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.

Code with output:

```
def count_elements(arr):
    element_set = set(arr)
    count = 0
    for x in arr:
        if x + 1 in element_set:
            count += 1
    return count
arr1 = [1, 2, 3]
print(count_elements(arr1))  # Output: 2

arr2 = [1, 1, 3, 3, 5, 5, 7, 7]
print(count_elements(arr2))  # Output: 0

#Time complexity:O(n)
#Space 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.

Code with output:

```
def perform_string_shifts(s, shift):
    net_shift = 0
```

```
    for direction, amount in shift:
        if direction == 0:
            net_shift -= amount
        else:
            net_shift += amount
    n = len(s)
    net_shift %= n
    if net_shift > 0:  # right shift
        s = s[-net_shift:] + s[:-net_shift]
    elif net_shift < 0:  # left shift
        net_shift = -net_shift
        s = s[net_shift:] + s[:net_shift]

    return s
s1 = "abc"
shift1 = [[0,1],[1,2]]
print(perform_string_shifts(s1, shift1))  # Output: "cab"

s2 = "abcdefg"
shift2 = [[1,1],[1,1],[0,2],[1,3]]
print(perform_string_shifts(s2, shift2))  # Output: "efgabcd"

#Time complexity:O(m+n)
#Space 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.

Code with output:

```
class 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])]

def leftMostColumnWithOne(binaryMatrix):
    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  # Move left
        else:
            current_row += 1  # Move down

    return leftmost_col
mat1 = [[0, 0], [1, 1]]
binaryMatrix1 = BinaryMatrix(mat1)
print(leftMostColumnWithOne(binaryMatrix1))  # Output: 0

mat2 = [[0, 0], [0, 1]]
binaryMatrix2 = BinaryMatrix(mat2)
print(leftMostColumnWithOne(binaryMatrix2))  # Output: 1

mat3 = [[0, 0], [0, 0]]
binaryMatrix3 = BinaryMatrix(mat3)
print(leftMostColumnWithOne(binaryMatrix3))  # Output: -1

#Time complexity:O(m+n)
#Space complexity:O(1)
```

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.

Code with output:

```
from collections import deque


class FirstUnique:
    def __init__(self, nums):
        self.queue = deque()
        self.counts = {}

        for num in nums:
            self.add(num)

    def showFirstUnique(self):
        while self.queue and self.counts[self.queue[0]] > 1:
            self.queue.popleft()

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

    def add(self, value):
        if value in self.counts:
            self.counts[value] += 1
        else:
```

```
            self.counts[value] = 1
            self.queue.append(value)
commands = ["FirstUnique", "showFirstUnique", "add", "showFirstUnique", "add", "showFirstUnique", "add", "showFirstUnique"]
args = [[[2, 3, 5]], [], [5], [], [2], [], [3], []]
outputs = [None]

firstUnique = FirstUnique(*args[0])
outputs.append(firstUnique.showFirstUnique())
firstUnique.add(5)
outputs.append(firstUnique.showFirstUnique())
firstUnique.add(2)
outputs.append(firstUnique.showFirstUnique())
firstUnique.add(3)
outputs.append(firstUnique.showFirstUnique())

print(outputs)  # [None, 2, None, 2, None, 3, None, -1]

#Time complexity:O(1)
#Space complexity:)O(n)
```

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.

Code with output:

```
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

arr2 = [0, 0, 1]
print(isValidSequence(root, arr2))  # Output: False

arr3 = [0, 1, 1]
print(isValidSequence(root, arr3))  # Output: False

#Time complexity:O(n)
#Space complexity:O(log n)
```

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.

Code with output:

```
def kidsWithCandies(candies, extraCandies):
    max_candies = max(candies)
    result = []

    for candy in candies:
```

```
        result.append(candy + extraCandies >= max_candies)


    return result
candies1 = [2, 3, 5, 1, 3]

extraCandies1 = 3

print(kidsWithCandies(candies1, extraCandies1))  # Output: [true, true, true, false, true]


candies2 = [4, 2, 1, 1, 2]

extraCandies2 = 1

print(kidsWithCandies(candies2, extraCandies2))  # Output: [true, false, false, false, false]


candies3 = [12, 1, 12]

extraCandies3 = 10

print(kidsWithCandies(candies3, extraCandies3))  # Output: [true, false, true]


#Time complexity:O(n)

#Space 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.

Code with output:

```
def maxDifference(num):
    num_str = str(num)
    max_val_str = num_str
    for digit in num_str:
        if digit != '9':
            max_val_str = num_str.replace(digit, '9')
            break
    max_val = int(max_val_str)
    if num_str[0] != '1':
        min_val_str = num_str.replace(num_str[0], '1')
    else:
        min_val_str = num_str
        for digit in num_str[1:]:
            if digit != '0' and digit != '1':
                min_val_str = num_str.replace(digit, '0')
                break
    min_val = int(min_val_str)

    return max_val - min_val
num1 = 555
print(maxDifference(num1))  # Output: 888

num2 = 9
print(maxDifference(num2))  # Output: 8

num3 = 123456
```

```
print(maxDifference(num3))  # Output: 820000


#Time complexity:O(n)
#Space complexity:O(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.

Code with output:

```
def checkIfCanBreak(s1, s2):
    # Sort both strings
    sorted_s1 = sorted(s1)
    sorted_s2 = sorted(s2)
    can_s1_break_s2 = all(c1 >= c2 for c1, c2 in zip(sorted_s1, sorted_s2))
    # Check if sorted_s2 can break sorted_s1
    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

s1 = "leetcodee"
s2 = "interview"
print(checkIfCanBreak(s1, s2))  # Output: True

#Time complexity:O(n log n)
#Space complexity:O(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.

Code with output:

```
def numberWays(hats):
    n = len(hats)
    all_masks = 1 << n
    dp = [0] * all_masks
    dp[0] = 1
    hats_map = [[] for _ in range(41)]
    for person, preferred_hats in enumerate(hats):
        for hat in preferred_hats:
            hats_map[hat].append(person)
```

```
    for hat in range(1, 41):
        for mask in range(all_masks - 1, -1, -1):
            if dp[mask] == 0:
                continue
            for person in hats_map[hat]:
                if mask & (1 << person) == 0:
                    new_mask = mask | (1 << person)
                    dp[new_mask] = (dp[new_mask] + dp[mask]) % MOD

    return dp[all_masks - 1]
hats1 = [[3, 4], [4, 5], [5]]
print(numberWays(hats1))  # Output: 1

hats2 = [[3, 5, 1], [3, 5]]
print(numberWays(hats2))  # Output: 4

hats3 = [[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4]]
print(numberWays(hats3))  # Output: 24

#Time complexity:O(2^n)
#Space complexity:O(2^n)
```

10. Next Permutation 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.

Code with output:

```
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]
```

```
nums2 = [3, 2, 1]
nextPermutation(nums2)
print(nums2)  # Output: [1, 2, 3]

nums3 = [1, 1, 5]
nextPermutation(nums3)
print(nums3)  # Output: [1, 5, 1]

#Time complexity:O(n)
#Space complexity:O(1)
```