a minimum, shall consist of the following:

- A procedure for generating a random $n \times n$ matrix with n^2 integer values in it.
- An ability to archive any generated matrices (this is needed to make sure that your algorithms are tested on the same set of matrix multiplication operations).
- An ability to time the execution of each of the matrix multiplication algorithms you implement.
- The actual harness: a procedure that takes as input a set of *experimental parameters*, performs performs the desired run-time experiment (see below), and collects the run-time data.
- Archiving of the results: the ability to save the results of the experiment in a way that avails them for further analysis (hint: CSV files work well).

Running Experiments

With the three implementations of the Matrix Multiplication algorithms, and with the test harness you build to test them, you will stage a run-time experiment designed to understand how the performance of the algorithms changes with the increase in the input size.

The experiment you will be staging has the following design.

[1] You *can* use Python's `NumPy` arrays if implementing in Python, but the use of `NumPy` does not absolve you from the other constraints.

[2] They already implement the algorithms you want to build.

1. Matrix sizes. Select between 10 and 20 different matrix sizes starting with something relatively small (e.g., $n = 10$, and ending with a reasonably large matrix size that still can fit comfortably in RAM (e.g., $n = 1000$). You may later need to revise the list of your matrix sizes to make sure that your experiment (a) can complete, and (b) will produce meaningful results. The list of matrix sizes is the first parameter of your experiment.

2. Repetitions. For each matrix size you will generate a number of pairs of matrices to be multiplied using your algorithm implementations. This process is controlled by the second parameter called Repeats. We want Repeats> 1 because we want to convince ourselves that the runtime of your implementation observed for a specific pair of matrices of size $n \times n$ is consistent with the runtimes observed on all other pairs of matrices of the same size.

3. Measurement. For each matrix size and for each repetition step, generate two matrices of the appropriate size and compute their product using all three of your methods. Time each computation.

For each matrix size, aggregate the results by computing the average running time, and standard deviation of the running time for each of the three algorithm implementations.

4. Reporting. For each algorithm you should obtain a set of average runtimes for each input size considered. Plot these runtime numbers. If possible, fit a curve to them to see if they follow the expected algorithm run-times.

5. Analysis. Observe the results. Answer the following questions.

- What is the observed runtime behavior for each algorith?
- Which algorithm is the fastest in your experiment? Which is the slowest?
- What are the asymptotic trends for each algorithm? Can you observe or predict if Strassen's Algorithm overtakes one or both of the remaining ones at some point?

6. Report. Prepare a properly formatted, and professionally presentable report describing your work in this lab. The report shall contain the precise description of the experiment your team has implemented, it shall contain the experimental results both in tabular and in graphical form, and it shall contain your analysis. **Both the obtained results, and the quality of writing will be graded!**

Note. Your test harness shall support the steps 1-3 described above.

Deliverables and Submission

You have two main deliverables: your code, and the report. Additionally, submit a simply `README` file.

README. Your `README` file shall contain the list of students in the team, as well as information about your implementation: language chosen, and the insturctions for how to run your implementation: both in order to perform a single matrix multiplication operation, and in order to run your test harness.

Code. Submit all your code in a `lab01.zip` or `lab01.tar.gz` archive.

Report. Submit your report as a PDF document named `lab01-report.pdf`

Note. Additional instructions concefning the internals of your implementation (which would make it possible for us to test your work automatically) may be released on Wednesday, September 27.

Use `handin` to submit all your deliverables

```
$ handin dekhtyar lab01 <files>
```

Good Luck!