CSCE 156 – Computer Science II

Lab 12.0 - Recursion

Prior to Lab

1. Review this laboratory handout prior to lab.
2. Review the lecture notes on recursion.

Lab Objectives & Topics

Following the lab, you should be able to:

- Be familiar with recursive methods in the Java programming language
- Be able to evaluate and empirically analyze recursive methods

Peer Programming Pair-Up

To encourage collaboration and a team environment, labs will be structured in a *pair programming* setup. At the start of each lab, you will be randomly paired up with another student (conflicts such as absences will be dealt with by the lab instructor). One of you will be designated the *driver* and the other the *navigator*.

The navigator will be responsible for reading the instructions and telling the driver what to do next. The driver will be in charge of the keyboard and workstation. Both driver and navigator are responsible for suggesting fixes and solutions together. Neither the navigator nor the driver is "in charge." Beyond your immediate pairing, you are encouraged to help and interact and with other pairs in the lab.

Each week you should alternate: if you were a driver last week, be a navigator next, etc. Resolve any issues (you were both drivers last week) within your pair. Ask the lab instructor to resolve issues only when you cannot come to a consensus.

Because of the peer programming setup of labs, it is absolutely essential that you complete any pre-lab activities and familiarize yourself with the handouts prior to coming to lab. Failure to do so will negatively impact your ability to collaborate and work with others which may mean that you will not be able to complete the lab.

Recursion

Recursion is a programming approach common to a "divide and conquer" algorithm strategy where a problem is deconstructed into smaller sub-problems until a "base case" is reached and the problem is solved directly. This lab will get you familiar with recursive algorithms by taking you through several exercises to design and analyze recursive algorithms.

Clone the starter code for this lab from GitHub using the following url: https://github.com/cbourke/CSCE156-Lab12.

Analyzing the Fibonacci Sequence

Recall that the Fibonacci sequence is a recursively defined sequence such that each term is the sum of the sequence?s two previous terms. That is, the sequence

$$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

A recursive method to compute the Fibonacci sequence has been provided for you (see `unl.cse.recursion.Fibonacci`). The `main()` method of this class also provides code to compute the execution time of the recursive method. Run the program for several input instances.

This recursive method is highly inefficient: the method is called many times on the same input. Your task will be to directly observe this by adding code to count the exact number of times the method is called for each input n.

To do this, declare a private integer array and increment entries on each call to the recursive method depending on the input n. You may assume that an array no larger than 50 will be needed.

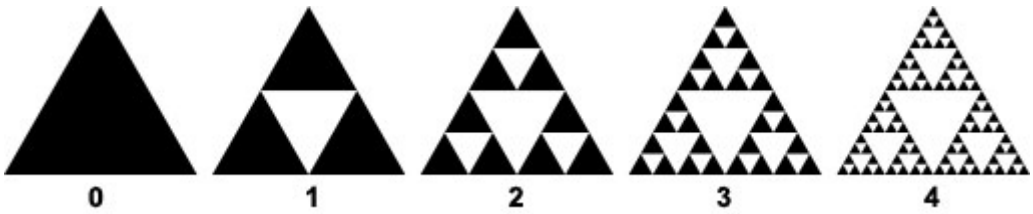

Figure 1: Sierpinski Triangle, 4 iterations

Palindromes

A *palindrome* is a string of characters that is the same string when reversed. Examples of palindromes: kayak, abba, noon. An empty string and any string of length one is a palindrome by definition.

Your task will be to design and implement a recursive algorithm to determine if a given string is a palindrome or not. Implement the method in the class `unl.cse.recursion.Palindrome`. You may find that Java's `String` class has several useful methods such as `charAt(int)` and `substring(int, int)`.

Sierpinski Triangle

A fractal is a geometric object that is self-similar. That is, if you zoom in on the object, it retains the same appearance or structure. One such fractal is the Sierpinski Triangle which is formed by drawing a triangle and removing an internal triangle drawn by connecting the midpoints of the outer triangle's sides. This process is repeated recursively ad infinitum. This process is illustrated for the first four iterations in Figure 1.

A Java applet has been provided to you (`unl.cse.recursion.SierpinskiTriangle`) that recursively draws the Sierpinski Triangle for a specified number of recursive iterations. Since this is an applet, you can run it without having a `main()` method. The depth of the recursion is specified in the `paint()` method. It will be your task to modify this program to count the total number of triangles that a recursion of depth n will ultimately render.

Pell Numbers

Another recursively defined sequence similar to the Fibonacci sequence are the Pell Numbers, defined as follows.

$$P_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ 2P_{n-1} + P_{n-2} & \text{otherwise} \end{cases}$$

A program has been provided (`PellNumbers`) that computes the n-th Pell Number using a recursive function. The method has been defined using Java's `BigInteger` class, a class that supports arbitrary precision integers. If we were to use this implementation to compute the 1000-th Pell Number, the computation would take not just centuries, but billions and billions of years!!

One alternative to such inefficient recursion is to use *memoization*. Memoization typically involves defining and filling a tableau of incremental values whose values are combined to compute subsequent values in the table.

For this exercise, we will instead use a Java `HashMap` to store values. A *map* is a data structure that allows you to define and retrieve key-value pairs. For this exercise, define a (static) `HashMap` that maps `Integer` types to `BigInteger` types (n to P_n) and use it in the `PellNumber` method as follows. If the value P_n is already defined in the map, use it as a return value. Otherwise, compute the value using recursion, but also place the result into the `HashMap` so that it will be available for subsequent recursive calls.

Answer the questions in your worksheet and demonstrate your working programs to a lab instructor.

Submission

We have included a test suite of unit tests written in JUnit (https://junit.org/junit5/) a popular unit testing framework for Java. Even though the test driver (in the `src/test` source folder) has no main method, you can still run it in Eclipse and get a report on how many of the tests passed, failed or resulted in an unexpected exception. Be sure all of the unit tests pass before submitting your source files through webhandin. You can rerun this test suite in the webgrader to ensure everything works.

Advanced Activity (Optional)

Bonus Activity 1: The Fibonacci method in Case Study 1 returns a primitive Java `int` value which has a maximum possible value of 2,147,483,647. Find the maximum value n such that this method returns an accurate value (the value n such that calling `Fibonacci(n+1)` results in overflow). Hint: you probably cannot use the recursive version provided (why?). Write an alternative method to compute the Fibonacci sequence that returns a Java `BigInteger` (an arbitrary-precision number class). Also

use memoization to eliminate repeated recursive calls.