Given an integer array num sorted in non-decreasing order. You can perform the following operation any number of times: Choose two indices, i and j, where nums[i] < nums[j]. Then, remove the elements at indices i and j from nums. The remaining elements retain their original order, and the array is reindexed. Return the minimum length of nums after applying the operation zero or more times.

Program:

def min\_length\_after\_removals(nums):

n = len(nums)

pairs = n // 2

return n - pairs \* 2

nums = [1, 2, 3, 4]

print(min\_length\_after\_removals(nums)) # Output: 1

Given an array of string words, return all strings in words that is a substring of another word. You can return the answer in any order. A substring is a contiguous sequence of characters within a string.

Program:

def string\_substrings(words):

result = []

for i in range(len(words)):

for j in range(len(words)):

if i != j and words[i] in words[j]:

result.append(words[i])

break

return result

words = ["mass", "as", "hero", "superhero"]

print(string\_substrings(words)) # Output: ['as', 'hero']

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):

if not mat or not mat[0]:

return mat

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

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

queue = deque()

# Initialize queue with all 0's and their distances

for i in range(m):

for j in range(n):

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

dist[i][j] = 0

queue.append((i, j))

# Directions for moving in the matrix: right, left, down, up

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

# BFS to find shortest path to 0

while queue:

x, y = queue.popleft()

for dx, dy in directions:

newX, newY = x + dx, y + dy

if 0 <= newX < m and 0 <= newY < n:

if dist[newX][newY] > dist[x][y] + 1:

dist[newX][newY] = dist[x][y] + 1

queue.append((newX, newY))

return dist

# Example usage

mat = [

[0, 0, 0],

[0, 1, 0],

[1, 1, 1]

]

print(updateMatrix(mat))

4. 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.sort()

dp = {-1: 0} # Initialize DP with -1 as the previous value and 0 operations

for num in arr1:

temp = defaultdict(lambda: float('inf'))

for key in dp:

if num > key:

temp[num] = min(temp[num], dp[key])

idx = bisect\_right(arr2, key)

if idx < len(arr2):

temp[arr2[idx]] = min(temp[arr2[idx]], dp[key] + 1)

dp = temp

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

5. 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):

# Minimum number of times 'a' needs to be repeated to cover length of 'b'

repeat\_count = -(-len(b) // len(a)) # Equivalent to math.ceil(len(b) / len(a))

# Create the repeated string of 'a'

repeated\_a = a \* repeat\_count

# Check if 'b' is a substring of the repeated string

if b in repeated\_a:

return repeat\_count

# Check if 'b' is a substring of the repeated string with one more repetition

repeated\_a += a

if b in repeated\_a:

return repeat\_count + 1

# If not found, return -1

return -1

# Example usage

a = "abc"

b = "cabcabca"

print(repeatedStringMatch(a, b)) # Output: 4