
NAME: S.DHARSHINI

REG NO: 192424258

SUB CODE: CSA0614

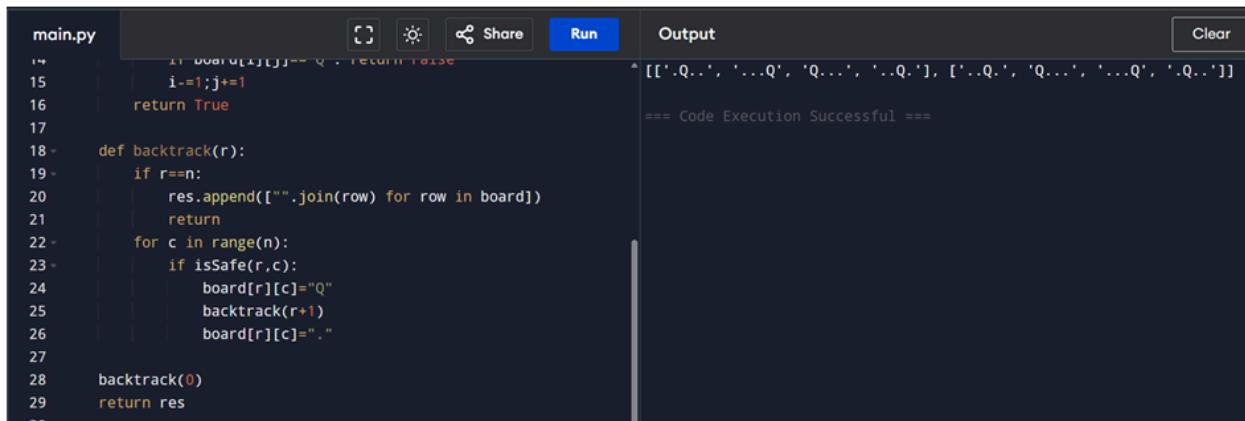
SUB NAME: DESIGN ANALYSIS AND ALGORITHM FOR APPROXIMATION PROBLEM

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 screenshot shows a code editor interface with a dark theme. On the left, the file 'main.py' is open, displaying the following Python code:

```
14     i-=1;j+=1
15     return True
16
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
```

The code implements a backtracking algorithm to solve the N-Queens problem. It uses a 2D list 'board' to represent the chessboard and a list 'res' to store the solutions. The 'isSafe' function checks if it's safe to place a queen at a given position. The 'backtrack' function tries to place queens row by row, backtracking when a conflict is found.

On the right, the 'Output' panel shows the results of running the code. It displays two solutions as 8x8 boards:

```
[["Q . . .", "...Q . .", "Q . . .", "...Q . ."], [". . Q . .", "...Q . .", "...Q . .", "...Q . ."]]
```

Below the boards, a message indicates successful execution: === 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
+--+
48
49 # Example input
50 board = [[5, 3, 0, 0, 7, 0, 0, 0, 0], [6, 0, 0, 1, 9, 5, 0, 0, 0], [0, 9, 8, 0, 0, 0, 0, 6, 0], [8, 0, 0, 0, 6, 0, 0, 0, 3], [4, 0, 8, 0, 3, 0, 0, 1, 0], [7, 0, 0, 0, 2, 0, 0, 0, 6], [0, 6, 0, 0, 0, 0, 2, 8, 0], [0, 0, 0, 4, 1, 9, 0, 0, 5], [0, 0, 0, 8, 0, 0, 7, 9]]
51
52
53
54
55
56
57
58
59
60 solve_sudoku(board)
61
62 # Print solved board
63 for row in board:
64     print(row)
65

```

[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]
[4, 2, 6, 8, 3, 7, 9, 5, 1]
[7, 1, 3, 9, 2, 4, 8, 6, 5]
[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 | Run | Output
+--+
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

```

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

The screenshot shows a code editor interface with a dark theme. The file is named 'main.py'. The code defines a function 'sumSubarrayMins' that iterates through all subarrays of the input array 'arr', calculates the minimum value for each, and sums them up. The output window shows the result '17' and a message '== Code Execution Successful =='.

```
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
```

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

The screenshot shows a code editor interface with a dark theme. The file is named 'main.py'. The code defines a function 'combinationSum' that uses backtracking to find all unique combinations of candidates that sum up to a target. The output window shows the result '[[2, 2, 3], [7]]' and a message '== Code Execution Successful =='.

```
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])
11    backtrack(0, [], 0)
12    return res
13
14 print(combinationSum([2,3,6,7], 7))
15
```

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 [ ] Share Run Output
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
20     backtrack(0, [], target)
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))
```

RESULT: Unique combinations summing to target are obtained.

EXP 9: Permutations

AIM: To generate all permutations of a given array.

CODE

```
main.py [ ] Share Run Output
12         BACKTRACK(path, remainingL-1, remainingL-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
```

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
    else:
        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))
```

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

The screenshot shows a code editor window with a dark theme. The left pane contains the code for 'main.py'. The right pane shows the output of the code execution.

```
main.py
  ...
  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))
```

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

The screenshot shows a code editor window with a dark theme. The left pane contains the code for 'main.py'. The right pane shows the output of the code execution.

```
main.py
  ...
  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))
```

Output:

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

The screenshot shows a code editor interface with a dark theme. On the left is the code file 'main.py' containing a backtracking algorithm to find subsets of a list 'E' that contain a specific element 'x'. On the right is the 'Output' panel which displays the results of the execution.

```
main.py
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
13 # Example Input
14 E = [2, 3, 4, 5]
15 x = 3
16 print("Subsets containing", x, ":", subsets_with_element(E, x))
17
18
19
20
21
```

Output

```
Subsets containing 3 : [[2, 3], [2, 3, 4], [2, 3, 4, 5], [2, 3, 5], [3, 4], [3, 5]]
==== Code Execution Successful ====
Clear
```

RESULT: All universal strings are correctly identified.