Lab Exercise #7
Recursion
Computer Science 2334

Group #: _____ Section #: _____

Members: _____

_____

_____

_____

_____

Learning Objectives:

- Understand the concept of recursion, and how it can be used to solve complex problems in a straightforward manner.
- Define what a base case is, and be able to develop one.

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. Recursion occurs when a function returns a call to itself, instead of returning a value. Some functions can call each other recursively, but for now, we will just concern ourselves with simple recursive methods.

Your textbook gives several examples of recursive solutions to common Computer Science problems, and you can find even more online. In fact, there is a strong connection between recursive code and iterative code, and most problems can be translated from one form to the other. The power of recursion, however, is that it simplifies code by breaking a large, complex problem into smaller and smaller pieces, which eventually can be solved quite easily.

Specifically, the problem is made smaller each time the method calls itself. This is one of the requirements of a recursive function. Each recursive function must:

1. Get smaller each time it calls itself, and

2. Terminate at some point.

For instance, when calculating a factorial by recursion, we start with the original problem, the factorial we would like to calculate. Let's call this number n. We can rewrite this problem as a combination of two (or more) smaller problems; n(factorial (n-1)). We then calculate the factorial of n-1 by calling the function itself, and passing n-1 as the argument. We hold onto the value of n as we do this. What we end up with is a series of function calls that finally reaches n-1 = 1, at which point we obviously don't want to go on any further. We then pass our results back up to each function call, combining the results until we reach the initial function call, and we have our answer! If this all sounds a little abstract, here is a walk-through with an initial value of n:

- Compute 5! This is 5(factorial 4). We call the factorial function, and pass the value 4

- factorial 4 = 4(factorial 3)

- factorial 3 = 3(factorial 2)

- factorial 2 = 2 (factorial 1)

- factorial 1 = 1. We stop the recursive calls here. This is the base case.

- Working back up, factorial 2 turns into 2(1) = 2. Factorial 3 = 3(2) = 6. We keep going, until we reach the initial call of factorial 5, which now equals 5(24) = 120.

This example meets both of our criteria—it gets smaller each time it calls itself (the number to calculate gets smaller), and it eventually terminates, in this case when n = 1. Now let's turn our attention to another application, binary search.

Binary search is a search algorithm that can search a sorted list for a specific value. It looks at the value in the middle of an array, and if that is the value it seeks, it returns it and exits. If not, the binary search algorithm calls itself recursively, passing half of the original list to the call. If the item being searched for is greater than the evaluated midpoint, then the larger half of the array is passed. If the item is less than the midpoint, the lesser half is passed. This continues until one of two things happens: either a) the value is found, or b) we get to a point where the array we evaluate has "shrunk away".

Today, we will complete some code that will implement a binary search. Your instructor will review the operation of the binary search method graphically, showing the array after each recursive call, and how the call itself is made. Then you should complete this lab handout. Have fun!

Instructions:

This lab exercise requires a laptop with an Internet connection. Once you have completed the exercises in this document, your group will submit it for grading. All group members should legibly write their names at the top of this lab handout.

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

1. One of the sheets in your lab packet has a small sample array drawn repeatedly down the front and back of the page. It is meant to simulate the steps of a recursive binary search method. For each line on which the array is drawn, for both sides of the page, you need to label the following values:

    a. The midpoint
    b. The lowest value in the array
    c. The largest value in the array

    Please keep in mind that these values (most likely) will change each time you re-draw the picture. This is a dynamic process; you are meant to be illustrating the way these values change during execution of the method. Do not just label the same ones over and over.

    Also, your labels should clearly show either a match, or the existence of a stopping condition, aka base case. Question (not to be answered in writing): Does the base case change slightly 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?

2. Read through the source code of the BinarySearch class posted on the website. Note the comments provided in the source code that give hints as to what needs to be done in the program.

3. Complete the source code 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

3. Test the program to make sure it correctly responds to user input. It should find all even numbers between 1 and 1000.

4. Submit the project archive following the steps given in the Submission Instructions by 2:30 pm, Friday, November 7.

5. Turn in this lab handout to your lab instructor or one of the other instructors by 2:30 pm, Friday, November 7.

value	0	2	4	6	8	10	12	14	16	18	20	22
index	0	1	2	3	4	5	6	7	8	9	10	11

value	0	2	4	6	8	10	12	14	16	18	20	22
index	0	1	2	3	4	5	6	7	8	9	10	11

value	0	2	4	6	8	10	12	14	16	18	20	22
index	0	1	2	3	4	5	6	7	8	9	10	11

value	0	2	4	6	8	10	12	14	16	18	20	22
index	0	1	2	3	4	5	6	7	8	9	10	11

value	0	2	4	6	8	10	12	14	16	18	20	22
index	0	1	2	3	4	5	6	7	8	9	10	11

value	0	2	4	6	8	10	12	14	16	18	20	22
index	0	1	2	3	4	5	6	7	8	9	10	11

value	0	2	4	6	8	10	12	14	16	18	20	22
index	0	1	2	3	4	5	6	7	8	9	10	11

value	0	2	4	6	8	10	12	14	16	18	20
index	0	1	2	3	4	5	6	7	8	9	10

value	0	2	4	6	8	10	12	14	16	18	20
index	0	1	2	3	4	5	6	7	8	9	10

value	0	2	4	6	8	10	12	14	16	18	20
index	0	1	2	3	4	5	6	7	8	9	10

value	0	2	4	6	8	10	12	14	16	18	20
index	0	1	2	3	4	5	6	7	8	9	10

value	0	2	4	6	8	10	12	14	16	18	20
index	0	1	2	3	4	5	6	7	8	9	10

value	0	2	4	6	8	10	12	14	16	18	20
index	0	1	2	3	4	5	6	7	8	9	10

value	0	2	4	6	8	10	12	14	16	18	20
index	0	1	2	3	4	5	6	7	8	9	10