WPP (AI104) LAB. PRACTICALS

ASSIGNMENT NO. 10

NAME:- KANADA PARAM RASIKLAL

ADMISSION NO.:- U24AI047

QUESTION 1:

Consider the 8 queen's problem, it is a 8*8 chess board where you need to place queens according to the following constraints.

a. Each row should have exactly only one queen.

b. Each column should have exactly only one queen.

c. No queens are attacking each other.

Write a program to place the queens randomly in the chess board so that all the conditions are satisfied. Find the solutions to the problem.

SOLUTION:

"""1. Consider the 8 queen's problem, it is a 8*8 chess board where you need to place queens

according to the following constraints.

a. Each row should have exactly only one queen.

b. Each column should have exactly only one queen.

c. No queens are attacking each other."""

```
import random


def is_safe(board, row, col):
    # Check the column
    for i in range(row):
        if board[i] == col or abs(board[i] - col) == abs(i - row):
            return False
    return True


def random_place_queens(n):
    while True:
        board = [-1] * n
        for row in range(n):
            possible_positions = [col for col in range(n) if is_safe(board, row, col)]
            if not possible_positions:
                break
            board[row] = random.choice(possible_positions)

        if -1 not in board:  # If all queens are placed successfully
            return board
```

```python
def print_board(board):
    n = len(board)
    for row in range(n):
        line = ["Q" if col == board[row] else "." for col in range(n)]
        print(" ".join(line))
    print()

n = 8  # Chessboard size (8x8)
solution = random_place_queens(n)
print("Random placement of queens:")
print_board(solution)
```

OUTPUT:

```
Random placement of queens:
. . Q . . . . .
. . . . Q . . .
. . . . . . . Q
. . . Q . . . .
Q . . . . . . .
. . . . . . Q .
. Q . . . . . .
. . . . . Q . .
```

Question 3:

A magic square is an N×N grid of numbers in which the entries in each row, column and main diagonal sum to the same number (equal to N(N^2+1)/2). Create a magic square for

N=4, 5, 6, 7, 8

Solution:

```
"""3. A magic square is an N×N grid of numbers in which the entries in each row, column and
main diagonal sum to the same number (equal to N(N^2+1)/2). Create a magic square for
N=4, 5, 6, 7, 8"""


import numpy as np


def generate_odd_magic_square(n):
    magic_square = np.zeros((n, n), dtype=int)
    i, j = 0, n // 2

    for num in range(1, n * n + 1):
        magic_square[i, j] = num
        i_new, j_new = (i - 1) % n, (j + 1) % n
        if magic_square[i_new, j_new] != 0:
            i += 1
        else:
            i, j = i_new, j_new
```

```
    return magic_square


def generate_doubly_even_magic_square(n):
    magic_square = np.arange(1, n * n + 1).reshape(n, n)
    for i in range(n):
        for j in range(n):
            if (i % 4 == j % 4) or (i % 4 + j % 4 == 3):
                magic_square[i, j] = n * n + 1 - magic_square[i, j]
    return magic_square


def generate_singly_even_magic_square(n):
    half_n = n // 2
    sub_square_size = half_n * half_n
    sub_square = generate_odd_magic_square(half_n)
    magic_square = np.zeros((n, n), dtype=int)

    for i in range(half_n):
        for j in range(half_n):
            magic_square[i, j] = sub_square[i, j]
            magic_square[i + half_n, j + half_n] = sub_square[i, j] +
sub_square_size
            magic_square[i + half_n, j] = sub_square[i, j] + 2 *
sub_square_size
            magic_square[i, j + half_n] = sub_square[i, j] + 3 *
sub_square_size
```

```
    k = (n - 2) // 4
    for i in range(half_n):
        for j in range(k):
            magic_square[i, j], magic_square[i + half_n, j] = (
                magic_square[i + half_n, j],
                magic_square[i, j],
            )
        for j in range(n - k, n):
            magic_square[i, j], magic_square[i + half_n, j] = (
                magic_square[i + half_n, j],
                magic_square[i, j],
            )

    for i in range(k):
        magic_square[i, k], magic_square[i + half_n, k] = (
            magic_square[i + half_n, k],
            magic_square[i, k],
        )

    return magic_square

def print_magic_square(n, magic_square):
    print(f"Magic Square for N={n}:")
```

```
    print(magic_square)
    print(f"Sum of each row/column/diagonal: {n * (n**2 + 1) // 2}")
    print()

for n in [4, 5, 6, 7, 8]:
    if n % 2 == 1:
        magic_square = generate_odd_magic_square(n)
    elif n % 4 == 0:
        magic_square = generate_doubly_even_magic_square(n)
    else:
        magic_square = generate_singly_even_magic_square(n)
    print_magic_square(n, magic_square)
```

Output:

```
Magic Square for N=8:
[[64  2  3 61 60  6  7 57]
 [ 9 55 54 12 13 51 50 16]
 [17 47 46 20 21 43 42 24]
 [40 26 27 37 36 30 31 33]
 [32 34 35 29 28 38 39 25]
 [41 23 22 44 45 19 18 48]
 [49 15 14 52 53 11 10 56]
 [ 8 58 59  5  4 62 63  1]]
Sum of each row/column/diagonal: 260

param@GREYRAT MINGW64 /c/WPP/ASSIGNMENT 10
```

```
param@GREYRAT MINGW64 /c/WPP/ASSIGNMEN
$ python -u "c:\WPP\ASSIGNMENT 10\Q3.py
Magic Square for N=4:
[[16  2  3 13]
 [ 5 11 10  8]
 [ 9  7  6 12]
 [ 4 14 15  1]]
Sum of each row/column/diagonal: 34

Magic Square for N=5:
[[17 24  1  8 15]
 [23  5  7 14 16]
 [ 4  6 13 20 22]
 [10 12 19 21  3]
 [11 18 25  2  9]]
Sum of each row/column/diagonal: 65

Magic Square for N=6:
[[26 19  6 35 28 15]
 [21  5  7 30 32 16]
 [22  9  2 31 36 11]
 [ 8  1 24 17 10 33]
 [ 3 23 25 12 14 34]
 [ 4 27 20 13 18 29]]
Sum of each row/column/diagonal: 111

Magic Square for N=7:
[[30 39 48  1 10 19 28]
 [38 47  7  9 18 27 29]
 [46  6  8 17 26 35 37]
 [ 5 14 16 25 34 36 45]
 [13 15 24 33 42 44  4]
 [21 23 32 41 43  3 12]
 [22 31 40 49  2 11 20]]
Sum of each row/column/diagonal: 175

Magic Square for N=8:
[[64  2  3 61 60  6  7 57]
 [ 9 55 54 12 13 51 50 16]
 [17 47 46 20 21 43 42 24]
```

Question 4:

Take N (N >= 10) random 2-dimensional points represented in cartesian coordinate space.

Store them in a numpy array. Convert them to polar coordinates.

Solution:

```
import numpy as np

N = 10  # You can change N to any value >= 10
cartesian_points = np.random.rand(N, 2) * 100  # Random points in range [0, 100]
print("Cartesian Points:")
print(cartesian_points)

def cartesian_to_polar(points):
    polar_points = np.zeros_like(points)
    for i, (x, y) in enumerate(points):
        r = np.sqrt(x**2 + y**2)  #Radius
        theta = np.arctan2(y, x)  #Angle in radians
        polar_points[i] = [r, theta]
    return polar_points
```

polar_points = cartesian_to_polar(cartesian_points)

print("\nPolar Coordinates:")

print(polar_points)

Output:

```
$ python Q4.py
Cartesian Points:
[[23.53186889 47.36315892]
 [64.01539917 79.42593313]
 [91.62684807  2.53204888]
 [69.42074272 71.89812752]
 [18.19695403 26.63590034]
 [52.6766231  56.29726707]
 [27.94110966 78.29816957]
 [ 3.22846342 97.24883123]
 [11.09233954 99.87720965]
 [13.82992394 80.15717741]]

Polar Coordinates:
[[5.28868384e+01 1.10968064e+00]
 [1.02012010e+02 8.92422065e-01]
 [9.16618272e+01 2.76273242e-02]
 [9.99428850e+01 8.02926828e-01]
 [3.22583373e+01 9.71452480e-01]
 [7.70986958e+01 8.18610822e-01]
 [8.31342827e+01 1.22802752e+00]
 [9.73024057e+01 1.53761055e+00]
 [1.00491278e+02 1.46018982e+00]
 [8.13415016e+01 1.39994335e+00]]
```

Question 5:

"""Write a program to make the length of each element 15 of a given Numpy array and the

```
string centred, left-justified, right-justified with paddings of _
(underscore)."""

import numpy as np

# Function to modify strings in the array
def format_strings(array):
    # Centered, Left-Justified, Right-Justified
    centred = [f"{str(item):_^15}" for item in array]
    left_justified = [f"{str(item):_<15}" for item in array]
    right_justified = [f"{str(item):_>15}" for item in array]

    return np.array(centred), np.array(left_justified),
np.array(right_justified)

# Example Input Array
input_array = np.array(["Python", "Numpy", "AI", "Code",
"Centered", "Left", "Right"])

# Formatting the strings
centred_array, left_array, right_array = format_strings(input_array)

# Printing Results
print("Original Array:")
print(input_array)
```

print("\nCentred Strings:")

print(centred_array)

print("\nLeft-Justified Strings:")

print(left_array)

print("\nRight-Justified Strings:")

print(right_array)

Output:

```
Original Array:
['Python' 'Numpy' 'AI' 'Code' 'Centered' 'Left' 'Right']

Centred Strings:
['____Python_____' '_____Numpy_____' '______AI_______' '_____Code______'
 '___Centered____' '_____Left______' '_____Right_____']

Left-Justified Strings:
['Python_________' 'Numpy__________' 'AI_____________' 'Code___________'
 'Centered_______' 'Left___________' 'Right__________']

Right-Justified Strings:
['_________Python' '__________Numpy' '_____________AI' '___________Code'
 '_______Centered' '___________Left' '__________Right']
```

Question 6:

"""The bisection method is a technique for finding solutions (roots) to equations with a single

unknown variable. Given a polynomial function f, try to find an initial interval off by random probe. Store all the updates in an Numpy array. Plot the root finding process using

the matplotlib/pyplot library."""

```
import numpy as np

import matplotlib.pyplot as plt


# Define the polynomial function

def f(x):

    return x**3 - 6*x**2 + 11*x - 6  # Example polynomial: (x - 1)(x -
2)(x - 3)


# Bisection method implementation

def bisection_method(func, a, b, tol=1e-6):

    updates = []

    if func(a) * func(b) >= 0:

        raise ValueError("The function must have opposite signs at
endpoints a and b.")


    while abs(b - a) > tol:

        c = (a + b) / 2  # Midpoint

        updates.append((a, b, c, func(c)))  # Store interval and function
value

        if func(c) == 0:  # Found the root

            break
```

```
        elif func(a) * func(c) < 0:
            b = c
        else:
            a = c

    updates.append((a, b, c, func(c)))  # Final update
    return np.array(updates)

# Plotting the process
def plot_root_finding(updates, func):
    x_vals = np.linspace(min(updates[:, 0]), max(updates[:, 1]), 500)
    y_vals = func(x_vals)

    plt.figure(figsize=(10, 6))
    plt.plot(x_vals, y_vals, label="f(x)", color="blue")
    plt.axhline(0, color="black", linestyle="--", linewidth=0.8)
    plt.title("Bisection Method Root-Finding Process")
    plt.xlabel("x")
    plt.ylabel("f(x)")

    for i, (a, b, c, _) in enumerate(updates):
        plt.plot([a, b], [0, 0], marker="o", label=f"Iteration {i+1}" if i == 0 else "")
```

```
    plt.legend()
    plt.grid()
    plt.show()

# Main program
try:
    a, b = np.random.uniform(-10, 10, 2)  # Random interval
    a, b = min(a, b), max(a, b)  # Ensure a < b
    updates = bisection_method(f, a, b)
    print("Updates (a, b, c, f(c)):")
    print(updates)

    # Plot the root-finding process
    plot_root_finding(updates, f)
except ValueError as e:
    print(e)
```