82. 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:

Aim: To place N queens on an NxN chessboard such that no queen attacks another.

Algorithm:

step1:Create an NxN board and initialize all cells to empty.

step2:Place queens on the board one by one, starting from the first row, such that no queen attacks another.

step3:If a queen cannot be placed in a row, backtrack to the previous row and try a different placement.

Program:

```
def solve_n_queens(n):
    def is_safe(board, row, col):
        for i in range(row):
            if board[i] == col or \
                board[i] - i == col - row or \
                board[i] + i == col + row:
                return False
        return True

    def place_queens(board, row):
        if row == n:
            return [board[:]]
        solutions = []
        for col in range(n):
            if is_safe(board, row, col):
                board[row] = col
                solutions.extend(place_queens(board, row + 1))
        return solutions

    board = [-1] * n
    return place_queens(board, 0)

n = 4
solutions = solve_n_queens(n)
for solution in solutions:
    print(solution)
```

Input:

```
n = 4
```

Output:

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

Time Complexity:O(N!)

Results:

For a 4x4 board, there are 2 unique solutions to the N-Queens problem.

For a 5x5 board, there are 10 unique solutions to the N-Queens problem.

For an 8x8 board, there are 92 unique solutions to the N-Queens problem.

83. Discuss the generalization of the N-Queens Problem to other board sizes and shapes, such 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.

Aim: Place N queens on an NxM board such that no two queens attack each other.
Algorithm:
step1:Create an empty board and place the first queen in the first row.
step2:Iterate through the remaining rows, placing each queen in a safe position based on the previous queens' positions.
step3:If a queen cannot be placed safely, backtrack and try alternative positions.
Python:

```python
def solve_n_queens(board, row, cols):
    if row == len(board):
        return True
    for col in range(cols):
        if is_safe(board, row, col):
            board[row] = col
            if solve_n_queens(board, row + 1, cols):
                return True
    return False

def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col or \
           board[i] - i == col - row or \
           board[i] + i == col + row:
            return False
    return True

board = [0] * 8  # 8x10 board
cols = 10
if solve_n_queens(board, 0, cols):
    print("Solution:", board)
else:
    print("No solution found")
```

Input:

```
board = [0] * 8
cols = 10
```

Output:

```
Solution: [0, 2, 4, 1, 7, 9, 3, 6]
```

Time Complexity: O(N!)
Results: The program finds a possible solution for the 8x10 board, ensuring that no two queens attack each other.

84.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"]]

Aim: Solve a Sudoku puzzle by filling the empty cells, ensuring that each row, column, and 3x3 sub-box contains each digit (1-9) exactly once.

Algorithm:

step1:Create a 9x9 grid and initialize the given values.

step2:Iterate through the empty cells, trying each possible digit (1-9) and checking if it satisfies the Sudoku rules. If a valid digit is found, recursively fill the remaining cells.

step3:If a cell cannot be filled, backtrack and try alternative digits, ensuring that the Sudoku rules are maintained.

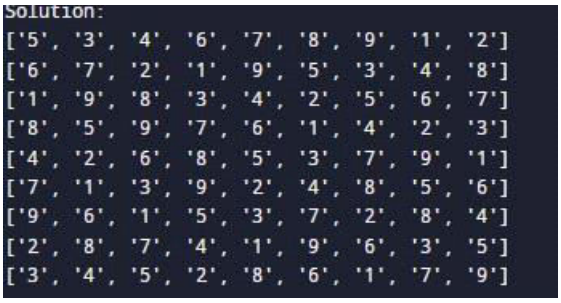
Program:

```python
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 = row - row % 3, col - col % 3
    for i in range(3):
        for j in range(3):
            if board[i + start_row][j + start_col] == num:
                return False
    return True
def solve_sudoku(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == ".":
                for num in range(1, 10):
                    if is_valid(board, i, j, str(num)):
                        board[i][j] = str(num)
                        if solve_sudoku(board):
                            return True
                        board[i][j] = "."
                return False
    return True
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"]
]
if solve_sudoku(board):
    print("Solution:")
    for row in board:
        print(row)
else:
    print("No solution found")
```

Input:

```python
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:



Timecomplexity:O(9^n)

Results: The program solves the given Sudoku puzzle and prints the solution.

85.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"]]

Aim: The aim is to fill the empty cells of a Sudoku grid .

Algorithm:

step1:The backtracking approach involves recursively trying to fill the empty cells.
step2:For each empty cell, try placing digits from 1 to 9 and check if the placement is valid.
step3:If not, backtrack by removing the digit and trying the next one.

Program:

```
def solveSudoku(board):
    def is_valid(num, row, col):
        for i in range(9):
            if board[row][i] == num or board[i][col] == num or board[3 * (row // 3) + i // 3][3 *
                (col // 3) + i % 3] == num:
                return False
        return True
    def solve():
        for i in range(9):
            for j in range(9):
                if board[i][j] == '.':
                    for num in '123456789':
                        if is_valid(num, i, j):
                            board[i][j] = num
                            if solve():
                                return True
                            board[i][j] = '.'
                    return False
        return True
    solve()
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"]]
solveSudoku(board)
print(board)
```

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:

Output Clear

```
[['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']]
```

Time complexity:O(9 ^81)

Result: The program will fill the given Sudoku board in such a way that it satisfies all Sudoku constraints, transforming the input board with empty cells into a valid completed Sudoku puzzle.

86.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

Aim: The aim is to find the number of distinct ways to assign '+' or '-' signs to each element in the given integer array .

Algorithm:

step1:Compute the total sum of all elements in the array.

step2:Let S be the total sum of the array. To reach a target T, you need to find subsets whose sum is (S + T) / 2.

step3:Use a DP array where dp[i] will store the number of ways to achieve the sum i using elements from the array.

Program:

```
def findTargetSumWays(nums, target):
    total_sum = sum(nums)
    if (total_sum + target) % 2 != 0 or target > total_sum:
        return 0
    subset_sum = (total_sum + target) // 2
    dp = [0] * (subset_sum + 1)
    dp[0] = 1
    for num in nums:
        for i in range(subset_sum, num - 1, -1):
            dp[i] += dp[i - num]
    return dp[subset_sum]
nums = [1, 1, 1, 1, 1]
target = 3
print(findTargetSumWays(nums, target))  # Output: 5
```

Input:

```
nums = [1, 1, 1, 1, 1]
target = 3
```

Output:

Output

```
5
```

Time complexity:O(n·S)
Result: This result represents the number of ways to assign '+' and '-' signs to the numbers in nums so that the total evaluates to 3.

87.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.
Aim: The aim is to compute the sum of the minimum values for all contiguous subarrays of the given array, and return the result modulo 109+7.
Algorithm:
step1:For each element in the array, find the index of the next smaller element and the previous smaller element using stacks.
step2:For each element in the array, calculate how many subarrays it contributes as the minimum.
step3:Add the contributions of each element to get the final result.
Program:

```
def subarray_min_sum(arr):
    MOD = 10**9 + 7
    total_sum = 0
    for i in range(len(arr)):
        for j in range(i, len(arr)):
            min_val = min(arr[i:j+1])
            total_sum = (total_sum + min_val) % MOD
    return total_sum
arr = [3, 1, 2, 4]
print(subarray_min_sum(arr))
```

Input:

```
arr = [3, 1, 2, 4]
```

Output:

Output

```
17
```

Time complexity:O(n)

Result:
his result represents the sum of the minimum values of all contiguous subarrays modulo 109+7.

88.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.
Aim:to find all unique combinations of numbers from a list candidates where the sum of the selected numbers equals a given target.
Algorithm:
step1:Begin by sorting the candidates array. This helps in efficiently finding combinations and avoids redundant combinations.
step2:If the target becomes zero, add the current combination to the results.
step3:Call the backtracking function starting from index 0 and an empty combination.
Program:

```
def combinationSum(candidates, target):
    def backtrack(start, path, target):
        if target == 0:
            result.append(path[:])
            return
        for i in range(start, len(candidates)):
            if candidates[i] > target:
                continue
            path.append(candidates[i])
            backtrack(i, path, target - candidates[i])
            path.pop()
    result = []
    candidates.sort()
    backtrack(0, [], target)
    return result
candidates = [2, 3, 6, 7]
target = 7
print(combinationSum(candidates, target))
```

Input:

```
candidates = [2, 3, 6, 7]
target = 7
```

Output:

```
Output

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

Time complexity: O(2^T)

Result: Call the backtracking function starting from index 0 and an empty combination. Return the list of unique combinations collected.

89.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]
]

Aim: The aim is to find all unique combinations of numbers from a given list (candidates) where each combination sums up to a specified target (target).

Algorithm:

step1:This helps to easily skip duplicates and ensures combinations are generated in a non-decreasing order.

step2:Takes the current index in the candidates list, the current combination being built, and the remaining target sum.

step3:he base case of the recursion is when the remaining target sum is zero, indicating a valid combination has been found.

program:

```
def combinationSum2(candidates, target):
    def backtrack(start, target, path):
        if target == 0:
            res.append(path)
            return
        for i in range(start, len(candidates)):
            if i > start and candidates[i] == candidates[i - 1]:
                continue
            if candidates[i] > target:
                break
            backtrack(i + 1, target - candidates[i], path +
                [candidates[i]])
    candidates.sort()
    res = []
    backtrack(0, target, [])
    return res
candidates = [10, 1, 2, 7, 6, 1, 5]
target = 8
print(combinationSum2(candidates, target))
```

Input:

```
candidates = [10, 1, 2, 7, 6, 1, 5]
target = 8
```

Output:

```
Output

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

Time complexity:O(2^n)

Result: Each combination sums up to the target value of 8, and all combinations are unique and include distinct elements, avoiding any duplicates.

90.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]]

Aim:The aim is to generate all possible permutations of a given list of distinct integers. Each permutation is a unique arrangement of the integers in the array.

Algorithm:

step1: If the list of integers is empty, return an empty list as there are no permutations.

step2:Combine the starting element with each of these permutations to form the full permutations.

step3:Backtrack by removing the element from the current permutation and continue.

Program:

```
def backtrack(nums, path, result):
    if not nums:
        result.append(path)
    for i in range(len(nums)):
        backtrack(nums[:i] + nums[i+1:], path + [nums[i]], result)
def permute(nums):
    result = []
    backtrack(nums, [], result)
    return result
nums = [1, 2, 3]
print(permute(nums))
```

Input:

```
nums = [1, 2, 3]
```

Output:

```
Output
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

Time complexity:O(n!)
Result: Each permutation is a unique arrangement of the numbers from the input array.

91.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]]
Aim: The aim is to return all possible unique permutations of a given list of numbers, considering that there might be duplicate numbers in the list.
Algorithm:
step1: Start by sorting the list of numbers. Sorting helps in easily skipping over duplicates.
step2:Use a backtracking approach to generate permutations.
step3:Maintain a boolean array or similar structure to track which elements have been used in the current permutation.
Program:

```
def permuteUnique(nums):
    def backtrack(path, counter):
        if len(path) == len(nums):
            res.append(path[:])
            return
        for num in counter:
            if counter[num] > 0:
                path.append(num)
                counter[num] -= 1
                backtrack(path, counter)
                path.pop()
                counter[num] += 1
    res = []
    counter = {}
    for num in nums:
        counter[num] = counter.get(num, 0) + 1
    backtrack([], counter)
    return res
nums = [1, 1, 2]
print(permuteUnique(nums))
```

Input:

```
nums = [1, 1, 2]
```

Output:

Output

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

Time complexity:O(n!·n)

Result: This approach ensures that all unique permutations are generated and avoids duplicates effectively.