Lab 8 Recursion

Goal

In this lab you will experiment writing recursive methods for a variety of applications.

Resources

- Chapter 7: Recursion

Java Files

- *Lab8.java*

Introduction

Recursion—the idea of a method invoking itself—can be challenging to master the first time we are exposed to it. Therefore, we're going to practice by tracing a couple recursive methods, and then writing a few of our own.

The main challenge in tracing a recursive method involves keeping track of the many activation records for different recursive calls to the method that build up on the runtime stack. As an example, here is a simple recursive method:

```
public static int sumOf(int n)
{
        int sum;
        if (n == 1)
                sum = 1; // base case
        else
                sum = sumOf(n - 1) + n; // recursive call
        return sum;
} // end sumOf
```

And here is a diagram of its computation as the JVM executes the call sum(4), which calculates and returns the sum 1+2+3+4. Here we indicate the depth of method calls on the stack by moving down a line every time sum is invoked, and moving up a line every time a value is returned.

```
sumOf(4)
        sum =  sumOf(3) + 4                                    return 10  (answer)
                sum = sumOf(2) + 3                      return 6
                        sum =  sumOf(1) + 2     return 3
                                sum=1     return 1
```

Another way to represent this is by showing the "activation records" as they build up on the runtime stack:

```
sum(1)
  return 1              (1)
sum(2)
  return sum(1) + 2  (3)
sum(3)
  return sum(2) + 3  (6)
sum(4)
  return sum(3) + 4  (10)
```

Here we start at the bottom of the stack with sum(4). The activation records "pile up" on the stack until values are returned to activation records located deeper in the stack. Returned values are indicated in parenthesis.

Pre-Lab Visualization

To reinforce this approach we will trace a different recursive method, the method f:

```
public int f(int n)
{
        int result = 0;
        if (n < 1)
                result = 1;
        else
                result = f(n / 2) + 1;
        return result;
} // end if
```

Notice that for each call in the general case, this method will invoke itself on a number half the size of the argument. For example, f(16) will require evaluating f(8), and f(8) will require evaluating f(4). How long will it take to get to the base case? Continue adding to the stack trace below as illustrated in the first two steps. When an activation frame returns a value, carry it down to the frame that requested it, then draw a line thru the frame that has been finished (indicating it has been popped from the stack).

(your first try)

```
f(8):
  return f(4) + 1;
f(16):
  return f(8) + 1;
```

(in case you mess up, we'll do it together here)

```
f(8):
  return f(4) + 1;
f(16):
  return f(8) + 1;
```

Directed Lab Work

1) Complete the method above main for `validInput`, which asks the user for integer input that is between 1 and 10, inclusive and reads a value using a Scanner object. If the input is out of range, the method should return a recursively invoked version of itself (asking for input all over again), otherwise the input value should be returned. NOTE: we are using if statements in this method. **NO LOOPS.** (Recursion itelf is a form of looping)

Algorithm for `validInput`
- declare a Scanner object set to keyboard
- ask for input between low and high, inclusive
- read an integer `value` from the Scanner
- either return `value` or return a recursively invoked `validInput` to read a new value

Test your method by activating the code in problem 1

2) Consider the method `displayRowOfCharacters` that displays any given character the specified number of times on one line. For example, the call `displayRowOfCharacters('*', 5);` produces the line *****

This method **is only allowed to print one copy of the char value at a time**, and uses recursion to repeatedly call itself (on a smaller size parameter) until the line is complete. Use print, not println to print the char value. Can you make it print a new line when the base case is invoked?

Implement this method in Java by using recursion and test in problem 2

3. Write a recursive method that writes a given array **backward** similar to displayArray in the demo file. Consider the last element of the array first. In other words, your method in the GENERAL case should (MAKE SURE YOU add a base case to the following!!)

a) print the last item in the array
b) then do a recursive call (invoke itself on the remaining items from 0 to last-1)

4. Another way to print an array backwards: repeat Exercise 3, but instead consider the first element of the array first. If we must only print the beginning of the array, we will have to hold off on printing it until the rest of the array has been printed first (backwards of course!). In other words, the GENERAL case should (MAKE SURE YOU add a base case to the following!!)

a) do a recursive call on rest of items (invoke itself on items first+1 thru the last)
b) when done with a, print the first item (this will print the first item last)

5. A palindrome is a string that reads the same forward and backward. For example *deed* and *level* are palindromes. Write an algorithm in pseudocode that tests whether a string is a palindrome. Implement your algorithm as a static method in Java. Remember to use .charAt to examine a character in a string. The basic idea is for every String, check the first and last items in the String. If they are different return false, otherwise, if they are the same, return the result of checking whether the the substring formed by removing the first and last letters is a palindrome.

Pseudocode

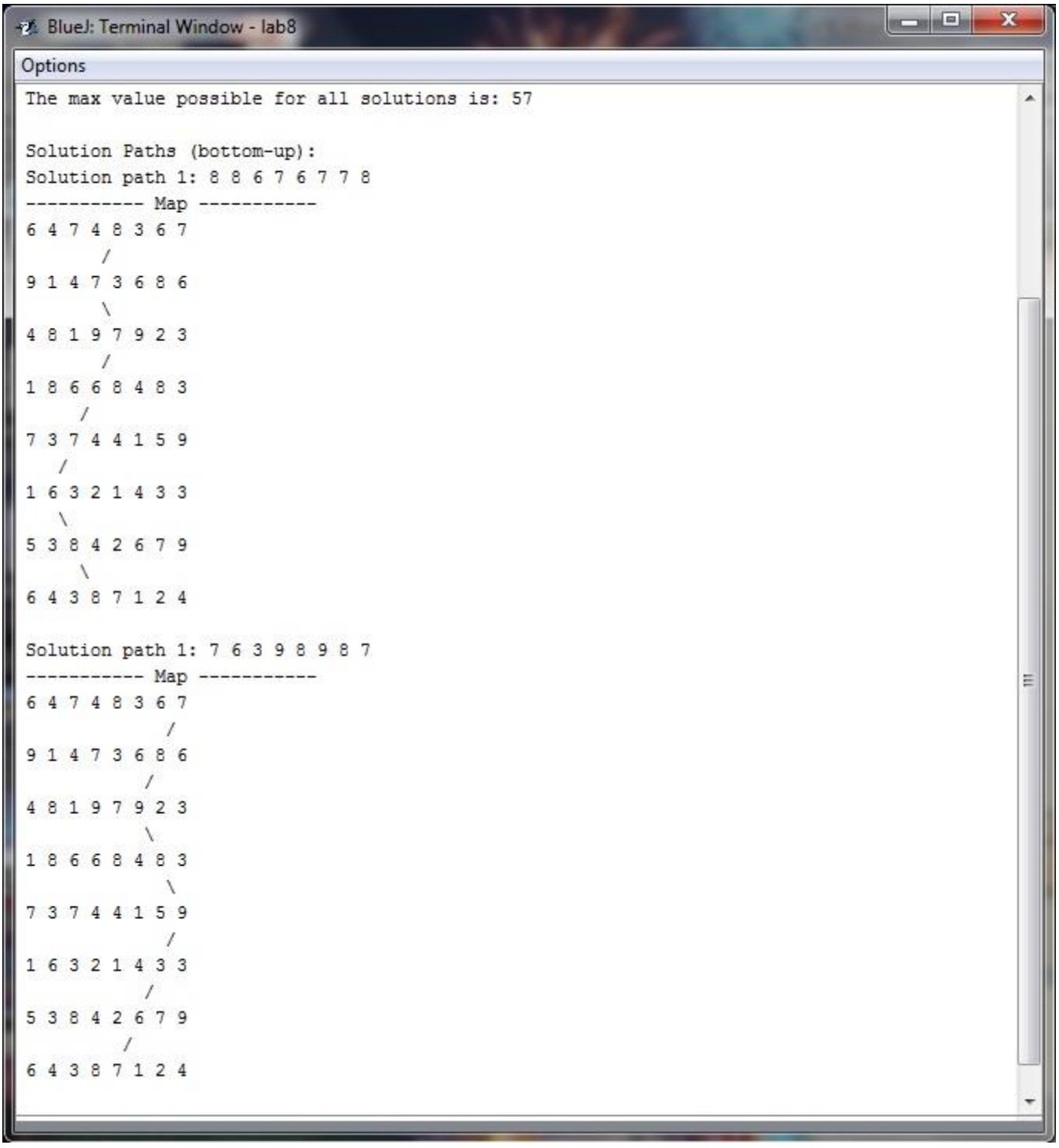
6. Coding Bat: Solve the following recursion problems: factorial , bunnyEars, count7, sumDigits

7. Consider an 8x8 checkerboard that has a dollar amount printed on each of its squares. (This is already defined at the top of lab 8 file—look for int board[][]). Imagine you can place a checker on the board anywhere you want and then move it across the board with standard diagonal moves. Once you reach the other side, you are finished. You will collect an amount of money equal to the sum of the values written on the squares that your checker traveled over. **a.** Give a recursive algorithm that will compute the maximum amount you can collect.

Hints: call the method maxValue. The method should accept a row and column number as input, and return the maximum dollar value possible by moving the piece to the far side of the board. Assume that row 7 is the bottom of the board and row 0 is the destination. The general case will involve invoking maxValue on the square found by moving up (subtract 1 from row) and left (subtract 1 from column) and moving up and right (add + to column) from the current row and column, and simply returning the larger of the two. You'll need to add a base case (row = 0) and figure out how to handle the edges so you don't fall off the board.

Here are some additional hints:
Base Case 1: **row is 0**. Just return the value of the board[row][col] (no further recursive calls)
Base Case 2: **col is either less than 0 or greater than 7** (fell off board). Just return zero (no further recursive calls)
General Case: set left equal to the maxValue (recursive call) of the cell above and to the left of board[row][col], and set right to the maxValue (recursive call) of the cell above and to the right of board[row][col]. Add whichever is bigger (left or right) to the value at board[row][col] and return this value.

For an extra challenge, print the sequence of coordinates in the maxValue path, as shown below:


BlueJ: Terminal Window - lab8
Options
The max value possible for all solutions is: 57
Solution Paths (bottom-up):
Solution path 1: 8 8 6 7 6 7 7 8
---------- Map ----------
6 4 7 4 8 3 6 7
/
9 1 4 7 3 6 8 6
\
4 8 1 9 7 9 2 3
/
1 8 6 6 8 4 8 3
/
7 3 7 4 4 1 5 9
/
1 6 3 2 1 4 3 3
\
5 3 8 4 2 6 7 9
\
6 4 3 8 7 1 2 4
Solution path 1: 7 6 3 9 8 9 8 7
---------- Map ----------
6 4 7 4 8 3 6 7
/
9 1 4 7 3 6 8 6
/
4 8 1 9 7 9 2 3
\
1 8 6 6 8 4 8 3
\
7 3 7 4 4 1 5 9
/
1 6 3 2 1 4 3 3
/
5 3 8 4 2 6 7 9
/
6 4 3 8 7 1 2 4