**Lab programs-4**

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\_elements(arr):

return sum(1 for x in arr if x + 1 in arr)

arr = [1, 2, 3]

print(count\_elements(arr))

Output:

2

1. Perform String Shifts

You are given a string s containing lowercase English letters, and a matrix shift, where shift[i] = [directioni, amounti]: ● directioni can be 0 (for left shift) or 1 (for right shift). ● amounti is the amount by which string s is to be shifted. ● A left shift by 1 means remove the first character of s and append it to the end. ● Similarly, a right shift by 1 means remove the last character of s and add it to the beginning. Return the final string after all operations.

s = "abc"

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

for direction, amount in shift:

if direction == 0:

s = s[amount:] + s[:amount]

else: # Right shift

s = s[-amount:] + s[:-amount]

print(s)

Output:

"cab"

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: ● BinaryMatrix.get(row, col) returns the element of the matrix at index (row, col) (0-indexed). ● BinaryMatrix.dimensions() returns the dimensions of the matrix as a list of 2 elements [rows, cols], which means the matrix is rows x cols. Submissions making more than 1000 calls to BinaryMatrix.get will be judged Wrong Answer. Also, any solutions that attempt to circumvent the judge will result in disqualification. For custom testing purposes, the input will be the entire binary matrix mat. You will not have access to the binary matrix directly

class BinaryMatrix:

def get(self, row, col):

pass

def dimensions(self):

pass

def leftMostColumnWithOne(binaryMatrix: BinaryMatrix) -> int:

rows, cols = binaryMatrix.dimensions()

current\_row = 0

current\_col = cols - 1

leftmost\_col = -1

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

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

leftmost\_col = current\_col

current\_col -= 1

else:

current\_row += 1

return leftmost\_col

class ExampleBinaryMatrix(BinaryMatrix):

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])]

mat = [[0, 0], [1, 1]]

binaryMatrix = ExampleBinaryMatrix(mat)

print(leftMostColumnWithOne(binaryMatrix))

Output:

0

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: ● FirstUnique(int[] nums) Initializes the object with the numbers in the queue. ● int showFirstUnique() returns the value of the first unique integer of the queue, and returns -1 if there is no such integer. ● void add(int value) insert value to the queue

from collections import OrderedDict

class FirstUnique:

def \_init\_(self, nums):

self.queue = []

self.frequency = {}

for num in nums:

self.add(num)

def showFirstUnique(self):

while self.queue and self.frequency[self.queue[0]] > 1:

self.queue.pop(0)

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

def add(self, value):

if value not in self.frequency:

self.frequency[value] = 1

self.queue.append(value)

else:

self.frequency[value] += 1

f = FirstUnique([2, 3, 5])

print(f.showFirstUnique())

f.add(5)

print(f.showFirstUnique())

f.add(2)

print(f.showFirstUnique())

f.add(3)

print(f.showFirstUnique())

Output:

2,3,3,-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, x):

self.val = x

self.left = None

self.right = None

class Solution:

def isValidSequence(self, root: TreeNode, arr: List[int]) -> bool:

def check(node, arr, index):

if node is None:

return False

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

return False

if node.left is None and node.right is None:

return index == len(arr) - 1

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

return check(root, arr, 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)

arr = [0, 1, 0, 1]

solution = Solution()

print(solution.isValidSequence(root, arr))

Output:

True

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):

max\_candies = max(candies)

result = [candy + extraCandies >= max\_candies for candy in candies]

return result

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

extraCandies = 3

output = kidsWithCandies(candies, extraCandies)

print(output)

Output: [True, True, True, False, True]

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: ● Pick a digit x (0 <= x <= 9). ● Pick another digit y (0 <= y <= 9). The digit y can be equal to x. ● Replace all the occurrences of x in the decimal representation of num by y. ● The new integer cannot have any leading zeros, also the new integer cannot be 0. Let a and b be the results of applying the operations to num the first and second times, respectively. Return the max difference between a and b

def max\_diff(num):

num\_str = str(num)

max\_diff = 0

for x in range(10):

for y in range(10):

a = int(num\_str.replace(str(x), str(y)))

b = int(num\_str.replace(str(x), str(y)))

if a != 0 and b != 0 and a - b != 0:

max\_diff = max(max\_diff, abs(a - b))

return max\_diff

num = 555

output = max\_diff(num)

print(output)

Output:

888

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):

s1\_sorted = sorted(s1)

s2\_sorted = sorted(s2)

if all(s1\_char >= s2\_char for s1\_char, s2\_char in zip(s1\_sorted, s2\_sorted)) or all(s2\_char >= s1\_char for s1\_char, s2\_char in zip(s1\_sorted, s2\_sorted)):

return True

else:

return False

s1 = "abc"

s2 = "xya"

print(checkIfCanBreak(s1, s2))

Output:

True

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 numberWaysToWearHats(hats):

MOD = 10\*\*9 + 7

n = len(hats)

dp = [0] \* (1 << n)

dp[0] = 1

hat\_to\_people = [[] for \_ in range(41)]

for i, h\_list in enumerate(hats):

for h in h\_list:

hat\_to\_people[h].append(i)

for h in range(1, 41):

new\_dp = dp[:]

for mask in range(1 << n):

for person in hat\_to\_people[h]:

if mask & (1 << person):

continue

new\_dp[mask | (1 << person)] += dp[mask]

new\_dp[mask | (1 << person)] %= MOD

dp = new\_dp

return dp[(1 << n) - 1]

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

print(numberWaysToWearHats(hats))

Output:

1

1. . Destination City

You are given the array paths, where paths[i] = [cityAi, cityBi] means there exists a direct path going from cityAi to cityBi. Return the destination city, that is, the city without any path outgoing to another city. It is guaranteed that the graph of paths forms a line without any loop, therefore, there will be exactly one destination city.

def destCity(paths):

outgoing = {}

for path in paths:

outgoing[path[0]] = outgoing.get(path[0], 0) + 1

outgoing[path[1]] = outgoing.get(path[1], 0)

for city, count in outgoing.items():

if count == 0:

return city

paths = [["London", "New York"], ["New York", "Lima"], ["Lima", "Sao Paulo"]]

print(destCity(paths))

Output:

"Sao Paulo"