University of Manitoba

ECE 3790: Engineering Algorithms **Winter 2021**

Lab 2: Divide-and-Conquer

February 22 - 23, 2021

The purpose of this lab is to give you practical experience programming and analyzing recursive algorithms, specifically divide-and-conquer algorithms.

Important: A portion of your marks for this lab, 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

So far in the course we have seen a handful of divide-and-conquer algorithms: merge sort, the maximum-subarray problem, tower of Hanoi, and recursive matrix multiplication. Each of these leads to a recurrence relation of the form:

$$T(n) = D(n) + aT(n/b) + C(n), \tag{2.1}$$

with a base case

$$T(n_0) = B(n_0),$$

where we have divided the problem of size n into a pieces of size n/b. $D(n)$ is the time (or operation count) to divide the problem, $C(n)$ is the time to combine the subproblems, and n_0 is some base case size that requires a constant number of operations $B(n_0)$. Note that here n_0 is just a constant problem size and should not be confused with the n_0 required for asymptotic notation definitions.

In class we have also learned a few techniques for solving recurrences (substitution method, recursion trees, master method) and as clever ECE 3790 students I'm sure you're all looking for a convenient (i.e., easy) way to check your solutions to those recurrences. Well, why don't we just program a recursive evaluation of the recurrence in equation (1) and use that to check if our answers make sense throughout the course? To make sure it works we can implement a recursive algorithm, say recursive block matrix multiplication, and see how well it predicts performance compared to actual performance.

Notes: This lab requires both programming and "on-paper" work. For this lab, you are only required to implement solutions in one language of your choice (Python, Java, C/C++, Matlab).

Problem 1 – Recurrence Relations

Functions

Implement a function called `evaluaterecurrence` that takes as input a vector of integers `nvec` and returns the total cost of evaluating a divide-and-conquer algorithm by evaluating some recurrence relation for each value of n in `nvec`. The goal is to determine the costs of evaluating a divide-and-conquer algorithms for different input sizes without actually running the algorithm. There are a couple of ways to approach this:

1. You have `evaluaterecurrence` call a hard-coded function that describes the recurrence of interest.

2. You pass `evaluaterecurrence` a function pointer so that it can evaluate any recurrence relation just by being passed the appropriate function.

The most general-purpose approach is to use function pointers/handles. **In this lab we require function pointers/handles for successful implementation, i.e., full marks.**

In case we are confused as to what needs to happen here, we are aiming to write a function that we can call as follows:

```
costs = evaluaterecurrence(myfunction, nvec)
```

where we want to evaluate `myfunction(n)` for each entry in the vector `nvec`. The function `myfunction` will, in general, calculate

```
value = D(n) + a*T(n/b) + C(n)
```

and return `value`. For example the function you write could be `mergesortrecurrence(n)` which, in the recursive case, returns

```
value = 1 + 2*mergesortrecurrence(n/2) + n.
```

Note:

- You will have to write an appropriate base case for each recurrence into your recurrence function.
- It is not necessary for your solution to be exact. For example it is possible to use floors/ceilings to your advantage in order to approximate the answers. Of course you could also be very precise in the way you code things up. You will be asked about any assumptions you made in the discussion.
- You are free to add any function arguments you need if they help.

Main Program

Write a main program `Lab_2_Problem_1` that specifies an array of values of n in `nvec` and uses your function(s) to evaluate and plot the cost trends as a function of n for the following recurrence relations:

1. $T_1(n) = 1 + 2T(n/2) + n$ (Merge Sort)
2. $T_2(n) = 2T(n-1) + 1$ (Tower of Hanoi)

3. $T_3(n) = 5T(n/3) + n$

4. The cost of a recursive algorithm of your choosing that has not yet been covered in class (research, references, and an algorithm explanation required).

Data Collection and Analysis

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

(pass in n in main())

- Call your main program on appropriate values of n to get the general trends of the recurrence relations (use your judgement to determine the ranges of n used so that you see the correct trend).

Paper work (master theorem?)

- Determine a closed-form (tight) asymptotic bound (upper or lower depending perhaps on what is convenient) for each of the 4 recurrence relations. If we have done a derivation in class then feel free to use the result (with reference). Otherwise we need a more-or-less complete proof. **Note**: that your code may be a good way to establish a guess instead of using recursion trees!

- For each of the 4 recurrences, plot the analytic solutions and the output of the main program in one figure, producing 4 total plots. Make sure to scale the curves (using a single multiplying factor) so that they line up for large values of n. Do this dynamically based on your data - do **not** scale by some arbitrary large constant. Make sure that a figure legend shows exactly what constant you are multiplying by and your closed-form function.

Evaluation and Discussion

Answer the following questions:

1. What assumptions, if any, did you make in coding up the solutions to the recurrence relations?

master theorem

2. For each of the four recurrences briefly discuss whether or not your analytic solution and numerical solution agree. If they do not, do your best to explain what is going on.

matlab

3. In order to line up the plots it is necessary to scale the results. Why is this an acceptable thing to do? How did you go about doing this?

4. Have you gained any intuition in analyzing recurrences from this exercise? That is, from the 4 recurrences given, can you summarize how features of the equation affect the change in asymptotic growth?

5. What base cases did you implement for each of the recurrences and why? For one of the recurrences try changing the base case. Does it change the overall trend? Why or why not?

Bonus

Evaluating functions recursively generally requires more overhead than is actually necessary: we have to push the state of the program to the stack every time we make a function call. It turns out that we can usually implement recursive programs without actually making recursive calls using either a stack or a queue. Describe and provide pseudocode for a technique that would replace your recursive function in this problem with an alternative method using a queue or stack (think about which one is appropriate here). If you really want to impress us, then implement it and show it gives the same results as your recursive program.

Hand in:

Hand in your derivations for the recurrence relation solutions, all code, plots, and your answers to the discussion questions.

Problem 2 – Recursive Matrix Multiplication

Functions

Implement a recursive block matrix multiplication function `recursivematrixmultiply` that takes as arguments matrices `A` and `B` as well as a minimum block size and produces the product `C = A*B` recursively, using automatic block partitioning into 4 blocks of roughly equal size. Details can be found in Lecture 9. The following notes apply:

- It is likely beneficial to pass the output C as a reference argument (depending on your language) so that you can update it as you go.
- An efficient solution to this problem will not copy submatrices but use indexes to appropriately offset into the matrices without copying. **For full marks, your solution should use indexing and not copy the submatrices**.
- 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., sizes and block offsets).
- 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)

Main Program

Write a main program `Lab_2_Problem_2` 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 recursively 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 (you can reuse anything you need from Lab 1).

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 your function to collect timing data for various square matrix sizes, for example $n = 100, 200, 400, 800, \ldots$.
- Argue the mathematical expression for the recurrence relation for this function in your own words. i.e. explain what each coefficient and term in the relation $T(n) = D(n) + aT(n/b) + C(n)$ for this algorithm.
- Use the results from Part 1 of this lab to verify that the recurrence relation agrees with the observed times. To do this, produce a plot showing the actual measured times, the times predicted from the recurrence, and the closed-form solution to the recurrence (you do not need to prove it) all appropriately scaled so they line up.

n^3, T(n), measured time

- Produce a plot comparing the run times of the recursive function to the non-recursive implementation from Lab 1. You can just reuse your numbers assuming you are on a computer with the same hardware as Lab 1, otherwise re-collect the Lab 1 results.

Evaluation and Discussion

Answer the following questions:

1. How does your analysis of the algorithm performance compare with the measured results? Discuss.
2. How does the performance of the recursive algorithm compare to the solution you implemented in Lab 1? What features of the algorithm/implementation contribute to the performance being either the same or different (depending on what you observe).
3. Strassen's algorithm improves upon the complexity of matrix multiplication by using a block matrix structure in addition to some clever substitutions. Briefly explain how Strassen's algorithm improves on the complexity. Make sure to provide references for your answer.

Hand in:

Hand in all code, plots, and your answers to the discussion questions.