U24AI044

NISHCHAY DEEPAK MITTAL

WPP ASSIGNMENT 10

Q1 & 2: Consider the 8 queen's problem, it is a 8*8 chess board where you need to place queens according to the following constraints.

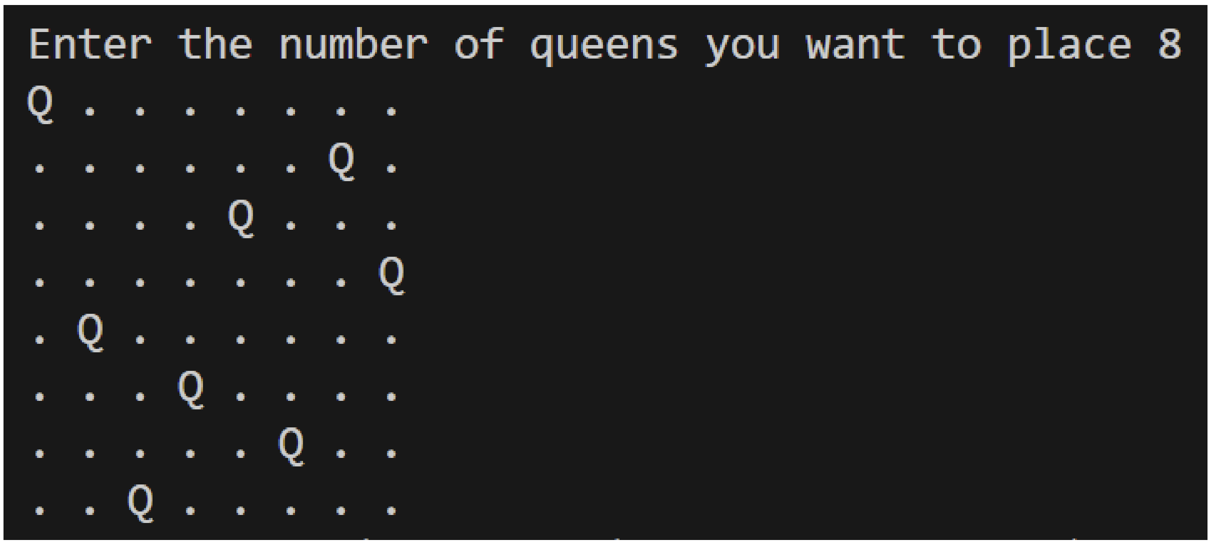
a. Each row should have exactly only one queen.

b. Each column should have exactly only one queen.

c. No queens are attacking each other.

```
import numpy as np

def is_valid(board, row, col, n):
    for i in range(col):
        if board[i] == row or \
           board[i] - i == row - col or \
           board[i] + i == row + col:
            return False
    return True

def solve_queen(board, col, n):
    if col == n:
        return True

    for row in range(n):
        if is_valid(board, row, col, n):
            board[col] = row
            if solve_queen(board, col + 1, n):
                return True
            board[col] = -1
    return False

def print_board(board, n):
    board_array = np.full((n, n), '.', dtype=str)
    for col in range(n):
        board_array[board[col], col] = 'Q'
    for row in board_array:
        print(' '.join(row))

def solve_n_queens(n):
    board = np.full(n, -1)
    if solve_queen(board, 0, n):
        print_board(board, n)
    else:
        print('No solution found')

n = int(input("Enter the number of queens you want to place "))
solve_n_queens(n)
```

OUTPUT



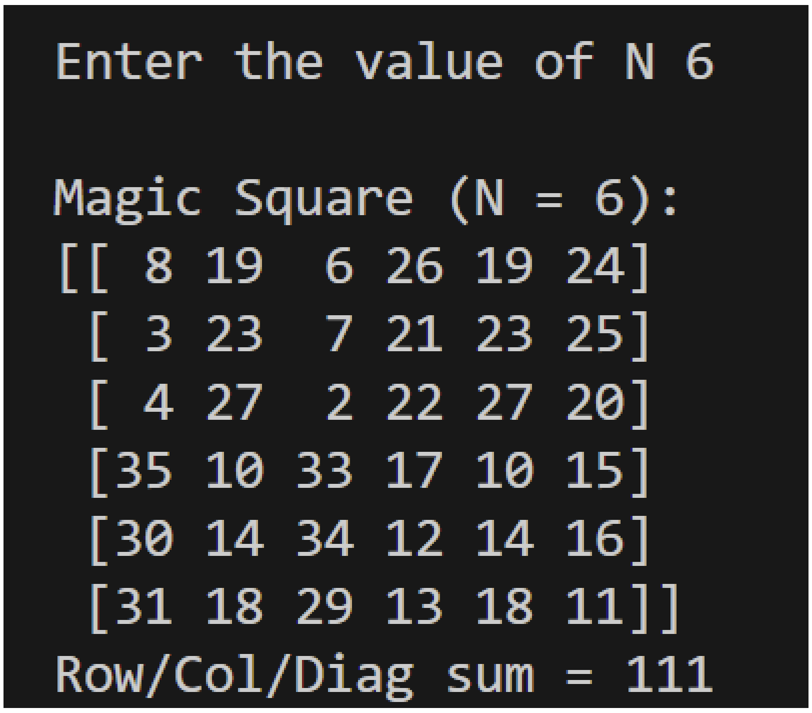
```
Enter the number of queens you want to place 8
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .
```

Q3: 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_magic_square(n):
    if n % 2 == 1:
        # Siamese method (odd n)
```

```python
        magic = np.zeros((n, n), dtype=int)
        i, j = 0, n // 2
        for num in range(1, n * n + 1):
            magic[i, j] = num
            i_new, j_new = (i - 1) % n, (j + 1) % n
            if magic[i_new, j_new]:
                i += 1
            else:
                i, j = i_new, j_new
        return magic

    elif n % 4 == 0:
        # Doubly even order
        magic = np.arange(1, n*n+1).reshape(n, n)
        mask = np.ones((n, n), dtype=bool)

        for i in range(n):
            for j in range(n):
                if ((i % 4 == j % 4) or (i % 4 + j % 4 == 3)):
                    mask[i, j] = False
        magic[mask] = (n*n + 1) - magic[mask]
        return magic

    elif n % 2 == 0:
        # Singly even (like 6, 10, etc.)
        half = n // 2
        mini_square = generate_magic_square(half)
        magic = np.zeros((n, n), dtype=int)
        add = [0, 2, 3, 1]
        for i in range(2):
            for j in range(2):
                magic[i*half:(i+1)*half, j*half:(j+1)*half] = (
                    mini_square + add[i*2 + j] * (half * half)
                )
        # Swap columns between top-left and bottom-left
        k = half // 2
        for i in range(half):
            for j in range(k):
                if j == k // 2:
                    continue
                magic[i, j], magic[i + half, j] = magic[i + half, j],
magic[i, j]

        for i in range(half):
            for j in range(n - k + 1, n):
                magic[i, j], magic[i + half, j] = magic[i + half, j],
magic[i, j]

        # Center column swap
        magic[:, [k]] = magic[:, [k + half]]
        magic[:, [k + half]] = magic[:, [k]]
        return magic

# Generate magic squares for N = 4, 5, 6, 7, 8
n = int(input("Enter the value of N "))
print(f"\nMagic Square (N = {n}):")
square = generate_magic_square(n)
print(square)
print("Row/Col/Diag sum =", n * (n**2 + 1) // 2)
```

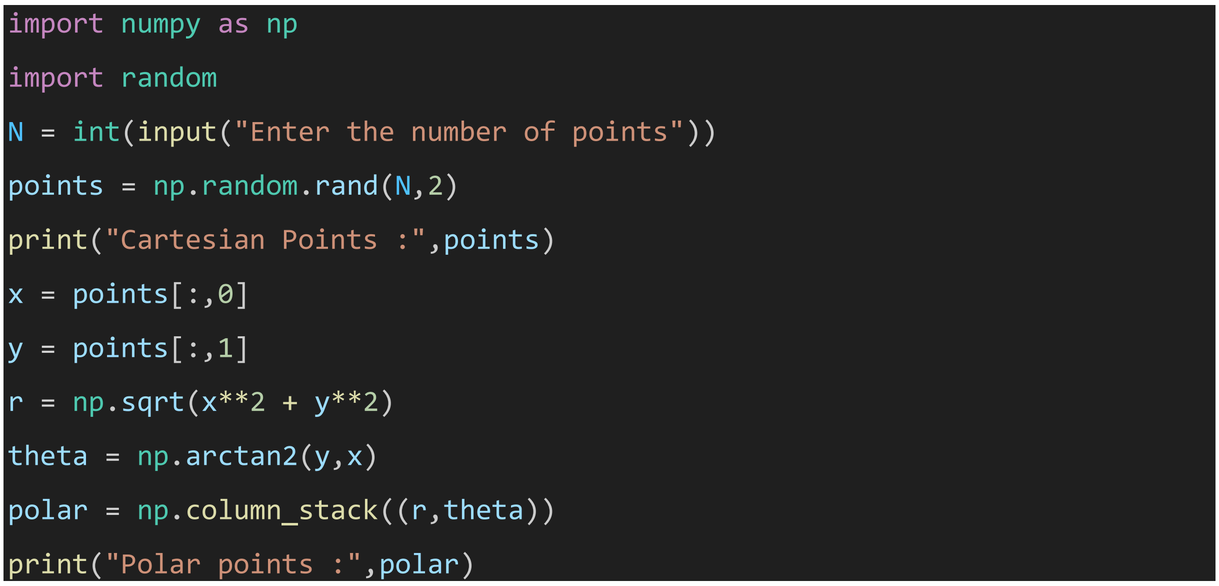
OUTPUT



```
Enter the value of N 6

Magic Square (N = 6):
[[ 8 19  6 26 19 24]
 [ 3 23  7 21 23 25]
 [ 4 27  2 22 27 20]
 [35 10 33 17 10 15]
 [30 14 34 12 14 16]
 [31 18 29 13 18 11]]
Row/Col/Diag sum = 111
```

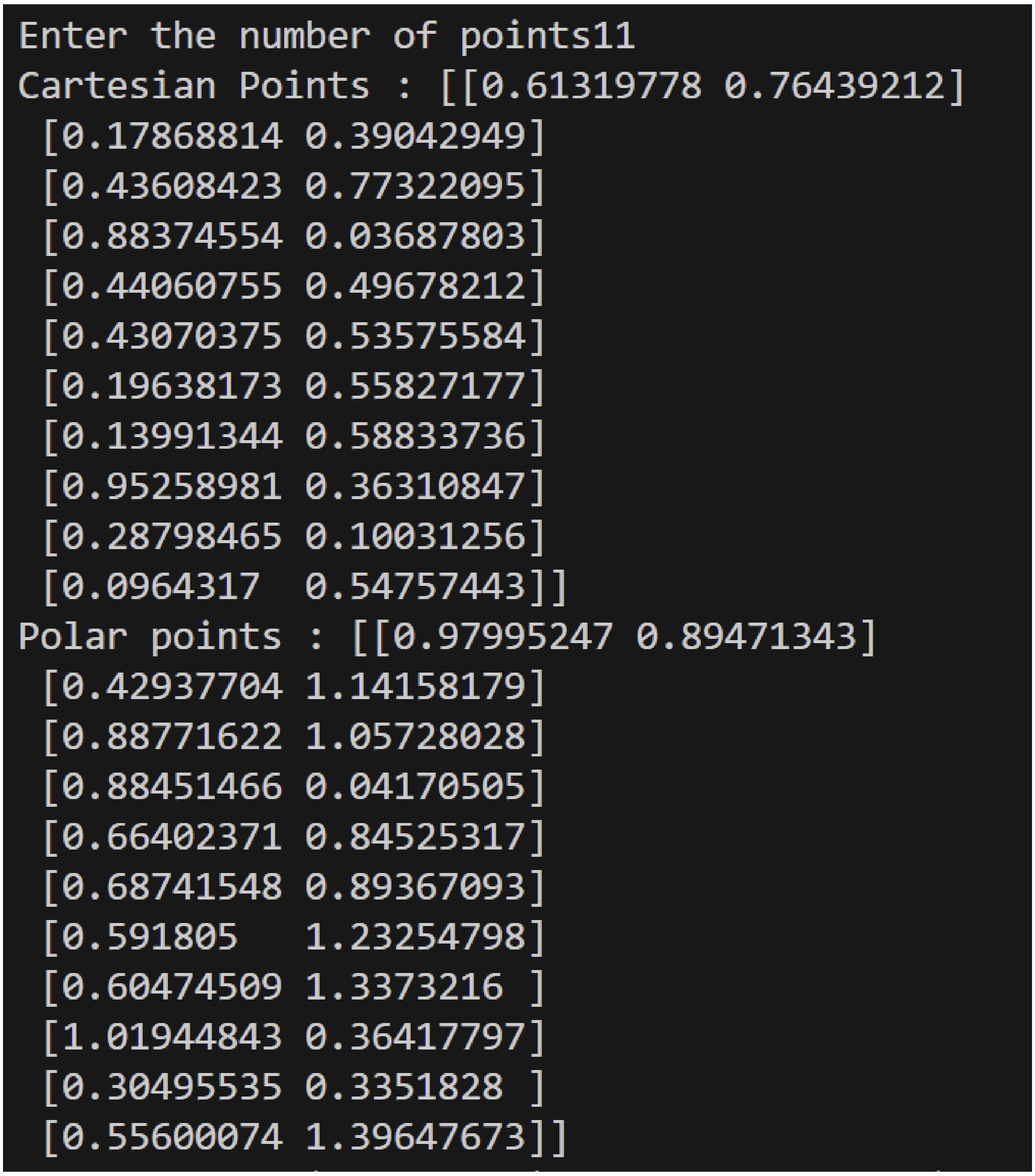
Q4: Take N (N >= 10) random 2-dimensional points represented in cartesian coordinate space.

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



```
import numpy as np
import random
N = int(input("Enter the number of points"))
points = np.random.rand(N,2)
print("Cartesian Points :",points)
x = points[:,0]
y = points[:,1]
r = np.sqrt(x**2 + y**2)
theta = np.arctan2(y,x)
polar = np.column_stack((r,theta))
print("Polar points :",polar)
```

OUTPUT



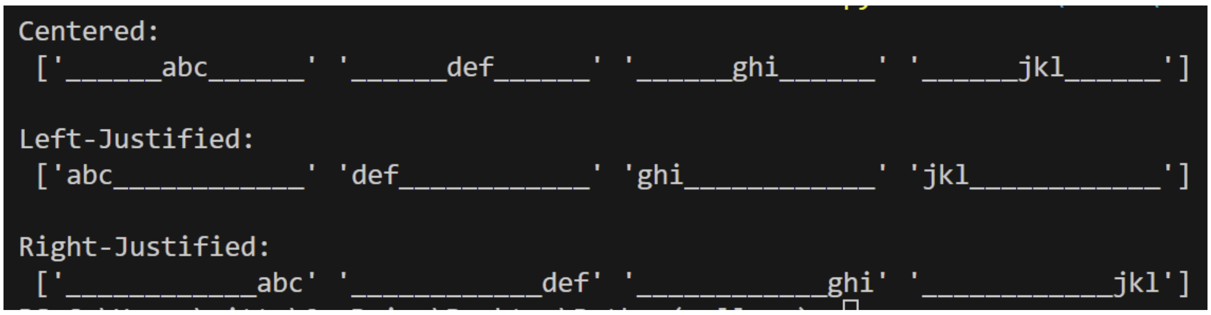
```
Enter the number of points11
Cartesian Points : [[0.61319778 0.76439212]
 [0.17868814 0.39042949]
 [0.43608423 0.77322095]
 [0.88374554 0.03687803]
 [0.44060755 0.49678212]
 [0.43070375 0.53575584]
 [0.19638173 0.55827177]
 [0.13991344 0.58833736]
 [0.95258981 0.36310847]
 [0.28798465 0.10031256]
 [0.0964317  0.54757443]]
Polar points : [[0.97995247 0.89471343]
 [0.42937704 1.14158179]
 [0.88771622 1.05728028]
 [0.88451466 0.04170505]
 [0.66402371 0.84525317]
 [0.68741548 0.89367093]
 [0.591805   1.23254798]
 [0.60474509 1.3373216 ]
 [1.01944843 0.36417797]
 [0.30495535 0.3351828 ]
 [0.55600074 1.39647673]]
```

Q5: 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



arr = np.array(['abc', 'def', 'ghi', 'jkl'])



width = 15
```

```python
# Centered
centered = np.array([s.center(width, '_') for s in arr])
print("Centered:\n", centered)

# Left-justified
left_justified = np.array([s.ljust(width, '_') for s in arr])
print("\nLeft-Justified:\n", left_justified)

# Right-justified
right_justified = np.array([s.rjust(width, '_') for s in arr])
print("\nRight-Justified:\n", right_justified)
```

OUTPUT



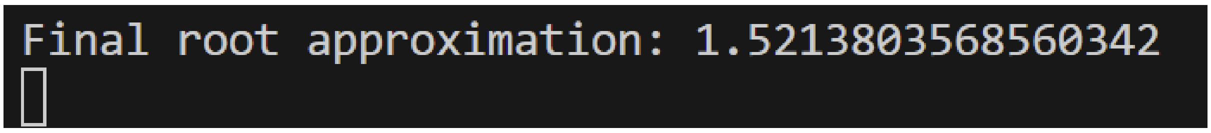
```
Centered:
 ['______abc______' '______def______' '______ghi______' '______jkl______']

Left-Justified:
 ['abc____________' 'def____________' 'ghi____________' 'jkl____________']

Right-Justified:
 ['____________abc' '____________def' '____________ghi' '____________jkl']
```

Q6: 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.

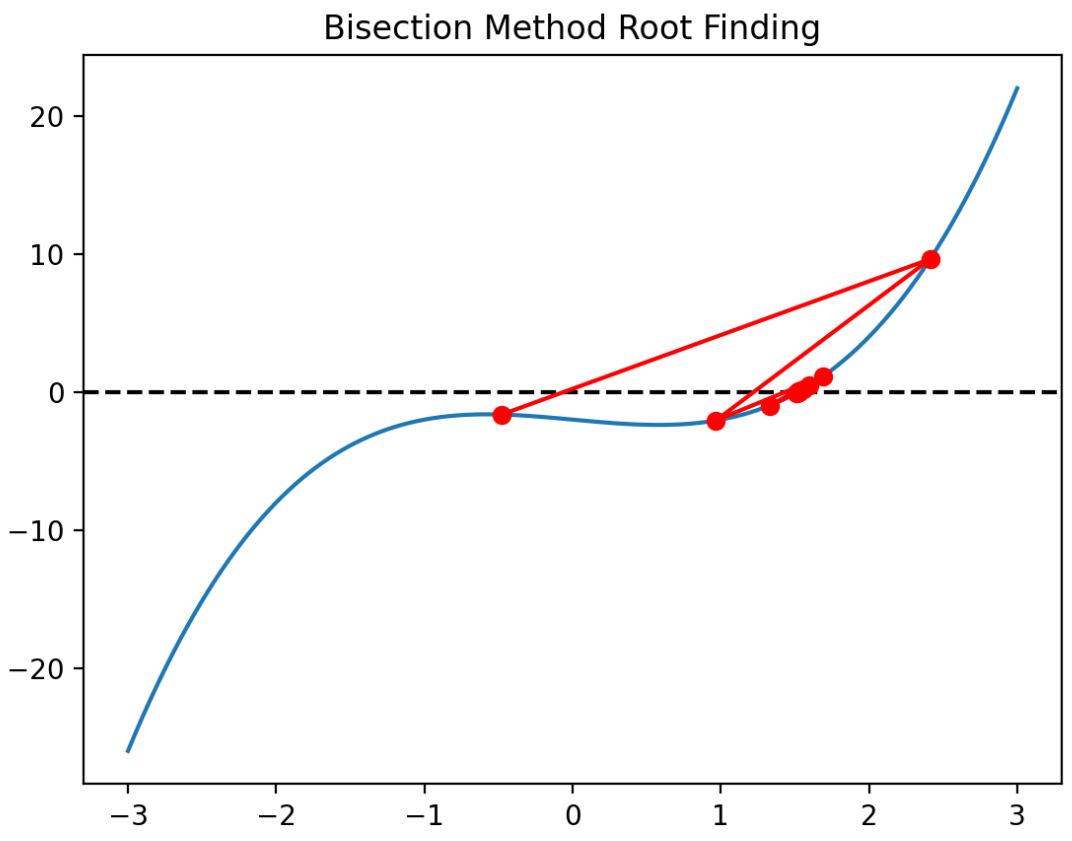
```python
import numpy as np
import matplotlib.pyplot as plt


def f(x):
    return x**3 - x - 2

a, b = np.random.uniform(-10, 0), np.random.uniform(1, 10)
```

```python
while f(a) * f(b) > 0:
    a, b = np.random.uniform(-10, 0), np.random.uniform(1, 10)

updates = []

for _ in range(20):
    c = (a + b) / 2
    updates.append(c)
    if f(c) == 0:
        break
    elif f(a) * f(c) < 0:
        b = c
    else:
        a = c

updates = np.array(updates)

print("Final root approximation:", updates[-1])

x = np.linspace(-3, 3, 400)
y = f(x)

plt.plot(x, y)
plt.axhline(0, color='black', linestyle='--')
plt.plot(updates, f(updates), 'ro-')
plt.title("Bisection Method Root Finding")
plt.show()
```

OUTPUT



```
Final root approximation: 1.5213803568560342
```


Bisection Method Root Finding
20
10
0
−10
−20
−3
−2
−1
0
1
2
3