CSE 101 Assignment
HW2: Place your queens with care

- All code must be written in C/C++.
- Please be careful about using built-in libraries or data structures. The assignment instructions will tell you what is acceptable, and what is not. If you have any doubts, please ask the instructors or TAs.

1 Problem description

Main objective: The n-queens problem is a classic computer science question. Recall how a queen moves in chess. It can move/attack along rows, columns, and diagonals. The usual problem is to place n queens on an $n \times n$ chessboard such that no two queens attack each other. Check out the following link if you want to read more about the problem: `https://en.wikipedia.org/wiki/Eight_queens_puzzle`

We have a twist in this assignment. We will take as input the position of many queens that are already placed on the chessboard, and your code has to the place the *remaining queens*. (Sometimes, this will not be possible.) Solve the n queens problems **without recursion**. You cannot make any recursive calls. If you make a recursive call, you will get no credit. There is no other restriction in this assignment. You can use whatever inbuilt libraries or data structures you want.

Important: the chessboard should be indexed starting from 1, in standard (x, y) coordinates. Thus, $(4, 3)$ refers to the square in the 4th column and 3rd row.

Setup: You can access a Codio unit (which I also call a Codio box) for this assignment. There is a directory "NQueens". You must write all your code in that directory, which is where the executable should be created. There are also some testing scripts and example input/output files. Please check out the README for more details on that.

Format: You should provide a Makefile. On running `make`, it should create an executable "nqueens". You should run the executable with *two* command line arguments: the first is an input file, the second is the output file. You must

provide a README with a short explanation of the usage and a description of the files involved.

All your files must be of the form *.c, *.cpp, *.h, *.hpp. When we grade, all other code files will be deleted. (So do not try to script some part in another language.)

Each line of the input file corresponds to a different instance. Each line of the input file will have three integers: the chessboard size, the column where the input queen is placed, and the row where the input queen is placed. For example, the file may look like:

8 4 4 6 3
11 4 4 6 3

The first line means we have a 8×8 chessboard, with two queens placed at $(4, 4), (6, 3)$. We wish to place 6 more queens without any attacking the other. The second line means we have a 11×11 chessboard, with two queens placed at $(4, 4), (6, 3)$. We wish to place 9 more queens without any attacks. So on and so forth.

Output: On running the command, the following should be printed in the output file. For each line of the input file, there is a line in the output file with the placement of queens.

- If there is no solution to the problem, print "No Solution" (with a newline at the end). Please capitalize exactly the same, to ensure that our grading scripts run correctly.
- If there is a solution, print the position of each queen as `<column> <space> <row> <space>` followed by the position for the next queen. The positions should be in increasing order of column, so first column first, second column second, etc. Please follow this format exactly, so that the checking script works for you.

For example, the output for the input describe above could be

No Solution
1 2 2 8 3 10 4 4 5 9 6 3 7 5 8 7 9 11 10 1 11 6

Interestingly, when you place queens as $(4, 4)$ and $(6, 3)$, there is no solution when the board has size 8. But there is a solution for a board of size 11. Note that when a solution exists, it may not be unique. It is ok to provide any solution (as long as it is correct). So you may get solutions different from the examples provided.

Suggestions for coding: Use a stack. *You can use any in-built stack that C/C++ provides, and can store the input chess pieces however you wish.* To stress, you do not need to write your own stack.

Think about the box-trace for the recursive solution for this problem. See how your stack solution will perform the same backtracking. It's not a bad idea to start from a recursive solution, and adapt it using stacks.

It is convenient to create on object that stores queens, as a pair of column and row values.

Helper code: In the Codio box, you will find some Java code that prints out your solution on a chessboard (with queen positions) directly on the console. This is extremely helpful in checking if your solution is correct. Compile and execute the java code on terminal directly using the commands below:

- `javac PrintSolutionHelper.java`
- `java PrintSolutionHelper <your output file>`

For each line in your output file that has a list of queens positions, the console will print the chessboard with queens on then. (The code is actually pretty interesting!)

2 Grading

You code should terminate within seconds for a chessboard of size at most 11. Roughly speaking, to process a file with 50 inputs (of chessboard size ≤ 11), your code should terminate within two minutes. You get no credit for a run that does not provide a proper output.

1. (10 points) For a full solution as described above.
2. (8 points) Takes too much time for boards of size 11, but works fine otherwise.
3. (6 points) Only solves the problem when the input is a single input queen.
4. (5 points) Only solves the problem when the input is a single input queen in the first column.