CS210A: Data Structure and Algorithms

Semester I, 2014-15, CSE, IIT Kanpur

Programming Assignment 3

N Queens Problem

Usually we come across an algorithmic problem whose efficient solution is based on some data structure. But, this data structure is usually not obvious from the problem statement. However, if we explore the problem carefully, it may guide us to the *right* data structure. The aim of this assignment is to make you experience this fact.

We shall solve a problem which is commonly called n **Queens Problem:** We are given an $n \times n$ chess board. The aim is to place n queens on this chessboard such that no two queens attack each other. A valid configuration thus requires that no two queens must share same row, column or diagonal. There may be many valid configurations. The aim is to **generate all such valid configurations**. Notice that no valid configuration exists for $n = 2$ and $n = 3$, therefore you can assume that $n > 3$.

A valid configuration for $n = 4$ is given below:

		Q	
Q			
			Q
	Q		

Figure 1: 4 queens in non-attacking positions on a 4×4 chess board.

Let us proceed to solve this problem. There are n queens to be placed on $n \times n$ chess board. So each row will have exactly one queen. Similarly, each column will have exactly one queen. So it can be observed that all (valid/invalid) configurations of n queens are $n!$ permutations.

First solution: We may enumerate all configurations from $(1, 1, 1, 1, 1, 1, 1, 1)$ to $(8, 8, 8, 8, 8, 8, 8, 8)$. For each configuration that we enumerate, we may verify whether it is a valid one or invalid one. In this way we may output all valid solutions.

Is there a better way to find all valid solutions ?

A better solution: Imagine the enumeration of all possible valid/invalid configurations in the first solution. Upon placing $i(< n)$ queens on the first i rows, it might not be possible to place the rest of $n - i$ queens to arrive at any valid configuration. So there is no point enumerating those $(n - i)!$ configurations. It would be very helpful if we could detect such bad partial-configurations early enough. This leads to the following question. (Think over it before proceeding further).

Question 1. *How to materialize the idea of "detecting bad partial-configurations early enough" ?*

To materialize the above idea, we may use a $n \times n$ matrix M representing a chess board. Whenever we place a queen, we mark with red color all those cells which are in attacking position by this queen. For example, when we place 1st queen at $(1, 1)$ position, we can mark entire first row, first column and the whole diagonal from $(1, 1)$ to (n, n). When we are going to place 2nd queen, we start traversing the second row from left to right. We realize from the matrix M that first as well as second cells of the second row are marked red. So we place the queen in the 3rd column. In this way, we avoided generating all those configurations of the form $(1, 2, *, *, *, *, *, *)$. In general, when we are going to place $(i + 1)$th queen, we explore $(i + 1)$th row from left to right in search for a safe cell and place the queen in leftmost safe cell found. If you explore this approach in more details, the following question will also arise. Think over this question before proceeding further.

Question 2. *Suppose upon placing i queens, we realize that entire $(i + 1)$th row is marked red, so what should our algorithm do ?*

There are two ways to address the above question.

1. **Recursive algorithm :**

   We may design recursive algorithm for this problem. For a given fixed position of first i queens, the recursive algorithm will enumerate all valid configurations with these positions of first i queens. In case, during the recursive call, we can't place $(i + 1)$th queen in any column of $(i + 1)$th row, the recursive call will terminate immediately. On the other hand, if we can place $(i+1)$th queen safely in some cell of $(i+1)$th row, we do the following. We mark with red color all cells of the row, column, and the two diagonals associated with the cell of $(i + 1)$th queen and proceed recursively for placing $(i + 2)$th queen.

   Note: Note that if we try to alter position of a queen, we need to undo the color-changes we did on the matrix M when we placed the queen at the current position. Realize that keeping colors does not achieve the desired objective. So think of storing suitable *numbers* instead of colors in the matrix so that you may undo the changes effectively. This note is meant for the following iterative algorithm as well.

2. **Iterative algorithm and a suitable data structure:**

   We may take an iterative approach using a suitable data structure. We may keep the history of all queens placed till now in this data structure. One thing you would have noticed from the discussion above that we are enumerating all valid expressions in *lexicographic* order. Upon placing i queens, if we find that all cells of $(i + 1)$th row are marked red, we should realize that the current configuration of first i queens does not lead to any valid configuration. Since we are enumerating valid solutions in lexicographic order, which one of the first i queens needs to be shifted to the right (if possible) ? Correct answer to this question will help you in the search of the data structure. Think over it for some time, before moving ahead.

Probably, by now, you would have realized that stack is the right choice of the data structure for the iterative algorithm of the given problem.

Objective: There are four objectives of this assignment.

1. (this part is to be submitted on Judge)

   You will implement the recursive and iterative solutions described above for enumerating all valid configurations of placing n queens on a $n \times n$ chess board. Both the solutions will make use of $n \times n$ matrix M as a global variable as described above. For the iterative solution, you must give an array based implementation of stack. Write functions for each operations on stack. Your algorithm **must** access stack only through these functions only. In other words, you are not allowed to access the array (used for implementing stack) in any way other than through these functions.

   Details of submissions of the code on Judge will be available by 9th September.

2. (this part is not to be submitted)

   You would have realized great amount of similarity between the two implementations. To get surprised, read the following Wikipedia article on Recursion - `http://en.wikipedia.org/wiki/Recursion_(computer_science)#Recursion_versus_iteration`. You will know how recursion is implemented in any programming language.

3. (this part is to be submitted as a hard copy)

   Which of the two implementations is faster ? Submit a half/one page report containing a table that states the time taken by the two implementations for various values of n. Please avoid the time spent by the algorithms on I/O (printing the valid configurations). More specifically, just don't print whenever you arrive at a valid configuration. Instead, just print the total number of valid configurations. Execute the recursive and iterative implementations up to that value of n for which the time taken is 5 minutes.

4. (this part is not to be submitted)

   Ponder over the following questions:

   (a) What might be the reason for the difference in the time taken by the two implementations ?

   (b) Can you realize that each recursive algorithm can be transformed into an iterative algorithm with the help of .... ?

Every art is beautiful and so is the art of algorithm design ... But those who google for every question/problem are unfortunately never able to see this beauty.