Sardar Vallabhbhai National Institute of Technology

Surat-395007

Web Programming and Python (AI104)

Assignment – 10
ROLL NO: I24AI001

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 numpy as np

# Check if it's safe to place a queen at (row, col)
def is_safe(board, row, col):
    # Check the column
    if col in board[:row]:
        return False

    # Check the diagonals
    for i in range(row):
        if abs(board[i] - col) == abs(i - row):
            return False

    return True

# Function to solve the N-Queens problem using backtracking
def solve_n_queens(board, row):
    n = len(board)

    # If all queens are placed, print the board
    if row == n:
        print_board(board)
        return True  # Stop after the first solution

    for col in range(n):
        if is_safe(board, row, col):
```

```
            board[row] = col  # Place queen at (row, col)
            if solve_n_queens(board, row + 1):  # Recur to place queens in
the next row
                return True  # Stop after the first solution
            board[row] = -1  # Backtrack if solution doesn't work

    return False

# Print the chessboard
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")

# Main function to initiate the N-Queens solution
def solve_n_queens_problem(n):
    board = np.full(n, -1)  # Initialize board with -1 (empty)
    if not solve_n_queens(board, 0):
        print(f"No solution found for {n}-Queens problem.")

# Driver code: User input for the size of the chessboard
if __name__ == "__main__":
    try:
        n = int(input("Enter the size of the chessboard (N) for the
N-Queens problem: "))
        if n < 4:
            print("There are no solutions for N < 4.")
        else:
            solve_n_queens_problem(n)
    except ValueError:
        print("Invalid input! Please enter a valid integer for the size of
the board.")
```

```
Enter the size of the chessboard (N) for the N-Queens problem: 10
Q . . . . . . . . .
. . Q . . . . . . .
. . . . . Q . . . .
. . . . . . . Q . .
. . . . . . . . . Q
. . . . Q . . . . .
. . . . . . . . Q .
. Q . . . . . . . .
. . . Q . . . . . .
. . . . . . Q . . .
```

2. 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.

```
import numpy as np
import random

# Check if the queens are attacking each other
def is_safe(board, n):
    # Check columns and diagonals
    for row in range(n):
        for prev_row in range(row):
            # Check if two queens are in the same column or on the same
diagonal
            if board[row] == board[prev_row] or \
               abs(board[row] - board[prev_row]) == abs(row - prev_row):
                return False
    return True

# Function to place queens randomly and check if the solution is valid
def solve_random_queens(n):
    while True:
        # Randomly place queens in each row
        board = np.random.permutation(n)  # Randomly shuffle column
positions for each row

        # Check if the placement is safe
        if is_safe(board, n):
```

```
            print_board(board, n)
            return board  # Return the valid board once found

# Print the chessboard
def print_board(board, n):
    for row in range(n):
        line = ['Q' if col == board[row] else '.' for col in range(n)]
        print(' '.join(line))
    print("\n")

# Driver code to solve the N-Queens problem with random placements
if __name__ == "__main__":
    try:
        n = int(input("Enter the size of the chessboard (N) for the
N-Queens problem: "))
        if n < 4:
            print("There are no solutions for N < 4.")
        else:
            solve_random_queens(n)
    except ValueError:
        print("Invalid input! Please enter a valid integer for the size of
the board.")
```

```
Enter the size of the chessboard (N) for the N-Queens problem: 10
. . . . . Q . . . .
. . . . . . . . . Q
. . . . . . Q . . .
. Q . . . . . . . .
. . . Q . . . . . .
. . . . . . . . Q .
Q . . . . . . . . .
. . . . . . . Q . .
. . . . Q . . . . .
. . Q . . . . . . .
```

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.

```python
import numpy as np

# Function to generate magic square for odd n using the Siamese method
def generate_magic_square_odd(n):
    magic_square = np.zeros((n, n), dtype=int)

    i, j = 0, n // 2

    for num in range(1, n**2 + 1):
        magic_square[i, j] = num

        new_i, new_j = (i - 1) % n, (j + 1) % n

        if magic_square[new_i, new_j] != 0:
            i += 1
        else:
            i, j = new_i, new_j

    return magic_square

# Function to generate magic square for even n (like 4, 6, 8 using
Strachey's method)
def generate_magic_square_even(n):
    magic_square = np.zeros((n, n), dtype=int)

    if n == 4:
        # Predefined for N=4, this is a known 4x4 solution
        magic_square = np.array([[1, 15, 14, 4],
                                 [12, 6, 7, 9],
                                 [8, 10, 11, 5],
                                 [13, 3, 2, 16]])
        return magic_square
    elif n == 6:
        # Strachey's method for N=6
        # Using a known pattern for 6x6 magic square
        magic_square = np.array([[35, 1, 6, 26, 19, 24],
                                 [3, 32, 7, 21, 23, 25],
                                 [31, 9, 8, 22, 27, 23],
                                 [4, 36, 2, 28, 20, 24],
                                 [29, 11, 25, 30, 5, 12],
```

```
                                [12, 20, 13, 17, 31, 18]])
        return magic_square
    elif n == 8:
        # Strachey's method for N=8
        magic_square = np.array([[1, 2, 3, 4, 5, 6, 7, 8],
                                [9, 10, 11, 12, 13, 14, 15, 16],
                                [17, 18, 19, 20, 21, 22, 23, 24],
                                [25, 26, 27, 28, 29, 30, 31, 32],
                                [33, 34, 35, 36, 37, 38, 39, 40],
                                [41, 42, 43, 44, 45, 46, 47, 48],
                                [49, 50, 51, 52, 53, 54, 55, 56],
                                [57, 58, 59, 60, 61, 62, 63, 64]])
        return magic_square
    else:
        raise ValueError(f"Magic square for N={n} is not currently
implemented.")

def generate_magic_square(n):
    if n % 2 != 0:  # For odd n, use Siamese method
        return generate_magic_square_odd(n)
    elif n % 2 == 0:  # For even n, use Strachey's method
        return generate_magic_square_even(n)

def print_magic_square(magic_square):
    for row in magic_square:
        print("\t".join(map(str, row)))

for n in [4, 5, 6, 7, 8]:
    print(f"\nMagic Square for N={n}:")
    magic_square = generate_magic_square(n)
    print_magic_square(magic_square)
```

```
Magic Square for N=4:
1       15      14      4
12      6       7       9
8       10      11      5
13      3       2       16

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

Magic Square for N=6:
35      1       6       26      19      24
3       32      7       21      23      25
31      9       8       22      27      23
4       36      2       28      20      24
29      11      25      30      5       12
12      20      13      17      31      18

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

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

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.

```
import numpy as np

def cartesian_to_polar(N):
    x = np.random.rand(N)
    y = np.random.rand(N)

    r = np.sqrt(x**2 + y**2)
    theta = np.arctan2(y, x)

    polar_coords = np.column_stack((r, theta))

    return polar_coords

N = int(input("Enter the no: "))
polar_coords = cartesian_to_polar(N)

print("Cartesian to Polar Coordinates:")
print("Radius (r) and Angle (theta) in radians:")
print(polar_coords)
```

```
Enter the no: 5
Cartesian to Polar Coordinates:
Radius (r) and Angle (theta) in radians:
[[0.98058463 0.01688263]
 [1.14606435 0.85251639]
 [0.40445487 1.01978411]
 [0.10862643 0.04900499]
 [0.91801351 1.50886761]]
```

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

def adjust_strings(array):
    left_justified = []
    centered = []
```

```python
    right_justified = []

    for string in array:
        left_justified.append(string.ljust(15, '_'))

        centered.append(string.center(15, '_'))

        right_justified.append(string.rjust(15, '_'))

    left_justified_array = np.array(left_justified)
    centered_array = np.array(centered)
    right_justified_array = np.array(right_justified)

    return left_justified_array, centered_array, right_justified_array

N = int(input("Enter the number of strings: "))
strings = []

for i in range(N):
    user_input = input(f"Enter string {i+1}: ")
    strings.append(user_input)

strings_array = np.array(strings)

left_justified, centered, right_justified = adjust_strings(strings_array)

print("\nLeft-Justified:")
print(left_justified)

print("\nCentered:")
print(centered)

print("\nRight-Justified:")
print(right_justified)
```

```
Enter the number of strings: 4
Enter string 1: i
Enter string 2: am
Enter string 3: akshaya
Enter string 4: balagopal

Left-Justified:
['i______________' 'am_____________' 'akshaya________' 'balagopal______']

Centered:
['_______i_______' '_______am______' '____akshaya____' '___balagopal___']

Right-Justified:
['______________i' '_____________am' '________akshaya' '______balagopal']
```

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 random
import matplotlib.pyplot as plt

#polynomial
def f(x, coefficients):
    return sum(c * x**i for i, c in enumerate(coefficients))

# Check if the function has opposite signs at two points, implying a root
exists between them
def has_opposite_signs(f, a, b, coefficients):
    return f(a, coefficients) * f(b, coefficients) < 0

# Bisection method
def bisection_method(f, a, b, coefficients, tolerance=1e-6,
max_iterations=100):
    updates = []  # To store midpoints for plotting
    iterations = 0
    root = None  # This will store the final root value
```

```
    # Continue while the interval is large enough and we haven't reached
the max iterations
    while (b - a) / 2 > tolerance and iterations < max_iterations:
        midpoint = (a + b) / 2
        updates.append(midpoint)

        # Check if midpoint is the root
        if f(midpoint, coefficients) == 0:
            root = midpoint
            break

        # Narrow down the interval based on the sign of the function at
the midpoint
        if has_opposite_signs(f, a, midpoint, coefficients):
            b = midpoint
        else:
            a = midpoint

        iterations += 1

    # If we exit the loop without finding the exact root, use the last
midpoint as the root
    if root is None:
        root = (a + b) / 2

    return updates, root

# Randomly find an initial interval [a, b] such that f(a) * f(b) < 0
def find_initial_interval(f, coefficients, lower=-10, upper=10):
    a = random.uniform(lower, upper)
    b = random.uniform(lower, upper)
    # Keep generating until f(a) and f(b) have opposite signs
    while f(a, coefficients) * f(b, coefficients) > 0:
        a = random.uniform(lower, upper)
        b = random.uniform(lower, upper)
    return a, b

if __name__ == "__main__":
    try:
```

```
        coefficients = list(map(float, input("Enter the coefficients of 
the polynomial (e.g., 1 -4 2 -1 for x^3 - 4x^2 + 2x - 1): ").split()))
        tolerance = float(input("Enter the tolerance for the root (e.g., 
1e-6): "))

        # Find a random initial interval
        a, b = find_initial_interval(f, coefficients)
        print(f"Initial interval: [{a}, {b}]")

        # Apply the bisection method
        updates, root = bisection_method(f, a, b, coefficients, tolerance)

        # Print the root and number of iterations
        print(f"Root found: {root}")
        print(f"Number of iterations: {len(updates)}")

        # Plot the bisection process
        x_vals = np.linspace(a - 2, b + 2, 400)
        y_vals = f(x_vals, coefficients)
        plt.plot(x_vals, y_vals, label="f(x)", color="blue")
        plt.axhline(0, color='black',linewidth=1)

        # Plot the midpoints found by the bisection method
        updates = np.array(updates)
        plt.scatter(updates, f(updates, coefficients), color="red", 
zorder=5, label="Midpoints")

        # Add labels and a title
        plt.title("Bisection Method Root Finding Process")
        plt.xlabel("x")
        plt.ylabel("f(x)")
        plt.legend()
        plt.grid(True)
        plt.show()

    except ValueError:
        print("Invalid input. Please enter valid numbers for coefficients 
and tolerance.")
```

```
Enter the coefficients of the polynomial (e.g., 1 -4 2 -1 for x^3 - 4x^2 + 2x - 1): 1 -4 2 -1
Enter the tolerance for the root (e.g., 1e-6): 1e-6
Initial interval: [6.40863889535882, -2.0105355368051274]
Root found: 2.199051679276846
Number of iterations: 0
```