LAB 4

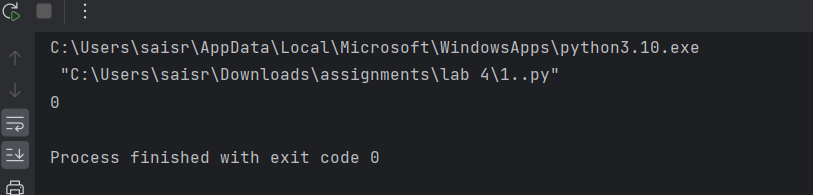
**1.Counting Elements**

**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.**

Coding:

ar =[1,1,3,3,5,5,7,7]  
c=0  
for i in range(len(ar)-1):  
 if ar[i]+ ar[i+1] in ar:  
 c+=2  
print(c)

Output:



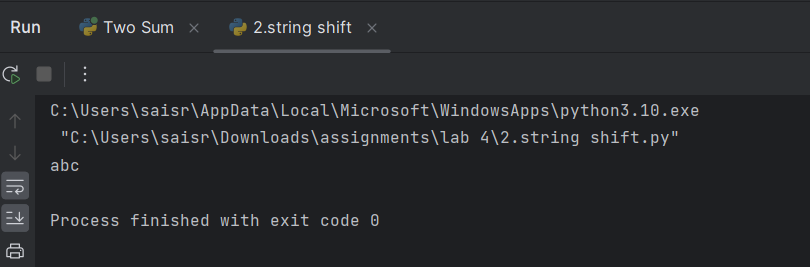
**2. Perform String Shifts**

**You are given a string s containing lowercase English letters, and a matrix shift, where shift[i] = [directioni, amounti]**

Coding:

s = "abc"  
shift = [[0, 1], [1, 2]]  
  
def left(a, s):  
 return s[a:] + s[:a]  
  
def right(a, s):  
 return s[-a:] + s[:-a]  
  
while shift:  
 for i in range(len(shift)):  
 if shift[i][0] == 0:  
 a = shift[i][1]  
 s = left(a, s)  
 else:  
 b = shift[i][1]  
 s = right(b, s)  
 shift.pop(0)  
  
print(s)

Output:



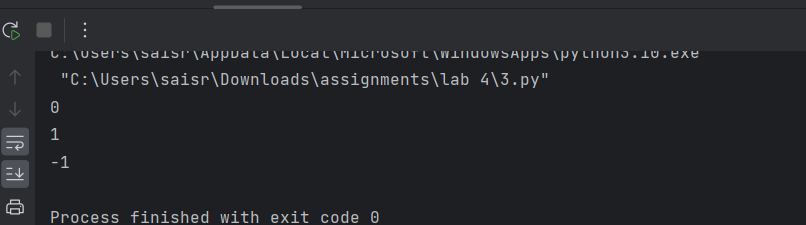
**3. Leftmost Column with at Least a One**

**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.**

Coding:

class BinaryMatrix:  
 def \_\_init\_\_(self, mat):  
 self.mat = mat  
  
 def get(self, row: int, col: int) -> int:  
 return self.mat[row][col]  
  
 def dimensions(self) -> list:  
 return [len(self.mat), len(self.mat[0])]  
  
  
def leftMostColumnWithOne(binaryMatrix):  
 rows, cols = binaryMatrix.dimensions()  
 current\_row = 0  
 current\_col = cols - 1  
 leftmost\_col\_with\_one = -1  
  
   
 while current\_row < rows and current\_col >= 0:  
 if binaryMatrix.get(current\_row, current\_col) == 1:  
 leftmost\_col\_with\_one = current\_col  
 current\_col -= 1   
 else:  
 current\_row += 1   
  
 return leftmost\_col\_with\_one  
  
  
  
mat1 = [[0, 0], [1, 1]]  
binaryMatrix1 = BinaryMatrix(mat1)  
print(leftMostColumnWithOne(binaryMatrix1))   
  
mat2 = [[0, 0], [0, 1]]  
binaryMatrix2 = BinaryMatrix(mat2)  
print(leftMostColumnWithOne(binaryMatrix2))   
  
  
mat3 = [[0, 0], [0, 0]]  
binaryMatrix3 = BinaryMatrix(mat3)  
print(leftMostColumnWithOne(binaryMatrix3))

Output:



**4. 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,**

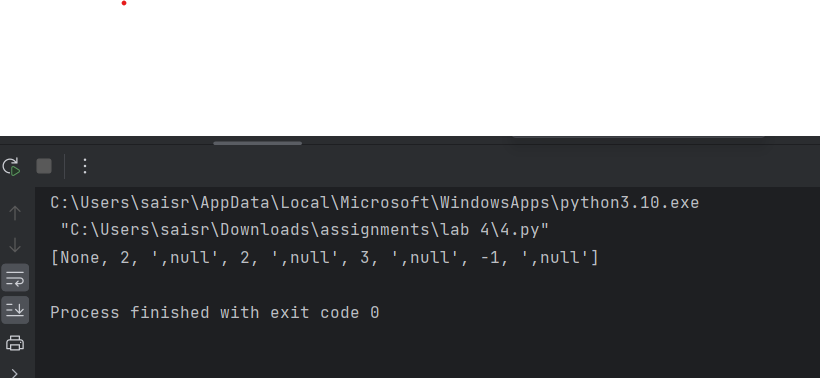
**and returns -1 if there is no such integer.**

**● void add(int value) insert value to the queue.**

Coding:

from collections import deque  
  
class Queue:  
 def \_\_init\_\_(self, nums):  
 self.queue = deque(nums)  
 self.unique\_elements = set(nums)  
  
 def showUnique(self):  
 if self.unique\_elements:  
 return self.queue[0]  
 return -1  
  
 def add(self, value):  
 if value in self.unique\_elements:  
 self.unique\_elements.remove(value)  
 else:  
 self.queue.append(value)  
 self.unique\_elements.add(value)  
  
 while self.queue and self.queue[0] not in self.unique\_elements:  
 self.queue.popleft()  
  
  
s = ["FirstUnique", "showFirstUnique", "add", "showFirstUnique", "add", "showFirstUnique", "add", "showFirstUnique"]  
ar = [[2, 3, 5], [], [5], [], [2], [], [3], []]  
  
firstUnique = None  
ans = []  
  
for i, op in enumerate(s):  
 if op == "FirstUnique":  
 firstUnique = Queue(ar[i])  
 ans.append(None)  
 elif op == "showFirstUnique":  
 ans.append(firstUnique.showUnique())

ans.append(“,null”)  
 elif op == "add":  
 firstUnique.add(ar[i][0])  
  
print(ans)

Output: 

**5. 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**

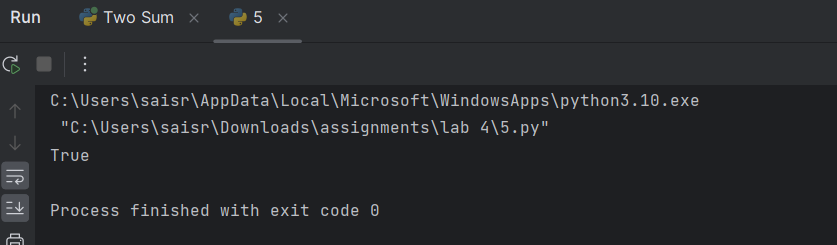
**concatenation of all values of the nodes along a path results in a sequence in the given**

**binary tree.**

Coding:

class TreeNode:  
 def \_\_init\_\_(self, x):  
 self.val = x  
 self.left = None  
 self.right = None  
  
def construct\_tree(lst):  
 if not lst:  
 return None  
 root = TreeNode(lst[0])  
 queue = [root]  
 i = 1  
 while i < len(lst):  
 node = queue.pop(0)  
 if lst[i] is not None:  
 node.left = TreeNode(lst[i])  
 queue.append(node.left)  
 i += 1  
 if i < len(lst) and lst[i] is not None:  
 node.right = TreeNode(lst[i])  
 queue.append(node.right)  
 i += 1  
 return root  
  
def isValidSequence(root, arr):  
 if not root or not arr:  
 return False  
 if root.val != arr[0]:  
 return False  
 if len(arr) == 1:  
 return not root.left and not root.right  
 return isValidSequence(root.left, arr[1:]) or isValidSequence(root.right, arr[1:])  
  
  
lst = [0,1,0,0,1,0,None,None,1,0,0]  
arr = [0,1,0,1]  
root = construct\_tree(lst)  
print(isValidSequence(root, arr))

Output:



**6. Kids With the Greatest Number of Candies**

**There are n kids with candies. You are given an integer array candies, where each**

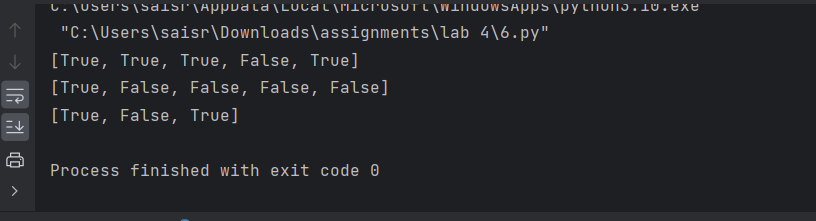
**candies[i] represents the number of candies the ith kid has, and an integer extraCandies, denoting the number of extra candies that you have.Return a boolean array result of length n, where result[i] is true if, after giving the ith kid all the extraCandies, they will have the greatest number of candies among all the kids, or**

**false otherwise. Note that multiple kids can have the greatest number of candies**.

Coding:

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  
print(kidsWithCandies(candies, extraCandies))   
candies = [4,2,1,1,2]  
extraCandies = 1  
print(kidsWithCandies(candies, extraCandies  
candies = [12,1,12]  
extraCandies = 10  
print(kidsWithCandies(candies, extraCandies))

Output:



**7. 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,**

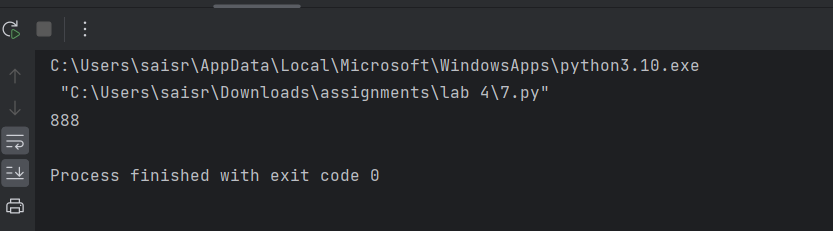
**respectively.**

**Return the max difference between a and b.**

Coding:

def maximumGap(num: int) -> int:  
 num\_str = str(num)  
 max\_num = int(''.join('9' if c != '0' else c for c in num\_str))  
 min\_num = int(''.join('1' if c == '9' else '0' if c == '0' else '1' for c in num\_str))  
 return max\_num - min\_num  
  
  
num = 555  
print(maximumGap(num))

Output:



**8. Check If a String Can Break Another String**

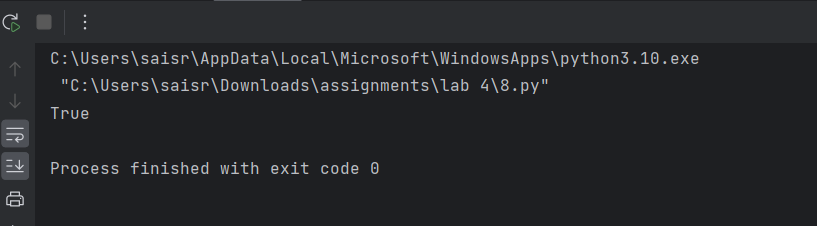
**Given two strings: s1 and s2 with the same size, check if some permutation of string s1**

**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.**

Coding:

def checkIfCanBreak(s1: str, s2: str) -> bool:  
 s1\_sorted = sorted(s1)  
 s2\_sorted = sorted(s2)  
  
 return (all(x >= y for x, y in zip(s1\_sorted, s2\_sorted)) or  
 all(x >= y for x, y in zip(s2\_sorted, s1\_sorted)))  
  
  
s1 = "abc"  
s2 = "xya"  
print(checkIfCanBreak(s1, s2))

Output:



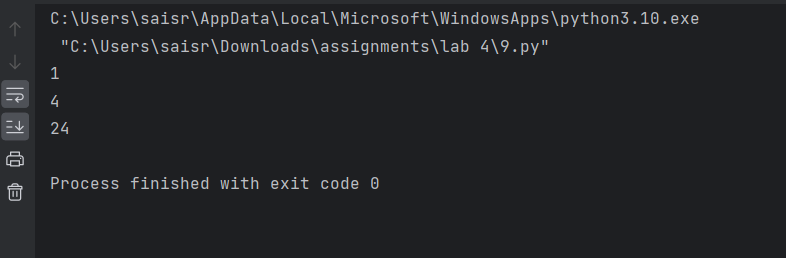
**9. Number of Ways to Wear Different Hats to Each Other**

**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.**

Coding:

MOD = 10 \*\* 9 + 7  
  
  
def numberWays(hats):  
 n = len(hats)  
 max\_hat = 40  
 hat\_to\_people = [[] for \_ in range(max\_hat + 1)]  
  
 for person, hat\_list in enumerate(hats):  
 for hat in hat\_list:  
 hat\_to\_people[hat].append(person)  
  
 dp = [0] \* (1 << n)  
 dp[0] = 1  
  
 for hat in range(1, max\_hat + 1):  
 for mask in range((1 << n) - 1, -1, -1):  
 for person in hat\_to\_people[hat]:  
 if mask & (1 << person) == 0:  
 dp[mask | (1 << person)] += dp[mask]  
 dp[mask | (1 << person)] %= MOD  
  
 return dp[(1 << n) - 1]  
  
  
hats = [[3, 4], [4, 5], [5]]  
print(numberWays(hats))   
  
hats = [[3, 5, 1], [3, 5]]  
print(numberWays(hats))   
  
hats = [[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]  
print(numberWays(hats))

Output:



**10. 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 willbe exactly one destination city.**

Coding:

def destCity(paths):  
 outgoing = set()  
  
 for path in paths:  
 outgoing.add(path[0])  
  
 for path in paths:  
 if path[1] not in outgoing:  
 return path[1]  
  
paths = [["London", "New York"], ["New York", "Lima"], ["Lima", "Sao Paulo"]]  
print(destCity(paths))   
  
paths = [["B", "C"], ["D", "B"], ["C", "A"]]  
print(destCity(paths))   
  
paths = [["A", "Z"]]  
print(destCity(paths))

Output:

