**TOPIC 6 : BACKTRACKING**

* 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.

1. Visualization for 4-Queens:

Input: N = 4

Output:

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

1. Visualization for 5-Queens:

Input: N = 5

Output:

1. Visualization for 8-Queens:

Input: N = 8

Output:

global N

N = 4

def printSolution(board):

for i in range(N):

for j in range(N):

if board[i][j] == 1:

print("Q", end=" ")

else:

print(".", end=" ")

print()

def isSafe(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, N, 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solveNQUtil(board, col):

if col >= N:

return True

for i in range(N):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQUtil(board, col + 1) == True:

return True

board[i][col] = 0

return False

def solveNQ():

board = [[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0],

[0, 0, 0, 0]]

if solveNQUtil(board, 0) == False:

print("Solution does not exist")

return False

printSolution(board)

return True

if \_name\_ == '\_main\_':

solveNQ()

* 1. 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.

1. 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.

1. 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.

1. 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.

N = 8

ROWS = 8

COLS = 10

def printSolution(board):

for i in range(ROWS):

for j in range(COLS):

if board[i][j] == 1:

print("Q", end=" ")

else:

print(".", end=" ")

print()

def isSafe(board, row, col):

for i in range(col):

if board[row][i] == 1:

return False

for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

if board[i][j] == 1:

return False

for i, j in zip(range(row, ROWS, 1), range(col, -1, -1)):

if board[i][j] == 1:

return False

return True

def solveNQUtil(board, col):

if col >= COLS:

return False

if sum(sum(row) for row in board) == N:

return True

for i in range(ROWS):

if isSafe(board, i, col):

board[i][col] = 1

if solveNQUtil(board, col + 1):

return True

board[i][col] = 0

return False

def solveNQ():

board = [[0 for \_ in range(COLS)] for \_ in range(ROWS)]

if not solveNQUtil(board, 0):

print("Solution does not exist")

return False

printSolution(board)

return True

if \_name\_ == '\_main\_':

    solveNQ()

* 1. 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"]]

N = 9

def printing(arr):

for i in range(N):

for j in range(N):

print(arr[i][j], end=" ")

print()

def isSafe(grid, row, col, num):

for x in range(9):

if grid[row][x] == num:

return False

for x in range(9):

if grid[x][col] == num:

return False

startRow = row - row % 3

startCol = col - col % 3

for i in range(3):

for j in range(3):

if grid[i + startRow][j + startCol] == num:

return False

return True

def solveSudoku(grid, row, col):

if (row == N - 1 and col == N):

return True

if col == N:

row += 1

col = 0

if grid[row][col] > 0:

return solveSudoku(grid, row, col + 1)

for num in range(1, N + 1, 1):

if isSafe(grid, row, col, num):

grid[row][col] = num

if solveSudoku(grid, row, col + 1):

return True

grid[row][col] = 0

return False

grid = [

[3, 0, 6, 5, 0, 8, 4, 0, 0],

[5, 2, 0, 0, 0, 0, 0, 0, 0],

[0, 8, 7, 0, 0, 0, 0, 3, 1],

[0, 0, 3, 0, 1, 0, 0, 8, 0],

[9, 0, 0, 8, 6, 3, 0, 0, 5],

[0, 5, 0, 0, 9, 0, 6, 0, 0],

[1, 3, 0, 0, 0, 0, 2, 5, 0],

[0, 0, 0, 0, 0, 0, 0, 7, 4],

[0, 0, 5, 2, 0, 6, 3, 0, 0]

]

if (solveSudoku(grid, 0, 0)):

printing(grid)

else:

print("no solution exists")

* 1. 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"]]

N = 9

def printing(arr):

for i in range(N):

for j in range(N):

print(arr[i][j], end=" ")

print()

def isSafe(grid, row, col, num):

for x in range(9):

if grid[row][x] == num:

return False

for x in range(9):

if grid[x][col] == num:

return False

startRow = row - row % 3

startCol = col - col % 3

for i in range(3):

for j in range(3):

if grid[i + startRow][j + startCol] == num:

return False

return True

def solveSudoku(grid, row, col):

if (row == N - 1 and col == N):

return True

if col == N:

row += 1

col = 0

if grid[row][col] > 0:

return solveSudoku(grid, row, col + 1)

for num in range(1, N + 1, 1):

if isSafe(grid, row, col, num):

grid[row][col] = num

if solveSudoku(grid, row, col + 1):

return True

grid[row][col] = 0

return False

grid = [

[3, 0, 6, 5, 0, 8, 4, 0, 0],

[5, 2, 0, 0, 0, 0, 0, 0, 0],

[0, 8, 7, 0, 0, 0, 0, 3, 1],

[0, 0, 3, 0, 1, 0, 0, 8, 0],

[9, 0, 0, 8, 6, 3, 0, 0, 5],

[0, 5, 0, 0, 9, 0, 6, 0, 0],

[1, 3, 0, 0, 0, 0, 2, 5, 0],

[0, 0, 0, 0, 0, 0, 0, 7, 4],

[0, 0, 5, 2, 0, 6, 3, 0, 0]

]

if (solveSudoku(grid, 0, 0)):

printing(grid)

else:

print("no solution exists")

* 1. 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

class Solution:

def dp(self, i, nums, tr, dct):

if tr == 0:

return 1

if i == 0:

if tr - nums[i] == 0:

return 1

return 0

if (i, tr) in dct:

return dct[(i, tr)]

x = self.dp(i - 1, nums, tr - nums[i], dct)

y = self.dp(i - 1, nums, tr, dct)

dct[(i, tr)] = x + y

return dct[(i, tr)]

def findTargetSumWays(self, nums, target):

sm = sum(nums)

n = len(nums)

if target > sm:

return 0

if (sm - target) % 2:

return 0

tr = (sm - target) // 2

nums.sort()

dp = [[0] \* (tr + 1) for i in range(n)]

for i in range(n):

dp[i][0] = 1

if nums[0] <= tr:

dp[0][nums[0]] = 1

for i in range(1, n):

for j in range(tr + 1):

x = 0

if nums[i] <= j:

x = dp[i - 1][j - nums[i]]

y = dp[i - 1][j]

dp[i][j] = x + y

zero = nums.count(0)

if zero:

return dp[-1][tr] \* 2

return dp[-1][tr]

# return self.dp(n-1,nums,tr,{})(2\*zero)

# Example input to demonstrate usage

nums = [1]

target = 1

sol = Solution()

print(sol.findTargetSumWays(nums, target)) # Output should be 5

* 1. 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

class Solution:

def sumSubarrayMins(self, arr):

n = len(arr)

left = [-1] \* n

right = [n] \* n

stack = []

for i, value in enumerate(arr):

while stack and arr[stack[-1]] >= value:

stack.pop()

if stack:

left[i] = stack[-1]

stack.append(i)

stack = []

for i in range(n - 1, -1, -1):

while stack and arr[stack[-1]] > arr[i]:

stack.pop()

if stack:

right[i] = stack[-1]

stack.append(i)

mod = 10\*\*9 + 7

result = sum((i - left[i]) \* (right[i] - i) \* value for i, value in enumerate(arr)) % mod

return result

# Example input to demonstrate usage

arr = [3, 1, 2, 4]

sol = Solution()

print(sol.sumSubarrayMins(arr)) # Expected output should be 17

* 1. 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:

class Solution:

def combinationSum(self, candidates, target):

self.res = set()

self.candidates = candidates

self.targetSum = target

for i, num in enumerate(candidates):

self.backtrack([num], i, num)

return [list(tup) for tup in self.res]

def backtrack(self, path, i, currentSum):

if currentSum >= self.targetSum:

if currentSum == self.targetSum:

pathTup = tuple(path)

if pathTup not in self.res:

self.res.add(pathTup)

return

for j in range(i, len(self.candidates)):

num = self.candidates[j]

path.append(num)

self.backtrack(path, j, currentSum + num)

path.pop()

# Example input to demonstrate usage

candidates = [2, 3, 6, 7]

target = 7

sol = Solution()

print(sol.combinationSum(candidates, target)) # Expected output should be [[2, 2, 3], [7]]

* 1. 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]

]

class Solution:

    def combinationSum2(self, candidates, target):

        candidates.sort()

        stack = []

        ans = []

        def com2(ind, target):

            if target == 0:

                ans.append(stack[:])

                return

            for i in range(ind, len(candidates)):

                if i > ind and candidates[i] == candidates[i - 1]:

                    continue

                if candidates[i] <= target:

                    stack.append(candidates[i])

                    com2(i + 1, target - candidates[i])

                    stack.pop()

                else:

                    break

        com2(0, target)

        return ans

# Example input to demonstrate usage

candidates = [10, 1, 2, 7, 6, 1, 5]

target = 8

sol = Solution()

print(sol.combinationSum2(candidates, target))  # Expected output should be [[1, 1, 6], [1,

* 1. 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]]

class Solution:

def permute(self, nums):

if len(nums) == 1:

return [nums[:]]

res = []

for \_ in range(len(nums)):

n = nums.pop(0)

perms = self.permute(nums)

for p in perms:

p.append(n)

res.extend(perms)

nums.append(n)

return res

nums = [1, 2, 3]

sol = Solution()

print(sol.permute(nums))

* 1. 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]]

class Solution:

def permuteUnique(self, nums):

ans = []

nums.sort()

def subset(p, up):

if len(up) == 0:

if p not in ans:

ans.append(p)

return

ch = up[0]

for i in range(len(p) + 1):

subset(p[0:i] + [ch] + p[i:], up[1:])

subset([], nums)

return ans

# Example input to demonstrate usage

nums = [1, 1, 2]

sol = Solution()

print(sol.permuteUnique(nums)) # Expected output should be [[1, 1, 2], [1, 2, 1],

* 1. You and your friends are assigned the task of coloring a map with a limited number of colors. The map is represented as a list of regions and their adjacency relationships. The rules are as follows: At each step, you can choose any uncolored region and color it with any available color. Your friend Alice follows the same strategy immediately after you, and then your friend Bob follows suit. You want to maximize the number of regions you personally color. Write a function that takes the map's adjacency list representation and returns the maximum number of regions you can color before all regions are colored. Write a program to implement the Graph coloring technique for an undirected graph. Implement an algorithm with minimum number of colors. edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)] No. of vertices, n = 4

def color(e, n, k):

     graph = [[] for \_ in range(n)]

     for u, v in e:

        graph[u].append(v)

        graph[v].append(u)

     c  = {}

     a  = set(range(k))

     for r  in range(n):

        if r  not in c :

             a  = {c [adj] for adj in graph[r ] if adj in c }

             for color in range(k):

                if color not in a :

                    c [r ] = color

                    a .discard(color)

                    break

     return c

e  = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)]

n = 4

k = 3

result = color(e , n, k)

print("Coloring:",result)

* 1. You and your friends are tasked with coloring a map using a limited set of colors, with the following rules: At each step, you can choose any region of the map that hasn't been colored yet and color it with any available color. Your friend Alice will then color the next region using the same strategy, followed by your friend Bob. You aim to maximize the number of regions you color. Given a map represented as a list of regions and their adjacency relationships, write a function to determine the maximum number of regions you can color. Write a program to implement the Graph coloring technique for an undirected graph. Implement an algorithm with minimum number of colors. edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)] No. of vertices, n = 4, k = 3

class Graph():

def \_init\_(self, vertices):

self.graph = [[0 for column in range(vertices)]

for row in range(vertices)]

self.V = vertices

def isSafe(self, v, pos, path):

if self.graph[path[pos-1]][v] == 0:

return False

for vertex in path:

if vertex == v:

return False

return True

def hamCycleUtil(self, path, pos):

if pos == self.V:

if self.graph[path[pos-1]][path[0]] == 1:

return True

else:

return False

for v in range(1, self.V):

if self.isSafe(v, pos, path):

path[pos] = v

if self.hamCycleUtil(path, pos+1):

return True

path[pos] = -1

return False

def hamCycle(self):

path = [-1] \* self.V

path[0] = 0

if not self.hamCycleUtil(path, 1):

print("Solution does not exist\n")

return False

self.printSolution(path)

return True

def printSolution(self, path):

print("Solution Exists: Following is one Hamiltonian Cycle")

for vertex in path:

print(vertex)

# Example 1

g1 = Graph(5)

g1.graph = [[0, 1, 0, 1, 0],

[1, 0, 1, 1, 1],

[0, 1, 0, 0, 1],

[1, 1, 0, 0, 1],

[0, 1, 1, 1, 0]]

g1.hamCycle()

# Example 2

g2 = Graph(5)

g2.graph = [[0, 1, 0, 1, 0],

[1, 0, 1, 1, 1],

[0, 1, 0, 0, 1],

[1, 1, 0, 0, 0],

[0, 1, 1, 0, 0]]

g2.hamCycle()

* 1. You are given an undirected graph represented by a list of edges and the number of vertices n. Your task is to determine if there exists a Hamiltonian cycle in the graph. A Hamiltonian cycle is a cycle that visits each vertex exactly once and returns to the starting vertex. Write a function that takes the list of edges and the number of vertices as input and returns true if there exists a Hamiltonian cycle in the graph, otherwise return false. Example: Given edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2), (2, 4), (4, 0)] and n = 5
  2. You are given an undirected graph represented by a list of edges and the number of vertices n. Your task is to determine if there exists a Hamiltonian cycle in the graph. A Hamiltonian cycle is a cycle that visits each vertex exactly once and returns to the starting vertex. Write a function that takes the list of edges and the number of vertices as input and returns true if there exists a Hamiltonian cycle in the graph, otherwise return false. Example:edges = [(0, 1), (1, 2), (2, 3), (3, 0), (0, 2)] and n = 4
  3. You are tasked with designing an efficient coading to generate all subsets of a given set S containing n elements. Each subset should be outputted in lexicographical order. Return a list of lists where each inner list is a subset of the given set. Additionally, find out how your coading handles duplicate elements in S. A = [1, 2, 3] The subsets of [1, 2, 3] are: [], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]

def findSubsets(nums, n):

for i in range(1 << n):

subset = []

for j in range(n):

if (i & (1 << j)) != 0:

subset.append(nums[j])

print(subset)

# Driver Code

arr = [1, 2, 3]

n = len(arr)

findSubsets(arr, n)

* 1. Write a program to implement the concept of subset generation. Given a set of unique integers and a specific integer 3, generate all subsets that contain the element 3. Return a list of lists where each inner list is a subset containing the element 3 E = [2, 3, 4, 5], x = 3, The subsets containing 3 : [3], [2, 3], [3, 4], [3,5], [2, 3, 4], [2, 3, 5], [3, 4, 5], [2, 3, 4, 5] Given an integer array nums of unique elements, return all possible subsets(the power set). The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

Input: nums = [1,2,3]

Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

Example 2:

Input: nums = [0]

Output: [[],[0]]

* 1. You are given two string arrays words1 and words2. A string b is a subset of string a if every letter in b occurs in a including multiplicity. For example, "wrr" is a subset of "warrior" but is not a subset of "world". A string a from words1 is universal if for every string b in words2, b is a subset of a. Return an array of all the universal strings in words1. You may return the answer in any order.

Example 1:

Input: words1 = ["amazon","apple","facebook","google","leetcode"], words2 = ["e","o"]

Output: ["facebook","google","leetcode"]

Example 2:

Input: words1 = ["amazon","apple","facebook","google","leetcode"], words2 = ["l","e"]

Output: ["apple","google","leetcode"]

class Solution:

def is\_subset(self, a, b):

# Function to check if string a is a subset of string b

count\_a = {}

count\_b = {}

# Count characters in string a

for char in a:

if char in count\_a:

count\_a[char] += 1

else:

count\_a[char] = 1

# Count characters in string b

for char in b:

if char in count\_b:

count\_b[char] += 1

else:

count\_b[char] = 1

# Check if every character in b is in a with enough count

for char in count\_b:

if char not in count\_a or count\_a[char] < count\_b[char]:

return False

return True

def universalStrings(self, words1, words2):

result = []

# Check each string in words1

for a in words1:

is\_universal = True

# Check if a is universal for all strings in words2

for b in words2:

if not self.is\_subset(a, b):

is\_universal = False

break

# If a is universal, add it to the result

if is\_universal:

result.append(a)

return result

# Example usage:

words1 = ["amazon", "apple", "facebook", "google", "leetcode"]

words2 = ["e", "o"]

sol = Solution()

print(sol.universalStrings(words1, words2)) # Output: ["facebook", "google", "leetcode"]

words2 = ["l", "e"]

print(sol.universalStrings(words1, words2)) # Output: ["apple", "google", "leetcode"]