CSCE 156 Lab 12: Recursion

0. Lab Objectives& Topics

Upon completion of this 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

1. Problem Statement

Recursion is a programming approach common to a "divide and conquer" algorithm strategy where a problem is deconstructed into smaller instances 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.

Case Study 1: 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, …

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*.

Hint: 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.

Case Study 2: 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`.

Hint: You may find that Java's String class has several useful methods such as `charAt(int)` and `substring(int, int)`.

Case Study 3: 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 the figure below.

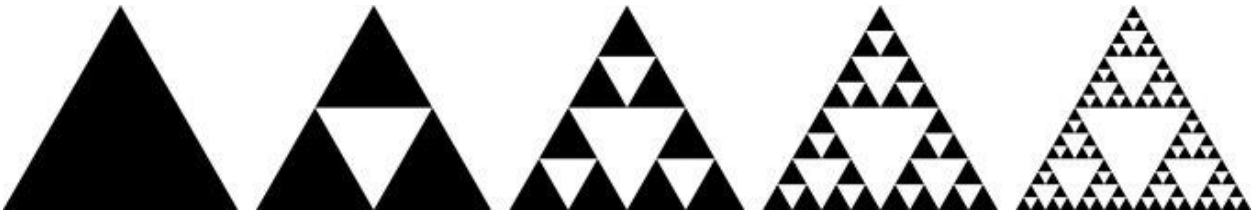

Figure 1Sierpinski Triangle, 4 iterations

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.

Case Study 4: 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: if the value 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 `Map` so that it will be available for subsequent recursive calls.

Demonstrate your working program by computing the 1000-th Pell Number.

2. Instructions

1. Import the project “Lab12-Recursion” from the GIT repository.
   https://github.com/cdr23859/cse-Lab12.git
2. If required, follow instructions from “Using Eclipse and Git for labs” document attached in Canvas.
3. Modify the code for each of the three case studies and complete the worksheet.
4. Get a lab instructor to sign-off on your lab worksheet.