
NAME: S.DHARSHINI

REG NO: 192424258

SUB CODE: CSA0614

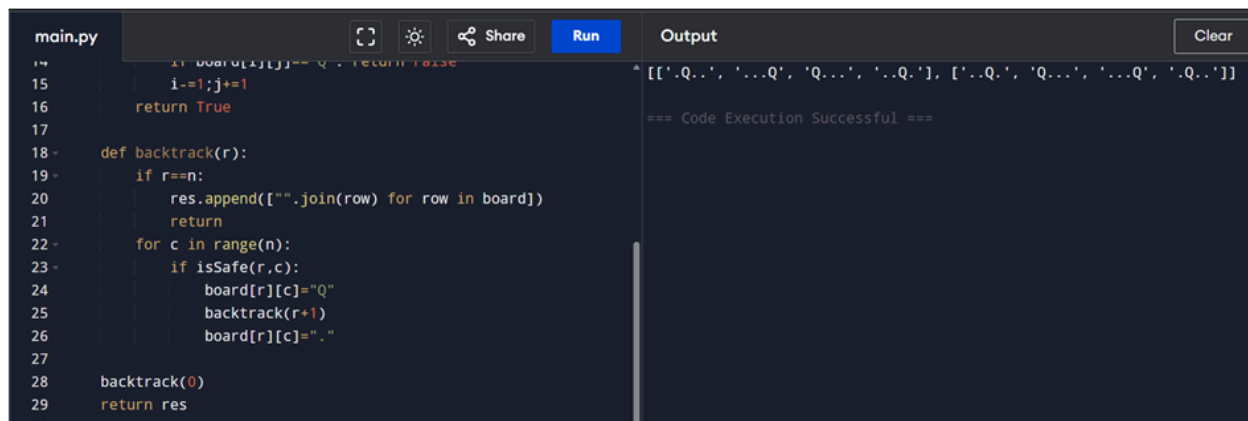
SUB NAME: DESIGN ANALYSIS AND ALGORITHM FOR APPROXIMATION PROBLEM

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:

The image shows a code editor interface with a dark theme. On the left, a file named 'main.py' is open, displaying Python code for solving the N-Queens problem using backtracking. The code includes a recursive function 'backtrack(r)' that iterates through columns 'c' and checks if a queen can be placed at (r, c) without conflicts. If a valid position is found, it proceeds to the next row. The main function 'backtrack(0)' initiates the process, and the results are stored in 'res'. On the right, the 'Output' pane shows the execution result: a list of four board configurations for a 4x4 grid, each represented as a string where '.' is an empty cell and 'Q' is a queen. The output also includes a success message: '=== Code Execution Successful ==='.

```
main.py
14     if board[i][j] == 'Q': return False
15     i -= 1; j += 1
16     return True
17
18 def backtrack(r):
19     if r == n:
20         res.append(''.join(row) for row in board)
21         return
22     for c in range(n):
23         if isSafe(r, c):
24             board[r][c] = 'Q'
25             backtrack(r+1)
26             board[r][c] = '.'
27
28 backtrack(0)
29 return res
30
```

```
[[['.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	Run	Output
<pre> 48 49 # Example input 50 board = [51 ["5","3",".", ".", ".", "7", ".", ".", ".", "."], 52 ["6",".", ".", ".", "1","9","5",".", ".", ".", "."], 53 [".","9","8",".", ".", ".", ".", ".", "6","."], 54 ["8",".", ".", ".", "6",".", ".", ".", "3"], 55 ["4",".", ".", "8",".", "3",".", ".", "1"], 56 ["7",".", ".", ".", "2",".", ".", ".", "6"], 57 [".","6",".", ".", ".", "2","8","."], 58 [".",".", ".", "4","1","9",".", ".", "5"], 59 [".",".", ".", ".", "8",".", ".", "7","9"] 60] 61 62 # Print solved board 63 for row in board: 64 print(row) 65 </pre>	Run	<pre> ['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 === </pre>

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	Run	Output
<pre> 1- def findTargetSumWays(nums, target): 2- count = 0 3- def backtrack(i, total): 4- nonlocal count 5- if i == len(nums): 6- if total == target: 7- count += 1 8- return 9- backtrack(i+1, total + nums[i]) 10- backtrack(i+1, total - nums[i]) 11- backtrack(0, 0) 12- return count 13- 14- print(findTargetSumWays([1,1,1,1,1], 3)) 15- </pre>	Run	<pre> 5 === Code Execution Successful === </pre>

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

main.py	Output
<pre>1 def sumSubarrayMins(arr): 2 ans = 0 3 for i in range(len(arr)): 4 m = arr[i] 5 for j in range(i, len(arr)): 6 m = min(m, arr[j]) 7 ans += m 8 return ans 9 10 print(sumSubarrayMins([3,1,2,4])) 11</pre>	<pre>17 === Code Execution Successful ===</pre>

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

main.py	Output
<pre>1 def combinationSum(candidates, target): 2 res = [] 3 def backtrack(start, path, total): 4 if total == target: 5 res.append(path[:]) 6 return 7 if total > target: 8 return 9 for i in range(start, len(candidates)): 10 backtrack(i, path+[candidates[i]], total 11 +candidates[i]) 12 backtrack(0, [], 0) 13 return res 14 print(combinationSum([2,3,6,7], 7)) 15</pre>	<pre>[[2, 2, 3], [7]] === Code Execution Successful ===</pre>

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

```

main.py
15     path.append(candidates[i])
16     backtrack(i + 1, path, target_remain - candidates[i]
17         ) # i+1 because each number used once
18     path.pop() # backtrack
19     backtrack(0, [], target)
20     return result
21
22
23 # Example 1
24 candidates1 = [10,1,2,7,6,1,5]
25 target1 = 8
26 print(combination_sum2(candidates1, target1))
27
28 # Example 2
29 candidates2 = [2,5,2,1,2]
30 target2 = 5
31 print(combination_sum2(candidates2, target2))

```

```

[[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]
[[1, 2, 2], [5]]

=== Code Execution Successful ===

```

RESULT: Unique combinations summing to target are obtained.

EXP 9: Permutations

AIM: To generate all permutations of a given array.

CODE

```

main.py
12     backtrack(path, remaining[1:], remaining[1:])
13     # undo / backtrack
14     path.pop()
15
16     backtrack([], nums)
17     return result
18
19 # Example 1
20 nums1 = [1,2,3]
21 print(permute(nums1))
22
23 # Example 2
24 nums2 = [0,1]
25 print(permute(nums2))
26
27 # Example 3
28 nums3 = [1]
29 print(permute(nums3))
30

```

```

[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
[[0, 1], [1, 0]]
[[1]]

=== Code Execution Successful ===

```

RESULT: All possible permutations are generated.

EXP 10: Unique Permutations

AIM: To generate unique permutations when duplicate elements exist.

CODE

```

main.py
return
10- for i in range(len(nums)):
11-     # skip used numbers
12-     if used[i]:
13-         continue
14-     # skip duplicates: only use first unused duplicate
15-     if i > 0 and nums[i] == nums[i-1] and not used[i-1]:
16-         continue
17-     used[i] = True
18-     path.append(nums[i])
19-     backtrack(path)
20-     path.pop()
21-     used[i] = False
22-
23- backtrack([])
24- return result
25- nums = [1,1,2]
26- print(permute_unique(nums))
27-
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
22- else:
23-     backtrack(coloring, idx + 1, your_turn_count)
24-     coloring[idx] = 0 # backtrack
25-
26- coloring = [0] * n
27- backtrack(coloring, 0, 0)
28- return max_count[0]
29-
30-
31- # Example Usage
32- n = 4
33- edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]
34- k = 3
35-
36- print("Maximum number of regions you can color:",
      max_regions_you_can_color(n, edges, k))

```

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

```
main.py  [Icons] [Run] [Clear]
1  # skip duplicates
2  if i > start and A[i] == A[i - 1]:
3      continue
4  path.append(A[i])
5  backtrack(i + 1, path)
6  path.pop() # backtrack
7
8  backtrack(0, [])
9  return result
10
11 # Example Input
12 A = [1, 2, 3]
13 print("Subsets:", subsets(A))
14
15 # Example with duplicates
16 B = [1, 2, 2]
17 print("Subsets handling duplicates:", subsets(B))
18
19
20
21
22
23
24
25
26
```

```
Output
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

```
main.py  [Icons] [Run] [Clear]
1  result = []
2
3  def backtrack(start, path):
4      if x in path:
5          result.append(path[:]) # only add if x is in subset
6      for i in range(start, len(nums)):
7          path.append(nums[i])
8          backtrack(i + 1, path)
9          path.pop() # backtrack
10
11  backtrack(0, [])
12  return result
13
14 # Example Input
15 E = [2, 3, 4, 5]
16 x = 3
17 print("Subsets containing", x, ":", subsets_with_element(E, x))
18
19
20
21
```

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

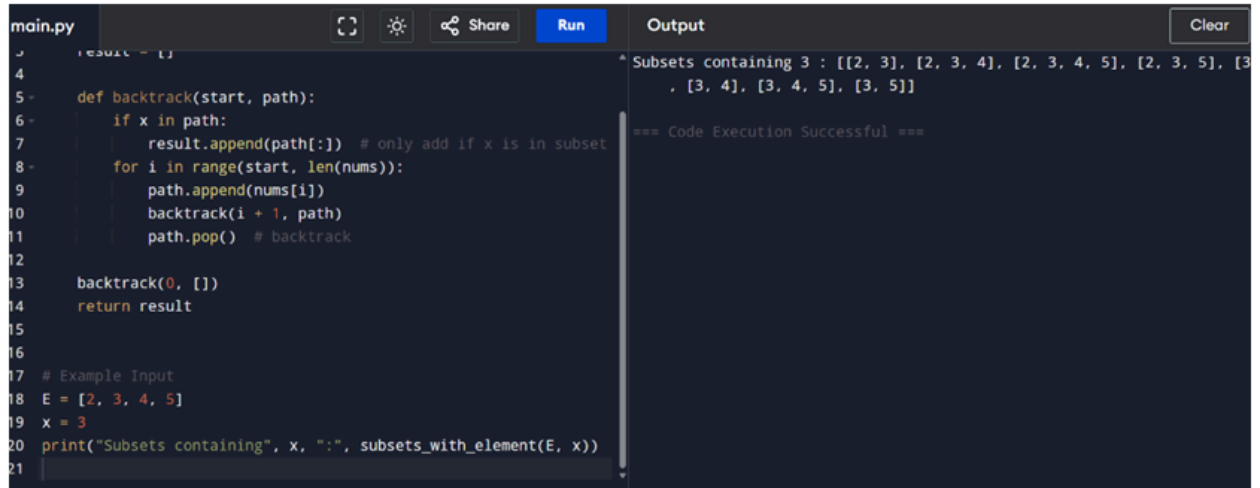
=== Code Execution Successful ===
```

RESULT: Required subsets are obtained.

EXP 17: Word Subsets

AIM: To find universal strings from words1.

CODE



```
main.py  [Icons]  Share  Run  Output  Clear
3 result = []
4
5 def backtrack(start, path):
6     if x in path:
7         result.append(path[:]) # only add if x is in subset
8     for i in range(start, len(nums)):
9         path.append(nums[i])
10        backtrack(i + 1, path)
11        path.pop() # backtrack
12
13    backtrack(0, [])
14    return result
15
16
17 # Example Input
18 E = [2, 3, 4, 5]
19 x = 3
20 print("Subsets containing", x, ":", subsets_with_element(E, x))
21
```

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

=== Code Execution Successful ===

RESULT: All universal strings are correctly identified.