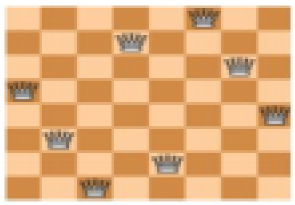
Activity #1

Data Structures

Instructor: Prof. Shahryar Rahnamayan, PEng, SMIEEE

Due Date: Friday, Feb. 5, 2021, @11:59 PM

One submission per group.



Problem statement: Solve N-Queen problem using: 1) an iterative method, and 2) recursive method for N=8 and N=9.

Your report should include: a) pseudocode for both methods, **b)** programs with comments, **c)** all solutions for N=8 and 9 for both methods, **d)** comparing running time (in ms) for the following eight cases listed in the table, **e)** discussion on the reported running times.

Algorithms/Pseudocode: -

Pseudocode for iterative program: -

Algorithm solve(8)

```
Input: column, row,start←0
Output: N-Queen Problem←new stack
while column≤size
    placed=true
    for(int row=start+1; row<=size; row++)
        If placed?=true
        place row
        column++
        for loop ends
    If !placed
        start←backtrack to the previous queen
        column--
    else
        start←0
```

Pseudocode for recursive program: -

Input: Size of chess board to be solved recursively
Output: All N-sized chess boards with solution to queen problem

```
Algorithm solve(n)
    While n = firstcolumn
        if queens = placed
            return True;
    foreach column do
        foreach row in column do
            if canPlaceQueen then
                add [row,column] to solution array solutionArray[row][column] =
                True
                solve(column + 1)
```

```
                                solutionArray[row][column] = False
                        return False;
```

Programs: -

Program for an iterative algorithm: -

```
import time

filename = None


class NQueenIterative:
   timeOfSolution: float
   size: object

   def __init__(mat, size):
       # defines the size of the chess board
       mat.size = size
       # defines the rows and columns
       # determines that the time starts from 0
       mat.timeOfSolution = 0
       # array of the chess board NxN
       mat.queenChess = []

   def isFunc(mat, a, b):
       # It goes through options in the rows and columns to decide where the
Queen should go
       for rowA, colB in enumerate(mat.queenChess):
           if abs(a - rowA) != abs(colB - b) and not (rowA == a):
               if colB != b:
                   continue
           return False
       # If the queen is placed in the same row or column or diagonal of
another queen
       # then false otherwise it is true
       return True

   # It prints the solution of the chess board NxN
   def printSolution(mat, calculate: object) -> object:
       """

       :rtype: object
       """
       # the word calculate in the string prints the number of solution
       # example; Solution #100
       print('\nSOLUTION #{0}:'.format(str(calculate)))
       filename.write('\nSOLUTION #{0}:\n'.format(str(calculate)))
```

```
        a: int
        # calculates the size/range for the rows
        for a in range(mat.size):
            row_num: object = [" / "] * mat.size
            if not a >= len(mat.queenChess):
                row_num[mat.queenChess[a]] = " Q "
                # Organizes the rows in such a matter where it won't look
clustered
            print("{0}\n".format(''.join(row_num)))
            filename.write("{0}\n".format(''.join(row_num)))

    def solve(mat, size):
        # b = col it will be print out as an integer number
        b: int = 0
        # a = row it will be print out as an integer number
        a: int = 0
        # Calculates the number of solutions starting from 1 then 2, then ...
        calculate = 1
        while True:
            # will not be placed at the ending of a column or that will not be
considered a safe spot for the Queen
            while not (b >= mat.size or mat.isFunc(a, b)):
                # calculates if the next column in the row will work or not
                b = b + 1
                # if it is not on the end of the column
            if b >= mat.size:
                pass
            else:
                # then add the column as a merger
                mat.queenChess.append(b)
                # queen's have been added to all the rows
                if a != mat.size - 1:
                    # goes on to the next row to see if that works
                    a = a + 1
                    # recalibrates from the beginning of the row
                    b = 0


                else:
                    if calculate == 1:
                        mat.timeOfSolution = time.time()

                    # prints the calculated solution
                    mat.printSolution(calculate)
                    calculate += 1
                    # the last queen that was placed is removed
                    mat.queenChess.pop()
                    # begins at the end of the column to find another solution
                    b = mat.size
                    # done going through all columns and rows. Also, the
queen's have been placed in the rows/columns
            if mat.size <= b:
                if not a == 0:
                    pass
                # starting a new chessboard solution, with different
solutions/combinations
                else:
                    # all possible solutions have been attempted,
                    # so the program stops calculating new solutions
                    return
                    # return to last calculated column + 1
                b = mat.queenChess.pop() + 1
                # go backwards one row
                a = a - 1
                # all possible solutions have been calibrated for that size of
chess board (NxN)

# asks which size chess board to calculate
chessSize = int(input("PLEASE ENTER THE SIZE OF THE CHESS BOARD: "))
# size of 3x3 and under aren't valid
if chessSize > 3:
   queen = NQueenIterative(chessSize)
   # calculating the starting time which is a decimal
   startingTime: float = time.time()
   filename = open("answers.txt", "w+")
   queen.solve(queen.size)
   filename.close()
   # calculating the starting time which is a decimal
   endingTime: float = time.time()
   # Prints the time for the first solution for whatever size that was
previously chosen
   print('Time to first: {0}'.format(str(queen.timeOfSolution -
startingTime)))
   # Prints the time for how long it took to come up with all the solutions
   # for whatever size that was previously chosen
   print('Time to all: {0}'.format(str(endingTime - startingTime)))
else:
   # If size is 3 or less than 3 run the program again and put a whole number
greater than 3
   chessSize = input('Run Again and the size of the Chess board size has to be
greater than 3.')
```

<u>Program for a recursive algorithm: -</u>

import time

```
class NQueens:
 #constructor function
 def __init__(self, size):
 # Store the puzzle (problem) size and the number of valid solutions
self.size = size #stores the size of the n*n chess board
 self.solutions = 0 #stores the number of solutions
 self.timeSpent = 0 #variable in order to calculate the time spent
 self.solve() #method to solve the number of possible solutions

 def solve(self):
 #solving the n queen puzzle and printing the number of solutions which were found
positions = [-1] * self.size
 self.put_queen(positions, 0)
 print("Found", self.solutions, "solutions.")

 def put_queen(self, positions, target_row):
 #this method try to put a queen on target_row by checking all the N possible cases
#if a valid case is found the function calls itself
 #the method calls itself until the next row until all the queens are placed on the chess board
# Base (stop) case - all N rows are occupied
 if target_row == self.size:
 self.show_full_board(positions)
 # self.show_short_board(positions)
 self.solutions += 1
 if(self.solutions == 1):
 self.timeSpent = time.time()
 else:
 # For all N columns positions try to place a queen
 for column in range(self.size):
 # Reject all invalid positions
 if self.check_place(positions, target_row, column):
 positions[target_row] = column
 self.put_queen(positions, target_row + 1)


 def check_place(self, positions, ocuppied_rows, column):
 #this method checks if a given position of the queen is under attack or not
#from any of the previously places queens
 #this particular part of the code checks the column and diagonal positions  for
```

```
i in range(ocuppied_rows):
 if positions[i] == column or \
 positions[i] - i == column - ocuppied_rows or \
 positions[i] + i == column + ocuppied_rows:

 return False
 return True

 def show_full_board(self, positions):
 #this function just the full N*N board
 for row in range(self.size):
 line = ""
 for column in range(self.size):
 if positions[row] == column:
 line += "Q "
 else:
 line += ". "
 print(line)
 print("\n")

 def show_short_board(self, positions):
 #shows the positions of the queens on the chess board in compressed form
line = ""
 for i in range(self.size):
 line += str(positions[i]) + " "
 print(line)

def main():
 #this is the main function which initializes and solves for the n queen puzzle
StartTime = time.time()
 NQ = NQueens(int(input("Enter the size of your N")))
 EndTime = time.time()
 print("time to first:" + str(NQ.timeSpent - StartTime))
 print("time to print them all:" + str(EndTime - StartTime))
print("Done!")

if __name__ == "__main__":
 main()
```
Runtime Comparison

Method	**N=8**		**N=9**	
	First Solution	**All Solutions**	**First Solution**	**All Solutions**
Iterative	0.033909559	0.102753400	0.002970457	0.376017093
Recursive	1.962002038	1.981019020	1.815871953	1.990029573

<u>Discussion</u>

In both recursive and iterative programs, 92 solutions were found for N = 8 queen problem and 352 solutions were found for N = 9 queen problem.

For the iterative method, it took 0.03390 s or 33.9 ms in order to find the first solution and it took 0.10275 s or 102.7 ms for all the solutions for N = 8

And for N = 9 it took 0.00297 s or 2.97 ms for the first solution and 0.37601 s or 376.01 ms for all the solutions.

For the recursive method, it took 1.96200 s or 1962 ms in order to find the first solution and 1.98101 s or 1981.01 ms for all the solutions for N = 8

And for N = 9 it took 1.81587 s or 1815.87 ms in order to find the first solution and 1.99002 s or 1990.02 ms for all the solutions.

It is observed that the time difference between the first solution and all solutions for recursive method is less as compared to the time difference for the iterative method.