TRINITY COLLEGE
DEPARTMENT OF COMPUTER SCIENCE

CPSC 215: Data Structures and Algorithms
Instructor: Dr. Chandranil Chakraborttii
Spring 2025

Laboratory 7

Academic Honesty Policy

In working on programming assignments, you may discuss broad issues of interpretation and understanding and general approaches to a solution. However, conversion to a specific solution or to program code must be your own work. Programming assignments are expected to be the work of the individual student, designed, and coded by him or her alone. Violations are easy to identify and will be dealt with promptly according to the Academic Integrity and Intellectual Dishonesty outlined in the Student Handbook.

- Copying another person's programs or encouraging or assisting another person to commit plagiarism is cheating. In particular, the following activities are strictly prohibited:
- Giving and receiving help in the actual development of code or writing of an assignment.
- Looking at another person's code or showing your code to another person.
- Sharing a copy of all or part of your code regardless of whether that copy is on paper or in a computer file.
- Turning in the work of any other person(s) (former students, friends, textbook authors, people on the Internet, etc.) and representing it as your own work.
- Fabricating compilation or execution results.
- Use of AI ChatBots for any help regarding the assignment is **strictly prohibited.**

The penalty for cheating is a failing grade (F) for the course, and the student is asked to appear at an Academic Dishonesty Hearing. In addition, the College also places a record of the incident in the student's permanent record. It is your responsibility to protect your work from unauthorized access. Do not discard copies of your programs in public places. Do not leave computers unattended and copies of output lying around.

N Queens Problem Implementation using Recursion

Objective:

The objective of this assignment is to implement the N Queens problem using recursion in Java. The N Queens problem is a classic problem in computer science and involves placing N queens on an N×N chessboard in such a way that no two queens attack each other.

Input Files (Starter Code)

- NQueens.java ***(Need to change)***
- NQueensTest.java – Junit Test file ***(Need to change)***
- Tester.java - Contains the main function. ***(Completed)***

Getting Started

- Open Eclipse and create a Java Project (File -> New -> Java Project)
- Give a project name (For example: Lab_7_CPSC_215)
- Go to Project -> Right Click -> New Package
- Give a package name (For example: Lab_7_CPSC_215)
- Now to create the classes, go to package -> Right Click -> New Java class.
- Paste contents of the given files. Keep the first line in place (package “package_name”;)
- Understand the code structure before starting to write code.

Tasks:

1. **Implement Class NQueens.java** Implement the recursive solution for the N Queens problem in a class named NQueens. This class should contain the following methods:

    - **public void solveNQueens(int n):** This method initiates the process of solving the N Queens problem for a given board size n. It recursively explores all possible configurations of placing N queens on an N×N chessboard, printing each valid solution found.

    - **private void solve(int row, int n, int[] queens):** This recursive method is called internally by solveNQueens. It attempts to place queens on the board row by row, ensuring that no two queens threaten each other. It backtracks when it encounters a dead-end and continues exploring other possibilities.

    - **private boolean isValid(int row, int col, int[] queens):** This method checks whether it's valid to place a queen at a given position (row, col) on the chessboard. It verifies

whether the new queen is not threatened by any other queen already placed on the board.

- **private void printSolution(int[] queens):** This method prints a valid solution configuration of placing queens on the chessboard.

2. **Tester Class:** Review the **Tester.java** class used to demonstrate the usage of the NQueens class. In this class, we prompt the user to enter the size of the chessboard (n), then call the solveNQueens method to find and print all solutions. Also print the total number of solutions found.

Expected Output

```
(base) nilchakraborttii@Adiministrator's-MacBook-Air Solution % javac *.java
(base) nilchakraborttii@Adiministrator's-MacBook-Air Solution % java Tester
Enter the size of the chessboard (n): 3
No solutions found!
(base) nilchakraborttii@Adiministrator's-MacBook-Air Solution % java Tester
Enter the size of the chessboard (n): 4

Solution 1
. Q . .
. . . Q
Q . . .
. . Q .
Solution 2
. . Q .
Q . . .
. . . Q
. Q . .
Total Solutions = 2
```

JUnit Testing

3. **JUnitTesting Class NQueensTest.java** Complete JUnit test case to validate the correctness of the **NQueens** class. Complete the method **testNQueensSolution** named **NQueensTest** containing test methods to ensure that the **solveNQueens** method produces correct number of solutions for different input sizes.

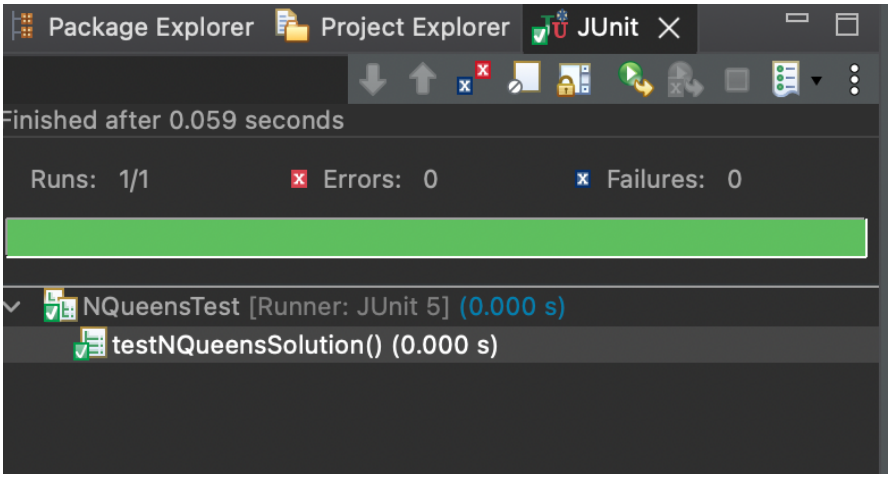
Test Steps:

- **Initialize Test Parameters:**
  - Set n to 4 for the first test case and 8 for the second test case.
- **For n =3, 4 and 8, execute Test Cases:**
  - Create an instance of the NQueens class.
  - Call the solveNQueens method with the specified input size n.

    - Use the getSolutionsCount method to retrieve the number of solutions found.
    - Assert that the number of solutions matches the expected count for each test case.
- **Validate Results:**
    - For the test case with 3 queens, assert that the number of solutions found should be 0.
    - For the test case with 4 queens, assert that the number of solutions found should be 2.
    - For the test case with 8 queens, assert that the number of solutions should be is 92.

Expected Outcome:

- If the solveNQueens method is implemented correctly, it should produce the expected number of solutions for each test case.
- The test case should pass without any assertion failures if the method behaves as expected.



How to approach solving the N-Queens problem:

To approach solving the N-Queens problem step by step using the provided code, follow these steps:

- **Understand the Problem**:
    - The N-Queens problem involves placing N chess queens on an N×N chessboard so that no two queens threaten each other. This means that no two queens should share the same row, column, or diagonal.

- **Understand the Solution Approach**:
    - The code uses a backtracking algorithm to solve the N-Queens problem recursively.
    - The main idea is to place queens on the chessboard row by row, ensuring that each placement is valid according to the rules of the game.
    - If a valid solution is found, it is printed, and the algorithm continues to search for more solutions.
    - The solveNQueens method initiates the solving process by creating an array queens to store the column index of each queen in each row, then calls the solve method to recursively find solutions starting from the first row.
    - The solve method checks each column in the current row to see if placing a queen there is valid. If it is, it updates the queen's array and recursively calls solve for the next row.
    - The isValid method checks if a queen placed at a given position conflicts with any previously placed queens.
    - The printSolution method prints a valid solution in a readable format.

- **Hints for completing the methods:**
    - **solveNQueens(int n):** This is the entry point of the algorithm. It initializes an array queens of size n to represent the positions of queens in each row. It then calls the solve method to start the backtracking process from the first row.
    - **solve(int row, int n, int[] queens):** This method performs the backtracking search. It iterates through each column in the current row and tries to place a queen if it's a valid position. If a valid position is found, it updates the queens array and recursively calls solve for the next row. If no valid position is found, it backtracks.
    - **isValid(int row, int col, int[] queens):** This method checks whether placing a queen at position (row, col) is valid. It ensures that no two queens threaten each other by checking if there's a queen in the same column or diagonal.
    - **printSolution(int[] queens):** This method prints the current solution represented by the positions of queens in the queens array.
    - **getSolutionsCount():** This method returns the total number of solutions found.

- **Running the Program**:
    - Compile and run the Tester class.
    - Enter the size of the chessboard (N) when prompted.
    - The program will print all valid solutions for the N-Queens problem on an N×N chessboard.

- **Understanding the Output**:
    - The program will print each valid solution, representing the positions of the queens on the chessboard with 'Q' and empty squares with '.'.
    - After printing all solutions, the total number of solutions found will be displayed.

- **Analyzing Performance**:
    - The time complexity of the backtracking algorithm used in this solution is exponential, typically O(N!), where N is the size of the chessboard.

- As the size of the chessboard increases, the number of solutions and the time taken to find them will also increase rapidly.
- For larger values of N, the program may take a considerable amount of time to find and print all solutions.

Additional hints: Step-by-step description (NQueens.java):

- **solveNQueens(int n):**
  - Begin the process of solving the N Queens problem for a given board size n.
  - Initialize a counter to keep track of the number of solutions found.
  - Create an array queens of size n to represent the board configuration.
  - Call the solve method with initial parameters to start the recursive exploration of solutions.

- **solve(int row, int n, int[] queens):**
  - Base case: If row equals n, a solution has been found.
    - Increment the solutions counter.
    - Print the current solution.
    - Return to explore other possible configurations.
  - For each column col in the current row:
  - Check if it's valid to place a queen at position (row, col).
  - If valid, update the queens array and recursively call solve for the next row.
  - After exploring all possibilities for the current row, backtrack and try the next column.

- **isValid(int row, int col, int[] queens):**
  - Iterate through each previously placed queen up to row - 1.
    - Check if the current queen threatens any other queen already placed.
    - Check if the current column matches any previously placed queen's column.
    - Check if the absolute difference between the current queen's column and any previously placed queen's column equals the row difference.
  - If no threats are found, return true; otherwise, return false.

- **printSolution(int[] queens):**
  - Iterate through the queens array to print the board configuration.
    - For each row, print a Q if there's a queen at that position, and . otherwise.

———————————————————————————————————

Submission Guidelines:

- Make sure you completed all sections with “To do” statements.
- Make sure you have added JavaDoc in your code for each class.
- Upon the completion of your lab, upload your files to the Moodle course website as a single Zip file.
- See your TA and get your Assignment graded before leaving the lab. Otherwise, inform the instructor.

———————————————————————————————————