

NAME :Harsha Mukundha

REG.NO : 192324070

COURSE CODE : CSA0689.

SUB NAME : DESIGN AND ANALYSIS OF ALGORITHM.

DAY-10



1. 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.

a. Visualization for 4-Queens:

Input: N = 4

Output:

Explanation: Each 'Q' represents a queen, and '.' represents an empty space.

b. Visualization for 5-Queens:

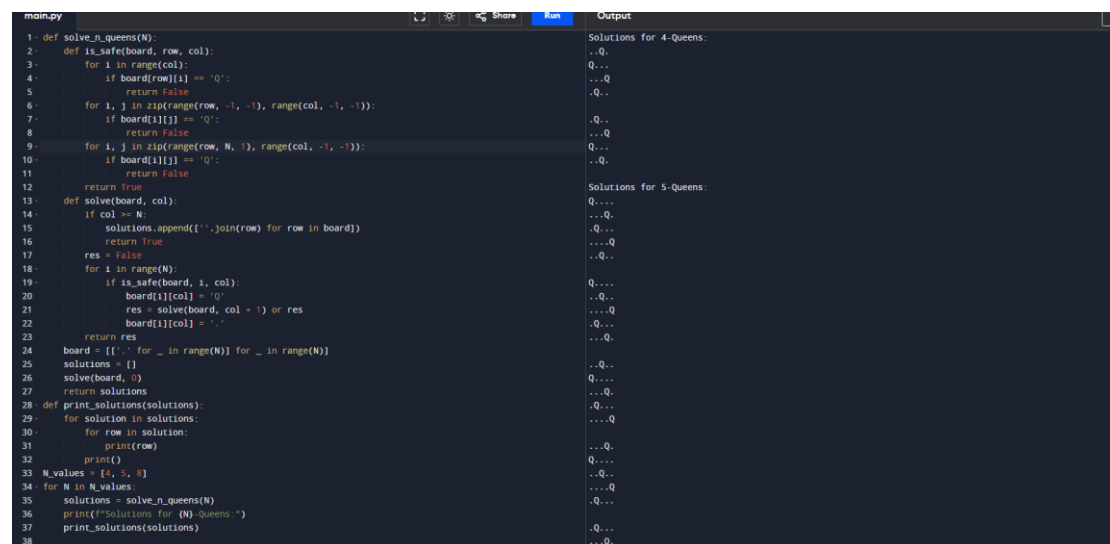
Input: N = 5

Output:

c. Visualization for 8-Queens:

Input: N = 8

Output:



```
main.py
1: def solve_n_queens(N):
2:     def is_safe(board, row, col):
3:         for i in range(col):
4:             if board[row][i] == 'Q':
5:                 return False
6:         for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
7:             if board[i][j] == 'Q':
8:                 return False
9:         for i, j in zip(range(row, N, 1), range(col, -1, -1)):
10:             if board[i][j] == 'Q':
11:                 return False
12:         return True
13:     def solve(board, col):
14:         if col == N:
15:             solutions.append(''.join(row) for row in board)
16:             return True
17:         res = False
18:         for i in range(N):
19:             if is_safe(board, i, col):
20:                 board[i][col] = 'Q'
21:                 res = solve(board, col + 1) or res
22:                 board[i][col] = '.'
23:         return res
24:     board = [['.' for _ in range(N)] for _ in range(N)]
25:     solutions = []
26:     solve(board, 0)
27:     return solutions
28: def print_solutions(solutions):
29:     for solution in solutions:
30:         for row in solution:
31:             print(row)
32:             print()
33: N_values = [4, 5, 8]
34: for N in N_values:
35:     solutions = solve_n_queens(N)
36:     print(f"Solutions for {N} Queens:")
37:     print_solutions(solutions)
38:
```

Solutions for 4-Queens:

```
..Q.
Q...
...Q
.Q..
```

Solutions for 5-Queens:

```
Q....
...Q.
.Q...
....Q
..Q..
.Q...
```

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.

a. 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.


```

["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"]

```

```

main.py
1 def solve_sudoku(board):
2     def is_valid(board, row, col, num):
3         for i in range(9):
4             if board[row][i] == num or board[i][col] == num:
5                 return False
6         start_row, start_col = 3 * (row // 3), 3 * (col // 3)
7         for i in range(3):
8             for j in range(3):
9                 if board[start_row + i][start_col + j] == num:
10                    return False
11        return True
12
13    def solve(board):
14        for row in range(9):
15            for col in range(9):
16                if board[row][col] == '.':
17                    for num in map(str, range(1, 10)):
18                        if is_valid(board, row, col, num):
19                            board[row][col] = num
20                            if solve(board):
21                                return True
22                            board[row][col] = '.'
23        return True
24    return True
25
26    solve(board)
27    return board
28
29    board = [
30        ['5','3','.','.','7','.','.','.','.'],
31        ['6','.','.','1','9','5','.','.','.'],
32        ['.','.','8','.','.','.','6','.','.'],
33        ['8','.','.','6','.','.','.','3','.'],
34        ['4','.','.','8','.','.','3','.','.'],
35        ['7','.','.','2','.','.','.','.','6'],
36        ['.','6','.','.','.','.','2','.','8','.'],
37        ['.','.','4','.','1','9','.','.','5'],
38        ['.','.','.','8','.','.','.','7','.']
39    ]
40    solved_board = solve_sudoku(board)
41    for row in solved_board:
42        print(row)

```

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"]

```

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"]]

```

main.py	Run	Output
<pre> 2- def is_valid(board, row, col, num): 3- for i in range(9): 4- if board[row][i] == num or board[i][col] == num: 5- return False 6- start_row, start_col = 3 * (row // 3), 3 * (col // 3) 7- for i in range(3): 8- for j in range(3): 9- if board[start_row + i][start_col + j] == num: 10- return False 11- return True 12- 13- def solve(board): 14- for row in range(9): 15- for col in range(9): 16- if board[row][col] == '.': 17- for num in map(str, range(1, 10)): 18- if is_valid(board, row, col, num): 19- board[row][col] = num 20- if solve(board): 21- return True 22- board[row][col] = '.' 23- return False 24- return True 25- 26- solve(board) 27- return board 28- 29- board = [30- ["5","3",".", "7",".", "6",".", "8","."], 31- ["6",".", "1","9","5",".", "3","4","8"], 32- ["1","9","8",".", "3","4","2","5","6","7"], 33- ["8",".", "6",".", "2","8",".", "3"], 34- ["4",".", "8",".", "3",".", "1"], 35- ["7",".", "2",".", "6"], 36- [".", "6",".", "2","8","."], 37- [".", "4","1","9",".", "5"], 38- [".", "8",".", "7","9"] 39-] 40- 41- solved_board = solve_sudoku(board) 42- for row in solved_board: 43- print(row) 44- </pre>	<div>Run</div>	<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>

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.

$-1 + 1 + 1 + 1 + 1 = 3$

$+1 - 1 + 1 + 1 + 1 = 3$

$+1 + 1 - 1 + 1 + 1 = 3$

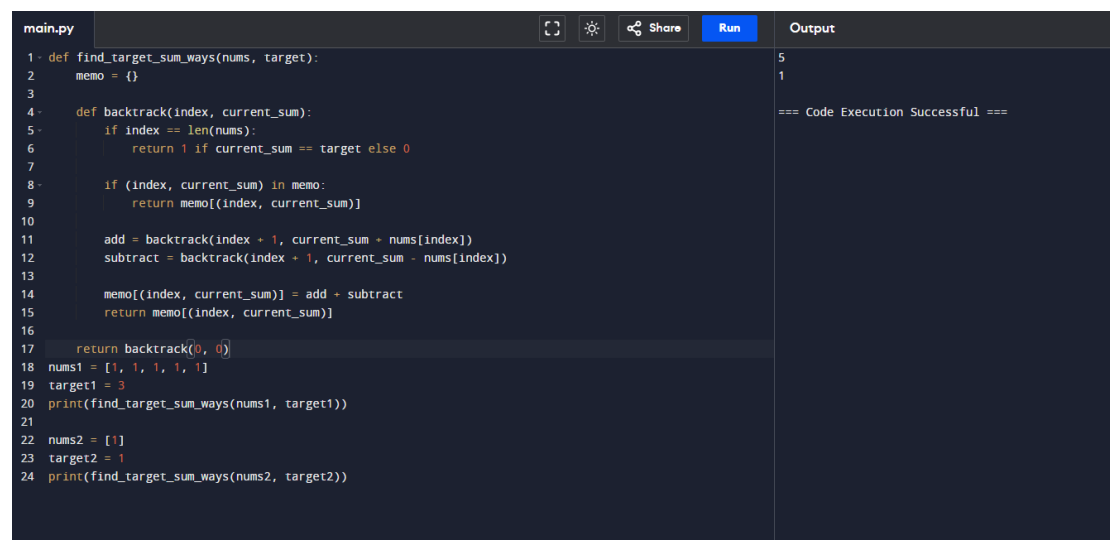
$+1 + 1 + 1 - 1 + 1 = 3$

$+1 + 1 + 1 + 1 - 1 = 3$

Example 2:

Input: `nums = [1]`, `target = 1`

Output: 1



```
main.py
1 def find_target_sum_ways(nums, target):
2     memo = {}
3
4     def backtrack(index, current_sum):
5         if index == len(nums):
6             return 1 if current_sum == target else 0
7
8         if (index, current_sum) in memo:
9             return memo[(index, current_sum)]
10
11         add = backtrack(index + 1, current_sum + nums[index])
12         subtract = backtrack(index + 1, current_sum - nums[index])
13
14         memo[(index, current_sum)] = add + subtract
15         return memo[(index, current_sum)]
16
17     return backtrack(0, 0)
18
19 nums1 = [1, 1, 1, 1, 1]
20 target1 = 3
21 print(find_target_sum_ways(nums1, target1))
22
23 nums2 = [1]
24 target2 = 1
25 print(find_target_sum_ways(nums2, target2))
```

Output

```
5
1
=== Code Execution Successful ===
```

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 $10^9 + 7$.

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].

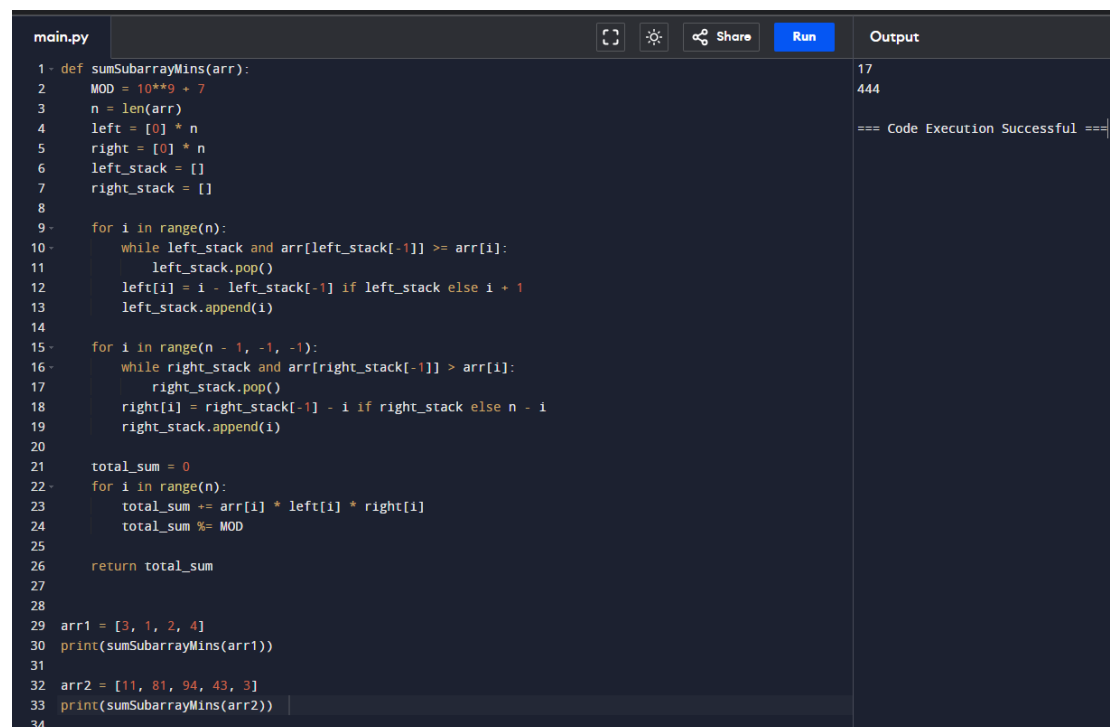
Minimums are 3, 1, 2, 4, 1, 1, 2, 1, 1, 1.

Sum is 17.

Example 2:

Input: arr = [11,81,94,43,3]

Output: 444



```
main.py
1 - def sumSubarrayMins(arr):
2     MOD = 10**9 + 7
3     n = len(arr)
4     left = [0] * n
5     right = [0] * n
6     left_stack = []
7     right_stack = []
8
9     for i in range(n):
10         while left_stack and arr[left_stack[-1]] >= arr[i]:
11             left_stack.pop()
12         left[i] = i - left_stack[-1] if left_stack else i + 1
13         left_stack.append(i)
14
15     for i in range(n - 1, -1, -1):
16         while right_stack and arr[right_stack[-1]] > arr[i]:
17             right_stack.pop()
18         right[i] = right_stack[-1] - i if right_stack else n - i
19         right_stack.append(i)
20
21     total_sum = 0
22     for i in range(n):
23         total_sum += arr[i] * left[i] * right[i]
24         total_sum %= MOD
25
26     return total_sum
27
28
29 arr1 = [3, 1, 2, 4]
30 print(sumSubarrayMins(arr1))
31
32 arr2 = [11, 81, 94, 43, 3]
33 print(sumSubarrayMins(arr2))
34
```

Output

```
17
444
=== Code Execution Successful ===
```

7. 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]]

```
main.py  [Icons] [Share] [Run] [Output]
1 def combinationSum(candidates, target):
2     def backtrack(remaining, start, path, result):
3         if remaining == 0:
4             result.append(path)
5             return
6         elif remaining < 0:
7             return
8
9         for i in range(start, len(candidates)):
10            backtrack(remaining - candidates[i], i, path + [candidates[i]], result)
11
12    result = []
13    backtrack(target, 0, [], result)
14    return result
15
16 candidates1 = [2, 3, 6, 7]
17 target1 = 7
18 print(combinationSum(candidates1, target1))
19
20 candidates2 = [2, 3, 5]
21 target2 = 8
22 print(combinationSum(candidates2, target2))
23
```

[[2, 2, 3], [7]]
[[2, 2, 2, 2], [2, 3, 3], [3, 5]]
=== Code Execution Successful ===

8. 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:

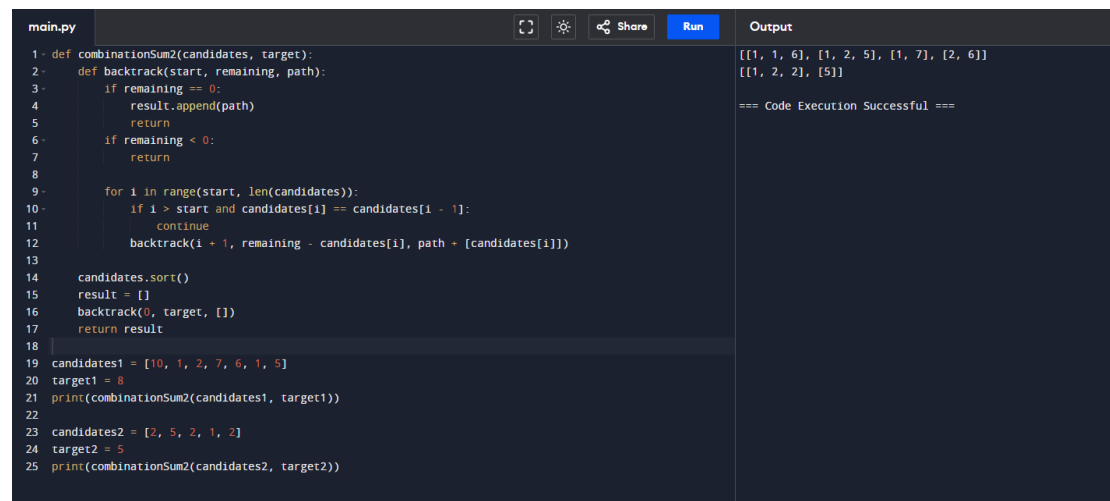
```
[  
[1,1,6],  
[1,2,5],  
[1,7],  
[2,6]  
]
```

Example 2:

Input: candidates = [2,5,2,1,2], target = 5

Output:

```
[  
[1,2,2],  
[5]  
]
```



The screenshot shows a code editor with a file named 'main.py'. The code defines a function 'combinationSum2' that takes 'candidates' and 'target' as input. It uses a recursive backtracking approach to find all combinations of candidates that sum up to the target. The code includes a 'backtrack' function that explores different paths, adding elements to the 'path' list and subtracting their values from the 'remaining' target. The 'main' part of the code tests the function with two sets of inputs: candidates1 = [10, 1, 2, 7, 6, 1, 5] and target1 = 8, and candidates2 = [2, 5, 2, 1, 2] and target2 = 5. The output of the code execution is displayed on the right, showing the combinations [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]] for the first test case and [[1, 2, 2], [5]] for the second test case. A message '=== Code Execution Successful ===' is also shown.

```
main.py  
1 def combinationSum2(candidates, target):  
2     def backtrack(start, remaining, path):  
3         if remaining == 0:  
4             result.append(path)  
5             return  
6         if remaining < 0:  
7             return  
8  
9         for i in range(start, len(candidates)):  
10            if i > start and candidates[i] == candidates[i - 1]:  
11                continue  
12            backtrack(i + 1, remaining - candidates[i], path + [candidates[i]])  
13  
14        candidates.sort()  
15        result = []  
16        backtrack(0, target, [])  
17        return result  
18  
19 candidates1 = [10, 1, 2, 7, 6, 1, 5]  
20 target1 = 8  
21 print(combinationSum2(candidates1, target1))  
22  
23 candidates2 = [2, 5, 2, 1, 2]  
24 target2 = 5  
25 print(combinationSum2(candidates2, target2))  
  
Output  
[[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]  
[[1, 2, 2], [5]]  
=== Code Execution Successful ===
```

9. 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]]`



```
main.py
1 def permute(nums):
2     def backtrack(path, remaining):
3         if not remaining:
4             result.append(path)
5             return
6
7         for i in range(len(remaining)):
8             backtrack(path + [remaining[i]], remaining[:i] + remaining[i+1:])
9
10    result = []
11    backtrack([], nums)
12    return result
13
14    nums1 = [1, 2, 3]
15    print(permute(nums1))
16
17    nums2 = [0, 1]
18    print(permute(nums2))
19    nums3 = [1]
20    print(permute(nums3))
21
```

Output

```
[[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 ===
```

10. 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]]`

```
1- def permuteUnique(nums):
2-     def backtrack(path, used):
3-         if len(path) == len(nums):
4-             result.append(path)
5-             return
6-
7-         for i in range(len(nums)):
8-             if used[i]:
9-                 continue
10-            if i > 0 and nums[i] == nums[i - 1] and not used[i - 1]:
11-                continue
12-
13-            used[i] = True
14-            backtrack(path + [nums[i]], used)
15-            used[i] = False
16-
17-         nums.sort()
18-         result = []
19-         used = [False] * len(nums)
20-         backtrack([], used)
21-         return result
22-
23-     nums1 = [1, 1, 2]
24-     print(permuteUnique(nums1))
25-
26-     nums2 = [1, 2, 3]
27-     print(permuteUnique(nums2))
28-
```

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

=== Code Execution Successful ===