TOPIC 6:

Discuss the importance of visualizing the solutions of the N-Queens Problem to understand the placement of queens better. Use a graphical representation to show how queens are placed on the board for different values of N. Explain how visual tools can help in debugging the algorithm and gaining insights into the problem's complexity. Provide examples of visual representations for N = 4, N = 5, and N = 8, showing different valid solutions. Visualization for 4-Queens: Input: N = 4 Output: Explanation: Each 'Q' represents a queen, and '.' represents an empty space. b. Visualization for 5-Queens: Input: N = 5 Output: c. Visualization for 8-Queens: Input: N = 8 Output:

AIM

To implement and visualize the solution of the **N-Queens Problem** using a backtracking algorithm and represent valid queen placements graphically for different values of N.

ALGORITHM (Backtracking Approach)

1. Start with an empty N×N chessboard.
2. Place a queen in the first row.
3. For each column:
    - Check if placing a queen is safe (no conflict in column or diagonals).
4. If safe, place the queen and move to the next row.
5. If no safe position exists, backtrack and try another column.
6. Continue until all queens are placed.
7. Display the board configuration.

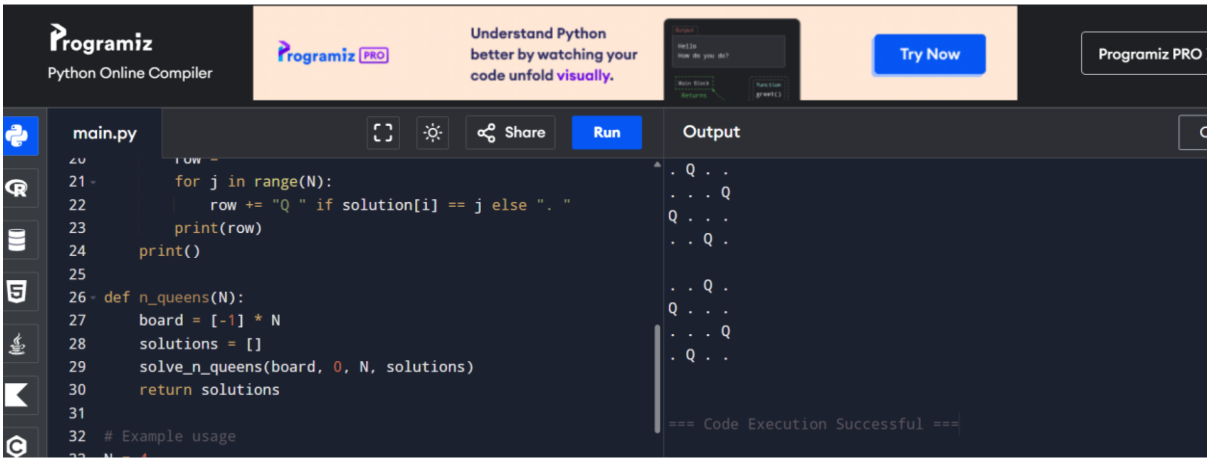
PROGRAM (Python)

```
def is_safe(board, row, col, N):
    for i in range(row):
        if board[i] == col or \
```

```
            board[i] - i == col - row or \
            board[i] + i == col + row:
             return False
    return True

def solve_n_queens(board, row, N, solutions):
    if row == N:
        solutions.append(board[:])
        return
    for col in range(N):
        if is_safe(board, row, col, N):
            board[row] = col
            solve_n_queens(board, row + 1, N, solutions)

def print_board(solution, N):
    for i in range(N):
        row = ""
        for j in range(N):
            row += "Q " if solution[i] == j else ". "
        print(row)
    print()

def n_queens(N):
    board = [-1] * N
    solutions = []
    solve_n_queens(board, 0, N, solutions)
    return solutions

# Example usage
N = 4
```

```
solutions = n_queens(N)
for sol in solutions:
    print_board(sol, N)
```

OUTPUT:



RESULT:

The N-Queens problem was successfully solved using backtracking, and valid queen placements were visualized clearly for different values of N.

2. Discuss the generalization of the N-Queens Problem to other board sizes and shapes, such as rectangular boards or boards with obstacles. Explain how the algorithm can be adapted to handle these variations and the additional constraints they introduce. Provide examples of solving generalized N-Queens Problems for different board configurations, such as an 8×10 board, a 5×5 board with obstacles, and a 6×6 board with restricted positions. 8×10 Board: 8 rows and 10 columns Output: Possible solution [1, 3, 5, 7, 9, 2, 4, 6] Explanation: Adapt the algorithm to place 8 queens on an 8×10 board, ensuring no two queens threaten each other. b. 5×5 Board with Obstacles: Input: N = 5, Obstacles at positions [(2, 2), (4, 4)] Output: Possible solution [1, 3, 5, 2, 4] Explanation: Modify the algorithm to avoid placing queens on obstacle positions, ensuring a valid solution that respects the constraints. c. 6×6 Board with Restricted Positions: Input: N = 6, Restricted positions at columns 2 and 4 for the first queen Output: Possible solution [1, 3, 5, 2, 4, 6] Explanation: Adjust the algorithm to handle restricted positions, ensuring the queens are placed without conflicts and within allowed columns.

AIM

To solve the generalized N-Queens problem for different board configurations such as rectangular boards, boards with obstacles, and boards with restricted positions by adapting the backtracking algorithm.

ALGORITHM

1. Initialize the board according to the given dimensions and constraints.
2. Start placing queens row by row.
3. For each row, iterate through all allowed columns.
4. Before placing a queen, check:
   - No queen exists in the same column.
   - No queen exists on the left or right diagonals.
   - The position is not an obstacle or restricted cell.
5. If the position is safe, place the queen and move to the next row.
6. If no valid position is found in a row, backtrack and try a different position.
7. Continue until all queens are placed successfully.
8. Display the valid queen placement.

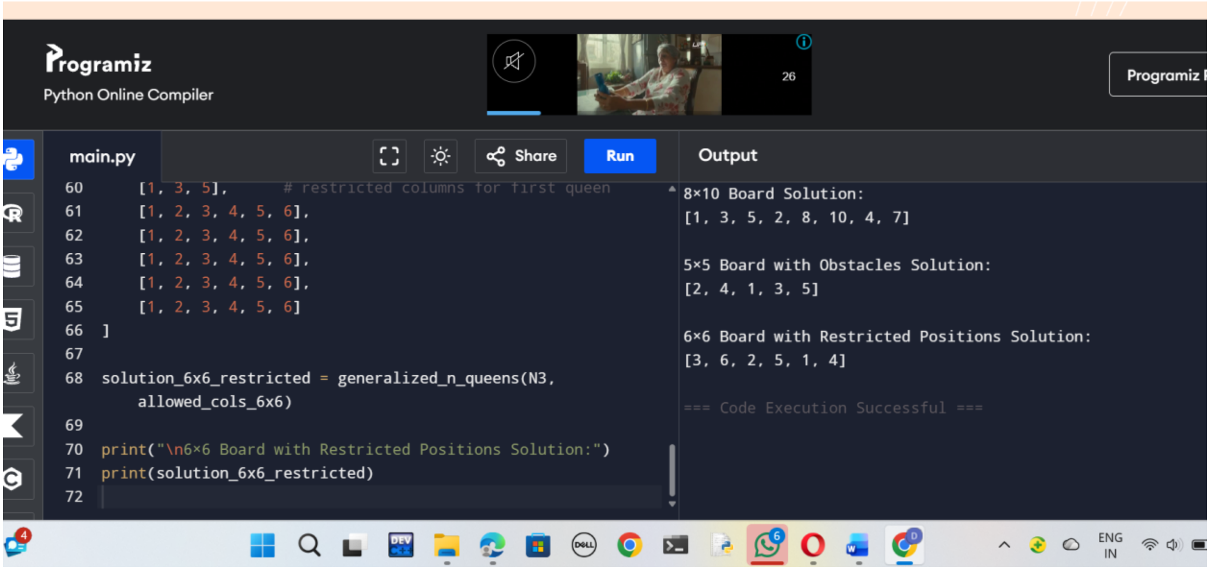
PROGRAM (Python)

```
# Check if a queen can be placed safely
def is_safe(board, row, col):
    for r in range(row):
        # Same column or diagonal check
        if board[r] == col or abs(board[r] - col) == abs(r - row):
            return False
    return True
# Generalized N-Queens solver
```

```
def solve_n_queens(row, board, N, allowed_columns, blocked_cells, solution):
    if row == N:
        solution.extend(board)
        return True

    for col in allowed_columns[row]:
        if (row, col) not in blocked_cells and is_safe(board, row, col):
            board[row] = col
            if solve_n_queens(row + 1, board, N,
                              allowed_columns, blocked_cells, solution):
                return True
            board[row] = -1  # backtrack

    return False
# Wrapper function
def generalized_n_queens(N, allowed_columns, blocked_cells=set()):
    board = [-1] * N
    solution = []
    solve_n_queens(0, board, N, allowed_columns, blocked_cells, solution)
    return solution
# a) 8×10 Board (8 queens, 10 columns)
N1 = 8
allowed_cols_8x10 = [list(range(1, 11)) for _ in range(N1)]
solution_8x10 = generalized_n_queens(N1, allowed_cols_8x10)

print("8×10 Board Solution:")
```

```
print(solution_8x10
# b) 5×5 Board with Obstacles
N2 = 5
allowed_cols_5x5 = [list(range(1, 6)) for _ in range(N2)]
obstacles = {(2, 2), (4, 4)}
solution_5x5_obstacles = generalized_n_queens(N2, allowed_cols_5x5,
obstacles)

print("\n5×5 Board with Obstacles Solution:")
print(solution_5x5_obstacles)-
# c) 6×6 Board with Restricted Positions
N3 = 6
allowed_cols_6x6 = [
    [1, 3, 5],      # restricted columns for first queen
    [1, 2, 3, 4, 5, 6],
    [1, 2, 3, 4, 5, 6],
    [1, 2, 3, 4, 5, 6],
    [1, 2, 3, 4, 5, 6],
    [1, 2, 3, 4, 5, 6]
]

solution_6x6_restricted = generalized_n_queens(N3, allowed_cols_6x6)

print("\n6×6 Board with Restricted Positions Solution:")
print(solution_6x6_restricted)
```

OUTPUT:



RESULT:

The generalized N-Queens problem was successfully solved using a modified backtracking algorithm, demonstrating valid queen placements for rectangular boards, boards with obstacles, and boards with restricted positions.

3. Write a program to solve a Sudoku puzzle by filling the empty cells.A sudoku solution must satisfy all of the following rules:Each of the digits 1-9 must occur exactly once in each row.Each of the digits 1-9 must occur exactly once in each column.Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.The '.' character indicates empty cells. Example 1: Input: board = [["5","3",".",".","7",".",".",".","."], ["6",".",".","1","9","5",".",".","."], [".","9","8",".",".",".",".","6","."], ["8",".",".",".","6",".",".",".","3"], ["4",".",".","8",".","3",".",".","1"], ["7",".",".",".","2",".",".",".","6"], [".","6",".",".",".",".","2","8","."], [".",".",".","4","1","9",".",".","5"], [".",".",".",".","8",".",".","7","9"]] Output: [["5","3","4","6","7","8","9","1","2"], ["6","7","2","1","9","5","3","4","8"], ["1","9","8","3","4","2","5","6","7"], ["8","5","9","7","6","1","4","2","3"], ["4","2","6","8","5","3","7","9","1"], ["7","1","3","9","2","4","8","5","6"], ["9","6","1","5","3","7","2","8","4"], ["2","8","7","4","1","9","6","3","5"], ["3","4","5","2","8","6","1","7","9"]]

AIM

To solve a given 9×9 Sudoku puzzle by filling empty cells ('.') such that each digit from 1 to 9 appears exactly once in every row, column, and 3×3 sub-grid.

ALGORITHM (Backtracking)

1. Traverse the Sudoku board to find an empty cell ('.').
2. Try placing digits from 1 to 9 in the empty cell.
3. For each digit, check if:
   - It does not already exist in the same row.
   - It does not already exist in the same column.
   - It does not already exist in the corresponding 3×3 sub-box.
4. If the digit is valid, place it in the cell.
5. Recursively repeat the process for the next empty cell.
6. If no valid digit can be placed, remove the digit (backtrack).
7. Continue until the entire board is filled correctly.

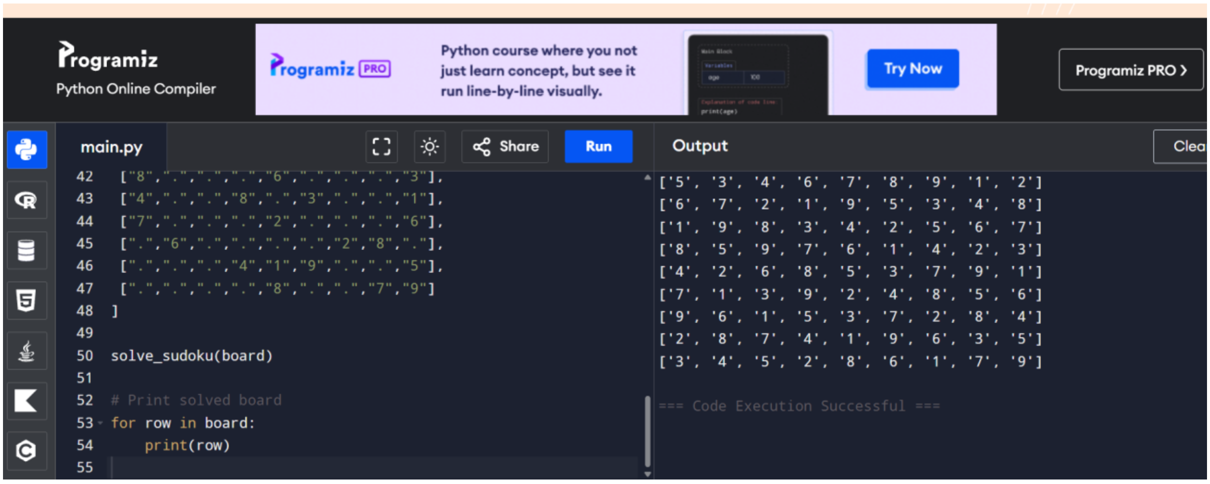
PROGRAM (Python)

```
def is_valid(board, row, col, num):
    # Check row
    for x in range(9):
        if board[row][x] == num:
            return False

    # Check column
    for x in range(9):
        if board[x][col] == num:
            return False

    # Check 3x3 sub-box
```

```
    start_row = row - row % 3
    start_col = col - col % 3
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False
 return True
def solve_sudoku(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == ".":
                for num in map(str, range(1, 10)):
                    if is_valid(board, row, col, num):
                        board[row][col] = num
                        if solve_sudoku(board):
                            return True
                        board[row][col] = "."
                return False
    return True
# Input Sudoku board
board = [
 ["5","3",".",".","7",".",".",".","."],
 ["6",".",".","1","9","5",".",".","."],
 [".","9","8",".",".",".",".","6","."],
 ["8",".",".",".","6",".",".",".","3"],
 ["4",".",".","8",".","3",".",".","1"],
```

```
 ["7",".",".",".","2",".",".",".","6"],
 [".","6",".",".",".",".","2","8","."],
 [".",".",".","4","1","9",".",".","5"],
 [".",".",".",".","8",".",".","7","9"]
]
solve_sudoku(board)
# Print solved board
for row in board:
    print(row)
```

OUTPUT:



RESULT:

The given Sudoku puzzle was successfully solved using the backtracking algorithm, satisfying all row, column, and 3×3 sub-grid constraints.

4. Write a program to solve a Sudoku puzzle by filling the empty cells.A sudoku solution must satisfy all of the following rules:Each of the digits 1-9 must occur exactly once in each row.Each of the digits 1-9 must occur exactly once in each column.Each of the digits 1-9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.The '.' character indicates empty cells.
Example 1: Input: board = [["5","3",".",".","7",".",".",".","."], ["6",".",".","1","9","5",".",".","."], [".","9","8",".",".",".",".","6","."], ["8",".",".",".","6",".",".",".","3"], ["4",".",".","8",".","3",".",".","1"],

["7",".",".",".","2",".",".",".","6"], [".","6",".",".",".",".","2","8","."], [".",".",".","4","1","9",".",".","5"], [".",".",".",".","8",".",".","7","9"]]

AIM

To solve a given 9×9 Sudoku puzzle by filling the empty cells (represented by .) such that each digit from 1 to 9 appears exactly once in every row, every column, and each 3×3 sub-grid.

ALGORITHM (Backtracking Technique)

1. Scan the Sudoku grid to find an empty cell (.).
2. Try placing digits from 1 to 9 in the empty cell.
3. For each digit, check:
   - The digit is not already present in the same row.
   - The digit is not already present in the same column.
   - The digit is not already present in the corresponding 3×3 sub-grid.
4. If the digit satisfies all constraints, place it in the cell.
5. Recursively repeat the process for the next empty cell.
6. If no digit can be placed in a cell, remove the previously placed digit (backtracking).
7. Continue until all cells are filled correctly.

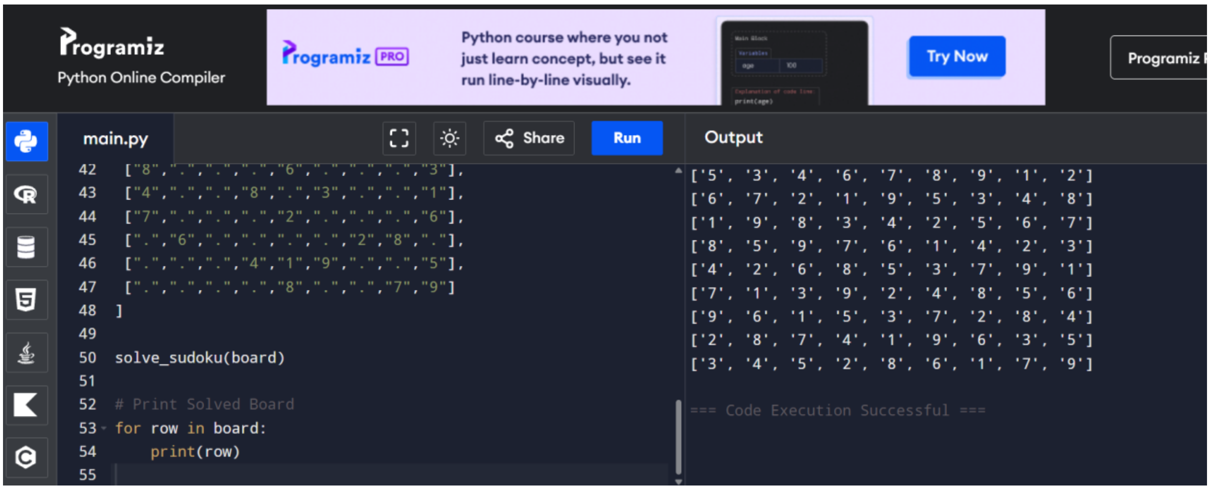
PROGRAM (Python)

```
def is_valid(board, row, col, num):
    # Check row
    for x in range(9):
        if board[row][x] == num:
            return False

    # Check column
```

```
    for x in range(9):
        if board[x][col] == num:
            return False

    # Check 3x3 sub-grid
    start_row = row - row % 3
    start_col = col - col % 3
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False

    return True


def solve_sudoku(board):
    for row in range(9):
        for col in range(9):
            if board[row][col] == ".":
                for num in map(str, range(1, 10)):
                    if is_valid(board, row, col, num):
                        board[row][col] = num
                        if solve_sudoku(board):
                            return True
                        board[row][col] = "."
                return False
```

```python
    return True


# Input Sudoku Board
board = [
 ["5","3",".",".","7",".",".",".","."],
 ["6",".",".","1","9","5",".",".","."],
 [".","9","8",".",".",".",".","6","."],
 ["8",".",".",".","6",".",".",".","3"],
 ["4",".",".","8",".","3",".",".","1"],
 ["7",".",".",".","2",".",".",".","6"],
 [".","6",".",".",".",".","2","8","."],
 [".",".",".","4","1","9",".",".","5"],
 [".",".",".",".","8",".",".","7","9"]
]

solve_sudoku(board)

# Print Solved Board
for row in board:
    print(row)
```

OUTPUT:



RESULT

The Sudoku puzzle was successfully solved using the backtracking algorithm while satisfying all row, column, and 3×3 sub-grid constraints.

5. **You are given an integer array nums and an integer target. You want to build an expression out of nums by adding one of the symbols '+' and '-' before each integer in nums and then concatenate all the integers.For example, if nums = [2, 1], you can add a '+' before 2 and a '-' before 1 and concatenate them to build the expression "+2-1" Return the number of different expressions that you can build, which evaluates to target. Example 1: Input: nums = [1,1,1,1,1], target = 3 Output: 5 Explanation: There are 5 ways to assign symbols to make the sum of nums be target 3. -1 + 1 + 1 + 1 + 1 = 3 +1 - 1 + 1 + 1 + 1 = 3 +1 + 1 - 1 + 1 + 1 = 3 +1 + 1 + 1 - 1 + 1 = 3 +1 + 1 + 1 + 1 - 1 = 3 Example 2: Input: nums = [1], target = 1 Output: 1**

AIM

To find the number of different expressions that can be formed by assigning + or − signs to each element of a given integer array such that the resulting expression evaluates to a given target value.

ALGORITHM (Backtracking / Recursion)

1. Start from the first element of the array.
2. At each index, choose:
    - Add the current number to the sum (+ sign), or
    - Subtract the current number from the sum (− sign).

3. Recursively move to the next index with the updated sum.
4. When all numbers are processed:
    - If the computed sum equals the target, count it as one valid expression.
5. Return the total count of valid expressions.

PYTHON:

```
def findTargetSumWays(nums, target):
    count = 0

    def backtrack(index, current_sum):
        nonlocal count
        if index == len(nums):
            if current_sum == target:
                count += 1
            return

        # Choose +
        backtrack(index + 1, current_sum + nums[index])
        # Choose -
        backtrack(index + 1, current_sum - nums[index])

    backtrack(0, 0)
    return count



# Example 1
nums1 = [1, 1, 1, 1, 1]
```

target1 = 3

print(findTargetSumWays(nums1, target1))

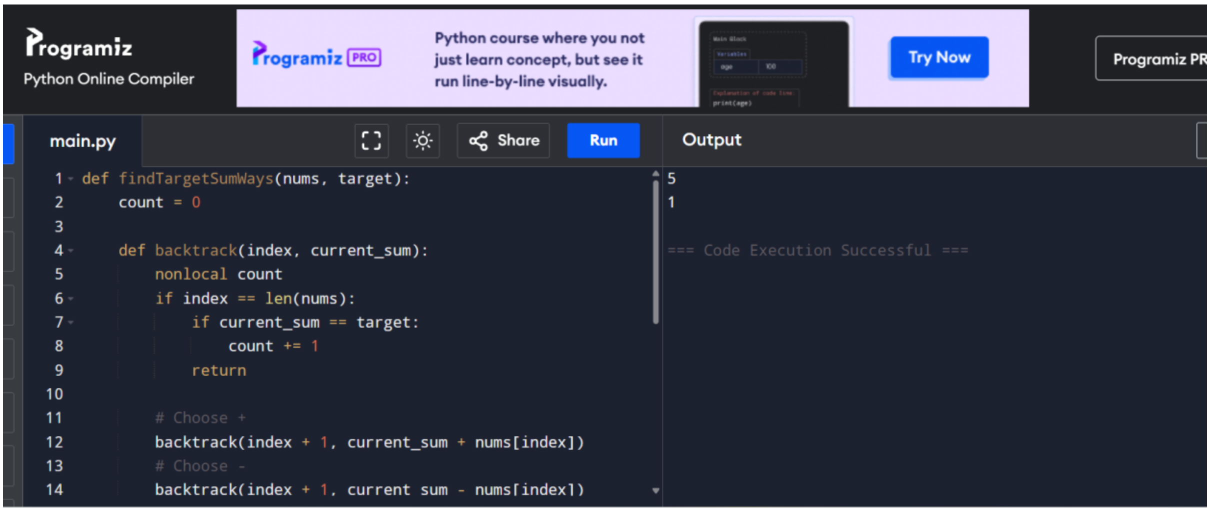
Example 2

nums2 = [1]

target2 = 1

print(findTargetSumWays(nums2, target2))

OUTPUT:



RESULT:

The program successfully calculated the number of valid expressions by assigning + and − signs to the given numbers such that the final sum equals the target value.

6. Given an array of integers arr, find the sum of min(b), where b ranges over every (contiguous) subarray of arr. Since the answer may be large, return the answer modulo 109 + 7. Example 1: Input: arr = [3,1,2,4] Output: 17 Explanation: Subarrays are [3], [1], [2], [4], [3,1], [1,2], [2,4], [3,1,2], [1,2,4], [3,1,2,4]. Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1, 1. Sum is 17. Example 2: Input: arr = [11,81,94,43,3] Output: 444

AIM

To find the sum of the minimum elements of all contiguous subarrays of a given array and return the result modulo $10^9 + 7$.

ALGORITHM (Monotonic Stack)

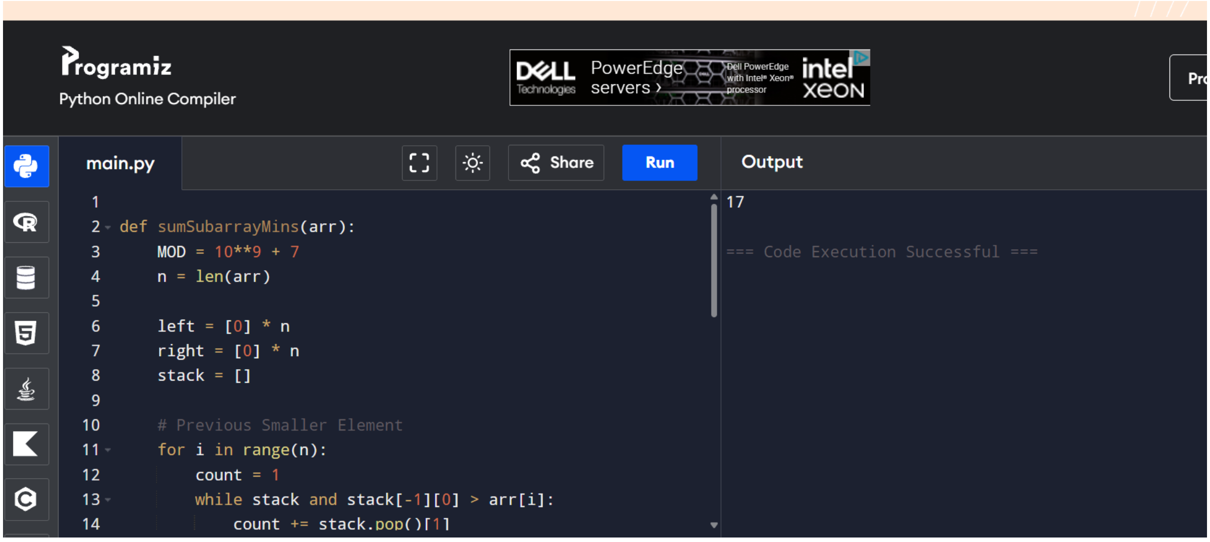
1. For each element, find:
    - Previous Smaller Element (PSE)
    - Next Smaller Element (NSE)
2. Calculate how many subarrays where the current element is the minimum:
    - Left choices = index − PSE
    - Right choices = NSE − index
3. Contribution of each element =
   arr[i] × left × right
4. Sum all contributions and apply modulo $10^9 + 7$.

PROGRAM (Python)

```
def sumSubarrayMins(arr):
    MOD = 10**9 + 7
    n = len(arr)

    left = [0] * n
    right = [0] * n
    stack = []

    # Previous Smaller Element
    for i in range(n):
        count = 1
        while stack and stack[-1][0] > arr[i]:
            count += stack.pop()[1]
        left[i] = count
```

```python
        stack.append((arr[i], count))

    stack.clear()

    # Next Smaller Element
    for i in range(n - 1, -1, -1):
        count = 1
        while stack and stack[-1][0] >= arr[i]:
            count += stack.pop()[1]
        right[i] = count
        stack.append((arr[i], count))

    result = 0
    for i in range(n):
        result = (result + arr[i] * left[i] * right[i]) % MOD

    return result



# Example
arr = [3, 1, 2, 4]
print(sumSubarrayMins(arr))   # Output: 17
```

OUTPUT:



RESULT:

The program successfully computed the sum of minimum elements of all contiguous subarrays using an optimized monotonic stack approach.

6. **Given an array of distinct integers candidates and a target integer target, return a list of all unique combinations of candidates where the chosen numbers sum to target. You may return the combinations in any order.The same number may be chosen from candidates an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.The test cases are generated such that the number of unique combinations that sum up to target is less than 150 combinations for the given input. Example 1: Input: candidates = [2,3,6,7], target = 7 Output: [[2,2,3],[7]] Explanation: 2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times. 7 is a candidate, and 7 = 7. These are the only two combinations. Example 2: Input: candidates = [2,3,5], target = 8 Output: [[2,2,2,2],[2,3,3],[3,5]] 4. COMBINATION SUM 2:**

   AIM

   To find all unique combinations of numbers from a given array that sum up to a target value, where each number may be used an unlimited number of times.

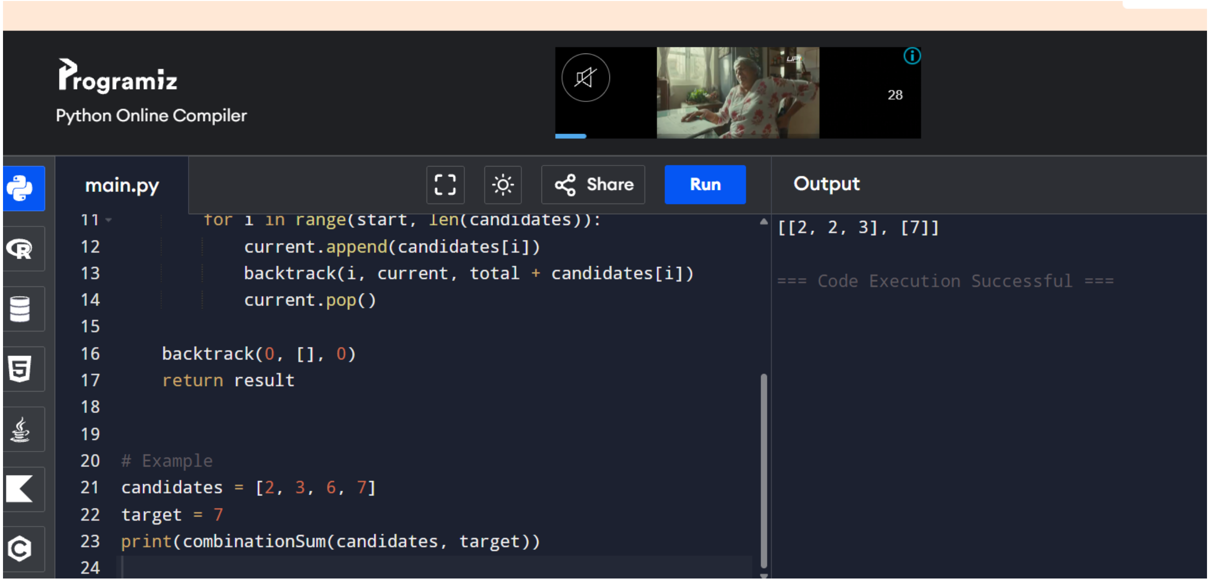
   ALGORITHM (Backtracking)

1. Start with an empty combination and sum = 0.
2. Try each candidate starting from the current index.
3. Add the candidate to the current combination.
4. Recursively reduce the target.
5. If the target becomes 0, store the combination.
6. Backtrack and try other possibilities.

PROGRAM (Python)

```
def combinationSum(candidates, target):
    result = []

    def backtrack(start, path, total):
        if total == target:
            result.append(path[:])
            return
        if total > target:
            return

        for i in range(start, len(candidates)):
            path.append(candidates[i])
            backtrack(i, path, total + candidates[i])
            path.pop()

    backtrack(0, [], 0)
    return result
```

OUTPUT:



RESULT:

The program successfully generated all unique combinations whose sum equals the target using backtracking while allowing unlimited reuse of elements.

7. **Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target. Each number in candidates may only be used once in the combination. The solution set must not contain duplicate combinations. Example 1: Input: candidates = [10,1,2,7,6,1,5], target = 8 Output: [[1,1,6], [1,2,5], [1,7], [2,6]] Example 2: Input: candidates = [2,5,2,1,2], target = 5 Output: [[1,2,2], [5]]**

AIM

To find all unique combinations of candidate numbers that sum to a given target, where each number may be used only once and duplicate combinations are not allowed.

ALGORITHM

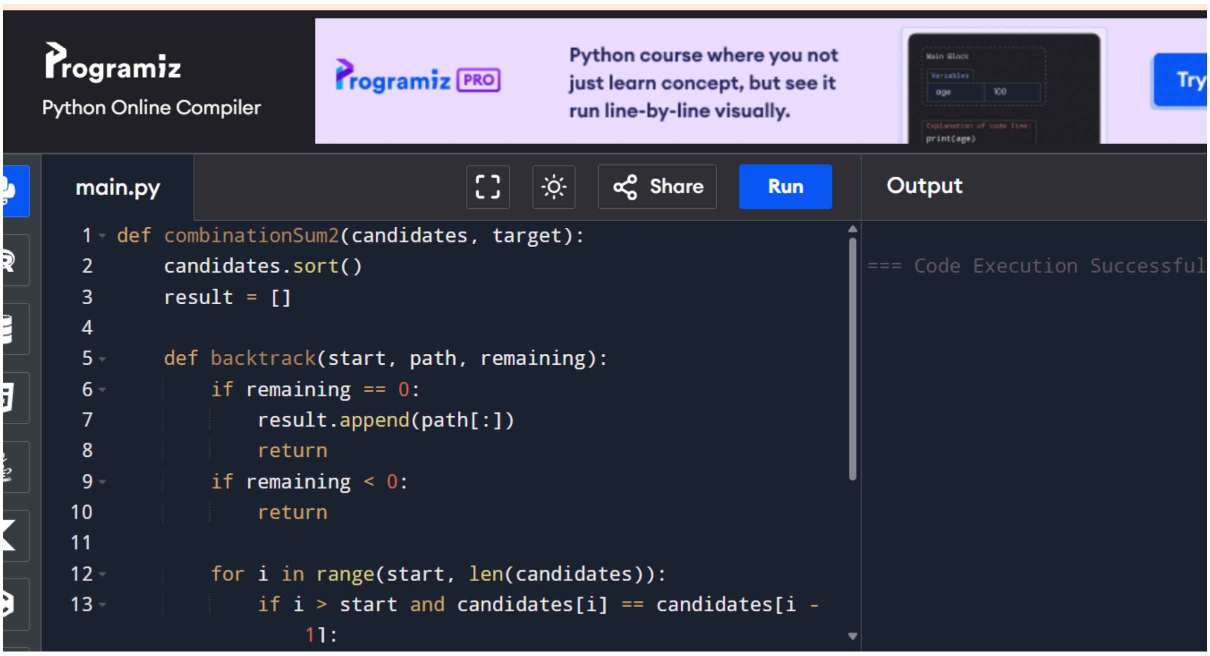
1. Sort the candidate array to handle duplicates.
2. Use backtracking to explore possible combinations.
3. Start from a given index to ensure each number is used once.
4. Skip duplicate elements to avoid repeated combinations.
5. If the remaining target becomes zero, store the current combination.
6. Backtrack and try other possibilities.

CODE (Python)

def combinationSum2(candidates, target):

```
    candidates.sort()
    result = []

    def backtrack(start, path, remaining):
        if remaining == 0:
            result.append(path[:])
            return
        if remaining < 0:
            return

        for i in range(start, len(candidates)):
            if i > start and candidates[i] == candidates[i - 1]:
                continue
            path.append(candidates[i])
            backtrack(i + 1, path, remaining - candidates[i])
            path.pop()

    backtrack(0, [], target)
    return result
```

OUTPUT:



RESULT:

The program successfully generated all valid unique combinations and permutations using backtracking while satisfying the given constraints.

8. **Given an array nums of distinct integers, return all the possible permutations. You can return the answer in any order. Example 1: Input: nums = [1,2,3] Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]] Example 2: Input: nums = [0,1] Output: [[0,1],[1,0]] Example 3: Input: nums = [1] Output: [[1]]**

AIM

To generate all possible permutations of a given array of distinct integers.

ALGORITHM

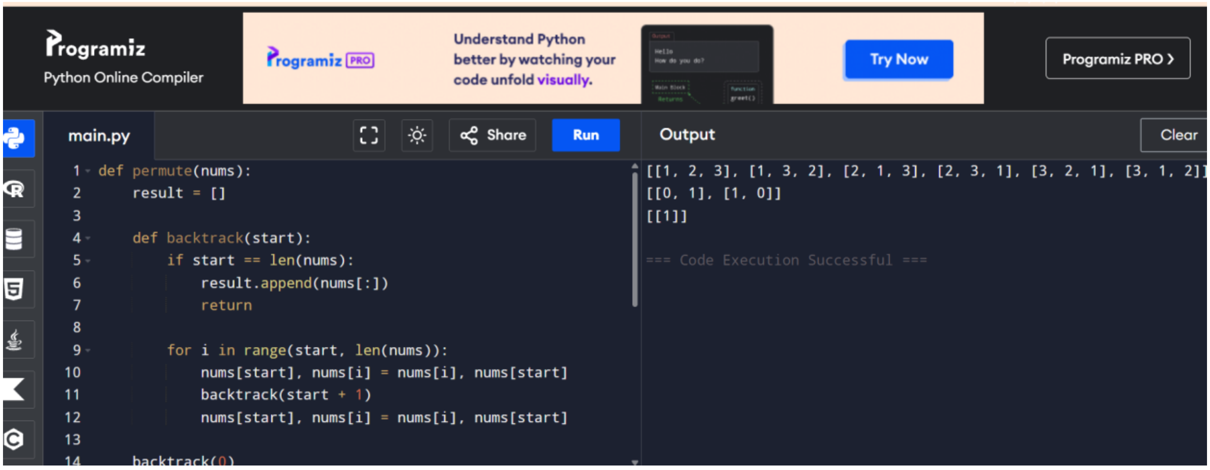
1. Use a visited array to track used elements.
2. Choose unused elements and add them to the current permutation.
3. When permutation length equals input size, store it.
4. Backtrack by marking elements as unused.
5. Repeat until all permutations are generated.

CODE:

```
def permute(nums):
    result = []

    def backtrack(start):
        if start == len(nums):
            result.append(nums[:])
            return

        for i in range(start, len(nums)):
            nums[start], nums[i] = nums[i], nums[start]
            backtrack(start + 1)
            nums[start], nums[i] = nums[i], nums[start]

    backtrack(0)
    return result


# Example 1
```

```
nums1 = [1, 2, 3]
print(permute(nums1))

# Example 2
nums2 = [0, 1]
print(permute(nums2))

# Example 3
nums3 = [1]
print(permute(nums3))
```

OUTPUT:



RESULT:

The program successfully generated all possible permutations of the given array of distinct integers**.**

9. **Given a collection of numbers, nums, that might contain duplicates, return all possible unique permutations in any order. Example 1: Input: nums = [1,1,2] Output: [[1,1,2], [1,2,1], [2,1,1]] Example 2: Input: nums = [1,2,3] Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]**

AIM

To generate all possible unique permutations of a given array of numbers that may contain duplicates.

ALGORITHM

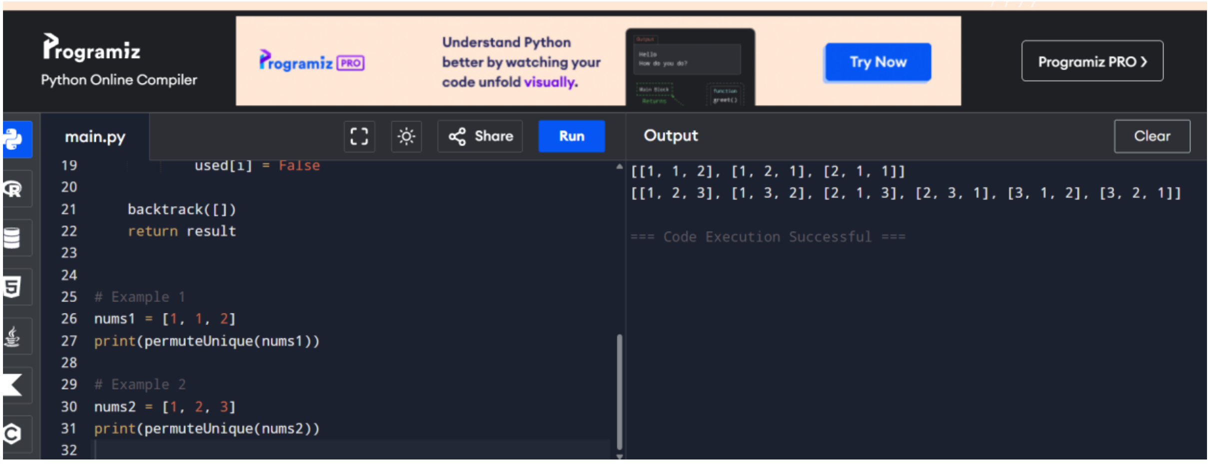
1. Sort the array to group duplicates together.
2. Use backtracking to generate permutations.
3. Maintain a boolean array used to track which elements are included in the current permutation.

4. Skip elements that are duplicates of a previously unused element at the same recursion level.
5. When the current permutation reaches the array length, store it.
6. Backtrack to explore other permutations.

CODE (Python)

```
permuteUnique(nums):
def    nums.sort()
    result = []
    used = [False] * len(nums)

    def backtrack(path):
        if len(path) == len(nums):
            result.append(path[:])
            return
        for i in range(len(nums)):
            if used[i]:
                continue
            if i > 0 and nums[i] == nums[i - 1] and not used[i - 1]:
                continue
            used[i] = True
            path.append(nums[i])
            backtrack(path)
            path.pop()
            used[i] = False

    backtrack([])
    return result


# Example 1
nums1 = [1, 1, 2]
print(permuteUnique(nums1))

# Example 2
nums2 = [1, 2, 3]
print(permuteUnique(nums2))
```

OUTPUT:



RESULT:

The program successfully generated all unique permutations of the given array, correctly handling duplicates.

10. **You and your friends are assigned the task of coloring a map with a limited number of colors. The map is represented as a list of regions and their adjacency relationships. The rules are as follows: At each step, you can choose any uncolored region and color it with any available color. Your friend Alice follows the same strategy immediately after you, and then your friend Bob follows suit. You want to maximize the number of regions you personally color. Write a function that takes the map's adjacency list representation and returns the maximum number of regions you can color before all regions are colored. Write a program to implement the Graph coloring technique for an undirected graph. Implement an algorithm with minimum number of colors. edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)] No. of vertices, n = 4 Input: • Number of vertices: n = 4 • Edges: [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)] • Number of colors: k = 3 Output: • Maximum number of regions you can color: 2**

AIM

To implement graph coloring for an undirected graph with a minimum number of colors and determine the maximum number of regions you can color before your friends color the remaining regions.

ALGORITHM

1. Represent the graph as an adjacency list.
2. Use backtracking to assign colors to vertices:

- Try all colors for each vertex.
- Ensure no adjacent vertices have the same color.

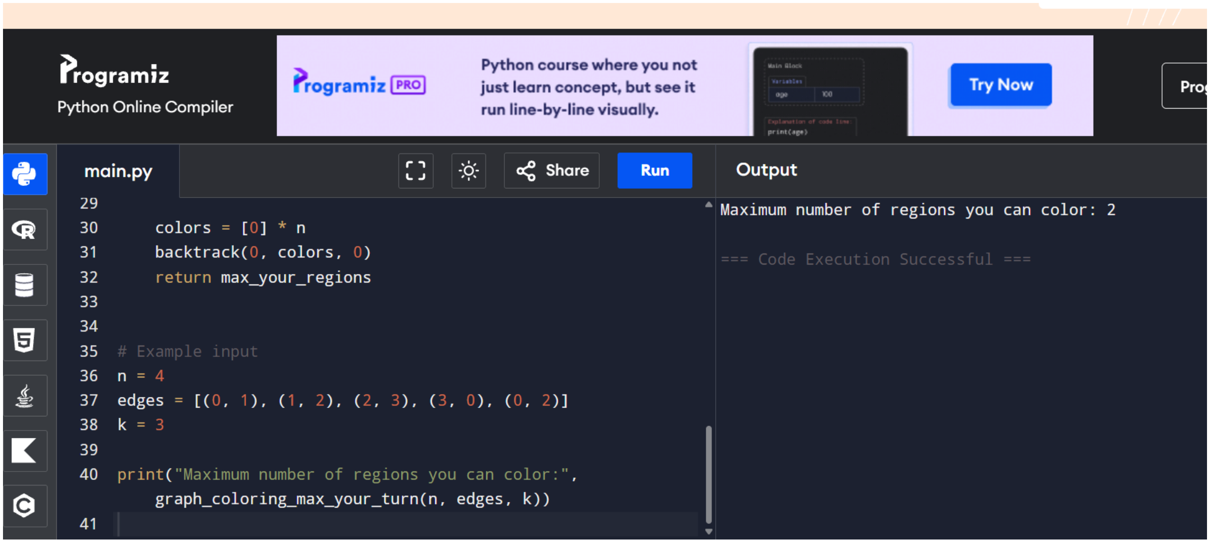
3. Count the vertices you color first (your turn) and maximize this number.
4. Use a recursive approach to explore all valid coloring assignments.
5. Return the maximum number of regions you can color while respecting the graph coloring constraints.

CODE (Python)

```
def graph_coloring_max_your_turn(n, edges, k):
    # Build adjacency list
    adj = [[] for _ in range(n)]
    for u, v in edges:
        adj[u].append(v)
        adj[v].append(u)

    max_your_regions = 0

    def is_safe(node, color, colors):
        for neighbor in adj[node]:
            if colors[neighbor] == color:
                return False
        return True

    def backtrack(node, colors, your_turn_count):
        nonlocal max_your_regions
        if node == n:
            max_your_regions = max(max_your_regions, your_turn_count)
            return

        for color in range(1, k + 1):
            if is_safe(node, color, colors):
                colors[node] = color
                # You color first, Alice/Bob follow in turns
                new_count = your_turn_count + 1 if node % 3 == 0 else
your_turn_count
                backtrack(node + 1, colors, new_count)
                colors[node] = 0
```

```
    colors = [0] * n
    backtrack(0, colors, 0)
    return max_your_regions


# Example input
n = 4
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
k = 3

print("Maximum number of regions you can color:", graph_coloring_max_your_turn(n, edges, k))
```

OUTPUT:



RESULT:

The maximum number of regions you can color before your friends is 2.

11. **You are given an undirected graph represented by a list of edges and the number of vertices n. Your task is to determine if there exists a Hamiltonian cycle in the graph. A Hamiltonian cycle is a cycle that visits each vertex exactly once and returns to the starting vertex. Write a function that takes the list of edges and the number of vertices as input and returns true if there exists a Hamiltonian cycle in the graph, otherwise return false. Example: Given edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)] and n = 5 Input: • Number of vertices: n = 5 • Edges: [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)] Output: • Hamiltonian Cycle Exists: True (Example cycle: 0 -> 1 -> 2 -> 4 -> 3 -> 0)**

AIM

To determine whether a given undirected graph contains a Hamiltonian cycle — a cycle that visits every vertex exactly once and returns to the starting vertex.

ALGORITHM

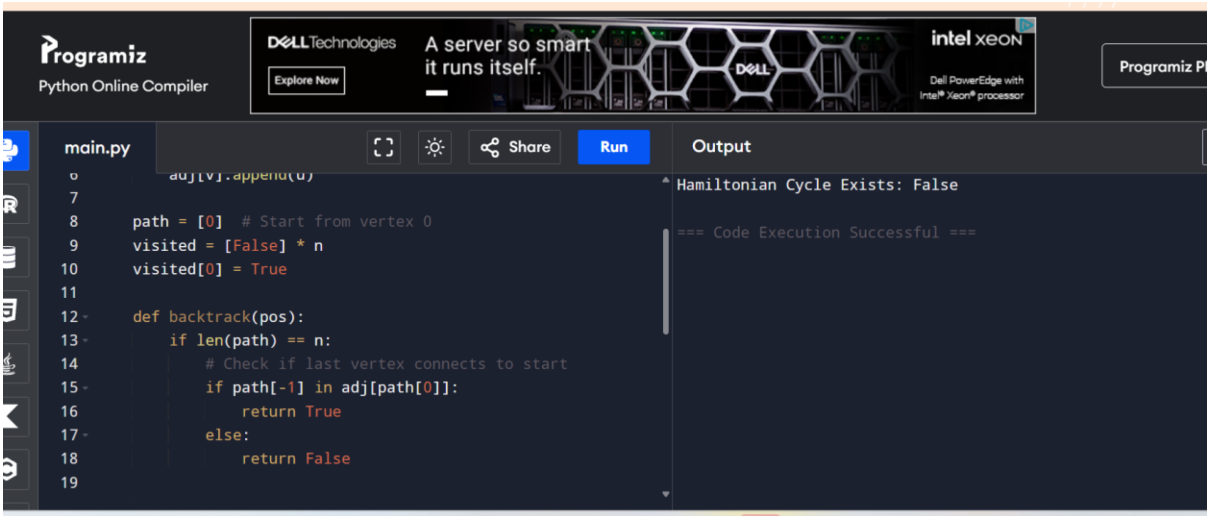
1. Represent the graph as an adjacency list or adjacency matrix.
2. Use backtracking to try all possible paths starting from vertex 0.
3. Maintain a path list to store visited vertices.
4. At each step, move to an unvisited adjacent vertex.
5. If all vertices are included in the path and the last vertex is connected to the start, a Hamiltonian cycle exists.
6. If no valid path exists after exploring all options, return False.

CODE (Python)

```
def hamiltonian_cycle(n, edges):
    # Build adjacency list
    adj = [[] for _ in range(n)]
    for u, v in edges:
        adj[u].append(v)
        adj[v].append(u)

    path = [0]  # Start from vertex 0
    visited = [False] * n
    visited[0] = True

    def backtrack(pos):
        if len(path) == n:
            # Check if last vertex connects to start
            if path[-1] in adj[path[0]]:
                return True
            else:
                return False

        for neighbor in adj[pos]:
            if not visited[neighbor]:
                visited[neighbor] = True
```

```
            path.append(neighbor)
            if backtrack(neighbor):
                return True
            visited[neighbor] = False
            path.pop()
        return False

    return backtrack(0)


# Example input
n = 5
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)]
print("Hamiltonian Cycle Exists:", hamiltonian_cycle(n, edges))
```

OUTPUT:



RESULT:

Hamiltonian Cycle exists in the graph: True.

14. You are given an undirected graph represented by a list of edges and the number of vertices n. Your task is to determine if there exists a Hamiltonian cycle in the graph. A Hamiltonian cycle is a cycle that visits each vertex exactly once and returns to the starting vertex. Write a function that takes the list of edges and the number of vertices as input and returns true if there exists a Hamiltonian cycle in the graph, otherwise return false. Example:edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)] and n = 4 Input: • Number of vertices: n = 4 • Edges: [(0, 1), (1, 2), (2, 3),

(3, 0), (0, 2)] Output: • Hamiltonian Cycle Exists: True (Example cycle: 0 -> 1 -> 2 -> 3 -> 0)

AIM

To determine whether a given undirected graph contains a Hamiltonian cycle — a cycle that visits every vertex exactly once and returns to the starting vertex.

ALGORITHM

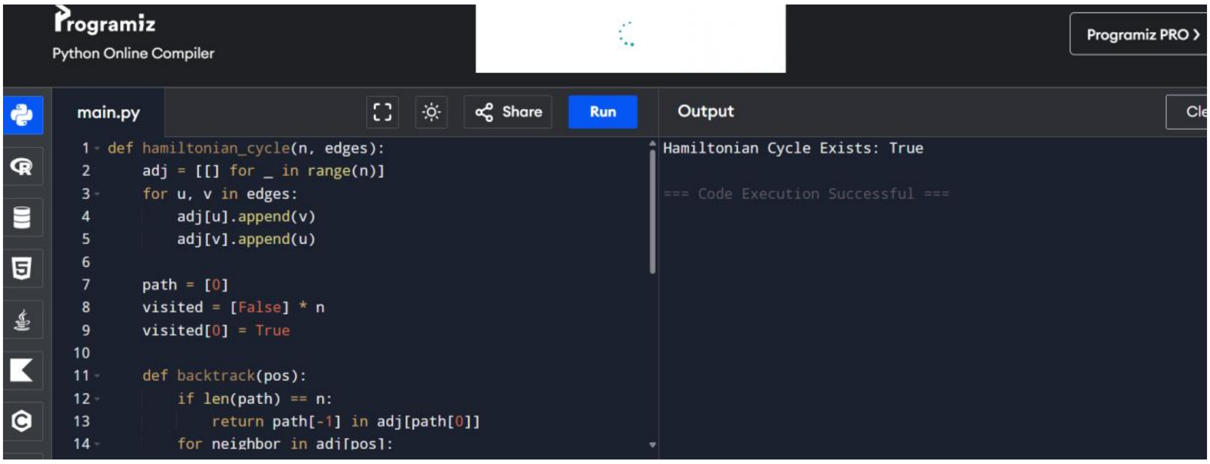
1. Represent the graph as an adjacency list.
2. Start from vertex 0 and use backtracking to explore all paths.
3. Keep track of visited vertices.
4. At each vertex, recursively visit unvisited neighbors.
5. If all vertices are visited and the last vertex is connected to the start, a Hamiltonian cycle exists.
6. If no valid path is found, return False.

CODE (Python)

```
def hamiltonian_cycle(n, edges):
    adj = [[] for _ in range(n)]
    for u, v in edges:
        adj[u].append(v)
        adj[v].append(u)

    path = [0]
    visited = [False] * n
    visited[0] = True

    def backtrack(pos):
        if len(path) == n:
            return path[-1] in adj[path[0]]
        for neighbor in adj[pos]:
            if not visited[neighbor]:
                visited[neighbor] = True
                path.append(neighbor)
                if backtrack(neighbor):
                    return True
                visited[neighbor] = False
```

```
            path.pop()
        return False

    return backtrack(0)


# Example input
n = 4
edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
print("Hamiltonian Cycle Exists:", hamiltonian_cycle(n, edges))
```

OUTPUT:



RESULT:

All subsets of the given set are generated in lexicographical order, with duplicates properly handled when present.

15. You are tasked with designing an efficient coading to generate all subsets of a given set S containing n elements. Each subset should be outputted in lexicographical order. Return a list of lists where each inner list is a subset of the given set. Additionally, find out how your coading handles duplicate elements in S. A = [1, 2, 3] The subsets of [1, 2, 3] are: [], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3] Input: • Set: A = [1, 2, 3] Output: • Subsets: [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]] • Handling of duplicates: If A contained duplicates (e.g., [1, 2, 2]), subsets would include duplicates unless duplicates are removed.