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

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()

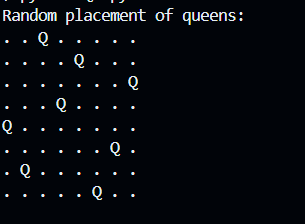
n = 8  # Chessboard size (8x8)

solution = random\_place\_queens(n)

print("Random placement of queens:")

print\_board(solution)

OUTPUT:



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:

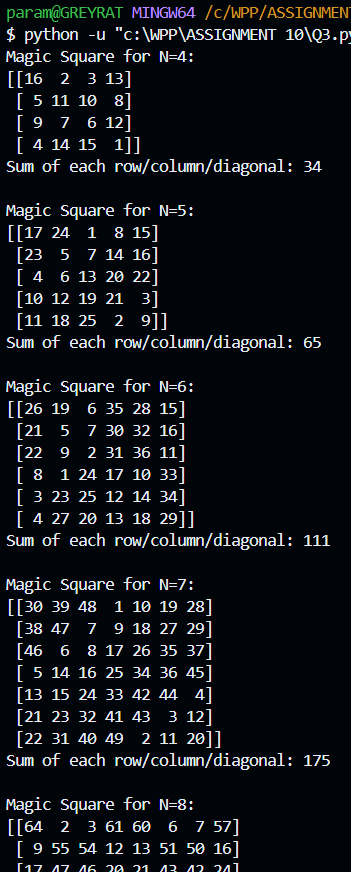
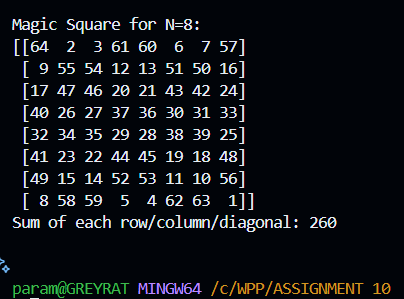
        magic\_square = generate\_doubly\_even\_magic\_square(n)

    else:

        magic\_square = generate\_singly\_even\_magic\_square(n)

    print\_magic\_square(n, magic\_square)

Output:



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]

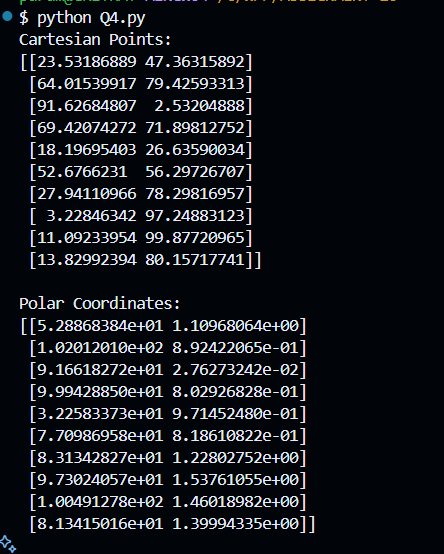
    return polar\_points

polar\_points = cartesian\_to\_polar(cartesian\_points)

print("\nPolar Coordinates:")

print(polar\_points)

Output:



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)

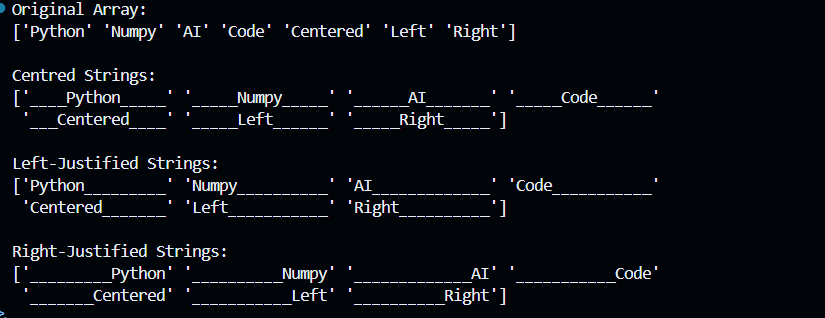
print("\nLeft-Justified Strings:")

print(left\_array)

print("\nRight-Justified Strings:")

print(right\_array)

Output:



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)