Objective

- Programs, which let you think recursive.

Task-1

We discussed the Case Study: "Rat in a Maze" in the class. You are required to implement this task as follows

Input

A file named 'in.txt' will contain the input matrix, which represents the Maze. First line of input shows the order of matrix; second and third line represents the starting and ending (goal) position of rat respectively. From 4th line to onward the matrix of given order is written.

Output

You will store your output in a file named 'out.txt'. The 1st, 2nd and 3rd line are same as 'in.txt' file. Then you will have to write the path in the file after which you will write the matrix such that * will be placed in place of path.

Sample Input

8

0 0

7 7

1 1 1 0 0 1 1 1

1 0 1 1 0 1 1 1

1 1 0 1 1 1 0 1

1 1 0 0 0 0 0 1

1 1 1 1 0 1 1 1

1 1 1 1 0 0 0 1

1 0 0 1 1 1 1 1

1 1 1 0 1 1 1 1

Sample Output

8

0 0

7 7

(0,0)(0,1)(0,2)(2,2)(2,3)(3,3)(3,4)(3,5)(2,5)(1,5)(1,6)(1,7)(2,7)(3,7)(4,7)(5,7)(6,7)(7,7)

* * * 0 0 * * *

1 0 * * 0 * 1 *

1 1 0 * * * 0 *

1 1 0 0 0 0 0 *

1 1 1 1 0 1 1 *

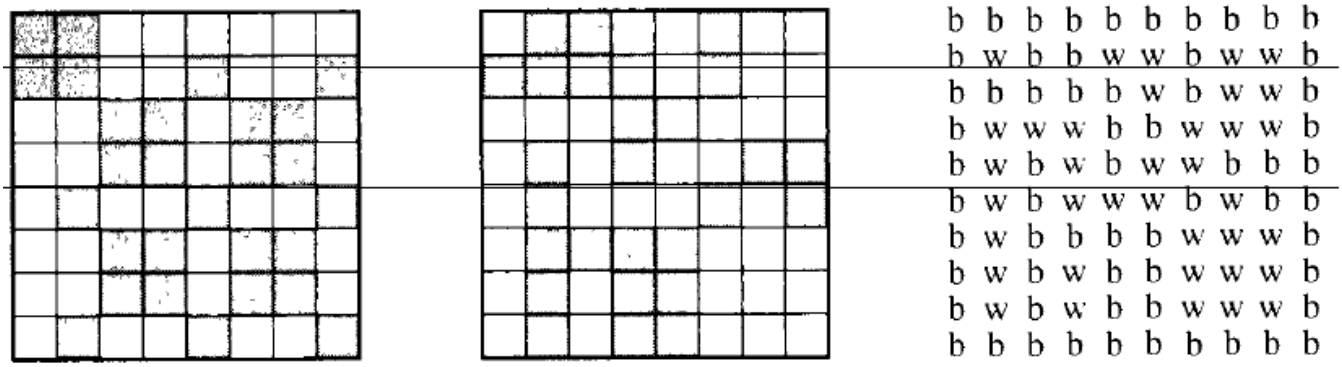
1 1 1 1 0 0 0 *

1 0 0 1 1 1 1 *

1 1 1 0 1 1 1 *

Task-2

(a–b) Two $n \times n$ squares of black and white cells and (c) an $(n + 2) \times (n + 2)$ array implementing square (b).



(a) (b)

b	b	b	b	b	b	b	b	b	b
b	w	b	b	w	w	b	w	w	b
b	b	b	b	b	w	b	w	w	b
b	w	w	w	b	b	w	w	w	b
b	w	b	w	b	w	w	b	b	b
b	w	b	w	w	w	b	w	b	b
b	w	b	b	b	b	w	w	w	b
b	w	b	w	b	b	w	w	w	b
b	w	b	w	b	b	w	w	w	b
b	b	b	b	b	b	b	b	b	b

(c)

An $n \times n$ square consists of black and white cells arranged in a certain way. The problem is to determine the number of white areas and the number of white cells in each area. For example, a regular 8×8 chessboard has 32 one-cell white areas; the square in Figure 5.22a consists of ten areas, two of them of ten cells, and eight of two cells; the square in Figure 5.22b has five white areas of one, three, twenty-one, ten, and two cells.

Write a program that, for a given $n \times n$ square, outputs the number of white areas and their sizes. Use an $(n + 2) \times (n + 2)$ array with properly marked cells. Two additional rows and columns constitute a frame of black cells surrounding the entered square to simplify your implementation. For instance, the square in Figure 5.22b is stored as the square in Figure 5.22c.

Traverse the square row by row and, for the first unvisited cell encountered, invoke a function that processes one area. The secret is in using four recursive calls in this function for each unvisited white cell and marking it with a special symbol as visited (counted).

Task-3

Write a recursive method sumDigits that has one integer parameter and returns the sum of the digits in the integer specified. Remember, your method should not use loops. For example, if the integer is 15121, then this method should return 10.

Task-4

Write a method printSquares that has an integer parameter n, and prints the squares of the integers from 1 to n, separated by commas. It should print the squares of the odd integers in descending order first and then following with the squares of the even integers in ascending order. It does not print a newline character.

For example, printSquares(4) should print 9, 1, 4, 16 printSquares(1) should print 1 printSquares(7) should print 49, 25, 9, 1, 4, 16, 36

You may NOT use helper methods to solve this problem; write a single method.

Task-5
Write a recursive function reverseLines that accepts a file input stream and prints the lines of that file in reverse order.

Task-6
Write a recursive function evenDigits that accepts an integer and returns a new number containing only the even digits, in the same order. If there are no even digits, return 0.
– Example: evenDigits(8342116) returns 8426

Task-7
The subset sum problem is an important and classic problem in computer theory. Given a set of integers and a target number, your goal is to find a subset of those numbers that sum to that target number. For example, given the numbers {3, 7, 1, 8, -3} and the target sum 4, the subset {3, 1} sums to 4. On the other hand, if the target sum were 2, the result is false since there is no subset that sums to 2. The prototype for this function is

int canMakeSum(int* array, int targetSum)

Task-8
You're standing at the base of a staircase and are heading to the top. A small stride will move up one stair, a large stride advances two. You want to count the number of ways to climb the entire staircase based on different combinations of large and small strides. For example, a staircase of three steps can be climbed in three different ways: via three small strides or one small stride followed by one large stride or one large followed by one small. A staircase of four steps can be climbed in five different ways (enumerating them is an exercise left to reader :-).
Write the recursive function **int countWays(int numStairs)** that takes a positive **numStairs** value and returns the number of different ways to climb a staircase of that height taking strides of one or two stairs at a time.
Here's a hint about the recursive structure of the problem: consider the options you have at each stair. You must either take a small stride or a large stride; either will take you closer to the goal and therefore represents a simpler instance of the same problem that can be handled recursively. What is the simplest possible situation and how is it handled?

int CountWays(int numStairs)

Task-9

This question is about a one-dimensional puzzle which you can think about as an array of integers. For example,

3	6	4	1	3	4	2	5	3	0

The circle on the first cell in the above array indicates the position of a marker. At each step in the puzzle you are allowed to move the marker the number of squares indicated by the integer present in the location it currently occupies. The marker may move either to the left or to the right, but it can not move beyond the two ends of the array. For example, in the puzzle configuration given above the marker can only move 3 places to the right, because there is no room to move 3 places to the left.

The goal of the puzzle is to move the marker to the 0 which is present in the last location of the array. You can assume that 0 is always placed at the last location of the array and no other array location contains a 0 in it. You can also assume that all the numbers (except 0) in the array are positive integers.

The above configuration can be solved by making following moves:

Starting position	3	6	4	1	3	4	2	5	3	0
Step 1: Move right	3	6	4	1	3	4	2	5	3	0
Step 2: Move left	3	6	4	1	3	4	2	5	3	0
Step 3: Move right	3	6	4	1	3	4	2	5	3	0
Step 4: Move right	3	6	4	1	3	4	2	5	3	0
Step 5: Move left	3	6	4	1	3	4	2	5	3	0
Step 6: Move right	3	6	4	1	3	4	2	5	3	0

Although the above configuration is solvable, there are some configurations of this puzzle which can not be solved. For example, consider this initial configuration:

3	1	2	3	0

With this initial configuration the marker will go back and forth between the two 3's but it will never reach 0. So, the puzzle is unsolvable in this case.

Your task is to write a *recursive* function, which takes as an argument the array which represents an initial configuration of the puzzle and decides whether the puzzle is solvable or not (this function will return true if the puzzle is solvable and return false if it is not). The function prototype will look something like this:

```
bool Puzzle (const int array[], int n, int current);
```

Here, `array` contains the configuration of the puzzle (note that this array is constant and you can not change it inside the function), **`n`** is the number of elements in the array, and **`current`** indicates the current position of the marker. So, in the case of the first configuration given above the initial function call will be **`Puzzle(array,10,0)`** where array contains the ten elements that have been shown above.

Note: Be sure to keep track of the array elements that you have already visited otherwise you may end up with infinite recursion. (Hint: For this you can use a temporary array. You are allowed to change the above function prototype slightly to incorporate the temporary array).

Task-10

Google the following:

- http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=48
- http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&page=show_problem&problem=320
- 8 puzzle problem
- N Queens Problems
- Suduko Solver

Task-11

- Have a look at following links which shows lot of beautiful recursive patterns
  http://arjay.bc.ca/Modula-2/Text/Ch18/Ch18.4.html
  http://www.coderholic.com/recursively-drawing-trees-with-javascript-and-canvas/
  http://www.cis.upenn.edu/~matuszek/cit594-2009/Assignments/02-recursive-drawings.html
  http://library.thinkquest.org/18222/root/story3/page2/page2.htm