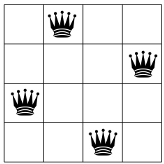
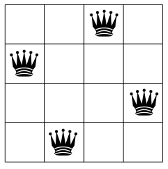
**Problem 8.12:** N Queens - Write an algorithm to print all thr ways of arranging N queens on an N × N chess board so that no two queens share the same row, column or diagonal.

Example Solutions for a 4 × 4 board (2 distinct solutions):





**Part 2: High Level Design Overview:**

|  |
| --- |
| **Method Descriptions:**   1. The board is constructed using a double char array. If there is a queen at a particular location, that location’s char value is set to ‘Q’. Otherwise the location’s char value is set an empty space. 2. There is a location object that specifies a location on the board with an integer variable for row and an integer variable for column. It also contains a “to string” method that makes it really easy to print locations. In addition, it’s easier to have a list of locations than some list with a bunch of rows and columns and you have to map them some way to eachother. 3. There is the operation “print board” which takes in a board and prints eat in a ways that is easy to read. 4. There are the operations “set queen” and “clear queen” which can either set or remove a queen at the specified location. 5. The method “diagonal Occupied” takes in a location and a board, and checks if the location has any queens on its diagonals. The diagonals are up-right, up-left, down-right, down-left. 6. The method “get available locations” takes in a board and returns a list of all the locations that do not have a queen in the same row, column or diagonal. 7. The “count ways” method actually solves the problem and computes all the possible arrangements. 8. The main method just calls count ways to make it easy to test. |
| **public** **class** NQueens {  **public** **static** **class** Location {  **int** row;  **int** column;  **public** Location(**int** row, **int** column){  **this**.row = row;  **this**.column = column;  }    @Override  **public** String toString(){  **return** ("[" + **this**.row + "," + **this**.column + "] ");  }  }  **public** **static** **void** printBoard(**char**[][] currentBoard){}  **public** **static** **void** setQueen(**char**[][] board, Location location){}  **public** **static** **void** clearQueen(**char**[][] board, Location location){}  **public** **static** **boolean** diagonalOccupied(**char**[][] board, **int** row, **int** column){}  **public** **static** List<Location> getAvailableLocations(**char**[][] board){}  **public** **static** **int** countWays(**char**[][] currentBoard, **int** currentDecision){}  **public** **static** **void** main(String[] args) {}  } |

**Part 3: Low Level Implementation of Trivial Methods:**

|  |
| --- |
| **public** **static** **void** printBoard(**char**[][] currentBoard){  System.*out*.println("State of Board: ");  **for**(**int** i = 0; i < *DIMENSIONS*; i++){  System.*out*.print(" | ");  **for**(**int** j = 0; j < *DIMENSIONS*; j++){  System.*out*.print(currentBoard[i][j] + " | ");  }  System.*out*.println();  }  System.*out*.println("\n");  }  **public** **static** **void** setQueen(**char**[][] board, Location location){  board[location.row][location.column] = 'Q';  }  **public** **static** **void** clearQueen(**char**[][] board, Location location){  board[location.row][location.column] = ' ';  } |

**Part 4: Low Level of Implementation for Checking Diagonals:**

|  |
| --- |
| **Description:** You initialize a Boolean variable “diagonal occupied” to false. You check if there is a queen in the up-left, up-right, down-left, down-right positions respectively. If any of them have a queen, you set the Boolean variable to true. Also, you do some out of bounds checks to make sure you don’t get an array out of bounds exception. These checks take place in the if-conditions. |
| **public** **static** **boolean** diagonalOccupied(**char**[][] board, Location location){  **boolean** diagonalOccupied = **false**;  **if**(location.row != 0 && location.column != 0 && (board[location.row - 1][location.column - 1] == 'Q')){  diagonalOccupied = **true**;  }  **if**(location.row != 0 && location.column != *DIMENSIONS* - 1 && (board[location.row - 1][location.column + 1] == 'Q')){  diagonalOccupied = **true**;  }  **if**(location.row != *DIMENSIONS* - 1 && location.column != 0 && (board[location.row + 1][location.column - 1] == 'Q')){  diagonalOccupied = **true**;  }  **if**(location.row != *DIMENSIONS* - 1 && location.column != *DIMENSIONS* - 1 && (board[location.row + 1][location.column + 1] == 'Q')){  diagonalOccupied = **true**;  }  **return** diagonalOccupied;  } |

**Part 5: Low Level of Implementation for Getting List of Available Locations:**

|  |
| --- |
| **Description:**   1. Have two Boolean arrays for rows occupied and columns occupied. These are set to true if the row at the index has a queen. 2. Iterate around the board. If there is a queen, set its row and column to be occupied in the Boolean arrays. 3. Now iterate again over each position in the board. For each location, check if its row, column or any diagonals are occupied. If they are not occupied, add the location to a list. 4. Return the list of available locations. |
| **public** **static** List<Location> getAvailableLocations(**char**[][] board){  List<Location> availableLocations = **new** ArrayList<Location>();  **boolean**[] rowOccupied = **new** **boolean**[*DIMENSIONS*];  **boolean**[] columnOccupied = **new** **boolean**[*DIMENSIONS*];  //set which rows and columns are occupied  **for**(**int** i = 0; i < *DIMENSIONS*; i++){  **for**(**int** j = 0; j < *DIMENSIONS*; j++){  **if**(board[i][j] == 'Q'){  rowOccupied[i] = **true**;  columnOccupied[j] = **true**;  }  }  }  //For each location that's row,column and diagonals are free,add to list  **for**(**int** i = 0; i < *DIMENSIONS*; i++){  **for**(**int** j = 0; j < *DIMENSIONS*; j++){  Location location = **new** Location(i,j);  **if**(!rowOccupied[i] && !columnOccupied[j] && !*diagonalOccupied*(board, location)){  availableLocations.add(location);  }  }  }  **return** availableLocations;  } |

**Part 6: Count Ways:**

|  |
| --- |
| **Description:** You have to place N queens on the board. So you are making N consecutive decisions. Each decision, you a list of available locations. So you iterate around all your possible decisions (available locations) and for each one you make the decision, recurse and undo the decision. This is similar to the typical process for recursion problems.  **Base Case:** You get to the base case when you making your last decision. Unlike normal recursion problems, you do not return one. In this case you could possibly have zero, one or more available locations, so you return the number of available locations. If this number is zero, it means all the previous decision you made set you up to be stuck with no available moves and this whole path is useless and yield no extra ways. The for loop in the if-condition is used to print these moves to allow the user to see them.  Note: If you would like to see the moves, pass “print + location.toString” in the parameter which would pass on the sequence of locations/decisions that yield to a good solution. |
| **public** **static** **int** countWays(**char**[][] currentBoard, **int** currentDecision){  List<Location> availableLocations = *getAvailableLocations*(currentBoard);  **if**(currentDecision == *DIMENSIONS* - 1){  **for**(Location location : availableLocations){  *setQueen*(currentBoard, location);  *printBoard*(currentBoard);  *clearQueen*(currentBoard, location);  }  **return** availableLocations.size();  }  **int** ways = 0;  **for**(Location location : availableLocations){  *setQueen*(currentBoard, location);  ways += *countWays*(currentBoard, currentDecision + 1);  *clearQueen*(currentBoard, location);  }  **return** ways;  }  **public** **static** **void** main(String[] args) {  **char**[][] board = **new** **char**[*DIMENSIONS*][*DIMENSIONS*];  System.*out*.println("Start: ");  **int** ways = *countWays*(board, 0);  System.*out*.println("Total Ways: " + ways);  } |

**Part 7: Interpreting Results**

If you run the program for a 4 by 4 board, it will return and print 48. However, if you read up online, you will see that there are 2 distinct solutions for a 4 by 4 grid that are in the image on the first page. The reason is because with recursion/combination problems, order matters. In the classical “compute all permutations of string”, “ABCD” is different from “DCBA”. In this problem, it could put the queens at the same locations on the boards in the first page but all in different order which would yield different “ways” in our recursive programs. So when you account for all the different ways to put the queens on the same four positions, it is 4! or 24. That means are program would return is actually returning

Return = ( # distinct solutions ) ( N ! different permutations of the same distinct solution ). So always divide the return by N! to get the number of distinct solutions.