Home Activity -N- Queens Problem

Data Structures
By
Okikioluwa Ojo (100790236)
Johnathan Howe (100785128)
Johnathon Tannous (100707434)

a) Pseudocode

Algorithm *PrintBoard*(*solutions*[])
Input: An array ***solutions*** holding multiple array tokens which hold each queen position by column in order of row for each possible solution.

```
n ← solutions[0].length
result ← “”
for each board in solutions do
      for each queen in board do
            for column ← 0 to n - 1 do
                  if column = queen then
                        result ← result + “Q”
                  else
                        result ← result + “-”
            result ← result + “\n”
      result ← result + “\n”
print(result, n, solutions.length)
End
```

Algorithm *IsValidPosition*(*row*, *column*, *queenArray*)
Input: An integer ***row*** which represents the row position of the current queen. An integer ***column*** which represents the column position of the current queen. An array ***queenArray*** which holds the queens that are currently placed, where the index represents the queen’s row position and the value represents the queen’s column position.
Output: **True** if the current queen has no collision with other queens. **False** if the current queen collides with other queens.

```
for i ← 0 to row - 1 do
      if queenArray[i] = column then
            return False

      diff_x ← queenArray[i] - column
      diff_y ← i - row
```

```
        if abs(diff_y / diff_x) = 1 then
            return False
        else
            return True
End
```

```
Algorithm PlaceQueen(row, queenArray, n) {recursive algorithm}
Input: An integer row which represents the current queen's row. An array
queenArray which holds the queens that are currently placed, where the index
represents the queen's row position and the value represents the queen's
column position.
Output: An array of all found solutions, formatted so that each token is an
array of n length holding each queen's position where the index represents
the queen's row position and the value represents the queen's column
position.

solutions = []
if row = n then
    solutions.push(queenArray)
else
    for column ← 0 to n - 1 then
        if isValidPosition(row, column, queenArray) then
            queenArray.push(column)
            solutions = PlaceQueen(row+1, queenArray, n)
            queenArray.pop()
return solutions
End
```

```
Algorithm QueensRecursive(n)
Input: An integer n which represents the number of queens and the length of
the chessboard.
Output: An array of all possible solutions, formatted so that each token is
an array of n length holding each queen's position where the index represents
the queen's row position and the value represents the queen's column
position.

queenArray ← []
solutions ← PlaceQueen(0, queenArray, n)
return solutions
End
```

Algorithm ***QueensIterative(n)***

Input: An integer *n* which represents the number of queens and the length of the chessboard.

Output: An array of all possible solutions, formatted so that each token is an array of *n* length holding each queen's position where the index represents the queen's row position and the value represents the queen's column position.

```
solutions ← []
queenArray ← []
row ← 0
column ← 0
infinitely loop
      while column < n do
            if isValidPosition(row, column, queenArray) then
                  queenArray.push(column)
                  row ← row + 1
                  column ← 0
                  break
            else
                  column ← column + 1

      if row > n - 1 then
            solutions.push(queenArray)

      if row = 0 then
            return solutions

      if (column > n - 1) or (row > n - 1) then
            column = queenArray.pop() + 1
            row ← row - 1
End
```

b) Source Code

GitHub Source Code: github.com/okikio-school/queens-problem

```python
# Data Structures - N-Queens assignment

# By
# - Okikioluwa Ojo (100790236)
# - Johnathan Howe (100785128)
# - Johnathon Tannous (100707434)

# Print the chessboards
def PrintBoard(solutions):
    n = len(solutions[0])

    # Create chessboard from solutions
    result = ""
    for board in solutions:
        for queen in board:
            for col in range(n):
                if col == queen:
                    result += "Q"
                else:
                    result += "-"
            result += "\n"
        result += "\n"

    # Print chessboard
    print(result)

    # e.g. For a(n) 8x8 chessboard we get 92 solutions
    print("For a(n) " + str(n) + "x" + str(n) +
          " chessboard we get " + str(len(solutions)) + " solutions")


# Checks if the row and col given are valid spots to place a queen
def isValidPosition(curr_row, curr_col, queensArr):
    # Iterate through each previous queen to validate that the position for the
new queen doesn't intersect
```

```
    for i in range(curr_row):
        # If column is in the same vertical path as other queens then this
position isn't valid
        if queensArr[i] == curr_col:
            return False

        # Change in X
        diff_x = queensArr[i] - curr_col

        # Change in Y
        diff_y = i - curr_row

        # Slope of a pure diagonal line ("/" or "\") is 1 or -1.
        # Using slope formula `m = Δy / Δx = (y2 - y1) / (x2 - x1)`,
        # we can determine the slope of the line between our queen and previous
queens based on column and row position,
        # thus to find whether the queen intersects diagonally with previous
queens and if so the row and column specified aren't valid
        if abs(diff_y / diff_x) == 1:
            return False

    return True

# queens recursive n problem
def QueensRecursive(n):
    # Array of all valid queens positions
    solutions = []

    # The index of queens array represents the row, while the value represents
the column
    # e.g. queensArr[row] = col
    queensArr = []

    # Run recursive method
    def PlaceQueen(row, queensArr):
        # If row is n, it has successfully iterated through each row and thus has
created a valid solution
        if row == n:
            solutions.append(queensArr.copy())
```

```
        else:
            # Iterate through each column in an n by n chessboard
            for col in range(n):
                # Is this column a valid position to place queen? If true do so
                if isValidPosition(row, col, queensArr):
                    # Add valid position to array of queen positions
                    queensArr.append(col)

                    # Recursively place queens until a solution where row reaches
n
                    # or until no possible solutions are found from current board
placement
                    PlaceQueen(row + 1, queensArr)

                    # Backtrack regardless of whether a solution is valid or not
in order to find new solutions.
                    # This is because once a solution is valid, it is pushed to
the solutions array,
                    # so `queensArr` is no longer necessary for the solution and
should focus on finding
                    # a new solution
                    queensArr.pop()
                    # Continue iterating through columns

    PlaceQueen(0, queensArr)
    return solutions


# queens iterative n problem
def QueensIterative(n):
    # Array of all valid queens positions
    solutions = []

    # The index of queens array represents the row, while the value represents
the column (STACK)
    # e.g. queensArr[row] = col
    queensArr = []

    row = 0
```

```
    col = 0

    # We don't know how many iterations are required to find all solutions,
    # so we use an infinite while loop, which stops when all the solutions have
been found
    while True:
        # Iterate through each column in an n by n chessboard
        while col < n:
            # Is this column a valid position to place queen? If true do so
            if (isValidPosition(row, col, queensArr)):
                # Add valid position to array of queen positions
                queensArr.append(col)
                row += 1  # Go down one row
                col = 0  # Reset the column

                # Stop looping through columns if a valid queen can be placed
                break
            else:
                # Continue iterating through columns if not valid
                col += 1

        # If row > n - 1, it has successfully iterated through each row and has
found a valid solution
        # Note: (n - 1) is the largest array index allowed,
        # as an index of n (e.g. row: 8) would cause an array overflow exception
        if row > n - 1:  # e.g. row: 8
            solutions.append(queensArr.copy())

        # If row == 0 after the loop has started, then the loop has backtracked
and moved forward through all the possible solutions,
        # thus it returns the solutions in an array
        if row == 0:
            return solutions

        # If col or row go past the largest array index allowable (n - 1),
        # then they weren't able to find any valid places to place a queen
        # thus they should backtrack
        if col > n - 1 or row > n - 1:
            # Move to the previous queens' row and column position,
```

```
            # then move to the next available column for a queen
            col = queensArr.pop() + 1
            row -= 1

PrintBoard(QueensIterative(8))
PrintBoard(QueensIterative(9))

PrintBoard(QueensRecursive(8))
PrintBoard(QueensRecursive(9))
```

c) Solutions

8x8	9x9
- - - - - - - Q - - - Q - - - - Q - - - - - - - - - Q - - - - - - - - - - Q - - - Q - - - - - - - - - - - - Q - - - - - Q - - -	- - - - - - - - Q - - - - - - Q - - - - - Q - - - - - - Q - - - - - - - - - - - - - - Q - - - - - - Q - - - Q - - - - - - - - - - Q - - - - - - - - - - Q - - - -
- - - - - - Q - - - Q - - - - - - - - - - - - Q - Q - - - - - - - - - - Q - - - Q - - - - - - - - - - - - Q - - - - - Q - - - -	- - - - - - - - Q - - - - - - Q - - - - Q - - - - - - - - - - - - - Q - - Q - - - - - - - - - - - Q - - - - Q - - - - - - - - - - - - - Q - - - - - - Q - - - - -
- - - - - Q - - - - - Q - - - - - - - - - - Q - Q - - - - - - - - - - - - - - Q - Q - - - - - - - - - - Q - - - - - Q - - - - -	- - - - - - - Q - - - - - - Q - - - - - Q - - - - - - - - - - - - - - Q - - - - - - Q - - Q - - - - - - - - - - - Q - - - - - - Q - - - - - - - - - - - Q - - - -

- - - - - Q - - - - - Q - - - - Q - - - - - - - - - - - Q - - - - - - - - - - Q - Q - - - - - - - - - - - - Q - - - Q - - - - -	- - - - - - - Q - - - - - - Q - - - Q - - - - - - - - - - Q - - - - - - - - - - - - Q - - - - - - - - - - Q - - - Q - - - - - - Q - - - - - - - - - - - Q - - - -
- - - - - Q - - - - Q - - - - - - - - - - - Q - - Q - - - - - - - - - - - - - Q - - - - Q - - - Q - - - - - - - - - - Q - - - -	- - - - - - Q - - - - - - - - - - Q - Q - - - - - - - - - - - - Q - - - Q - - - - - - - - - - Q - - - - - - - - - - Q - - - - - - - - - - - Q - - - - Q - - - - -

d) Runtime Comparison

Method	Finding first solution (N=8)	Finding first solution (N=9)	Finding all solution (N=8)	Finding all solution (N=9)
Iterative	1.995200 ms	1.035000 ms	295.9931 ms	1217.0013 ms
Recursive	2.040900 ms	1.042800 ms	346.998400 ms	1246.003000 ms

e) Discussion

JIT languages like Python are better able to optimize for iterative loops, as they can predict how they will occur. But, for recursive loops it's more difficult for them to predict, so each recursion takes longer. Altogether the differences are more pronounced.