Homework 2: Recursion (due Monday, July 31st at 11:59:00 PM)

1. Here are five functions, with descriptions of what they are supposed to do. They are incorrectly implemented. Replace the incorrect implementations of these functions with correct ones that use recursion in a useful way; your solution must not use the keywords while, for, or goto. You must not use global variables or variables declared with the keyword static, and you must not modify the function parameter lists. You must not use any references or pointers as parameters except for the parameters representing arrays. For the string functions, you may only use the substr() and size()/length() functions, as I want you to become familiar with utilizing array manipulation. I strongly discourage you using helper functions for this assignment, as it can be solved without it.

```
// Returns the product of two non-negative integers, m and n,
// using only repeated addition.
int productOfTheSystem(unsigned int m, unsigned int n)
{
    return -1;  // This is incorrect.
}

// Returns the number of occurrences of digit in the decimal
// representation of num. digit is an int between 0 and 9
// inclusive.
//
// Pseudocode Example:
//    accountForADigit(18838, 8) => 3
//    accountForADigit(55555, 3) => 0
//    accountForADigit(0, 0)     => 0 or 1 (either if fine)
//
int accountForADigit(int num, int digit)
{
    return -1;  // This is incorrect.
}

// Returns a string where the same characters next each other in
// string n are separated by "88" (you can assume the input
// string will not have 8's)
//
// Pseudocode Example:
//    eightClap("goodbye") => "go88odbye"
//    eightClap("yyuu")    => "y88yu88u"
//    eightClap("aaaa")    => "a88a88a88a"
//
string eightClap(string n)
```

```
{
    return "";  // This is not always correct.
}

// str contains a single pair of the less than and greater than
// signs, return a new string made of only the less than and
// greater than sign and whatever is contained in between
//
//  Pseudocode Example:
//     coneHeads("abc<ghj>789") => "<ghj>"
//     coneHeads("<x>7")        => "<x>"
//     coneHeads("4agh<y>")     => "<y>"
//     coneHeads("4agh<>")      => "<>"
//
string coneHeads(string str)
{
    return "*";  // This is incorrect.
}

// Return true if the total of any combination of elements in
// the array a equals the value of the target.
//
//  Pseudocode Example:
//     conglomerateOfNumbers({2, 4, 8}, 3, 10) => true
//     conglomerateOfNumbers({2, 4, 8}, 3, 6)  => true
//     conglomerateOfNumbers({2, 4, 8}, 3, 11) => false
//     conglomerateOfNumbers({2, 4, 8}, 3, 0)  => true
//     conglomerateOfNumbers({}, 0, 0)         => true
//
bool conglomerateOfNumbers(const int a[], int size, int target)
{
    return false;  // This is not always correct.
}
```

2. Write a C++ function named `findAWay` that determines whether or not a there's a path from start to finish in a rectangular maze. Here is the prototype:

```
bool findAWay(string maze[], int nRows, int nCols,
                int sr, int sc, int er, int ec);
// Return true if there is a path from (sr,sc) to (er,ec)
// through the maze; return false otherwise
```

The parameters are:

- A rectangular maze of Xs and dots that represents the maze. Each string of the array is a row of the maze.
- Each 'X' represents a wall, each '@' represents a bottomless trap hole, and each '.' represents a walkway.
- The number of rows in the maze.
- The number of columns in the maze. Each string in the maze parameter must be this length.
- The starting coordinates in the maze: `sr, sc`; the row number is in the range 0 through nRows – 1 and the column number is in the range 0 through nCols – 1.
- The ending coordinates in the maze: `er, ec`; the row number is in the range 0 through nRows – 1, and the column number is in the range 0 through nCols – 1.

Here is an example of a simple maze with 5 rows and 7 columns:

```
"XXXXXXX"
"X...X@X"
"XXX.X.X"
"X@....X"
"XXXXXXX"
```

The function must return true if in the maze as it was when the function was called, there is a path from `maze[sr][sc]` to `maze[er][ec]` that includes only walkways, no walls or bottomless trap holes; otherwise it must return false. The path may turn north, east, south, and west, but not diagonally. When the function returns, it is allowable for the maze to have been modified by the function.

(Our convention is that (0,0) is the northwest (upper left) corner, with south (down) being the increasing r direction and east (right) being the increasing c direction.)

Here is pseudocode for your function:

```
If the start location is equal to the ending location, then
we've solved the maze, so return true.
     Mark the start location as visited.
     For each of the four directions,
          If the location one step in that direction (from the
            start location) is unvisited,
               then call findAWay starting from that
                 location and ending at the same ending location
                 as in the current call).
                    If that returned true,
                         then return true.
     Return false.
```

(If you wish, you can implement the pseudocode for loop with a series of four if statements instead of a loop.)

Here is how a client might use your function:

```
int main()
{
   string maze[10] = {
      "XXXXXXXXXX",
      "X.......@X",
      "XX@X@@.XXX",
      "X..X.X...X",
      "X..X...@.X",
      "X....XXX.X",
      "X@X....XXX",
      "X..XX.XX.X",
      "X...X....X",
      "XXXXXXXXXX"
   };

   if (findAWay(maze, 10, 10, 6, 4, 1, 1))
      cout << "Solvable!" << endl;
   else
      cout << "Out of luck!" << endl;

   return 0;
}
```

Because the focus of this homework is on practice with recursion, we won't demand that your function be as robust as we normally would. In particular, your function may make the following simplifying assumptions (i.e., you do not have to check for violations):

- the maze array contains nRows rows (you couldn't check for this anyway);
- each string in the maze is of length nCols;
- the maze contains only Xs, @s, and dots when passed in to the function;
- the top and bottom rows of the maze contain only Xs, as do the left and right columns;
- `sr` and `er` are between 0 and nRows – 1 and `sc` and `ec` are between 0 and nCols – 1;
- `maze[sr][sc]` and `maze[er][ec]` are '.' (i.e., not walls or bottomless trap holes)

(Of course, since your function is not checking for violations of these conditions, make sure you don't pass bad values to the function when you test it.)

Turn It In

You will use BruinLearn to turn in this homework. Turn in one zip file that contains your solutions to the homework problem. The zip file itself must be named in the following format (no spaces): LastNameFirstName_SID_AssignmentTypeAssignmentNumber.zip (AssignmentType: P=project, H=homework; AssignmentNumber = {1,2,3,4}). An example is BruinJoe_123456789_H1.

The zip file must contain only the file `recursion.cpp`, if you don't finish it all you should still have the empty headers that return dummy values. Your code must be such that if we insert it into a suitable test framework with a main routine and appropriate #include directives, it compiles. In other words, it must have no missing semicolons, unbalanced parentheses, undeclared variables, etc., as well as a main function that has the minimum:

```
int main() {
   return 0;
}
```