**Saveetha School of Engineering**

**Saveetha Institute of Medical And Technical Science**

**LAB PROGRAM-04**

**PROGRAMMING LANGUAGE**

Python

**COURSE CODE / NAME**

CSA0666 - Design And Analysis Of Algorithm For Divide And Conquer Techniques

**SUBMITTED BY**

T Svetha – 192321155

1. **Counting Elements Given an integer array arr, count how many elements x there are, such that x + 1 is also in arr. If there are duplicates in arr, count them separately.**

def count(arr):

elem= set(arr)

c= 0

for x in arr:

if x + 1 in elem:

c += 1

return c

arr1 = [1, 2, 3]

print(count(arr1))

**Output:** 2

**Time complexity:** O(n)

1. **Perform String Shifts You are given a string s containing lowercase English letters, and a matrix shift, where shift[i] = [directioni, amounti]:**

def string(s: str, shift: List[List[int]]) -> str:

total = 0

for direction, amount in shift:

total+= (-1 if direction == 0 else 1) \* amount

total %= len(s)

return s[total:] + s[:total]

s = "abcde"

shift = [[0,1],[1,2]]

shifted = string(s, shift)

print(shifted)

**Output:** "cdeab"

**Time complexity**:O(n)

1. **Leftmost Column with at Least a One A row-sorted binary matrix means that all elements are 0 or 1 and each row of the matrix is sorted in non-decreasing order. Given a row-sorted binary matrix binaryMatrix, return the index (0-indexed) of the leftmost column with a 1 in it. If such an index does not exist, return -1. You can't access the Binary Matrix directly. You may only access the matrix using a BinaryMatrix interface:**

class bmatrix:

def \_\_init\_\_(self, mat):

self.mat = mat

def get(self, row, col):

return self.mat[row][col]

def dimensions(self):

return [len(self.mat), len(self.mat[0])]

def left(matrix):

rows, cols = matrix.dimensions()

current\_row = 0

current\_col = cols - 1

leftmost\_col = -1

while current\_row < rows and current\_col >= 0:

if matrix.get(current\_row, current\_col) == 1:

leftmost\_col = current\_col

current\_col -= 1

else:

current\_row += 1

return leftmost\_col

mat1 = [[0, 0], [1, 1]]

binary = bmatrix(mat1)

print(left(binary))

**Output:** 0

**Time complexity:** O(m+n)

1. **First Unique Number You have a queue of integers, you need to retrieve the first unique integer in the queue. Implement the FirstUnique class**

from collections import deque

class unique:

def \_\_init\_\_(self):

self.queue = deque([])

self.seenonce = {}

self.seentwice = set()

def show(self) -> int:

while self.queue and self.queue[0] in self.seentwice:

self.queue.popleft()

return self.queue[0] if self.queue else -1

def add(self, value: int) -> None:

if value not in self.seentwice:

self.queue.append(value)

if value not in self.seenonce:

self.seenonce[value] = 1

else:

del self.seenonce[value]

self.seentwice.add(value)

firstunique = unique()

firstunique.add(7)

firstunique.add(7)

firstunique.add(7)

firstunique.add(1)

firstunique.add(1)

firstunique.add( uniquenum = firstunique.show ()

print(uniquenum)

**Output:** 1

**Time complexity**:O(1)

1. **Check If a String Is a Valid Sequence from Root to Leaves Path in a Binary Tree Given a binary tree where each path going from the root to any leaf form a valid sequence, check if a given string is a valid sequence in such binary tree. We get the given string from the concatenation of an array of integers arr and the concatenation of all values of the nodes along a path results in a sequence in the given binary tree.**

class TreeNode:

def \_\_init\_\_(self, val=0, left=None, right=None):

self.val = val

self.left = left

self.right = right

def isValidSequence(root, arr):

def dfs(node, index):

if node is None:

return False

if index >= len(arr) or node.val != arr[index]:

return False

if index == len(arr) - 1:

return node.left is None and node.right is None

return dfs(node.left, index + 1) or dfs(node.right, index + 1)

return dfs(root, 0)

root = TreeNode(0)

root.left = TreeNode(1)

root.right = TreeNode(0)

root.left.left = TreeNode(0)

root.left.right = TreeNode(1)

root.right.left = TreeNode(0)

root.left.left.right = TreeNode(1)

root.left.right.left = TreeNode(0)

root.left.right.right = TreeNode(0)

arr1 = [0, 1, 0, 1]

print(isValidSequence(root, arr1))

**Output**: True

**Time complexity:**O(n)

1. **Kids With the Greatest Number of Candies There are n kids with candies. You are given an integer array candies, where each candies[i] represents the number of candies the ith kid has, and an integer extraCandies, denoting the number of extra candies that you have. Return a boolean array result of length n, where result[i] is true if, after giving the ith kid all the extraCandies, they will have the greatest number of candies among all the kids, or false otherwise. Note that multiple kids can have the greatest number of candies.**

def kidsWithCandies(candies, extraCandies):

mostcandies = max(candies)

result = [candy + extraCandies >= mostcandies for candy in candies]

return result

candies = [2, 1, 5, 6, 3]

extraCandies = 3

result = kidsWithCandies(candies, extraCandies)

print(result)

**Output:** [True, True, True, True, True]

**Time complexity**:O(n)

1. **Max Difference You Can Get From Changing an Integer You are given an integer num. You will apply the following steps exactly two times:**

def diff(num):

strnum = str(num)

maxdiff = max(int(largest \* '9' + str\_num[1:]) - num, int(str\_num[:1] + '0' \* (len(str\_num) - 1)) - num)

return maxdiff

num = 555

result = diff(num)

print(result)

**Output:** 888

**Time complexity**:O(n)

1. **Check If a String Can Break Another String Given two strings: s1 and s2 with the same size, check if some permutation of string s1 can break some permutation of string s2 or vice-versa. In other words s2 can break s1 or vice-versa. A string x can break string y (both of size n) if x[i] >= y[i] (in alphabetical order) for all i between 0 and n-1.**

def checkIfCanBreak(s1, s2):

sorted\_s1 = sorted(s1)

sorted\_s2 = sorted(s2)

can\_s1\_break\_s2 = all(c1 >= c2 for c1, c2 in zip(sorted\_s1, sorted\_s2))

can\_s2\_break\_s1 = all(c2 >= c1 for c1, c2 in zip(sorted\_s1, sorted\_s2))

return can\_s1\_break\_s2 or can\_s2\_break\_s1

s1 = "abc"

s2 = "xya"

print(checkIfCanBreak(s1, s2)) # Output: True

s1 = "abe"

s2 = "acd"

print(checkIfCanBreak(s1, s2))

**Output**: False

**Time complexity**:O(n log n)

1. **Number of Ways to Wear Different Hats to Each Other There are n people and 40 types of hats labeled from 1 to 40. Given a 2D integer array hats, where hats[i] is a list of all hats preferred by the ith person. Return the number of ways that the n people wear different hats to each other. Since the answer may be too large, return it modulo 109 + 7**.

def numberWays(hats):

MOD = 10\*\*9 + 7

n = len(hats)

dp = [[0] \* (1 << 40) for \_ in range(n + 1)]

dp[0][0] = 1

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

for mask in range(1 << 40):

available\_hats = mask & hats[i - 1][:]

for prev\_mask in range(mask & (~available\_hats)):

dp[i][mask] = (dp[i][mask] + dp[i - 1][prev\_mask]) % MOD

return sum(dp[n]) % MOD

hats = [[3,4],[4,5],[5]]

result = numberWays(hats)

print(result)

**Output:** 1

**Time complexity**:O(n\*2^40)

1. **Next Permutation A permutation of an array of integers is an arrangement of its members into a sequence or linear order.**

def nextPermutation(nums):

n = len(nums)

if n <= 1:

return

i = n - 2

while i >= 0 and nums[i] >= nums[i + 1]:

i -= 1

if i >= 0:

j = n - 1

while nums[j] <= nums[i]:

j -= 1

nums[i], nums[j] = nums[j], nums[i]

nums[i + 1:] = reversed(nums[i + 1:])

nums1 = [1, 2, 3]

nextPermutation(nums1)

print(nums1)

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

**Time complexity**:O(n)