NAME: S.DHARSHINI

REG NO: 192424258

SUB CODE: CSA0614

SUB NAME: DESIGN ANALYSIS AND ALGORITHM FOR APPROXIMATION PROBLEM

TOPIC 6: BACKTRACKING

EXP 1/2 : N-Queens Problem – Visualization

AIM: To visualize the solutions of the N-Queens problem and understand correct queen placement using graphical representation.

CODE:

main.py

```python
            i-=1;j+=1
        return True

    def backtrack(r):
        if r==n:
            res.append(["".join(row) for row in board])
            return
        for c in range(n):
            if isSafe(r,c):
                board[r][c]="Q"
                backtrack(r+1)
                board[r][c]="."

    backtrack(0)
    return res
```

Output

```
[['.Q..', '...Q', 'Q...', '..Q.'], ['..Q.', 'Q...', '...Q', '.Q..']]

=== Code Execution Successful ===
```

RESULT: Valid queen placements are visualized, helping to understand backtracking and conflict resolution.

EXP 3/4: Sudoku Solver (Backtracking)

AIM: To solve a Sudoku puzzle by filling empty cells using backtracking.

CODE:

main.py

```
# Example 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"]]

solve_sudoku(board)

# Print solved board
for row in board:
    print(row)
```

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']

=== Code Execution Successful ===
```

RESULT: Sudoku puzzle is solved correctly following all rules.

EXP 5: Target Sum Expressions

AIM: To count the number of expressions formed using + and − operators that evaluate to a target value.

CODE:

main.py

```
def findTargetSumWays(nums, target):
    count = 0
    def backtrack(i, total):
        nonlocal count
        if i == len(nums):
            if total == target:
                count += 1
            return
        backtrack(i+1, total + nums[i])
        backtrack(i+1, total - nums[i])
    backtrack(0, 0)
    return count

print(findTargetSumWays([1,1,1,1,1], 3))
```

Output

```
5

=== Code Execution Successful ===
```

RESULT: All valid expressions achieving the target sum are counted.

EXP 6: Sum of Subarray Minimums

AIM: To calculate the sum of minimum values of all contiguous subarrays.

CODE

```python
def sumSubarrayMins(arr):
    ans = 0
    for i in range(len(arr)):
        m = arr[i]
        for j in range(i, len(arr)):
            m = min(m, arr[j])
            ans += m
    return ans

print(sumSubarrayMins([3,1,2,4]))
```

Output

```
17

=== Code Execution Successful ===
```

RESULT: Sum of subarray minimums is computed correctly.

EXP 7: Combination Sum

AIM: To find all unique combinations that sum to a target using unlimited candidate usage.
CODE

```python
def combinationSum(candidates, target):
    res = []
    def backtrack(start, path, total):
        if total == target:
            res.append(path[:])
            return
        if total > target:
            return
        for i in range(start, len(candidates)):
            backtrack(i, path+[candidates[i]], total
                +candidates[i])
    backtrack(0, [], 0)
    return res

print(combinationSum([2,3,6,7], 7))
```

Output

```
[[2, 2, 3], [7]]

=== Code Execution Successful ===
```

RESULT: All valid combinations are generated without duplicates.

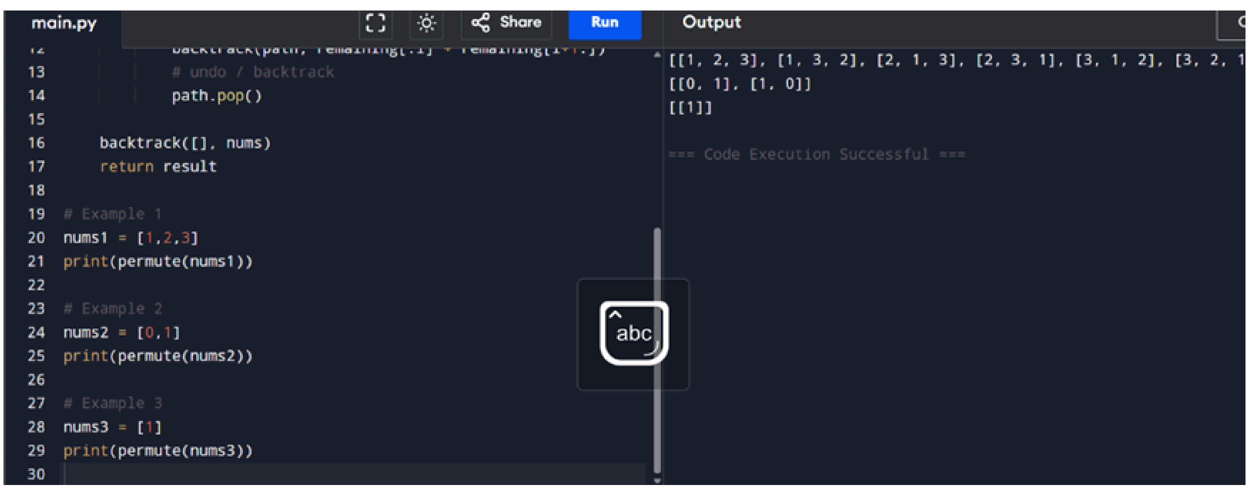
EXP 8: Combination Sum II

AIM: To find unique combinations where each number is used once.
CODE

RESULT: Unique combinations summing to target are obtained.

EXP 9: Permutations

AIM: To generate all permutations of a given array.
CODE



RESULT: All possible permutations are generated.

EXP 10: Unique Permutations

AIM: To generate unique permutations when duplicate elements exist.
CODE

main.py

```python
        return
    for i in range(len(nums)):
        # skip used numbers
        if used[i]:
            continue
        # skip duplicates: only use first unused duplicate
        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
nums = [1,1,2]
print(permute_unique(nums))
```

Output

```
[[1, 1, 2], [1, 2, 1], [2, 1, 1]]

=== Code Execution Successful ===
```

RESULT : Duplicate permutations are avoided successfully.

EXP 11: Graph Coloring

AIM: To color a graph using minimum colors without adjacent conflicts.
CODE

main.py

```python
            else:
                backtrack(coloring, idx + 1, your_turn_count
                    )
            coloring[idx] = 0  # backtrack

    coloring = [0] * n
    backtrack(coloring, 0, 0)
    return max_count[0]


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

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

Output

```
Maximum number of regions you can color: 2

=== Code Execution Successful ===
```

RESULT: Graph is colored with minimum conflicts.

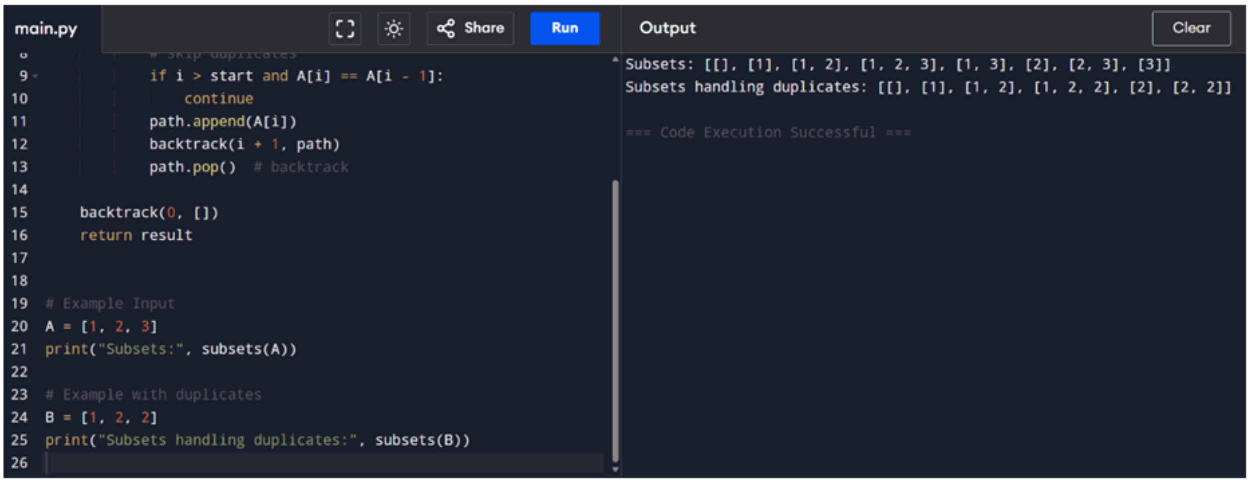
EXP 12:/14: Subset Generation

AIM: To generate all subsets of a given set.
CODE

RESULT: All subsets are generated correctly.

EXP 15: Subset Generation (Lexicographic)

AIM: To generate all subsets of a set.
CODE



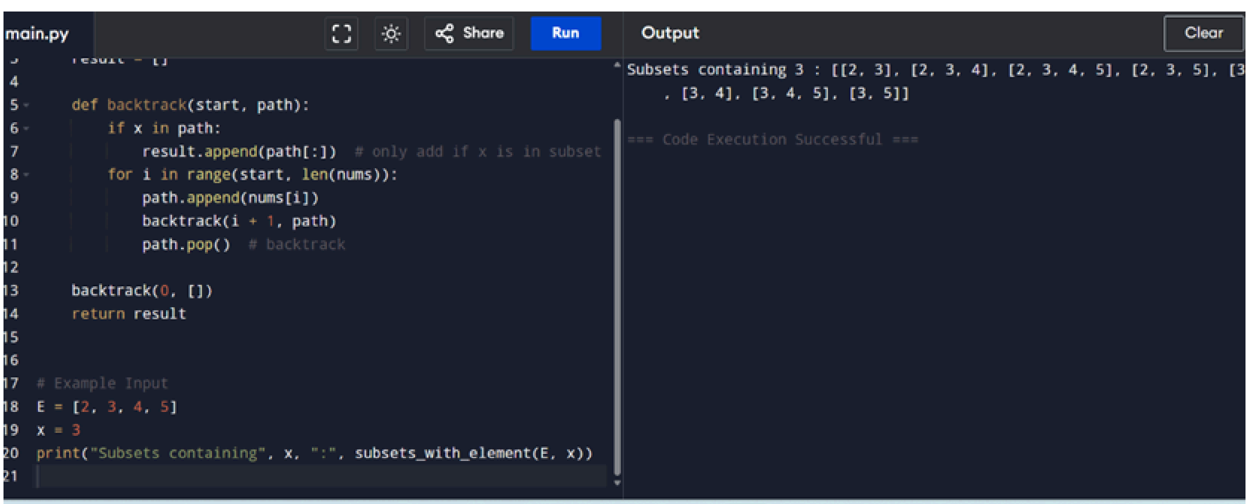
```python
            # skip duplicates
            if i > start and A[i] == A[i - 1]:
                continue
            path.append(A[i])
            backtrack(i + 1, path)
            path.pop()  # backtrack

    backtrack(0, [])
    return result


# Example Input
A = [1, 2, 3]
print("Subsets:", subsets(A))

# Example with duplicates
B = [1, 2, 2]
print("Subsets handling duplicates:", subsets(B))
```

```
Subsets: [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
Subsets handling duplicates: [[], [1], [1, 2], [1, 2, 2], [2], [2, 2]]

=== Code Execution Successful ===
```

RESULT: All subsets are generated successfully.

EXP 16: Subsets Containing a Given Element

AIM: To generate subsets containing a specific element.
CODE



```python
    result = []

    def backtrack(start, path):
        if x in path:
            result.append(path[:])  # only add if x is in subset
        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()  # backtrack

    backtrack(0, [])
    return result


# Example Input
E = [2, 3, 4, 5]
x = 3
print("Subsets containing", x, ":", subsets_with_element(E, x))
```

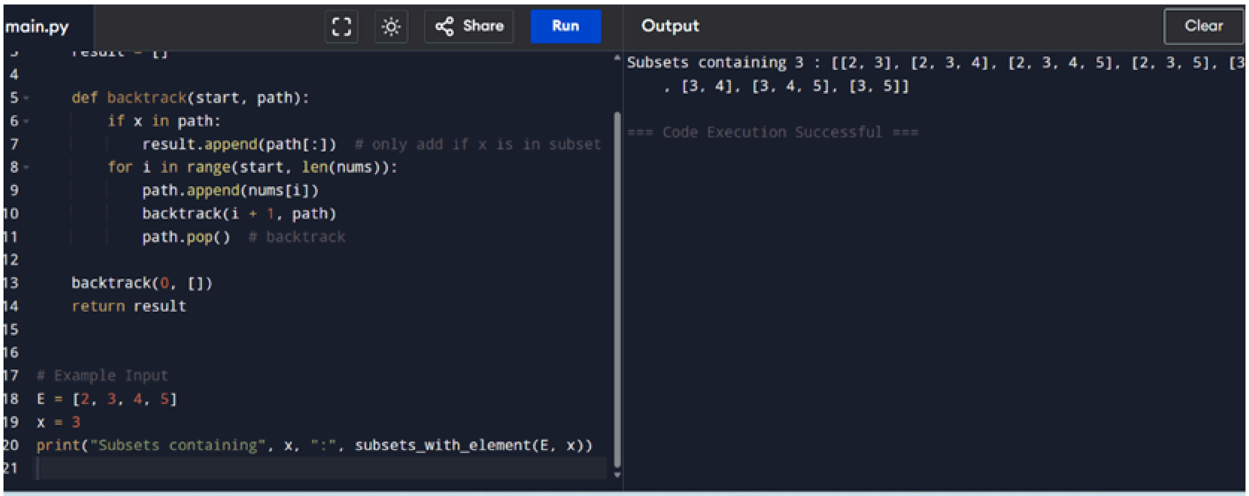
```
Subsets containing 3 : [[2, 3], [2, 3, 4], [2, 3, 4, 5], [2, 3, 5], [3
 , [3, 4], [3, 4, 5], [3, 5]]

=== Code Execution Successful ===
```

RESULT: Required subsets are obtained.

EXP 17: Word Subsets

AIM: To find universal strings from words1.

CODE



```python
    result = []

    def backtrack(start, path):
        if x in path:
            result.append(path[:])  # only add if x is in subset
        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()  # backtrack

    backtrack(0, [])
    return result


# Example Input
E = [2, 3, 4, 5]
x = 3
print("Subsets containing", x, ":", subsets_with_element(E, x))
```


RESULT: All universal strings are correctly identified.