Lab 7 – Recursion

Computer Science 2334

Due by: Friday, 14 April 2017, 11:59 pm

This lab is an individual exercise. Students must complete this assignment on their own.

Objectives:

1. To understand the concept of recursion and how it can be used to solve complex problems in a straightforward manner.
2. To understand what a base case is and be able to develop one.
3. To demonstrate this knowledge by completing a series of exercises.

Instructions:

This lab exercise requires a laptop with an Internet connection. Once you have completed the exercises in this document, you will submit it for grading.

Make sure you read this lab description and look at all of the source code posted on the class website for this lab exercise before you begin working.

Assignment:

Many problems in computer science require relatively complex code that is difficult to write and even more difficult to debug. However, we can solve many of these problems in elegant and simple fashion by using the concept of recursion. Direct recursion occurs when a method calls itself to accomplish some task, instead of calling another method to do that work or doing the work iteratively. Methods can use recursion indirectly by calling each other in a recursive loop but for now we will just concern ourselves with direct recursive methods. In this lab, our recursion will carry out a binary search. For bonus points, you may also do a recursive summation of a sub-array.

As you have learned in the lectures, binary search is a search algorithm that can search a sorted array for a specific value. It first looks at the value in the middle of an array and, if that is the value it seeks, it returns it (or a reference to it) and exits. If not, the binary search algorithm calls itself recursively, passing approximately half of the original array (or references to the end points of half of the original array) to itself in the call. If the item being searched for is greater than the evaluated midpoint, then (references to) the half of the array containing larger values is passed. If the item is less than the midpoint, (references to) the half containing smaller values is passed. This continues until one of two things happens: either a) the value is found, or b) the search gets to a point where the array (portion) to be evaluated is empty. Today, we will complete some code that will implement a binary search (and, for bonus points, a recursive summation).

1. The last two pages of these instructions have small sample arrays drawn repeatedly down the page. Each is meant to simulate the steps of a recursive binary search. For each array, label the following:
    a. The midpoint of the array.
    b. The location of the smallest value in the array.
    c. The location of the largest value in the array.

Done

For the array marked “Call 1,” the locations you label will be the initial locations of the midpoint and smallest and largest elements of the array when the binary search is first called. For the array marked “Call 2,” these locations will be the *new* locations for the midpoint, smallest, and largest elements after the binary search has recursively called itself. For “Call 3,” these will be the locations on the next recursive call, and so forth. Note that at least some of these values will change each time the method is called recursively. This is a dynamic

process; you are meant to be illustrating the way these values change during execution of the method. ***Do not just repeat the same labels over and over down the page!*** Note that it is okay if you stop before all the array copies shown on the page are used. On the call/array at which you stop, your labels should clearly show either a match to the value for which you were searching or the existence of another stopping condition (aka a base case).

Note: The arrays on the last two pages are *different!* One has 11 elements and the other has 10.

1

Answer the following question on this form: How does the base case change depending on how many elements we have in the array? In other words, how do we set up a stopping condition that will handle both even and odd numbers of elements?

> There is no need for the base case to change. As long as the code properly handles the endpoints of the array, even and odd numbers of elements will be handled the same way. This is because the values of the indices of the array are integer values, and as such the calculations done to find the midpoint should be integer addition/division, (e.x. `mid = (leftIndex + rightIndex)/2`). Integer division always rounds down, so in cases with even numbers of elements this rounding will happen, thus ensuring that both even and odd numbers of elements are handled.

2. Download the `Lab7-eclipse.zip` project archive from the class website. Import the project into your Eclipse workspace using the slides from Lab 2. You will submit the modified project archive when you are finished.

Done

3. Review the source code. Note the comments provided in the source code that give hints as to what needs to be done in the program.

Done

4. Complete the **IntegerSearcher** class by doing the following:
   a. Add code that calculates the midpoint.
   b. Add expressions to the `if` statements that will terminate the method and call it recursively.
   c. Add parameters to the two recursive calls of the `recursiveBinarySearch` method.

Done

5. Test the program yourself to make sure it works correctly. Do this by running the program and inputing an integer you would like to find in the array. As written, this program should be able to find the positive even integers between 0 and 1000.

Done

6. Ensure that there are no warnings generated for your code. **Do not suppress warnings.** Fix your code so that warnings are not necessary.

Done

7. BONUS EXERCISE:
   Complete the **Adder** class by doing the following:
   a. Add code that determines if the sub-array array[left,...,right] is empty.
   b. Add parameters to the recursive call of the `recursiveSum` method.

Done

8. Submit a completed electronic copy of this handout as part of the **project archive** following the steps given in the **Submission Instructions** by **Friday, 14 April 2017, at 11:59 pm** through D2L (`http://learn.ou.edu`).

Done

Carry out a binary search on the following array for the value 12.

Call 1

small mid large

value	0	1	4	7	9	10	12	13	17	21	22
index	0	1	2	3	4	5	6	7	8	9	10

Call 2

small mid large

value	0	1	4	7	9	10	12	13	17	21	22
index	0	1	2	3	4	5	6	7	8	9	10

Call 3 - match

small mid large

value	0	1	4	7	9	10	12	13	17	21	22
index	0	1	2	3	4	5	6	7	8	9	10

Call 4

value	0	1	4	7	9	10	12	13	17	21	22
index	0	1	2	3	4	5	6	7	8	9	10

Call 5

value	0	1	4	7	9	10	12	13	17	21	22
index	0	1	2	3	4	5	6	7	8	9	10

Call 6

value	0	1	4	7	9	10	12	13	17	21	22
index	0	1	2	3	4	5	6	7	8	9	10

Call 7

value	0	1	4	7	9	10	12	13	17	21	22
index	0	1	2	3	4	5	6	7	8	9	10

Call 8

value	0	1	4	7	9	10	12	13	17	21	22
index	0	1	2	3	4	5	6	7	8	9	10

***NOTE* This is different from the previous page. This array contains 10 elements, rather than 11.**

Carry out a binary search on the following array for the value 12.

Call 1

	small				mid					large
value	0	1	4	7	9	10	12	13	17	21
index	0	1	2	3	4	5	6	7	8	9

Call 2 - match

					small		mid			large
value	0	1	4	7	9	10	12	13	17	21
index	0	1	2	3	4	5	6	7	8	9

Call 3

value	0	1	4	7	9	10	12	13	17	21
index	0	1	2	3	4	5	6	7	8	9

Call 4

value	0	1	4	7	9	10	12	13	17	21
index	0	1	2	3	4	5	6	7	8	9

Call 5

value	0	1	4	7	9	10	12	13	17	21
index	0	1	2	3	4	5	6	7	8	9

Call 6

value	0	1	4	7	9	10	12	13	17	21
index	0	1	2	3	4	5	6	7	8	9

Call 7

value	0	1	4	7	9	10	12	13	17	21
index	0	1	2	3	4	5	6	7	8	9

Call 8

value	0	1	4	7	9	10	12	13	17	21
index	0	1	2	3	4	5	6	7	8	9