Programming Project 4

EE312 Fall 2013

Due Monday 10/7 before the end of the day

FIVE POINTS

General: This project will give you a chance to write some recursive functions. PLEASE NOTE: Your recursive functions must not use loops and must not use global variables!

Your Mission: Edit the file Project4.c. This project consists of five parts. For each part you should write both a recursive function and an iterative function to solve the problem (i.e., you'll write a total of eight functions for this project). Each of the projects is independent, and they're all very short. **NOTE: We will only grade the recursive version of each problem, so if you'd like, you can skip the iterative versions. For both part 4 and part 5, the iterative version is challenging (especially for part 5).**

Part 1: Write three functions to find the smallest element in a set. The first function, *minIt,* should use a loop (i.e., be an iterative solution). The second function *minRec1* should be a recursive function that uses a decomposition where the size of the smaller problem is n-1 (where n is the size of the original problem). The third function, *minRec2* should be a recursive function that uses a decomposition where the size of the smaller problem is n/2. Note that your recursive functions will not have any loops!

Part 2: Write two functions to calculate the square root of a number. The first function, *squareIt* should be iterative and the second function *squareRec* should be recursive. You'll be using floating point numbers (type **double**) for this question, so it is impossible to calculate the square root exactly. Instead you must calculate the square root accurate to 15 decimal digits. Your functions will use three parameters. The first parameter is the value for which you must calculate the square root. The other two parameters are "guesses". The first guess is guaranteed to be smaller than the actual square root. The second guess is guaranteed to be larger than the actual square root. These parameters should give you a big hint how to solve this problem.

Part 2: Write two functions to perform string comparison. Both functions must use only recursion (no loops or global variables). In the first function, write the conventional strcmp function, which compares characters using their ASCII values. The comparison is case sensitive, and punctuation is significant. So, for example, " zzz" is less than "aaa" because the former string has a space in front (ASCII 32) which is less than the ASCII for 'a'. Follow the conventions for strcmp, return -1 if the first string is less than the second, return 0 if the strings are the same and return 1 if the first string is greater than the second. Call your function strCompare (it's already declared for you in the project file).

Once you've finished writing the traditional strcmp function (using only recursion!) write a new version of the function that compares only the letters in the strings and ignores the case (upper/lower case) of the letters. For this version, " ZZZ" is greater than "aaa" since

the space is ignored and the capitalization is also ignored. Similarly, the two strings "C++ programming" and "c programming" are equal to each other. Name this function strCompare2 (it's also already declared for you in the project file).

Part 4: Write to functions to find the solution to a maze. The maze is represented by a two-dimensional matrix. The syntax for accessing an element of the matrix is `maze[row][col]` where *row* and *col* are the row number and column number for a cell in the matrix. If the value of an element in the matrix is 1, then the corresponding square in the matrix is a wall, and you cannot go into that square. If the value of the element is 0, then you can go into the square. The entrance to the maze is a suqare in row 0 and the exit is a square in the last row (MATRIX_SIZE – 1). The maze is square with MATRIX_SIZE rows and MATRIX_SIZE columns.

The maze is generated with some special properties that make finding a solution relatively easy. There is exactly one path between any two squares in the maze (except walls of course, there are no paths that allow you to walk into a wall). To look at the maze that's generated, call the *printMaze* function and you'll see what I mean.

For the iterative solution to the maze problem, you'll rely on this property. The algorithm you'll use is "follow the right hand wall". The image you should have is that of a blind person trying to get out of the maze. By sticking out their right arm and keeping their right hand along a wall, they'll eventually get out of the maze. To code this algorithm in C, you'll need to keep track of where you currently are in the maze, and what direction you are currently heading. I've written a few functions to help you "move" through the maze. The functions *turnRight* and *turnLeft* assume that you're using an integer to keep track of your direction. The four directions are 0 for up (rows get smaller, columns stay the same) 1 for moving right (the row stays the same, but the column increases), 2 for moving down (the row increases, the column stays the same) and three for moving left (the row stays the same and the column decreases). The function *adjacentCell* calculates the correct row and column for one of the four adjacent cells to your current position. Note that you are not allowed to move diagonally, only one of the four directions.

Anyway, the basic technique is to keep walking through the maze with your right hand on the wall until you get to the last row of the maze at which point you should stop (you've found the exit). To prove that you've found your path through the maze, you should leave a trail of "bread crumbs" on the path. For full credit, your bread crumbs should only be left along the correct shortest path to the exit (this is not as hard as it might sound – the maze is designed to ensure that there's only one path to the exit that does not involve backtracking). To indicate a bread crumb, you should set the element in the matrix equal to 2. When you run *printMaze* any of the squares with bread crumbs will be displayed as the letter "o".

For the recursive solution to the problem you'll write a much more general maze solver. Unlike the iterative version, there's a fairly straightforward recursive solution to mazes that will work with any type of maze (recall for this problem we're generating a special

maze so that the "follow the right hand wall" will work). In any event, for the recursive solution you'll drop a bread crumb in the current square and then check to see if any of your adjacent squares are the best path to the exit. If any square you encounter already has a bread crumb in it, then it is not on the best path to the exit (since you've already been to this square and dropped a bread crumb, you're walking around in circles, hardly the best path to anywhere). If all of the adjacent squares are either walls, or have bread crumbs in them, then this square is clearly not on the best path. Similarly, if all the adjacent squares are walls, or are known not to be on the best path, then this square is not on the best path. There's a bit more detailed hint in the comments in *Project8.cpp*.

Please note the maze is much more challenging than the first three. The iterative version is interesting, but will not be graded.

Part 5: Write a function that makes change. The input to the function is the amount of money (cents). The return value from the function must be a struct where the components indicate the number of coins which will add up to the amount of money. For this problem, we'll use a fictitious currency (Martian currency) which has 1-cent, 5-cent and 12-cent coins (pennies, nicks and dodeks). To receive credit, your function must use the minimum number of coins possible. For example, if the input to the function were 15, then you should return a Martian struct with the dodek and pennies components both set to zero and the nicks component set to 3 (since three nicks is the most efficient way to create 15 cents using martian currency). If the input were 17, then you should return a Martian struct with pennies equal to zero, nicks equal to 1 and dodeks equal to 1.

NOTE: I encourage you to think about how you would write an iterative solution to this problem. Short of building a lookup table (or an if-then-else with about 20 conditions in it), there really is no easy iterative solution. So, I encourage you to *think* about it, so that you can convince yourself that iteration isn't always the best approach. We won't grade your iterative solution to this problem (if you come up with one). The recursive solution is actually quite easy.