Lab: Recursion

This document defines the **in-class exercises** (lab) for the "Algortihms" course @ Software University.

Part I - Recursive Array Sum

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

Write a **`FindSum`** method. It will take as arguments the **input array** and the **current index**.

- The method should return the **current element** + the **sum of all next elements** (obtained by recursively calling **`FindSum`**).
- The recursion should stop when there are no more elements in the array.

```
static int FindSum(int[] arr,  )
{
    // TODO: Set bottom of recursion
    // TODO: Return sum of current element + sum of elements to the right
}
```

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

Part II - 8 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).

Problem 1. Learn about the "8 Queens" Puzzle

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

Problem 2. 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:

```
class EightQueens
{
    const int Size = 8;
    static bool[,] chessboard = new bool[Size, Size];
}
```

Problem 3. 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:

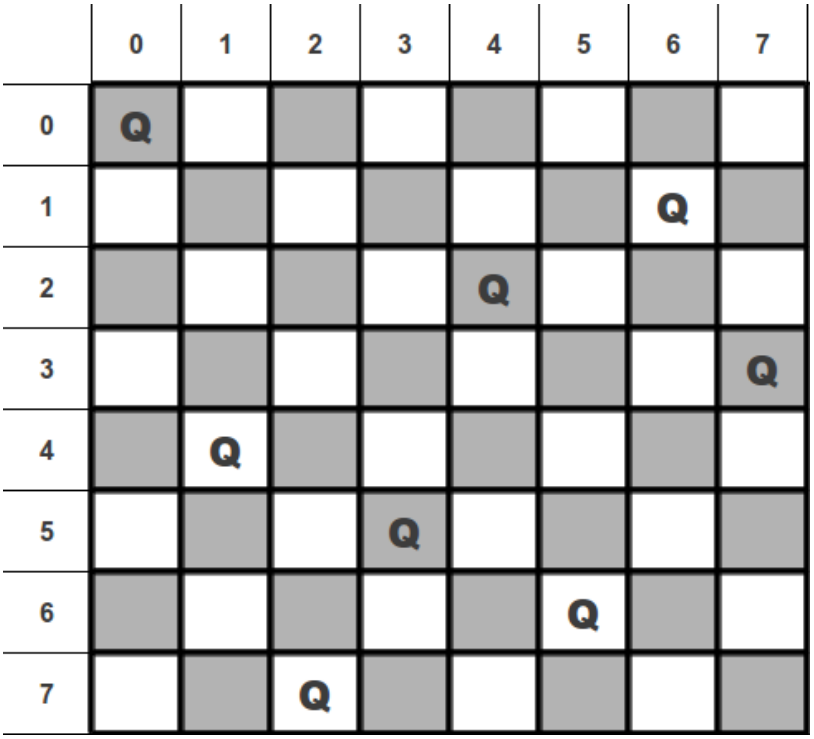
```
static HashSet<int> attackedRows = new HashSet<int>();
static HashSet<int> attackedColumns = new HashSet<int>();
static HashSet<int> attackedLeftDiagonals = new HashSet<int>();
static HashSet<int> attackedRightDiagonals = new HashSet<int>();
```

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}



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

Problem 4. 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. The code for putting the next queen at a certain row might look like this:

```
static void PutQueens(int row)
{
    if (row == Size)
    {
        PrintSolution();
    }
    else
    {
        for (int col = 0; col < Size; col++)
        {
            if (CanPlaceQueen(row, col))
            {
                MarkAllAttackedPositions(row, col);
                PutQueens(row + 1);
                UnmarkAllAttackedPositions(row, col);
            }
        }
    }
}
```

Initially, we invoke this method from row 0:

```
static void Main()
{
    PutQueens(0);
}
```

Problem 5. 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 is already occupied by another queen, the position is occupied. Otherwise it is free. A sample code might look like this:

```
static bool CanPlaceQueen(int row, int col)
{
    var positionOccupied =
         attackedRows.Contains(row) ||
         attackedColumns.Contains(col) ||
         attackedLeftDiagonals.Contains(col - row) ||
         attackedRightDiagonals.Contains(row + col);
    return !positionOccupied;
}
```

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

Problem 6. Mark / Unmark Attacked Positions

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

```
static void MarkAllAttackedPositions(int row, int col)
{
    attackedRows.Add(row);
    attackedColumns.Add(col);
    attackedLeftDiagonals.Add(col - row);
    attackedRightDiagonals.Add(row + col);
    chessboard[row, col] = true;
}
```

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

```
static void UnmarkAllAttackedPositions(int row, int col)
{
    // TODO
}
```

Problem 7. Print Solutions

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

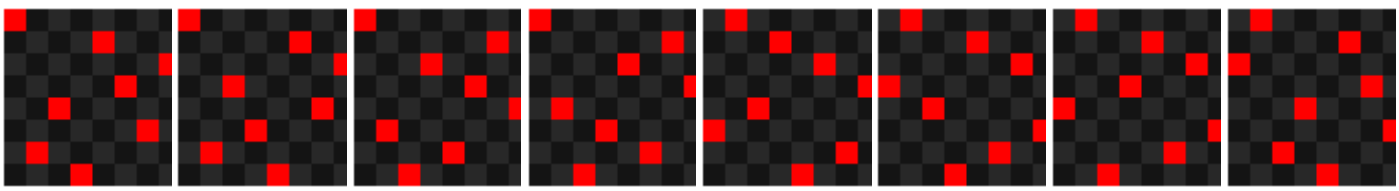
```
class EightQueens
{
    static int solutionsFound = 0;
```

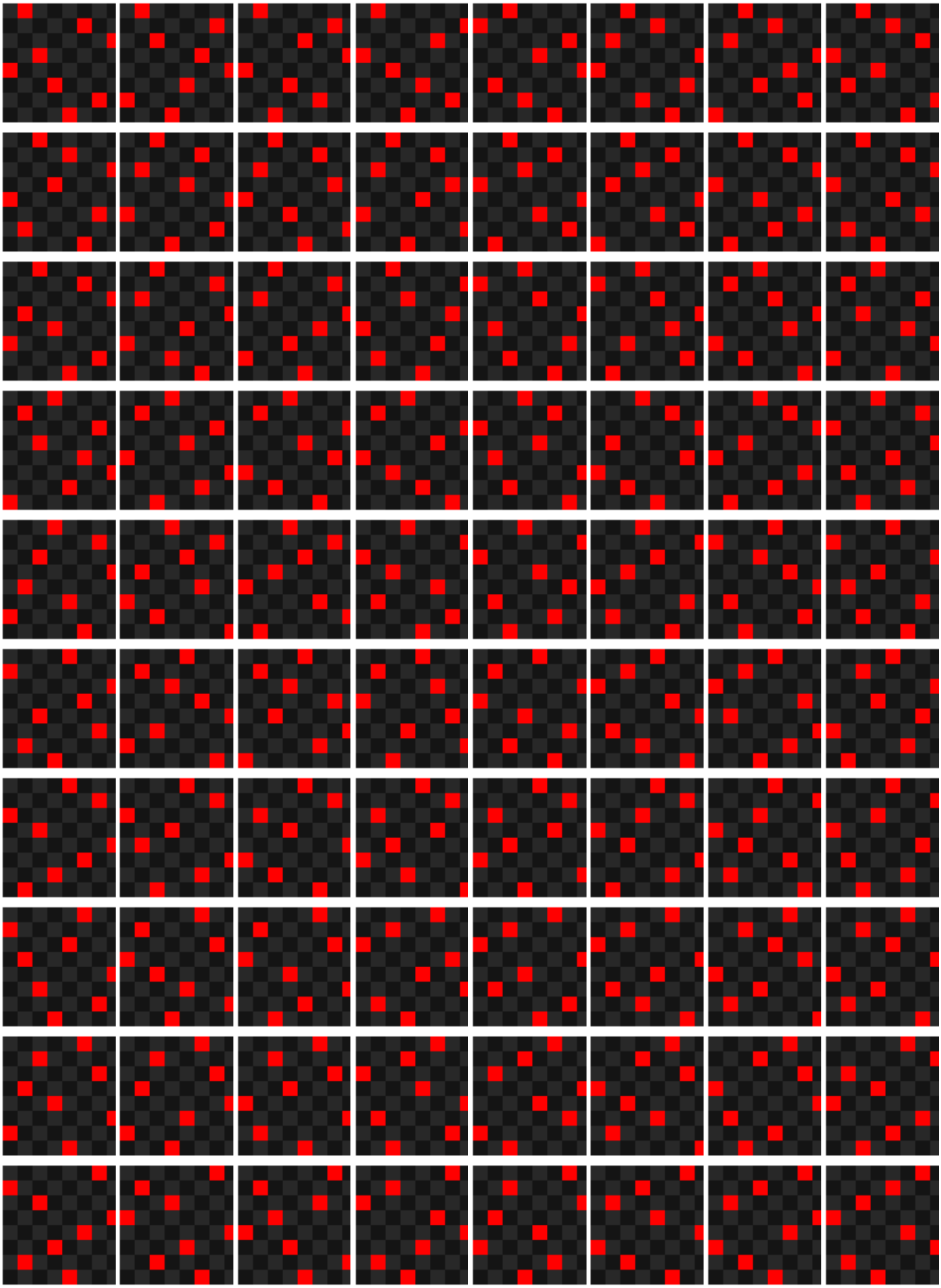
Next, pass through all rows and through all columns at each row and **print the chessboard cells**:

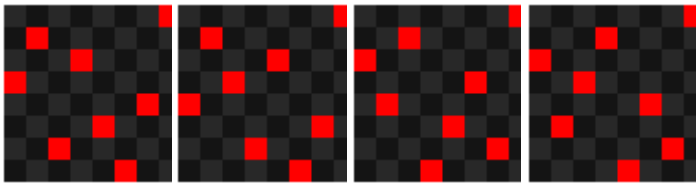
```
static void PrintSolution()
{
    for (int row = 0; row < Size; row++)
    {
        for (int col = 0; col < Size; col++)
        {
            // TODO: print '*' or '-' depending on scoreboard[row, col]
        }
        Console.WriteLine();
    }
    Console.WriteLine();

    solutionsFound++;
}
```

Problem 8. 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:







Problem 9. 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.

Problem 10.* Permutation Based Solution

Try to implement the more-efficient **permutation-based solution** of the "8 Queens" puzzle. Look at this code to grasp the idea: http://introcs.cs.princeton.edu/java/23recursion/Queens.java.html.