Lab: Recursion and Backtracking

This document defines the lab for the "Algorithms – Fundamentals (C#)" course @ Software University".

Please submit your solutions (source code) to all below-described problems in Judge.

1. Recursive Array Sum

Write a program that finds the sum of all elements in an integer array. Use **recursion**.

Note: In practice, this recursion should not be used here (instead use an **iterative solution**), this is just an exercise.

Examples

Input	Output
1 2 3 4	10
-1 0 1	0

Hints

Write a **recursive** method. It will take as arguments the **input array** and an **index**.

- The method should return the **current element** + the **sum of all next elements** (obtained by recursively calling it):

```
private static int GetSum(int[] arr, int index)
{
    return arr[index] + GetSum(arr, index: index + 1);
}
```

- The recursion should stop when there are no more elements in the array:

```
if (index >= arr.Length)
{
    return 0;
}
```

- This is how the complete solution should look:

```
private static int GetSum(int[] arr, int index)
{
    if (index >= arr.Length)
    {
        return 0;
    }

    return arr[index] + GetSum(arr, index: index + 1);
}
```

2. Recursive Drawing

Write a program that draws the figure below depending on **n**.

Examples

Input	Output
2	** * # ##
5	***** **** *** ** * # ## ### #### #####

Hints

- Set the bottom of the recursion:

```
if (n <= 0)
{
    return;
}
```

- Define **pre** and **post** recursive behavior:

```
Console.WriteLine(new string('*', n));
PrintFigure(n - 1);
Console.WriteLine(new string('#', n));
```

3. Generating 0/1 Vectors

Generate all **n**-bit vectors of **0** and **1** in **lexicographic order**.

Examples

Input	Output
3	000 001 010

	011 100 101 110 111
5	00000 00001 00010 … 11110 11111

Hints

- The method should receive as parameters the array which will be our vector and an index.
- The bottom of the recursion should be when the index is outside of the vector.
- To generate all combinations, create a for loop with a recursive call:

```
for (int i = 0; i <= 1; i++)
{
    arr[index] = i;
    Gen01(arr, index: index + 1);
}
```

4. Recursive Factorial

Write a program that calculates the recursively factorial of a given number.

NOTE: In practice, this recursion should not be used here (instead use an **iterative solution**).

Examples

Input	Output
5	120
10	3628800

Hints

Write a **recursive** method. It will take as arguments an integer number.

- The method should return the **current element** * the **result of calculating the factorial of current element - 1** (obtained by recursively calling it).
- The recursion should stop when the last element is reached.

5. Find All Paths in a Labyrinth

You are given a labyrinth. Your goal is to find all paths from the start (cell 0, 0) to the exit, marked with 'e'.

- Empty cells are marked with a dash '-'.
- Walls are marked with a star '*'.

On the first line, you will receive the dimensions of the labyrinth. Next, you will receive the actual labyrinth.

The order of the paths does not matter.

Examples

Input	Output
3 3 --- -*- --e	RRDD DDRR
3 5 -**-e ----- *****	DRRRRU DRRRUR

Hints

- Create methods for reading and finding all paths in the labyrinth.

```
static void Main(string[] args)
{
    lab = ReadLab();
    FindPaths(0, 0, 'S');
}
```

- Create a static list that will hold directions (basically the path).

```
static List<char> path = new List<char>();
```

- Finding all paths should be recursive.

```
private static void FindPaths(int row, int col, char direction)
{
    if (!IsInBounds(row, col))
        return;

    path.Add(direction);

    if (IsExit(row, col))
    {
        PrintPath();
    }
    else if (!IsVisited(row, col) && IsFree(row, col))
    {
        Mark(row, col);
        FindPaths(row, col + 1, 'R');
        FindPaths(row + 1, col, 'D');
        FindPaths(row, col - 1, 'L');
        FindPaths(row - 1, col, 'U');
        Unmark(row, col);
    }

    path.RemoveAt(path.Count - 1);
}
```

- Implement all helper methods that are present in the code above.

6. Queens Puzzle

In this lab, we will implement a recursive algorithm to solve the **"8 Queens" puzzle**. Our goal is to write a program to **find all possible placements of 8 chess queens** on a chessboard so that no two queens can attack each other (on a row, column, or diagonal).

Examples

Input	Output
(no input)	* - - - - - - - - - - - * - - - - - - - - - - * - - - - - * - - - - * - - - - - - - - - - - * - - * - - - - - - - - - * - - - - * - - - - - - - - - - - - * - - - - - - - - - * - - * - - - - - - - - - - - * -

	- - - * - - - - - * - - - - - - - - - - * - - - … *(90 solutions more)*

Hints

Learn about the "8 Queens" Puzzle

Learn about the "8 Queens" puzzle, e.g. from Wikipedia: http://en.wikipedia.org/wiki/Eight_queens_puzzle.

Define a Data Structure to Hold the Chessboard

First, let's define a data structure to hold the **chessboard**. It should consist of 8 x 8 cells, each either occupied by a queen or empty. Let's also define the size of the chessboard as a constant:

Define a Data Structure to Hold the Attacked Positions

We need to **hold the attacked positions** in some data structure. At any moment during the execution of the program, we need to know **whether a certain position {row, col} is under attack** by a queen or not.

There are many ways to **store the attacked positions**:

- By keeping **all currently placed queens** and checking whether the new position conflicts with some of them.
- By keeping an **`int[][]` matrix of all attacked positions** and checking the new position directly in it. This will be complex to maintain because the matrix should change many positions after each queen placement/removal.
- By keeping **sets of all attacked rows, columns, and diagonals**. Let's try this idea:

The above definitions have the following assumptions:

- **The Rows** are 8, numbered from 0 to 7.
- **The Columns** are 8, numbered from 0 to 7.
- The **left diagonals** are 15, numbered from -7 to 7. We can use the following formula to calculate the left diagonal number by row and column: **`leftDiag = col - row`**.
- The **right diagonals** are 15, numbered from 0 to 14 by the formula: **`rightDiag = col + row`**.

Let's take as an **example** the following chessboard with 8 queens placed on it at the following positions:

- {0, 0}; {1, 6}; {2, 4}; {3, 7}; {4, 1}; {5, 3}; {6, 5}; {7, 2}

	0	1	2	3	4	5	6	7
0	Q							
1							Q	
2					Q			
3								Q
4		Q						
5				Q				
6						Q		
7			Q					

Following the definitions above for our example, the **queen {4, 1}** occupies **row 4**, **column 1**, **left diagonal -3,** and **right diagonal 5**.

Write the Backtracking Algorithm

Now, it is time to write the recursive **backtracking algorithm** for placing the 8 queens.

The algorithm starts from row 0 and tries to place a queen at some column at row 0. On success, it tries to place the next queen at row 1, then the next queen at row 2, etc. until the last row is passed.

Check If a Position is Free

Now, let's write **the code to check whether a certain position is free**. A position is free when it is not under attack by any other queen. This means that if some of the rows, columns, or diagonals are already occupied by another queen, the position is occupied. Otherwise, it is free.

Recall that **`col-row`** is the number of the left diagonal and **`row+col`** is the number of the right diagonal.

Mark / Unmark Attacked Positions

After a queen is placed, we need to **mark as occupied all rows, columns, and diagonals** that it can attack.

On removal of a queen, we will need a method to mark as free all rows, columns, and diagonals that were attacked by it.

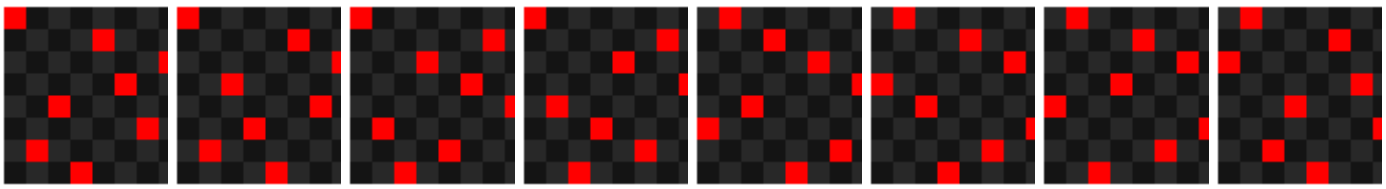
Print Solutions

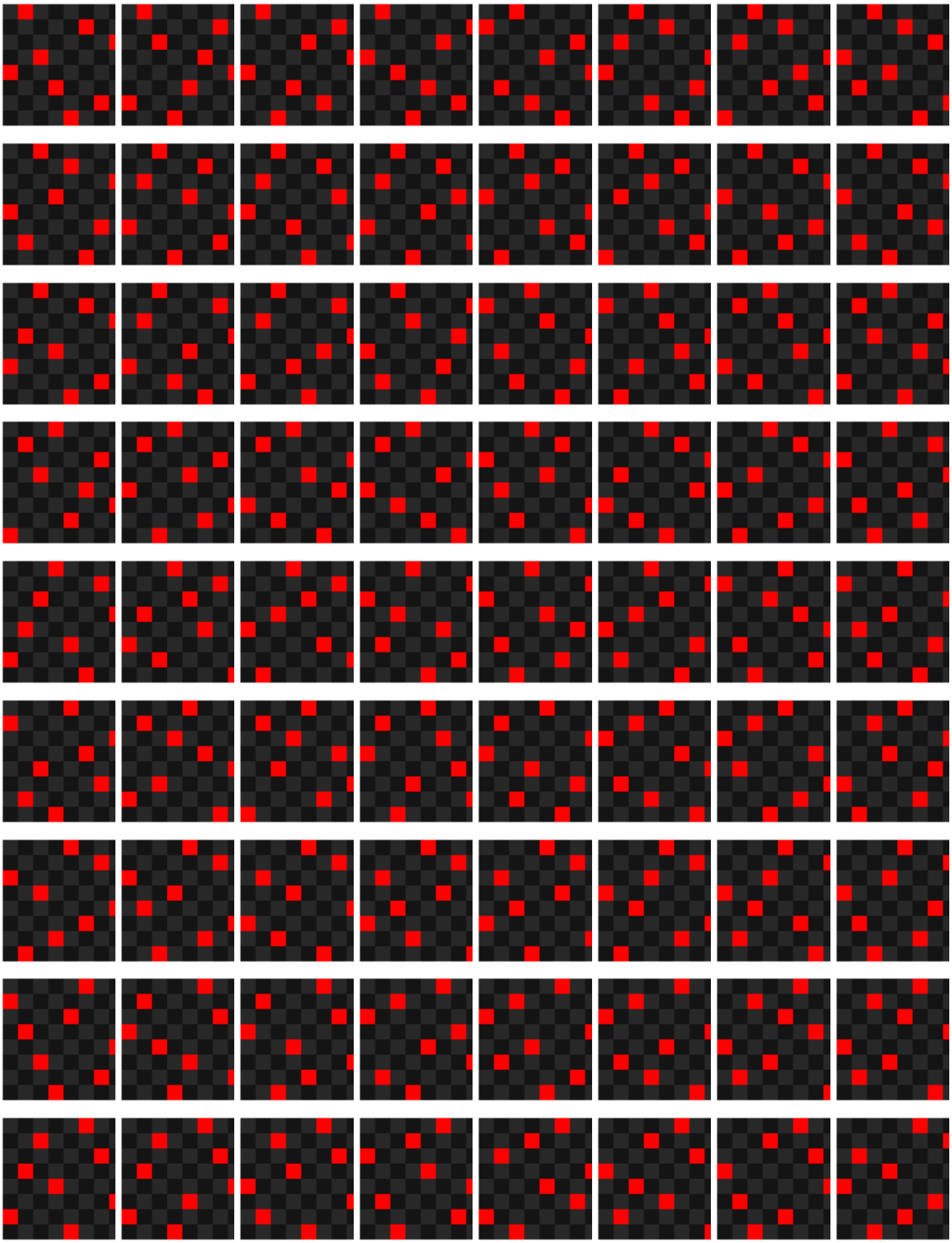
When a solution is found, it should be printed on the console. First, introduce a solutions counter to simplify checking whether the found solutions are correct.

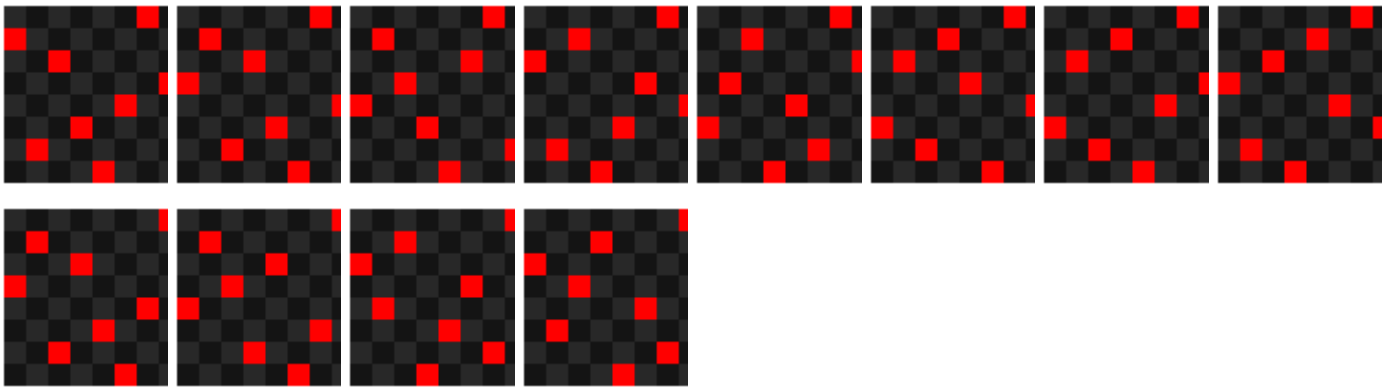
Next, pass through all rows and all columns at each row and **print the chessboard cells**:

Testing the Code

The "8 queens" puzzle has **92 distinct solutions**. Check whether your code generates and prints all of them correctly. The **`solutionsFound`** counter will help you check the number of solutions. Below are the 92 distinct solutions:







Submit your code in judge, printing all 92 solutions, separated by a single line.

Optimize the Solution

Now we can optimize our code:

- Remove the **`attackedRows`** set. It is not needed because all queens are placed consecutively at rows 0…7.
- Try to use **`bool[]`** array for **`attackedColumns`**, **`attackedLeftDiagonals`** and **`attackedRightDiagonals`** instead of sets. Note that arrays are indexed from 0 to their size and cannot hold negative indexes.

7. Recursive Fibonacci

Each member of the **Fibonacci sequence** is calculated from the **sum of the two previous members**. The first two elements are 1, 1. Therefore the sequence goes like 1, 1, 2, 3, 5, 8, 13, 21, 34…

The following sequence can be generated with an array, but that's easy, so **your task is to implement it recursively**.

If the function **GetFibonacci(n)** returns the n^{th} Fibonacci number, we can express it using **GetFibonacci(n) = GetFibonacci(n-1) + GetFibonacci(n-2)**.

However, this will never end and in a few seconds, a Stack Overflow Exception is thrown. In order for the recursion to stop it has to have a "bottom". The bottom of the recursion is getFibonacci(1), and should return 1. The same goes for getFibonacci(0).

Input

- On the only line in the input, the user should enter the wanted Fibonacci number N where $1 \leq N \leq 49$

Output

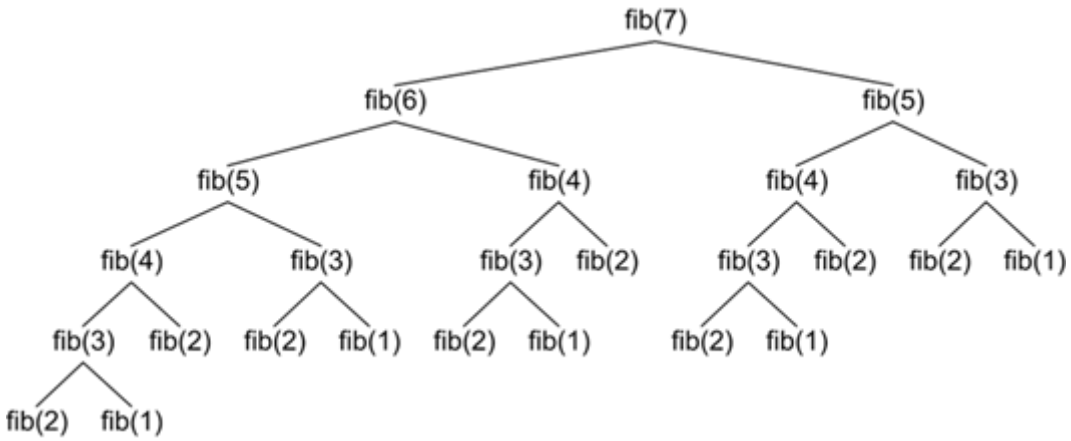
- The output should be the n^{th} Fibonacci number counting from 0

Examples

Input	Output
5	8
10	89
21	17711

Hint

For the nth Fibonacci number, we calculate the N-1st and the N-2nd number, but for the calculation of N-1st number we calculate the N-1-1st(N-2nd) and the N-1-2nd number, so we have a lot of repeated calculations.



If you want to figure out how to skip those unnecessary calculations, you can search for a technique called memoization.