ECE 3790: Engineering Algorithms **Winter 2021**

Lab 1: An Introduction to Algorithms

February 1 – February 2, 2021

The purpose of this lab is to give you a simple introduction to algorithms. Specifically, you will implement two algorithms in two different languages and analyze their behaviour for different problem instances.

Important: A portion of your marks for this, and every other lab in this course, will be based on evaluation during your specified lab period. **Please be sure to attend your lab section.**

Labs can be performed in groups of two and feel free to interact with your classmates. **Code should be well documented**.

Each group is responsible for turning in a lab report that includes the answers requested at the end of each problem and a title page. The due date is 11:59pm one week after the lab.

Important: Please include the following signed statement with your submission.

I (We), [insert name(s)] attest that the work I am (we are) submitting is my (our) own work and that it has not been copied/plagiarized from online or other sources. Any sourced material used for completing this work has been properly cited. [Signature(s)]

Also Important: Labs done in pairs must be submitted with a short paragraph outlining each partner's contribution.

Introduction

This programming laboratory will provide you an opportunity to brush up on your programming skills. If you haven't yet been introduced to Matlab, you will also get a learn-by-doing introduction.

For this first lab, you will be required to implement solutions in *two* languages: 1) Python, Java, or C/C++ (your choice) and 2) Matlab. It is not the purpose of this course to teach you these languages, however if you have questions/troubles with your code please don't hesitate to ask.

A crash course in Matlab programming can be found on the course webpage. If you are learning Matlab for the first time, it is highly suggested that you read the posted supplemental material before coming to the lab.

Problem 1 – Matrix Multiplication

Functions

Implement a matrix-matrix-product function called `matrixmultiply` that takes as arguments matrices `A` and `B` and returns the product `C = A*B`. Your implementation should directly implement the triply-nested for-loop algorithm presented in Lecture 1. The following notes apply:

- it is not necessary for the matrices to be square
- your function should identify and report erroneous problem instances
- you are free to add any function arguments required for your language of choice (e.g., matrix sizes)
- you can use derived types or classes to represent matrices but will have to clearly explain what you're using and where it comes from (if you're using a third-party library)

Note: Implement your matrix multiplication function in two languages, one of which must be Matlab.

Main Program

Write a main program `Lab_1_Problem_1` that:

- reads the sizes of matrices `A` and `B` from the command line (or other user input), allocates and fills `A` and `B` as random matrices, and evaluates the product `C = A*B`
- evaluates the computational time of the *matrix product* (excluding other operations) and clearly displays the elapsed time
- verifies the matrix product against a built-in/library routine (think carefully about how can you succintly show that the result is correct)
- evaluates the computational time of the *built-in/library* routine and clearly displays the elapsed time

Note: Implement your main program in two languages, one of which must be Matlab.

Validation and Data Collection

Once you have completed writing your function and main program you need to:

- show that your function works for both square and rectangular matrices (you will need to decide what is convincing)
- use both your method and the built-in/library method to collect timing data for various square matrix sizes, for example $n = 100, 200, 400, 800, \ldots$ and plot time versus problem size for the two methods.

Note: You should validate, collect data, and produce plots for **two languages**. The plots can be produced by a common tool (e.g., you could use Matlab to produce plots for both implementations).

Evaluation and Discussion

Answer the following questions: REPORT

~~1.~~ What computer hardware did you use to run the algorithms produced in this lab (CPU type, number of cores, amount of memory)?

~~2~~. How did you verify that your algorithm works? How can you be sure it works for all problem instances?
-test with all random-size & random-value instances
-test with erroneous cases to see errors reported
-test for possible scenarios first, assume works for all

3. How does your algorithm perform versus the built-in/library function? What factors/reasons contribute to any difference in performance?
pre-combined, scripting, vectorization math

4. How does the performance of your algorithm change as a function of problem size? Is the *change* the same or different from the built-in function? Why?

5. *Derive* the order-of-growth of *your* matrix multiplication algorithm. Justify why or why not the results you measured agree with your assessment of the order-of-growth. You can do this on just the square matrix-matrix multiplication for simplicity (i.e. $C^{n\times n} = A^{n\times n}B^{n\times n}$). Is this a tight upper bound? Justify.
use the "collected timing data vs sample size? does it look like n^3 ?

6. Under what conditions would order of growth *not* be useful for estimating the actual run-time scaling?

Note: Evaluation and discussion should occur for **two languages** corresponding to your validation/data collection for those languages. There is no need to repeat discussions that are common to both language implementations, just focus on any differences.

Hand in:

Hand in all code, plots, a table showing your measured times (two languages, two functions each language), and your answers to the discussion questions.

Problem 2 – Merge Sort

Functions

Implement an in-place merge sort function called `mergesort` that takes as arguments an array `A` and returns the sorted array in `A`. The following notes apply:

- your function can be implemented for sorting one specific type of data (it does not need to work for general inputs)
- your function should identify and report erroneous problem instances (erroneous might depend on your previous design decision)
- you are free to add any function arguments required for your language of choice (e.g., array size)

Note: Implement your merge sort function in two languages, one of which must be Matlab.

Main Program

Write a main program `Lab_1_Problem_2` that:

- reads the size of the array/list to be sorted from the command line (or other user input), allocates and fills `A` as a random list and sorts it using your merge sort function
- evaluates the computational time of the *sorting* (excluding other operations) and clearly displays the elapsed time
- verifies the *sorting*; you may consider writing a utility function for this
- evaluates the computational time for sorting the same list using a *built-in/library* routine and clearly displays the elapsed time

Note: Implement your main program in two languages, one of which must be Matlab.

Validation and Data Collection

Once you have completed writing your function and main program you need to:

- show that your function works
- use both your method and the built-in/library method to collect timing data for various array/list sizes, for example $n = 1000, 2000, 4000, 8000, \ldots$ and plot time versus problem size for the two methods.

Note: You should validate, collect data, and produce plots for **two languages**. The plots can be produced by a common tool (e.g., you could use Matlab to produce plots for both implementations).

Evaluation and Discussion

Answer the following questions:

~~1.~~ How did you verify that your algorithm works? How can you be sure it works for all problem instances? try different size of array. test if array not same data type

2. How does your algorithm perform versus the built-in/library function? What factors/reasons contribute to any difference in performance? Just use the default `sort` function from the language you are using.

3. How does the performance of your algorithm change as a function of problem size? Is the *change* the same or different from the built-in function? Why?

4. *Derive* the order-of-growth of *your* merge sort implementation. Justify why or why not the results you measured agree with your assessment of the order-of-growth.

use the "collected timing data vs sample size? Does the graph look like nlogn?

~~5.~~ Relate this result to what the algorithm is doing in plain English. As an example, for bubble sort ($O(n^2)$) we might say, "Each entry in a list of length n needs to be compared to/swapped with $\sim n$ other entries in the list. This means that for n list entries and $\sim n$ checks/swaps per list entry, we see some function of n^2 checks/swaps".

~~6~~. Prove (*with math*) that the base of the logarithm in Big-O analysis doesn't matter. (Hint: What else doesn't matter in Big-O analysis?)

Note: Evaluation and discussion should occur for **two languages** corresponding to your validation/data collection for those languages. There is no need to repeat discussions that are common to both language implementations, just focus on any differences.

Hand in:

Hand in all code, plots, a table showing your measured times (two languages, two functions each language), and your answers to the discussion questions.