University of Manitoba

ECE 3790: Engineering Algorithms **Winter 2021**

Lab 3: Dynamic Programming

March 8-9, 2021

The purpose of this lab is to give you practical experience programming and analyzing dynamic programming 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 two weeks 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

In class we have been introduced to dynamic programming as a way of solving problems by subdividing problems into subproblems similar in structure to the original problem. In addition to this standard property, subproblems must 1) overlap and 2) have optimal substructure. The basic idea of dynamic programming is to obtain the optimal solution to a problem by exploiting the fact that it can be obtained from optimal solutions to subproblems *and* that we can save time by keeping track of subproblems we have already solved.

Problem 1 – The Fibonacci Numbers

There are many problems that can benefit from the ideas that go into dynamic programming, even if they are not naturally dynamic programming problems. Consider the Fibonacci numbers, defined by the recurrence:

$$F_n = F_{n-1} + F_{n-2}, \qquad n = 2, 3, \ldots, \qquad F_0 = F_1 = 1$$

As discussed in class, this problem has overlapping subproblems. The concept of memoization can be applied to this problem with dramatic effects on performance.

Functions

A) Implement a function called `recursiveFibonacci` that, given an integer n, applies a divide-and-conquer strategy (without memoization) and returns F_n.

B) Implement a function called `memoizedFibonacci` that, given an integer n, applies a dynamic-programming strategy (top-down with memoization) and returns F_n. You may add whatever arguments you require, or a wrapper function as necessary, to make the memoized version of the function work to your liking.

C) Implement a function called `bottomupFibonacci` that, given an integer n, applies a bottom-up dynamic programming strategy and returns F_n.

Main Program

Write a main program `Lab_3_Problem_1` that specifies an array of values of n and collects the running times for your three functions A, B, and C for computing the Fibonacci numbers as a function of n.

Data Collection and Analysis

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

- Call your main program on appropriate values of n to get the general trends of the computational costs of each method for computing the nth Fibonacci number.
- Determine the best Big-O notation fit for each of the three approaches. Show your work! Remember that operation counting and order-of-growth is always a possibility.
- For each of the three approaches, plot the analytic and measured costs/times and the output of the main program in one figure, producing 3 total plots. Scale the curves appropriately and provide details in the legend.

Evaluation and Discussion

Answer the following questions:

1. How does your divide-and-conquer strategy solve the problem of computing the nth Fibonacci number? Explain briefly, relating your answer to the three steps of divide-and-conquer.
2. How does dynamic programming improve upon the divide-and-conquer solution? What makes this improvement possible? Explain briefly, relating your answer to the key features of dynamic programming.
3. How would you summarize the general performance of the three approaches? Why do they perform the way they do?
4. Which of the three approaches is "best"? Justify your reasoning.
5. Do any of the three approaches have the same complexity? If so, which one would you use? Assuming you are using the faster one, why is it faster if the complexity is the same?
6. What computer hardware are you running your program on, and what is the (approximate) index n of the largest Fibonacci number that you can compute on your system in one hour using each of the three methods? Justify your answer.

Hand in:

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

Problem 2 – Choose Your Own Adventure

Introduction

In class we have seen two interesting problems that can benefit from dynamic programming: Matrix Chain Multiplication (MCM) and the Longest Common Subsequence (LCS). Your task is to implement a solution to either problem.

Functions

A) Implement a function called `recursiveMCM` or `recursiveLCS` that returns the optimum value of the problem you have chosen to solve (this is the cost of the multiplication in MCM or the length of the LCS). We leave the input arguments to your discretion.

B) Implement a function called `memoizedMCM` or `memoizedLCS` that improves the top-down recursive version using memoization. You may add whatever arguments you require, or a wrapper function as necessary, to make the memoized version of the function work to your liking.

C) Augment B to also calculate the solution to the problem (i.e., the parenthesization for MCM or the actual longest subsequence for LCS).

Main Program

Write a main program `Lab_3_Problem_2` that:

- Sets up problems of various sizes and evaluates and stores the computational time of your functions as the input sizes grows.

Validation and Data Collection

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

- Show that your function works. If you don't want to program a brute-force answer you will need to show some examples that are convincing.
- Use your functions to collect timing data for various problem sizes.
- Determine the complexity of your solutions. Show your work!

- Plot the analytic and measured costs/times and the output of the main program in one figure, producing 2 plots (one for each implementation). Make sure to scale the curves appropriately.

Evaluation and Discussion

Answer the following questions:

1. How do the 4 general steps of dynamic programming apply to the case of the problem you chose to implement? That is, what do you specifically do in each of these 4 steps to solve the problem?
2. How did the performance of the algorithm improve using memoization? What technique did you use to determine the total cost of the algorithms? Is there an easy/logical "math-free" way to summarize the performance?
3. For the problem you chose to solve, is a bottom-up implementation possible? If so how would it work? If not, why not?
4. Research time: Find a problem that you find interesting and briefly explain how dynamic programming can be used to efficiently solve the problem focusing on optimal substructure and overlapping subproblems. Make sure to say things in your own words and provide references supporting your research.

Hand in:

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