Exercises: Recursive Algorithms and Backtracking

Problems for exercises and homework for the "Data Structures and Algorithms Basics" course from the official "Applied Programmer" curriculum.

You can check your solutions here: https://judge.softuni.bg/Contests/2726/Recursive-Algorithms-and-Backtracking

1. Generating 0/1 Vectors

Generate all **n**-bit vectors of zeroes and ones 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 a current **index**

```
private static void Gen01(int[] vector, int index)
```

Bottom of recursion should be when the index is outside of the vector

```
if (index > vector.Length - 1)
{
    // Print vector
}
```

To generate all combinations, create a for loop with a **recursive call**:

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

2. Combinations with Repetition

Write a **recursive** program for generating and printing all combinations **with duplicates** of **k** elements from a set of **n** elements (k <= n). In combinations, the **order of elements doesn’t matter**, therefore (1 2) and (2 1) are the same combination, meaning that once you print/obtain (1 2), (2 1) is no longer valid.

Examples

Input	Output	Comments	Solution with nested loops

3
2
	1 1
	1 2
	1 3
	2 2
	2 3
	3 3
		• n=3 => we have a set of three elements {1, 2, 3}
		• k=2 => we select two elements out of the three each time
		• Duplicates are allowed, meaning (1 1) is a valid combination.
			`int n = 3;`
			`int k = 2;`
			
			`// k == 2 => 2 nested for-loops`
			`for (int i1 = 1; i1 <= n; i1++)`
			`{`
			`    for (int i2 = i1; i2 <= n; i2++)`
			`    {`
			`        Console.WriteLine($"({i1} {i2})");`
			`    }`
			`}`

5
3
	1 1 1
	1 1 2
	1 1 3
	1 1 4
	1 1 5
	1 2 2
	…
	3 5 5
	4 4 4
	4 4 5
	4 5 5
	5 5 5
		Select 3 elements out of 5 – {1, 2, 3, 4, 5}, a total of 35 combinations
		(1 2 1) is not valid as it’s the same as (1 1 2)
			`int n = 5;`
			`int k = 3;`
			
			`// k == 3 => 3 nested for-loops`
			`for (int i1 = 1; i1 <= n; i1++)`
			`{`
			`    for (int i2 = i1; i2 <= n; i2++)`
			`    {`
			`        for (int i3 = i2; i3 <= n; i3++)`
			`        {`
			`            Console.WriteLine($"({i1} {i2} {i3})");`
			`        }`
			`    }`
			`}`

3. Combinations without Repetition

Modify the previous program to **skip duplicates, e.g. (1 1) is not valid.**

Examples

Input	Output	Comments	Solution with nested loops

3
2
	1 2
	1 3
	2 3
		• n=3 => we have a set of three elements {1, 2, 3}
		• k=2 => we select two elements out of the three each time
		• Duplicates are not allowed, meaning (1 1) is not a valid combination.
			`int n = 3;`
			`int k = 2;`
			
			`// k == 2 => 2 nested for-loops`
			`for (int i1 = 1; i1 <= n; i1++)`
			`{`
			`    for (int i2 = i1 + 1; i2 <= n; i2++)`
			`    {`
			`        Console.WriteLine($"({i1} {i2})");`
			`    }`
			`}`

5
3
	1 2 3
	1 2 4
	1 2 5
	1 3 4
	1 3 5
	1 4 5
	2 3 4
	2 3 5
	2 4 5
	3 4 5
		Select 3 elements out of 5 – {1, 2, 3, 4, 5}, a total of 10 combinations	`int n = 5;`
			`int k = 3;`
			
			`// k == 3 => 3 nested for-loops`
			`for (int i1 = 1; i1 <= n; i1++)`
			`{`
			`    for (int i2 = i1 + 1; i2 <= n; i2++)`
			`    {`
			`        for (int i3 = i2 + 1; i3 <= n; i3++)`
			`        {`
			`            Console.WriteLine($"({i1} {i2} {i3})");`
			`        }`
			`    }`
			`}`

4. Generating Combinations

Generate all **n choose k** combinations. Read the set of elements, then number of elements to choose.

Examples

Input	Output

1 2 3 4
2
	1 2
	1 3
	1 4
	2 3
	2 4
	3 4

10 20 30 40 50 3	10 20 30 10 20 40 10 20 50 … 30 40 50

Hints

The method could receive the following parameters:

```
private static void GenCombs(int[] set, int[] vector, int startNum, int index)
{

}
```

Set the **bottom** of the recursion:

```
if (index [redacted] vector.Length)
{
    Console.WriteLine(string.Join(" ", vector));
}
```

Loop through all possible picks for a given index of the vector:

```
for (int i = startNum; i < set.Length; i++)
{
    vector[index] = set[i];
    GenCombs(set, vector, i + 1, index + 1);
}
```

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).

Examples

Input	Output

(no input)	`* - - - - - - -`
	`- - - - * - - -`
	`- - - - - - - *`
	`- - - - - * - -`
	`- - * - - - - -`
	`- - - - - - * -`
	`- * - - - - - -`
	`- - - * - - - -`
	
	`* - - - - - - -`
	`- - - - - * - -`
	`- - - - - - - *`

```
- - * - - - - -
- - - - - - * -
- - - * - - - -
- * - - - - - -
- - - - * - - -

…

(90 solutions more)
```

About the "8 Queens" Puzzle

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

1. 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];
}
```

5. 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**:

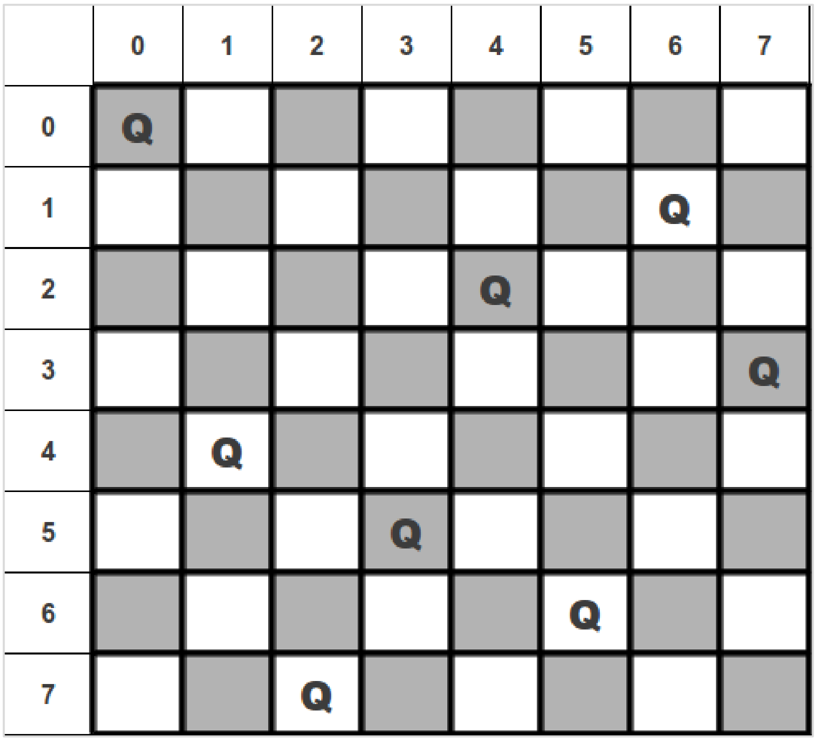
- By keeping **all currently placed queens** and checking whether the new position conflicts with some of them.
- By keeping an int[,] **matrix of all attacked positions** and checking the new position directly in it. This will be complex to maintain because the matrix should change many positions after 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**.

6. 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);
}
```

7. 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.

8. 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
}
```

9. 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;
```

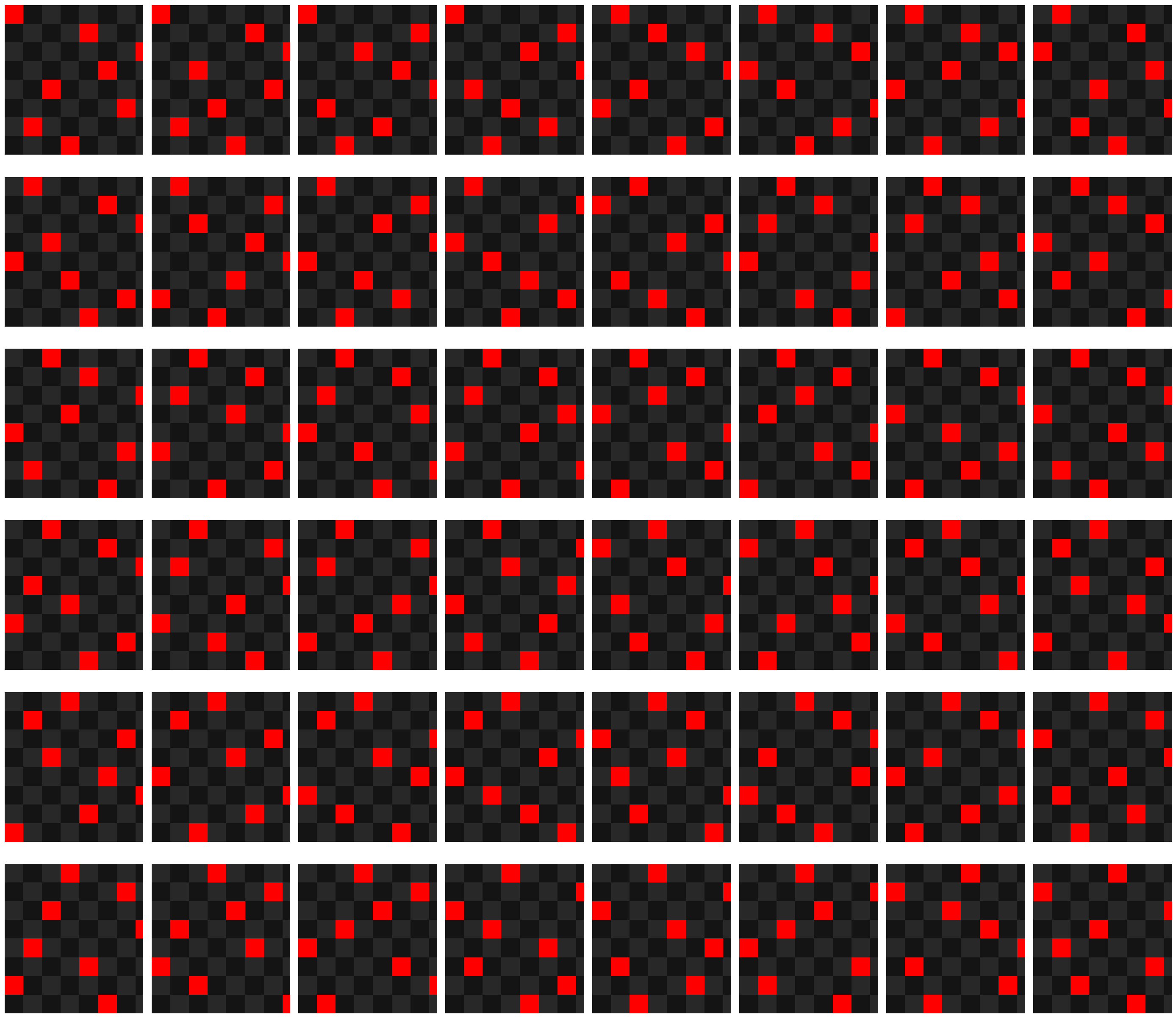
Next, pass through all rows and through all columns at each row and **print the chessboard cells**:

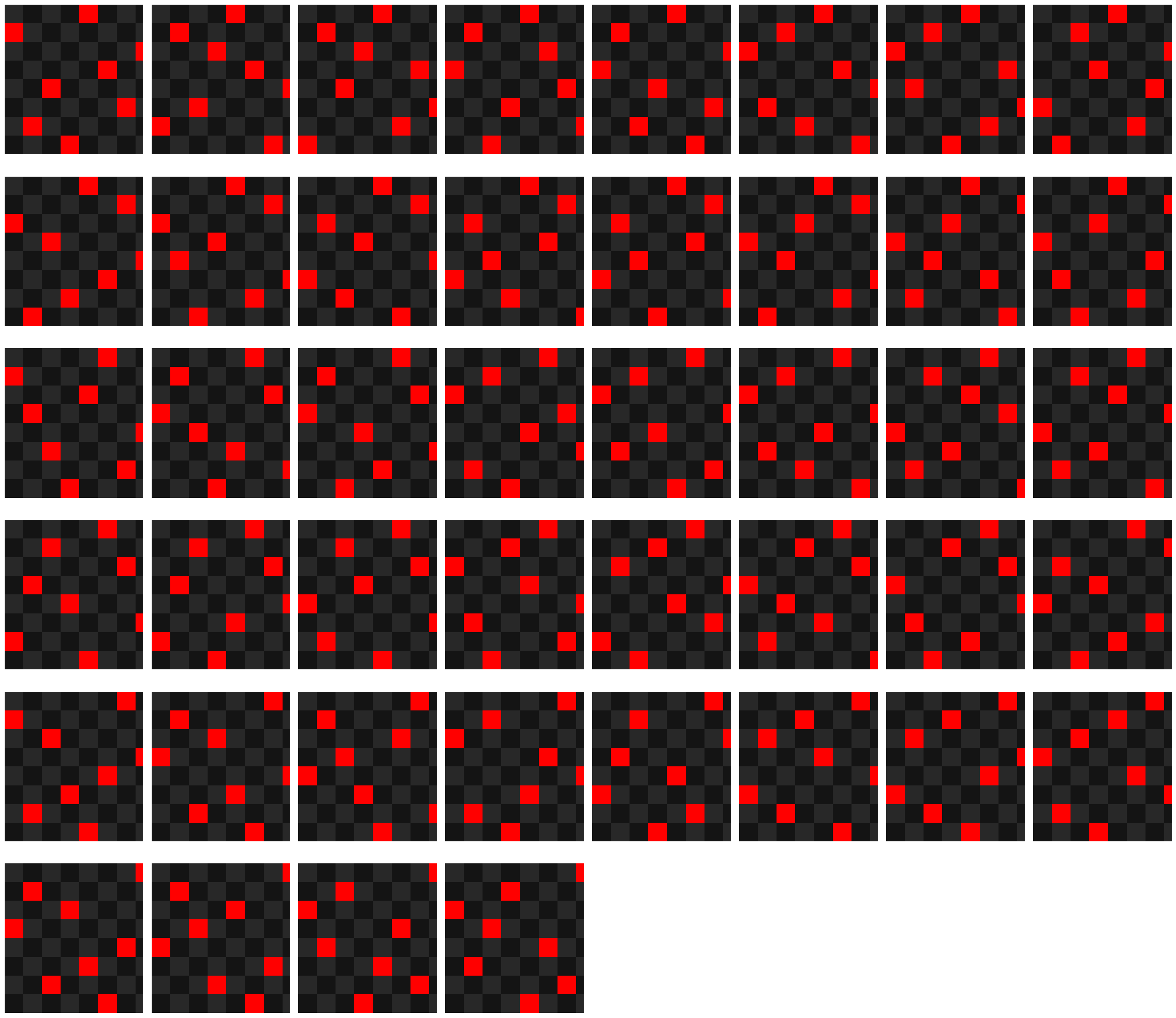
```
static void PrintSolution()
{
    for (int row = 0; row < Size; row++)
    {
        for (int col = 0; col < Size; col++)
        {
            // TODO: print '*' or '-' depending on scoreboard[row, col]
        }
        Console.WriteLine();
    }
    Console.WriteLine();

    solutionsFound++;
}
```

10. 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.

11. Optimize the Solution

Now we can optimize our code:

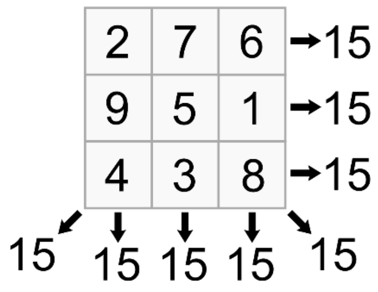
- Remove the **`attackedRows`** set. It is not needed because all queens are placed consecutively at rows 0…7.
- Try to use **`bool[]`** array for **`attackedColumns`**, **`attackedLeftDiagonals`** and **`attackedRightDiagonals`** instead of sets. Note that arrays are indexed from 0 to their size and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

* Permutation Based Solution

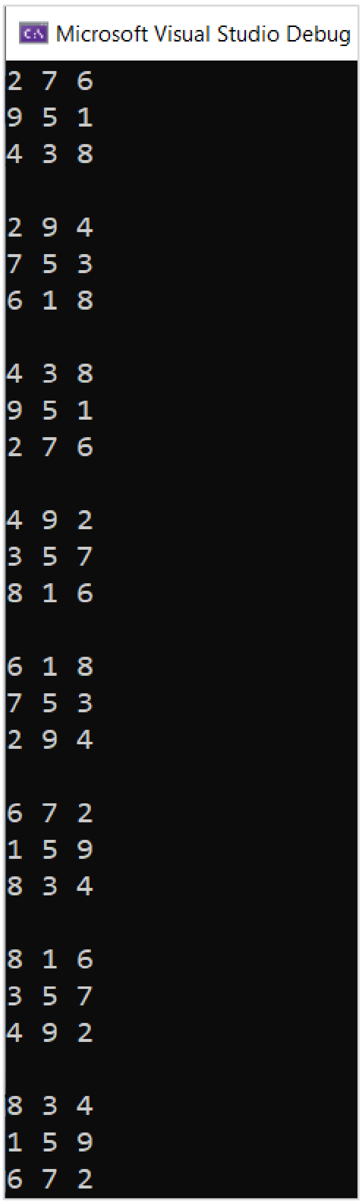
Try to implement the more-efficient **permutation-based solution** of the "8 Queens" puzzle. Look at this code to grasp the idea: http://introcs.cs.princeton.edu/java/23recursion/Queens.java.html.

Part III – * Magic Squares

Your final task is to use **recursion** to print all **3X3 magic squares** with numbers from **1** to **9** inclusively (without repetition). Magic squares are special squares, in which the sums of the numbers in each row, each column, and both main diagonals are the **same**. For more clarity, look at the example below:



This is one of the squares, which you should print. Magic squares with numbers from 1 to 9 inclusively are **8** in total. They are shown below and should be **printed** by your program on the console like this:



```
2 7 6
9 5 1
4 3 8

2 9 4
7 5 3
6 1 8

4 3 8
9 5 1
2 7 6

4 9 2
3 5 7
8 1 6

6 1 8
7 5 3
2 9 4

6 7 2
1 5 9
8 3 4

8 1 6
3 5 7
4 9 2

8 3 4
1 5 9
6 7 2
```

If you want, you can try to find **magic squares** with numbers from **0** to **9** inclusively. They should be **16** in total.

Hints

For more simplicity, you can keep squares as an **array** of integers. The array should have a size of **9** for all cells in the 3X3 square:

```
static int[] square = new int[9];
```

Note that you should also keep track of **numbers**, which have already been **used** in the current square, as they should not be repeated. You may create an **array**, where each number has value **0** (when the number is not used) or **1** (when the number is already used). Whenever you 'go to' a recursive call and 'come back' from a recursive call, you should **update** this array. The array should have a size of **9** to keep information for all numbers from 1 to 9:

```
static int[] visited = new int[9];
```

Then, create a method to **permute** numbers. You can use the instructions from this site to help you write the **recursive** algorithm: https://erwnerve.tripod.com/prog/recursion/magic.htm.

Write a **`TestMagic()`** method to check whether the **current square** is a **magic square**. It is a good idea to generate a **matrix** from the square array, so you can easily check if diagonal sums, row sums and column sums are all the **same**.

If a matrix is a magic square, **print** its numbers like a **3X3 square**. Numbers should be separated by **space** and magic squares should be separated by **an empty line**.

For checking whether a matrix is a **magic square** or not, you may use the following method:

```
static bool isMagicSquare(int[,] matrix)
{
    int N = 3;

    // calculate the sum of
    // the prime diagonal
    int sum = 0, sum2 = 0;

    for (int i = 0; i < N; i++)
        sum = sum + matrix[i, i];

    // the secondary diagonal
    for (int i = 0; i < N; i++)
        sum2 = sum2 + matrix[i, N - 1 - i];

    if (sum != sum2)
        return false;

    // For sums of Rows
    for (int i = 0; i < N; i++)
    {

        int rowSum = 0;
        for (int j = 0; j < N; j++)
            rowSum += matrix[i, j];

        // check if every row sum is
        // equal to prime diagonal sum
        if (rowSum != sum)
            return false;
    }

    // For sums of Columns
    for (int i = 0; i < N; i++)
    {

        int colSum = 0;
        for (int j = 0; j < N; j++)
            colSum += matrix[j, i];

        // check if every column sum is
        // equal to prime diagonal sum
        if (sum != colSum)
            return false;
    }

    return true;
}
```