FIT1045 Intro to Algorithms and Programming – Workshop 10

Objectives

The **objectives of this workshop** are:

- To discuss the use of backtracking in algorithms.
- To implement a backtracking algorithm for the *n*-Queens problem

Useful Links:

For this workshop, you may find it useful to review some of the following concepts:

- Recursion (Lecture 14)
- Backtracking (Lectures 16–17)

Task 0: To be completed before class

Write a Python function named `getRemainingIntegers` that takes as input a positive integer *n*, along with a list *nums*, and returns (as a list) the integers from 1 to n that are NOT part of *nums*.

In Task 1 you will have to write a function which does a similar thing, but extended to satisfy the requirements of the n-Queens problem. Therefore, the remaining part of this task is to review the n-Queens problem. Refer back to the week 6 workshop and the week 9 backtracking lecture. Recall the different ways to represent a partial solution to the problem, and how to check if a solution is valid when (in particular) it is represented as a list L where L[i] gives the row position of the Queen in column i.

Task 1:

Useful material: In this task you may find it useful to reuse some of your code from Workshop 6.

Task A:

Write a Python function named `getPositions` that takes as input a list *L* representing a partial solution to the *n*-Queens problem and a positive integer *n* representing the dimension of the $n \times n$ chessboard. The input list contains the positions of the Queens currently placed on the board (or is empty if no Queens have been placed). You should use the same list representation as you use in Task 3 of Workshop 6, that is, the entry L[i] gives the row position of the Queen in column i. You may assume that the partial solution gives the positions of the Queens in the first *i* columns and that the other columns are empty.

Your program should return a list containing possible row positions for a Queen to be placed in the next column on the board. (The index of the next column can be obtained by the length of *L*.)

For example:
If your list is $[5,3]$ and $n = 6$, your program should return the list $[0,1]$, that is, a Queen could be placed at row 0 column 2, or row 1 column 2, of the board.

Some further examples:
If your list is empty and $n = 5$, your program should return the list $[0,1,2,3,4]$.
If your list is $[3,1]$ and $n = 4$, your program should return an empty list.

Task B:

In order to test your program in Task A, modify the program in Task A so that it prints a table representing the chessboard. It should have Q in entries where a queen is placed, X in entries where a Queen may be placed in the next column and 0 in all other entries.

For example:
If your list is $[5,3]$ and $n = 6$, your program should print:

```
0 0 X 0 0 0
0 0 X 0 0 0
0 0 0 0 0 0
0 Q 0 0 0 0
0 0 0 0 0 0
Q 0 0 0 0 0
```

If your program `getPositions` is correctly implemented, a Queen placed on a position marked 'X' cannot "attack" any of the positions marked 'Q', so you can easily check if 'X' marks a valid position. What else should you check for?

Task 2:

Useful material: In this task you may find the Week 10 "Implementing Backtracking" lecture useful.

Task A:

The following is an algorithm `nQueens` that takes as input a list *partialSolution* and a positive integer n and prints all solutions to the n-Queens problem using backtracking.

```
Function nQueens(partialSolution, n)
{
    positions = getPositions(partialSolution, n)
    If positions is empty
    {
        If length(partialSolution) ==n
        {
            print(partialSolution)
        }
    }
    Else
    {
        For each element in positions
        {
            push(element, partialSolution)

            nQueens(partialSolution, n)

            pop(partialSolution)
        }
    }
}
```

Write a program in Python that takes as input a positive integer n and prints all possible solutions to the n-Queens problem. Your program should use *backtracking*.

For example, your program may output:

```
Enter value for n: 4
Solutions
------------
[1, 3, 0, 2]
[2, 0, 3, 1]
------------
```

and

```
Enter value for n: 2
Solutions
------------
------------
```

Or you might prefer to print them as tables. For example:

```
Enter value for n: 4
Solutions
-------------------------
0 0 Q 0
Q 0 0 0
0 0 0 Q
0 Q 0 0
-------------------------
0 Q 0 0
0 0 0 Q
Q 0 0 0
0 0 Q 0
-------------------------
```

Task B:

Modify your algorithm so that it counts the number of solutions. If there are no solutions, it should print "No Solutions".

Hint: You may have to change the input and output of the function `nQueens`

For example, your program may output:

```
Enter value for n: 4
-------------------------
0 0 Q 0
Q 0 0 0
0 0 0 Q
0 Q 0 0
-------------------------
0 Q 0 0
0 0 0 Q
Q 0 0 0
0 0 Q 0
-------------------------
There are 2 solutions.
```

and

```
Enter value for n: 3
-------------------------
There are no solutions.
```

Task 3 – N-queens brute forced!

In mathematics, the notion of *permutation* relates to the act of arranging all the members of a set into some sequence or order, or if the set is already ordered, rearranging (reordering) its elements, a process called *permuting*. The number of permutations on a set of n elements is given by $n!$ (read as n factorial). For example, there are $2! = 2 \cdot 1 = 2$ permutations of $\{1,2\}$, namely $\{1,2\}$ and $\{2,1\}$, and $3! = 3 \cdot 2 \cdot 1 = 6$ permutations of $\{1,2,3\}$, namely $\{1,2,3\}, \{1,3,2\}, \{2,1,3\}, \{2,3,1\}, \{3,1,2\}$, and $\{3,2,1\}$.

A brute force algorithm for N-queens problem goes as follows:

- Generate all possible permutations of $1, 2, \cdots, N_1, \cdots, N$. Set a counter to 0.
- For each permutation $\Pi_1, \Pi_2, \cdots, \Pi_{N_1}, \cdots, \Pi_N$, place a queen on row i and column Π_i, for $i = 1, 2, N_1, \cdots, N$.
- Test if this configuration is valid; if so, increment the counter.

Implement the brute force algorithm for 4-queens problem.

For example, your program may output:

```
Enter the size of N: 4
Number of solutions: 2
```

Additional Task – What happens if . . .

The n-Quasi-Queens problem is the problem of placing n-Quasi-Queens on an $n \times n$ chessboard so that no Quasi-Queen attacks another. A Quasi-Queen can attack another Quasi-Queen if they are on the same diagonal or in the same column. But a Quasi-Queen can only attack another Quasi-Queen in the same row if the row number is odd.

Suppose you were asked to modify your code to find solutions to the the n-Quasi-Queens problem. What function(s) would you have to modify?

Write a program to solve the n-Quasi-Queens problem.

For example, your program may output

```
Enter value for n: 3
-------------------------
0 Q 0
0 0 0
Q 0 Q
-------------------------
0 0 0
0 0 0
Q Q Q
-------------------------
There are 2 solutions.
```