ASSIGNMENT-3

1.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.

Example Input: arr = [1,2,3] Output: 2 Explanation: 1 and 2 are counted cause 2 and 3 are in arr. Example 2: Input: arr = [1,1,3,3,5,5,7,7] Output: 0 Explanation: No numbers are counted, cause there is no 2, 4, 6, or 8 in arr

PROGRAM:

```
def count_elements(arr):
    unique_nums = set(arr)
    count = 0
    for num in arr:
        if num + 1 in unique_nums:
            count += 1
    return count
arr1 = [1, 2, 3]
print(count_elements(arr1))
arr2 = [1, 1, 3, 3, 5, 5, 7, 7]
print(count_elements(arr2))
```

OUTPUT:

```
the no.of elements are: 2
the no.of elements are: 0
```

TIME COMPLEXITY: O(n)

2.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.

Example 1: Input: s = "abc", shift = [[0,1],[1,2]] Output: "cab" Explanation: [0,1] means shift to left by 1. "abc" -> "bca" [1,2] means shift to right by 2. "bca" -> "cab"

Example 2: Input: s = "abcdefg", shift = [[1,1],[1,1],[0,2],[1,3]] Output: "efgabcd" Explanation: [1,1] means shift to right by 1. "abcdefg" -> "gabcdef" [1,1] means shift to right by 1. "gabcdef" -> "fgabcde" [0,2] means shift to left by 2. "fgabcde" -> "abcdefg" [1,3] means shift to right by 3. "abcdefg" -> "efgabcd"

PROGRAM:

```
def stringShift(s, shift):
    total_shift = 0
    for direction, amount in shift:
        if direction == 0:
            total_shift -= amount
        else:
            total_shift += amount
    total_shift %= len(s)
    return s[-total_shift:] + s[:-total_shift]
s1 = "abc"
shift1 = [[0,1],[1,2]]
print(stringShift(s1, shift1))
s2 = "abcdefg"
shift2 = [[1,1],[1,1],[0,2],[1,3]]
print(stringShift(s2, shift2))
```

OUTPUT:

```
the string shift:  cab
the string shift:  efgabcd
```

TIME COMPLEXITY: O(n)

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. 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.

Example 1: Input: mat = [[0,0],[1,1]] Output: 0

Example 2: Input: mat = [[0,0],[0,1]] Output: 1 Example 3: Input: mat = [[0,0],[0,0]] Output: -1

PRPOGRAM:

```
from typing import List
```

```
class BinaryMatrix:
    def __init__(self, mat: List[List[int]]):
        self.mat = mat
    def get(self, row: int, col: int) -> int:
        return self.mat[row][col]
    def dimensions(self) -> List[int]:
        return [len(self.mat), len(self.mat[0])]
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
mat1 = BinaryMatrix([[0,0],[1,1]])
print(leftMostColumnWithOne(mat1))
mat2 = BinaryMatrix([[0,0],[0,1]])
print(leftMostColumnWithOne(mat2))
mat3 = BinaryMatrix([[0,0],[0,0]])
print(leftMostColumnWithOne(mat3))
```

OUTPUT:

TIME COMPLEXITY: O(rows+columns)

4.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. Example 1: Input: ["FirstUnique","showFirstUnique","add","showFirstUnique","add","showFirstUnique","a dd","showFirstUnique"] [[[2,3,5]],[],[5],[],[2],[],[3],[]] Output: [null,2,null,2,null,3,null,-1] Explanation: FirstUnique firstUnique = new FirstUnique([2,3,5]); firstUnique.showFirstUnique(); // return 2 firstUnique.add(5); // the queue is now [2,3,5,5] firstUnique.showFirstUnique(); // return 2 firstUnique.add(2); // the queue is now [2,3,5,5,2] firstUnique.showFirstUnique(); // return 3 firstUnique.add(3); // the queue is now [2,3,5,5,2,3] firstUnique.showFirstUnique(); // return -1

PROGRAM:

```
class ListNode:
    def __init__(self, val):
        self.val = val
        self.prev = None
        self.next = None
class FirstUnique:
    def __init__(self, nums):
        self.head = ListNode(-1)  # Dummy head node
        self.tail = ListNode(-1)  # Dummy tail node
        self.head.next = self.tail
        self.tail.prev = self.head
        self.num_count = {}
        for num in nums:
            self.add(num)
    def showFirstUnique(self) -> int:
        if self.head.next == self.tail:
            return -1
        return self.head.next.val
    def add(self, value: int) -> None:
        if value in self.num_count:
            node = self.num_count[value]
            if node:
                self.remove(node)
```

```
            self.num_count[value] = None
        else:
            node = ListNode(value)
            self.append(node)
            self.num_count[value] = node
    def append(self, node):
        node.prev = self.tail.prev
        node.next = self.tail
        self.tail.prev.next = node
        self.tail.prev = node
    def remove(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev
firstUnique = FirstUnique([2,3,5])
print(firstUnique.showFirstUnique())  # Output: 2
firstUnique.add(5)
print(firstUnique.showFirstUnique())
firstUnique.add(2)
print(firstUnique.showFirstUnique())
firstUnique.add(3)
print(firstUnique.showFirstUnique())
```

OUTPUT:

```
2
2
3
-1
```

TIME COMPLEXITY: O(1)

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.

Example 1: Input: root = [0,1,0,0,1,0,null,null,1,0,0], arr = [0,1,0,1] Output: true Explanation: The path 0 -> 1 -> 0 -> 1 is a valid sequence (green color in the figure). Other valid sequences are: 0 -> 1 -> 1 -> 0 0 -> 0 -> 0

PROGRAM:

```
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 not node or index >= len(arr) or node.val != arr[index]:
            return False
        if not node.left and not node.right and index == len(arr) - 1:
            return True
        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 = None
root.right.right = None
root.left.left.left = None
root.left.left.right = None
root.left.right.left = TreeNode(1)
root.left.right.right = TreeNode(0)
arr = [0, 1, 0, 1]
print(isValidSequence(root, arr))
```

OUTPUT:

```
False
```

TIME COMPLEXITY: O(n)

6. 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. Example 1: Input: candies = [2,3,5,1,3], extraCandies = 3 Output: [true,true,true,false,true] Explanation: If you give all extraCandies to: - Kid 1, they will have 2 + 3 = 5 candies, which is the greatest among the kids. - Kid 2, they will have 3 + 3 = 6 candies, which is the greatest among the kids. - Kid 3, they will have 5 + 3 = 8 candies, which is the greatest among the kids. - Kid 4, they will have 1 + 3 = 4 candies, which is not the greatest among the kids. - Kid 5, they will have 3 + 3 = 6 candies, which is the greatest among the kids.

Example 2: Input: candies = [4,2,1,1,2], extraCandies = 1 Output: [true,false,false,false,false] Explanation: There is only 1 extra candy. Kid 1 will always have the greatest number of candies, even if a different kid is given the extra candy. Example 3: Input: candies = [12,1,12], extraCandies = 10 Output: [true,false,true]

PROGRAM:

```
def kidsWithCandies(candies, extraCandies):
    max_candies = max(candies)
    result = []
    for candy in candies:
        if candy + extraCandies >= max_candies:
            result.append(True)
        else:
            result.append(False)
    return result
candies1 = [2, 3, 5, 1, 3]
extraCandies1 = 3
print(kidsWithCandies(candies1, extraCandies1))
candies2 = [4, 2, 1, 1, 2]
extraCandies2 = 1
print(kidsWithCandies(candies2, extraCandies2))
candies3 = [12, 1, 12]
extraCandies3 = 10
print(kidsWithCandies(candies3, extraCandies3))
```

OUTPUT:
```
[True, True, True, False, True]
[True, False, False, False, False]
[True, False, True]
```

TIME COMPLEXITY: O(n)

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.

Example 1: Input: num = 555 Output: 888 Explanation: The first time pick x = 5 and y = 9 and store the new integer in a. The second time pick x = 5 and y = 1 and store the new integer in b. We have now a = 999 and b = 111 and max difference = 888

Example 2: Input: num = 9 Output: 8 Explanation: The first time pick x = 9 and y = 9 and store the new integer in a. The second time pick x = 9 and y = 1 and store the new integer in b. We have now a = 9 and b = 1 and max difference =8

PROGRAM:

```
def maxDiff(num):
    num_str = str(num)
    max_a = num_str.replace(max(num_str), '9')
    min_b = num_str.replace('1' if num_str[0] != '1' else '0', '1')
    max_b = num_str.replace(min(num_str), '1')
    if max_b[0] == '0':
        for i in range(1, len(max_b)):
            if max_b[i] != '0':
                max_b = max_b.replace(max_b[i], '0')
                break
    min_a = num_str.replace('9', '1')
    return int(max_a) - int(min_b), int(max_b) - int(min_a)
num1 = 555
print(max(maxDiff(num1)))
num2 = 9
print(max(maxDiff(num2)))
```

OUTPUT:
```
the maximum defference is : 444
the maximum differance is : 0
```

TIME COMPLEXITY: O(log n)

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.

Example 1: Input: s1 = "abc", s2 = "xya" Output: true Explanation: "ayx" is a permutation of s2="xya" which can break to string "abc" which is a permutation of s1="abc".

Example 2: Input: s1 = "abe", s2 = "acd" Output: false Explanation: All permutations for s1="abe" are: "abe", "aeb", "bae", "bea", "eab" and "eba" and all permutation for s2="acd" are: "acd", "adc", "cad", "cda", "dac" and "dca". However, there is not any permutation from s1 which can break some permutation from s2 and vice-versa.

Example 3: Input: s1 = "leetcodee", s2 = "interview" Output: true

PROGRAM:

```
def canBreak(s1, s2):
    s1_chars = sorted(s1)
    s2_chars = sorted(s2)
    s1_breaks_s2 = True
    s2_breaks_s1 = True
    for char1, char2 in zip(s1_chars, s2_chars):
        if char1 < char2:
            s1_breaks_s2 = False
        elif char1 > char2:
            s2_breaks_s1 = False
        if not s1_breaks_s2 and not s2_breaks_s1:
            return False
    return True
s1 = "abc"
s2 = "xya"
s1 = "abe"
s2 = "acd"
print(canBreak(s1, s2))
```

s1 = "leetcodee"

s2 = "interview"

print(canBreak(s1, s2))

OUTPUT:

```
True
False
True
```

TIME COMPLEXITY: O(n log n)

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.

Example 1: Input: hats = [[3,4],[4,5],[5]] Output: 1 Explanation: There is only one way to choose hats given the conditions. First person choose hat 3, Second person choose hat 4 and last one hat 5. Example 2: Input: hats = [[3,5,1],[3,5]] Output: 4 Explanation: There are 4 ways to choose hats: (3,5), (5,3), (1,3) and (1,5)

Example 3: Input: hats = [[1,2,3,4],[1,2,3,4],[1,2,3,4],[1,2,3,4]] Output: 24 Explanation: Each person can choose hats labeled from 1 to 4. Number of Permutations of (1,2,3,4) = 24.

PROGRAM:

```
MOD = 10**9 + 7

def numberWays(hats):
    n = len(hats)
    max_hat = 40
    dp = [0] * (1 << n)
    dp[0] = 1
    hat_to_people = [[] for _ in range(max_hat + 1)]
    for i, person in enumerate(hats):
        for hat in person:
            hat_to_people[hat].append(i)
    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):
                    continue
```

```
            dp[mask | (1 << person)] += dp[mask]
            dp[mask | (1 << person)] %= MOD
    return dp[(1 << n) - 1]
hats1 = [[3, 4], [4, 5], [5]]
print(numberWays(hats1))
hats2 = [[3, 5, 1], [3, 5]]
print(numberWays(hats2))
hats3 = [[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
print(numberWays(hats3))
```

OUTPUT:

TIME COMPLEXITY: O(n*2^n*m)

10. A permutation of an array of integers is an arrangement of its members into a sequence or linear order. ● For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1]. The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order). ● For example, the next permutation of arr = [1,2,3] is [1,3,2]. ● Similarly, the next permutation of arr = [2,3,1] is [3,1,2]. ● While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement. Given an array of integers nums, find the next permutation of nums. The replacement must be in place and use only constant extra memory.
Example 1: Input: nums = [1,2,3] Output: [1,3,2]

Example 2: Input: nums = [3,2,1] Output: [1,2,3]

Example 3: Input: nums = [1,1,5] Output: [1,5,1]

PROGRAM:

```
def nextPermutation(nums):
    i = len(nums) - 2
    while i >= 0 and nums[i] >= nums[i + 1]:
        i -= 1
    if i >= 0:
        j = len(nums) - 1
        while nums[j] <= nums[i]:
```

```
            j -= 1
        nums[i], nums[j] = nums[j], nums[i]
    left, right = i + 1, len(nums) - 1
    while left < right:
        nums[left], nums[right] = nums[right], nums[left]
        left += 1
        right -= 1
nums1 = [1, 2, 3]
nextPermutation(nums1)
print(nums1)
nums2 = [3, 2, 1]
nextPermutation(nums2)
print(nums2)
nums3 = [1, 1, 5]
nextPermutation(nums3)
print(nums3)
```

OUTPUT:

```
[1, 3, 2]
[1, 2, 3]
[1, 5, 1]
```

TIME COMPLEXITY: O(n)