1.Given an m x n binary matrix mat, return the distance of the nearest 0 for each cell. The distance between two adjacent cells is 1.

Program:

from collections import deque

def updateMatrix(mat):

m, n = len(mat), len(mat[0])

dist = [[float('inf')] \* n for \_ in range(m)]

queue = deque()

# Enqueue all cells with 0s and set their distance to 0

for i in range(m):

for j in range(n):

if mat[i][j] == 0:

dist[i][j] = 0

queue.append((i, j))

# Directions array for moving up, down, left, right

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Perform BFS

while queue:

x, y = queue.popleft()

for dx, dy in directions:

new\_x, new\_y = x + dx, y + dy

if 0 <= new\_x < m and 0 <= new\_y < n and dist[new\_x][new\_y] > dist[x][y] + 1:

dist[new\_x][new\_y] = dist[x][y] + 1

queue.append((new\_x, new\_y))

return dist

# Example usage:

mat = [

[0, 0, 0],

[0, 1, 0],

[1, 1, 1]

]

print(updateMatrix(mat))

2. Given two integer arrays arr1 and arr2, return the minimum number of operations (possibly zero) needed to make arr1 strictly increasing. In one operation, you can choose two indices 0 <= i < arr1.length and 0 <= j < arr2.length and do the assignment arr1[i] = arr2[j]. If there is no way to make arr1 strictly increasing, return -1.

Program:

from bisect import bisect\_right

from collections import defaultdict

def makeArrayIncreasing(arr1, arr2):

arr2 = sorted(set(arr2))

dp = {-1: 0} # Using -1 as a placeholder for the initial state

for num in arr1:

new\_dp = defaultdict(lambda: float('inf'))

for key in dp:

if num > key:

new\_dp[num] = min(new\_dp[num], dp[key])

idx = bisect\_right(arr2, key)

if idx < len(arr2):

new\_dp[arr2[idx]] = min(new\_dp[arr2[idx]], dp[key] + 1)

dp = new\_dp

if dp:

return min(dp.values())

return -1

# Example usage:

arr1 = [1, 5, 3, 6, 7]

arr2 = [1, 3, 2, 4]

print(makeArrayIncreasing(arr1, arr2)) # Output: 1

3. Given two strings a and b, return the minimum number of times you should repeat string a so that string b is a substring of it. If it is impossible for b​ to be a substring of a after repeating it, return -1. Notice: string "abc" repeated 0 times is "", repeated 1 time is "abc" and repeated 2 times is "abcabc"

Program:

def repeatedStringMatch(a, b):

# Calculate the minimum number of repeats needed

min\_repeats = -(-len(b) // len(a)) # Equivalent to math.ceil(len(b) / len(a))

# Check if b is a substring of a repeated min\_repeats or min\_repeats + 1 times

for i in range(2):

if b in (a \* (min\_repeats + i)):

return min\_repeats + i

# If b is not found, return -1

return -1

# Example usage:

a = "abcd"

b = "cdabcdab"

print(repeatedStringMatch(a, b)) # Output: 3

4. Given an array nums containing n distinct numbers in the range [0, n], return the only number in the range that is missing from the array

Program:

def missingNumber(nums):

n = len(nums)

xor\_all = 0

xor\_nums = 0

for i in range(n + 1):

xor\_all ^= i

for num in nums:

xor\_nums ^= num

return xor\_all ^ xor\_nums

# Example usage:

nums = [3, 0, 1]

print(missingNumber(nums)) # Output: 2

5. You are given an n x n integer matrix grid.Generate an integer matrix maxLocal of size (n - 2) x (n - 2) such that: maxLocal[i][j] is equal to the largest value of the 3 x 3 matrix in grid centered around row i + 1 and column j + 1. In other words, we want to find the largest value in every contiguous 3 x 3 matrix in grid. Return the generated matrix.

Program:

def largestLocal(grid):

n = len(grid)

maxLocal = [[0] \* (n - 2) for \_ in range(n - 2)]

for i in range(n - 2):

for j in range(n - 2):

max\_val = float('-inf')

for k in range(i, i + 3):

for l in range(j, j + 3):

max\_val = max(max\_val, grid[k][l])

maxLocal[i][j] = max\_val

return maxLocal

# Example usage:

grid = [

[9, 9, 8, 1],

[5, 6, 2, 6],

[8, 2, 6, 4],

[6, 2, 2, 2]

]

print(largestLocal(grid))

6. You are given an array of strings words and a string pref. Return the number of strings in words that contain pref as a prefix. A prefix of a string s is any leading contiguous substring of s.

Program:

def count\_words\_with\_prefix(words, pref):

count = 0

for word in words:

if word.startswith(pref):

count += 1

return count

# Example usage:

words = ["apple", "ape", "apricot", "banana", "appetizer"]

pref = "app"

print(count\_words\_with\_prefix(words, pref)) # Output: 3

7. Given an m x n integer matrix matrix, if an element is 0, set its entire row and column to 0's. You must do it in place.

Program:

def set\_zeroes(matrix):

if not matrix or not matrix[0]:

return

m, n = len(matrix), len(matrix[0])

first\_row\_has\_zero = any(matrix[0][j] == 0 for j in range(n))

first\_col\_has\_zero = any(matrix[i][0] == 0 for i in range(m))

# Use first row and column as markers

for i in range(1, m):

for j in range(1, n):

if matrix[i][j] == 0:

matrix[i][0] = 0

matrix[0][j] = 0

# Set matrix elements to zero based on markers

for i in range(1, m):

for j in range(1, n):

if matrix[i][0] == 0 or matrix[0][j] == 0:

matrix[i][j] = 0

# Set first row to zero if needed

if first\_row\_has\_zero:

for j in range(n):

matrix[0][j] = 0

# Set first column to zero if needed

if first\_col\_has\_zero:

for i in range(m):

matrix[i][0] = 0

# Example usage:

matrix = [

[1, 2, 3],

[4, 0, 6],

[7, 8, 9]

]

set\_zeroes(matrix)

print(matrix) # Output: [[1, 0, 3], [0, 0, 0], [7, 0, 9]]

8.

8. Given two integer arrays nums1 and nums2, return an array of their intersection . Each element in the result must be unique and you may return the result in any order.

Program:

def intersection(nums1, nums2):

set1 = set(nums1)

set2 = set(nums2)

return list(set1.intersection(set2))

# Example usage:

nums1 = [1, 2, 2, 1]

nums2 = [2, 2]

print(intersection(nums1, nums2)) # Output: [2]

9. Given the strings s1 and s2 of size n and the string evil, return the number of good strings. A good string has size n, it is alphabetically greater than or equal to s1, it is alphabetically smaller than or equal to s2, and it does not contain the string evil as a substring. Since the answer can be a huge number, return this modulo 109 + 7

Program:

MOD = 10\*\*9 + 7

def count\_good\_strings(s1, s2, evil):

n = len(s1)

dp = [[[[0]\*2 for \_ in range(2)] for \_ in range(2)] for \_ in range(n+1)]

def contains\_evil(s):

m = len(evil)

for i in range(n - m + 1):

if s[i:i+m] == evil:

return True

return False

def count\_valid\_strings(len, prefix, larger, smaller):

if len == n:

return 1

if dp[len][prefix][larger][smaller] != -1:

return dp[len][prefix][larger][smaller]

total = 0

start = ord('a') if prefix == 0 else ord(s1[len])

end = ord('z') if smaller else ord(s2[len])

for c in range(start, end + 1):

new\_prefix = prefix or (c > ord(s1[len]))

new\_smaller = smaller or (c < ord(s2[len]))

if new\_prefix and not contains\_evil(chr(c)):

total = (total + count\_valid\_strings(len + 1, new\_prefix, True, new\_smaller)) % MOD

dp[len][prefix][larger][smaller] = total

return total

# Initialize dp table with -1

for i in range(n + 1):

for j in range(2):

for k in range(2):

for l in range(2):

dp[i][j][k][l] = -1

return count\_valid\_strings(0, 0, 0, 0)

# Example usage:

s1 = "abc"

s2 = "def"

evil = "xy"

print(count\_good\_strings(s1, s2, evil)) # Output: 6

10. Given a 2D integer array matrix, return the transpose of matrix. The transpose of a matrix is the matrix flipped over its main diagonal, switching the matrix's row and column indices.

Program:

def transpose(matrix):

if not matrix:

return []

m, n = len(matrix), len(matrix[0])

transpose\_matrix = [[0]\*m for \_ in range(n)]

for i in range(m):

for j in range(n):

transpose\_matrix[j][i] = matrix[i][j]

return transpose\_matrix

# Example usage:

matrix = [

[1, 2, 3],

[4, 5, 6],

[7, 8, 9]

]

print(transpose(matrix))