DESIGN AND ANALYSIS OF ALGORITHM

NAME : MALREDDY.P

REGISTER NO: 192372015

DAY -10 PROGRAMS

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:

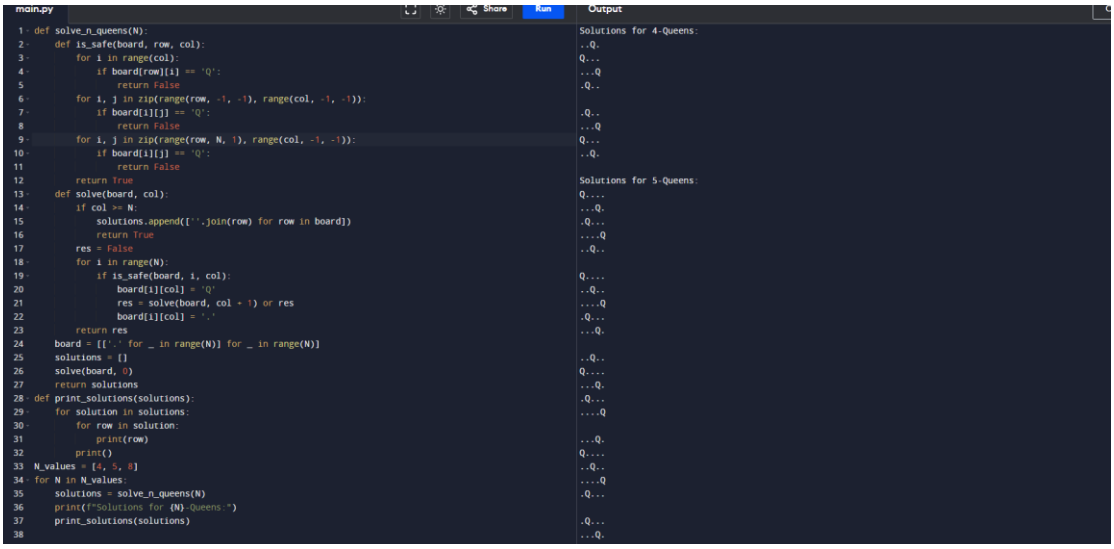
Input: N = 5

Output:

c. Visualization for 8-Queens:

Input: N = 8

Output:



```python
def solve_n_queens(N):
    def is_safe(board, row, col):
        for i in range(col):
            if board[row][i] == 'Q':
                return False
        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
            if board[i][j] == 'Q':
                return False
        for i, j in zip(range(row, N, 1), range(col, -1, -1)):
            if board[i][j] == 'Q':
                return False
        return True
    def solve(board, col):
        if col >= N:
            solutions.append([''.join(row) for row in board])
            return True
        res = False
        for i in range(N):
            if is_safe(board, i, col):
                board[i][col] = 'Q'
                res = solve(board, col + 1) or res
                board[i][col] = '.'
        return res
    board = [['.' for _ in range(N)] for _ in range(N)]
    solutions = []
    solve(board, 0)
    return solutions
def print_solutions(solutions):
    for solution in solutions:
        for row in solution:
            print(row)
        print()
N_values = [4, 5, 8]
for N in N_values:
    solutions = solve_n_queens(N)
    print(f"Solutions for {N}-Queens:")
    print_solutions(solutions)
```

```
Solutions for 4-Queens:
..Q.
Q...
...Q
.Q..

.Q..
...Q
Q...
..Q.

Solutions for 5-Queens:
Q....
...Q.
.Q...
....Q
..Q..

Q....
..Q..
....Q
.Q...
...Q.

..Q..
Q....
...Q.
.Q...
....Q

...Q.
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.

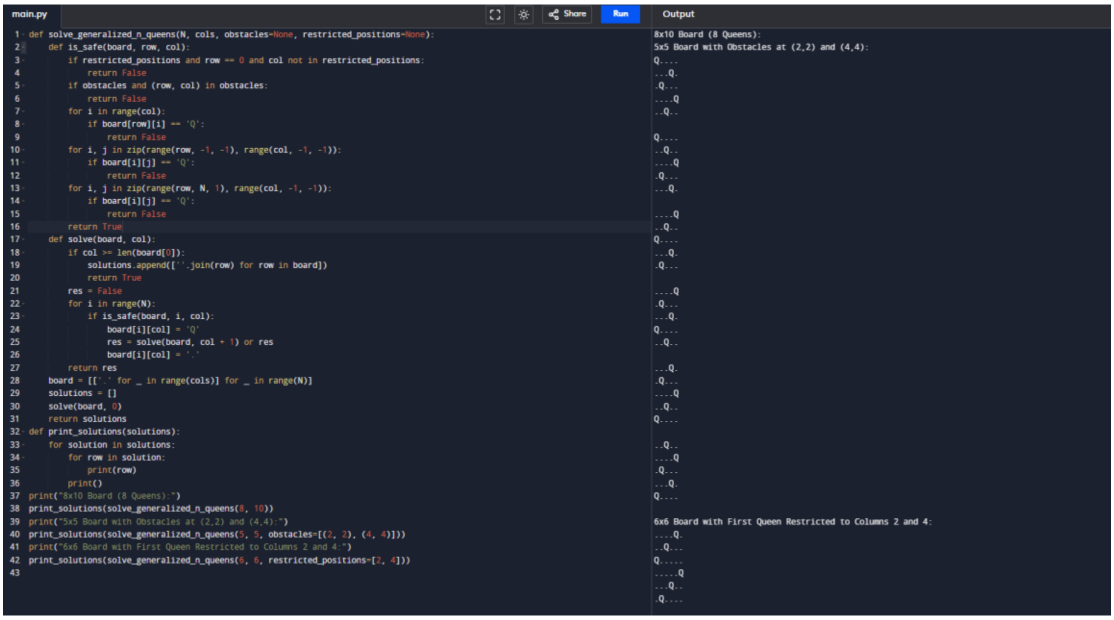
c. 6×6 Board with Restricted Positions:

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.



main.py

```python
def solve_generalized_n_queens(N, cols, obstacles=None, restricted_positions=None):
    def is_safe(board, row, col):
        if restricted_positions and row == 0 and col not in restricted_positions:
            return False
        if obstacles and (row, col) in obstacles:
            return False
        for i in range(col):
            if board[row][i] == 'Q':
                return False
        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
            if board[i][j] == 'Q':
                return False
        for i, j in zip(range(row, N, 1), range(col, -1, -1)):
            if board[i][j] == 'Q':
                return False
        return True
    def solve(board, col):
        if col >= len(board[0]):
            solutions.append([''.join(row) for row in board])
            return True
        res = False
        for i in range(N):
            if is_safe(board, i, col):
                board[i][col] = 'Q'
                res = solve(board, col + 1) or res
                board[i][col] = '.'
        return res
    board = [['.' for _ in range(cols)] for _ in range(N)]
    solutions = []
    solve(board, 0)
    return solutions
def print_solutions(solutions):
    for solution in solutions:
        for row in solution:
            print(row)
        print()
print("8x10 Board (8 Queens):")
print_solutions(solve_generalized_n_queens(8, 10))
print("5x5 Board with Obstacles at (2,2) and (4,4):")
print_solutions(solve_generalized_n_queens(5, 5, obstacles=[(2, 2), (4, 4)]))
print("6x6 Board with First Queen Restricted to Columns 2 and 4:")
print_solutions(solve_generalized_n_queens(6, 6, restricted_positions=[2, 4]))
```

Output

```
8x10 Board (8 Queens):
5x5 Board with Obstacles at (2,2) and (4,4):
Q....
...Q.
.Q...
....Q
..Q..

Q....
..Q..
....Q
.Q...
...Q.

....Q
..Q..
Q....
...Q.
.Q...

....Q
.Q...
...Q.
Q....
..Q..

...Q.
.Q...
....Q
..Q..
Q....

..Q..
....Q
.Q...
...Q.
Q....

6x6 Board with First Queen Restricted to Columns 2 and 4:
....Q.
..Q...
Q.....
.....Q
...Q..
.Q....
```

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"],

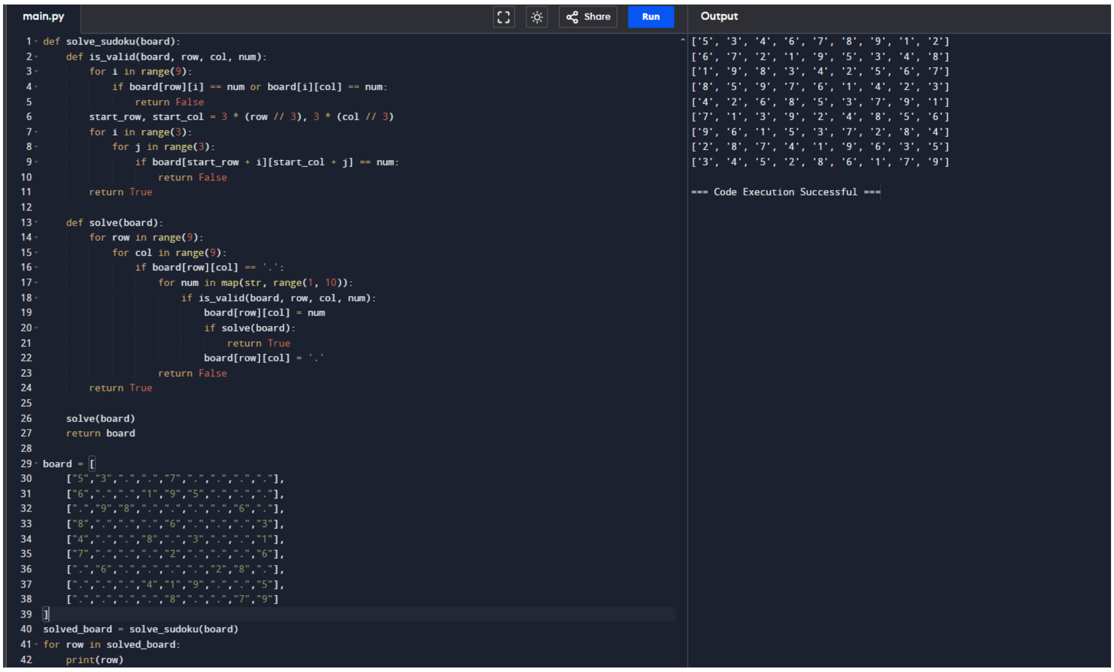
["4","2","6","8","5","3","7","9","1"],

["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"]]



main.py

```
def solve_sudoku(board):
    def is_valid(board, row, col, num):
        for i in range(9):
            if board[row][i] == num or board[i][col] == num:
                return False
        start_row, start_col = 3 * (row // 3), 3 * (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(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(board):
                                return True
                            board[row][col] = '.'
                    return False
        return True

    solve(board)
    return 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"]
]
solved_board = solve_sudoku(board)
for row in solved_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 ===
```

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"],

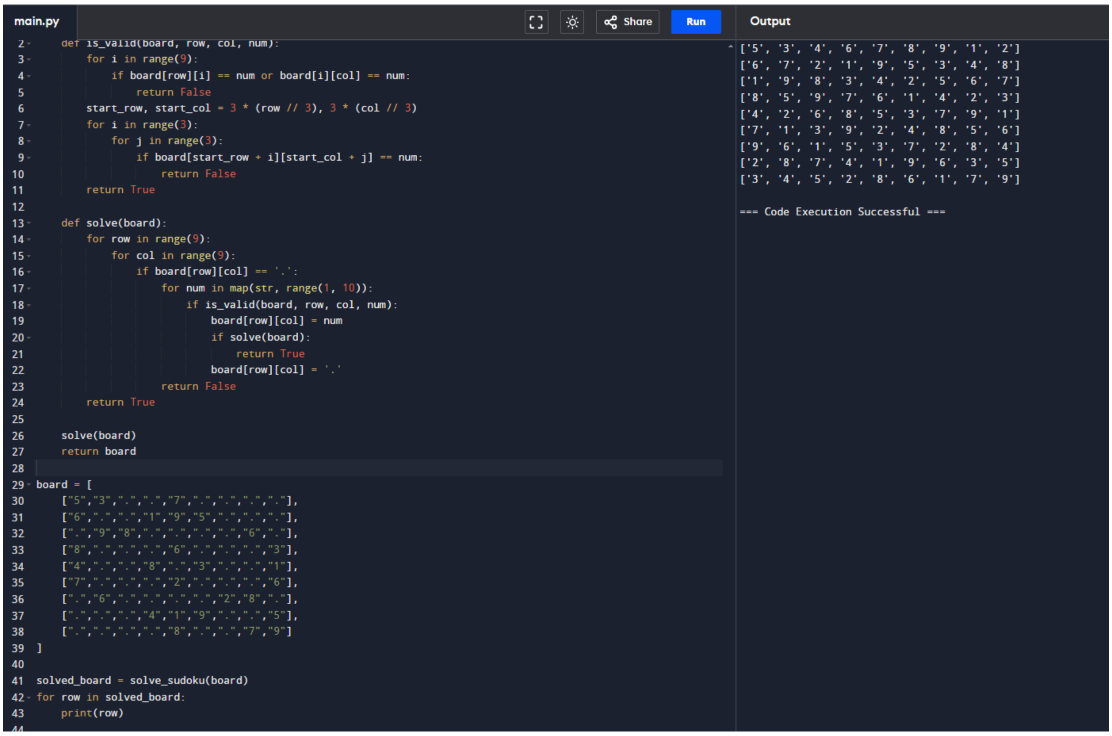
["4","2","6","8","5","3","7","9","1"],

["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"]]



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

```python
def find_target_sum_ways(nums, target):
    memo = {}

    def backtrack(index, current_sum):
        if index == len(nums):
            return 1 if current_sum == target else 0

        if (index, current_sum) in memo:
            return memo[(index, current_sum)]

        add = backtrack(index + 1, current_sum + nums[index])
        subtract = backtrack(index + 1, current_sum - nums[index])

        memo[(index, current_sum)] = add + subtract
        return memo[(index, current_sum)]

    return backtrack(0, 0)
nums1 = [1, 1, 1, 1, 1]
target1 = 3
print(find_target_sum_ways(nums1, target1))

nums2 = [1]
target2 = 1
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 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

main.py

```python
def sumSubarrayMins(arr):
    MOD = 10**9 + 7
    n = len(arr)
    left = [0] * n
    right = [0] * n
    left_stack = []
    right_stack = []

    for i in range(n):
        while left_stack and arr[left_stack[-1]] >= arr[i]:
            left_stack.pop()
        left[i] = i - left_stack[-1] if left_stack else i + 1
        left_stack.append(i)

    for i in range(n - 1, -1, -1):
        while right_stack and arr[right_stack[-1]] > arr[i]:
            right_stack.pop()
        right[i] = right_stack[-1] - i if right_stack else n - i
        right_stack.append(i)

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

    return total_sum


arr1 = [3, 1, 2, 4]
print(sumSubarrayMins(arr1))

arr2 = [11, 81, 94, 43, 3]
print(sumSubarrayMins(arr2))
```

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]]4. COMBINATION SUM 2:

main.py

```python
def combinationSum(candidates, target):
    def backtrack(remaining, start, path, result):
        if remaining == 0:
            result.append(path)
            return
        elif remaining < 0:
            return

        for i in range(start, len(candidates)):
            backtrack(remaining - candidates[i], i, path + [candidates[i]], result)

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

candidates1 = [2, 3, 6, 7]
target1 = 7
print(combinationSum(candidates1, target1))

candidates2 = [2, 3, 5]
target2 = 8
print(combinationSum(candidates2, target2))
```

Output

```
[[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]

]

main.py

```python
def combinationSum2(candidates, target):
    def backtrack(start, remaining, path):
        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
            backtrack(i + 1, remaining - candidates[i], path + [candidates[i]])

    candidates.sort()
    result = []
    backtrack(0, target, [])
    return result

candidates1 = [10, 1, 2, 7, 6, 1, 5]
target1 = 8
print(combinationSum2(candidates1, target1))

candidates2 = [2, 5, 2, 1, 2]
target2 = 5
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:

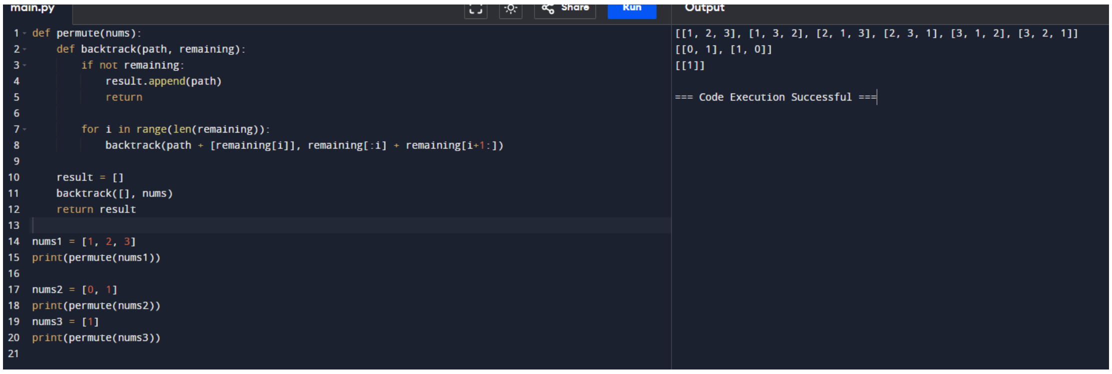
Input: nums = [0,1]

Output: [[0,1],[1,0]]

Example 3:

Input: nums = [1]

Output: [[1]]



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]]

```python
def permuteUnique(nums):
    def backtrack(path, used):
        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
            backtrack(path + [nums[i]], used)
            used[i] = False

    nums.sort()
    result = []
    used = [False] * len(nums)
    backtrack([], used)
    return result

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

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

```
[[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 ===
```