

DATE:8/6/24

ASSIGNMENT-3

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:

```
def countElements(arr,n):  
    # Initialize count as zero  
    count = 0  
    # Iterate over each element  
    for i in range(n):  
        # Store element in int x  
        x = arr[i]  
        # Calculate x + 1  
        xPlusOne = x + 1  
        # Initialize found as false  
        found = False  
        # Run loop to search for x + 1  
        # after the current element  
        for j in range(i + 1,n,1):  
            if (arr[j] == xPlusOne):  
                found = True  
                break  
        # Run loop to search for x + 1  
        # before the current element  
        k = i - 1  
        while(found == False and k >= 0):  
            if (arr[k] == xPlusOne):  
                found = True
```

```

break

k -= 1

# if found is true, increment count

if (found == True):

    count += 1

return count

# Driver program

if __name__ == '__main__':

    arr = [1, 2, 3]

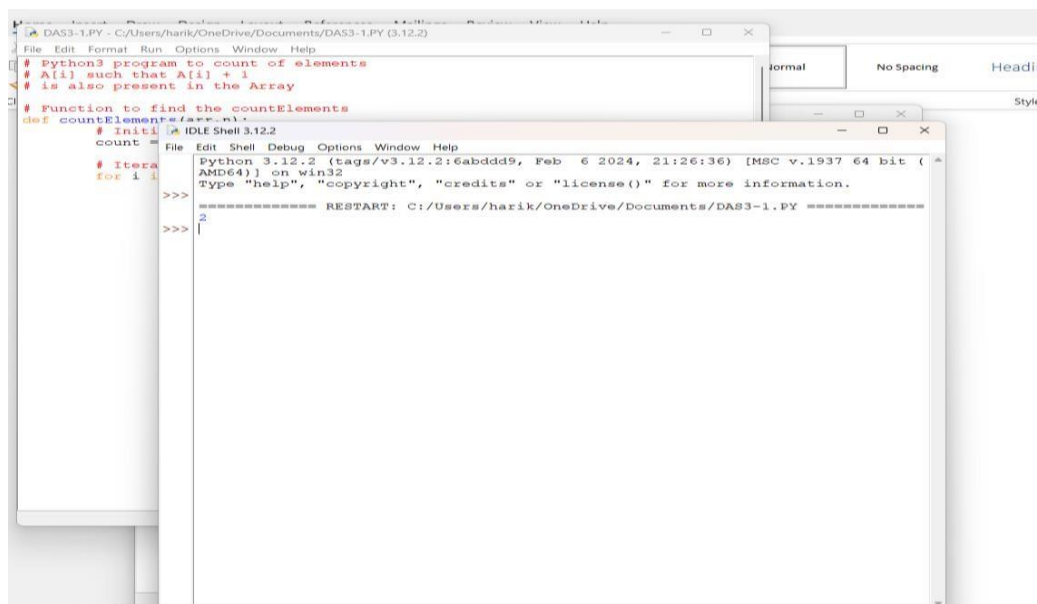
    n = len(arr)

    # call countElements function on array

    print(countElements(arr, n))

```

OUTPUT:



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.

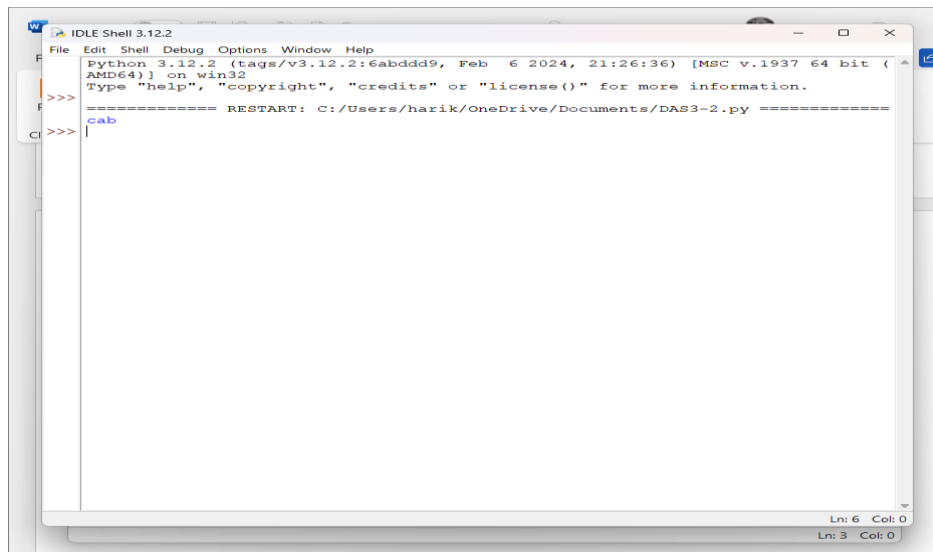
Return the final string after all operations.

CODE:

```
def stringShift(s, shift):
    val = 0
    for i in range(len(shift)):
        # If shift[i][0] = 0, then left shift
        # Otherwise, right shift
        val += -shift[i][1] if shift[i][0] == 0 else shift[i][1]
    # Stores length of the string
    Len = len(s)
    # Effective shift calculation
    val = val % Len
    # Stores modified string
    result = ""
    # Right rotation
    if (val > 0):
        result = s[Len - val:Len] + s[0: Len - val]
    # Left rotation
    else:
        result = s[-val: Len] + s[0: -val]
    print(result)

# Driver Code
s = "abc"
shift = [
    [ 0, 1 ],
    [ 1, 2 ]
]
stringShift(s, shift)
```

OUTPUT:



```
Python 3.12.2 (tags/v3.12.2:6abddd9, Feb 6 2024, 21:26:36) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/harik/OneDrive/Documents/DAS3-2.py =====
>>>
```

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

CODE:

```
# Python3 implementation to find the
```

```
# Leftmost Column with atleast a
```

```
# 1 in a sorted binary matrix
```

```
import sys
```

```
N = 3
```

```
# Function to search for the
```

```
# leftmost column of the matrix
```

```
# with atleast a 1 in sorted
# binary matrix
def search(mat, n, m):

    a = sys.maxsize

    # Loop to iterate over all the
    # rows of the matrix
    for i in range (n):
        low = 0
        high = m - 1
        ans = sys.maxsize

        # Binary Search to find the
        # leftmost occurrence of the 1
        while (low <= high):
            mid = (low + high) // 2

            # Condition if the column
            # contains the 1 at this
            # position of matrix
            if (mat[i][mid] == 1):

                if (mid == 0):
                    ans = 0
                    break

            elif (mat[i][mid - 1] == 0):
                ans = mid
                break
```

```
if (mat[i][mid] == 1):
    high = mid - 1
else:
    low = mid + 1

# If there is a better solution
# then update the answer
if (ans < a):
    a = ans

# Condition if the solution
# doesn't exist in the matrix
if (a == sys.maxsize):
    return -1
return a + 1

# Driver Code
if __name__ == "__main__":

    mat = [[0, 0, 0],
            [0, 0, 1],
            [0, 1, 1]]
    print(search(mat, 3, 3))
```

OUTPUT:

```
File Edit Shell Debug Options Window Help
Python 3.12.2 (tags/v3.12.2:6abddd9, Feb 6 2024, 21:26:36) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> = RESTART: C:/Users/harik/OneDrive/Documents/DAS3-3.PY
>>> |

# Store
len = 1
# Effect
val = v
# Store
result
# Right
if (val
# Left
else:
print(x
```

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:

```
class FirstUnique:
```

```
def __init__(self, nums: List[int]):
```

```
self.cnt = Counter(nums)
```

```
self.unique = OrderedDict({v: 1 for v in nums if self.cnt[v] == 1})
```

```
def showFirstUnique(self) -> int:
```

```
return -1 if not self.unique else next(v for v in self.unique.keys())
```

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

```
self.cnt[value] += 1
```

```
if self.cnt[value] == 1:
```

```
self.unique[value] = 1
```

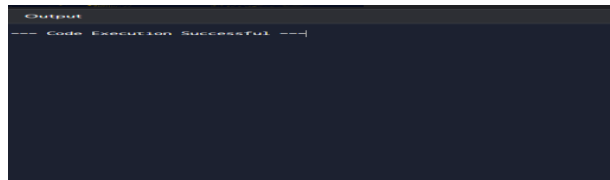
```
elif value in self.unique:
```

```
self.unique.pop(value)
```

Your FirstUnique object will be instantiated and called as such:

```
# obj = FirstUnique(nums)
# param_1 = obj.showFirstUnique()
# obj.add(value)
```

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.

CODE:

```
# Python program to see if
# there is a root to leaf path
# with given sequence
# Class of Node
class Node:
# Constructor to create a
# node in Binary Tree
def __init__(self, val):
    self.val = val
    self.left = None
    self.right = None
# Util function
def existPathUtil(root, arr, n, index):

# If root is NULL or reached
# end of the array
```



```
if not root or index == n:
```

```
    return False
```

```
# If current node is leaf
```

```
if not root.left and not root.right:
```

```
    if root.val == arr[index] and index == n-1:
```

```
        return True
```

```
    return False
```

```
# If current node is equal to arr[index] this means
```

```
# that till this level path has been matched and
```

```
# remaining path can be either in left subtree or
```

```
# right subtree.
```

```
return ((index < n) and (root.val == arr[index])) and \
```

```
(existPathUtil(root.left, arr, n, index+1) or \
```

```
existPathUtil(root.right, arr, n, index+1)))
```

```
# Function to check given sequence of root to leaf path exist
```

```
# in tree or not.
```

```
# index represents current element in sequence of root to
```

```
# leaf path
```

```
def existPath(root, arr, n, index):
```

```
    if not root:
```

```
        return (n == 0)
```

```
    return existPathUtil(root, arr, n, 0)
```

```
# Driver Code
```

```
if __name__ == "__main__":
```

```
    arr = [5, 8, 6, 7]
```

```

n = len(arr)
root = Node(5)
root.left = Node(3)
root.right = Node(8)
root.left.left = Node(2)
root.left.right = Node(4)
root.left.left.left = Node(1)
root.right.left = Node(6)
root.right.left.right = Node(7)

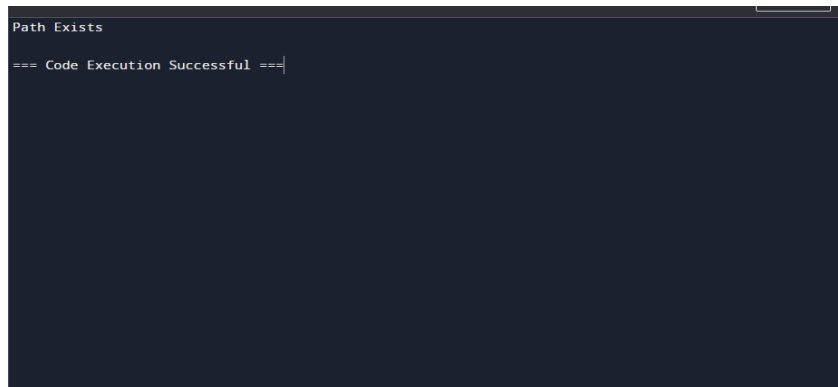
```

```

if existPath(root, arr, n, 0):
    print("Path Exists")
else:
    print("Path does not Exist")

```

OUTPUT:



```

Path Exists
=== Code Execution Successful ===

```

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 i th 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 i th 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:

```

class Solution:

```

```

def kidsWithCandies(self, candies: List[int], extraCandies: int) -> List[bool]:
    n = len(candies)
    result = [False]*n
    maxCandies = max(candies)
    for i in range(n):
        if candies[i] + extraCandies >= maxCandies:
            result[i] = True
    return result

```

OUTPUT:

[true,true,true,false,true]



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** ($0 \leq x \leq 9$).
- Pick another digit **y** ($0 \leq y \leq 9$). 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:

class Solution:

```

    def maxDiff(self, num: int) -> int:

```

```

        a, b = str(num), str(num)

```

```

        for c in a:

```

```

            if c != "9":

```

```

                a = a.replace(c, "9")

```

```

                break

```

```

        if b[0] != "1":

```

```

            b = b.replace(b[0], "1")

```

```

        else:

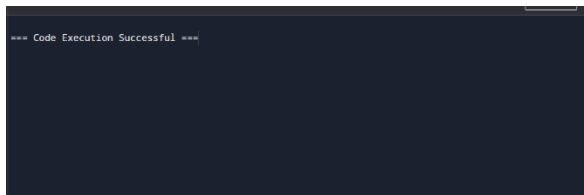
```

```

    for c in b[1:]:
        if c not in "01":
            b = b.replace(c, "0")
            break
    return int(a) - int(b)

```

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 $x[i] \geq y[i]$ (in alphabetical order) for all **i** between 0 and **n-1**.

CODE:

```
def arePermutation(str1, str2):
```

```
    # Get lengths of both strings
```

```
    n1 = len(str1)
```

```
    n2 = len(str2)
```

```
    # If length of both strings is not same,
```

```
    # then they cannot be Permutation
```

```
    if (n1 != n2):
```

```
        return False
```

```
    # Sort both strings
```

```
    a = sorted(str1)
```

```
str1 = " ".join(a)
```

```
b = sorted(str2)
```

```
str2 = " ".join(b)
```

```
# Compare sorted strings
```

```
for i in range(0, n1, 1):
```

```
if (str1[i] != str2[i]):
```

```
return False
```

```
return True
```

```
# Driver Code
```

```
if __name__ == '__main__':
```

```
str1 = "test"
```

```
str2 = "ttew"
```

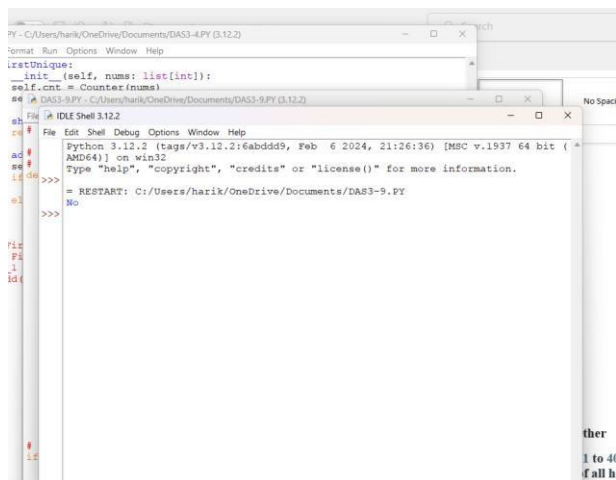
```
if (arePermutation(str1, str2)):
```

```
print("Yes")
```

```
else:
```

```
print("No")
```

```
OUTPUT:
```



```
Python 3.12.2 (tags/v3.12.2:6abddd9, Feb 6 2024, 21:26:36) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/harik/OneDrive/Documents/DAS3-9.PY
>>>
No
>>>
```

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 i th 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 $10^9 + 7$.

CODE:

```
class Solution:
    def numberWays(self, hats: List[List[int]]) -> int:
        g = defaultdict(list)
        for i, h in enumerate(hats):
            for v in h:
                g[v].append(i)
        mod = 10**9 + 7
        n = len(hats)
        m = max(max(h) for h in hats)
        f = [[0] * (1 << n) for _ in range(m + 1)]
        f[0][0] = 1
        for i in range(1, m + 1):
            for j in range(1 << n):
                f[i][j] = f[i - 1][j]
                for k in g[i]:
                    if j >> k & 1:
                        f[i][j] = (f[i][j] + f[i - 1][j ^ (1 << k)]) % mod
        return f[m][-1]
```

OUTPUT:



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:

```
def next_permutation(nums):
    # Find the first element from the right that is not in decreasing order
    i = len(nums) - 2
    while i >= 0 and nums[i] >= nums[i + 1]:
        i -= 1
    # If such an element is found, find the smallest element from the right that is greater than it
    if i >= 0:
        j = len(nums) - 1
        while nums[j] <= nums[i]:
            j -= 1
        # Swap the two elements
        nums[i], nums[j] = nums[j], nums[i]
    # Reverse the elements from i+1 to the end to get the next permutation
    nums[i + 1:] = reversed(nums[i + 1:])

nums = [3, 2, 1]
next_permutation(nums)
print(nums)
```

OUTPUT:

```
>>> ===== RESTART: C:/Users/harik/OneDrive/Documents/DAS3-S.PY ===== info
===== RESTART: C:/Users/harik/OneDrive/Documents/DAS3-109.PY =====
[1, 2, 3] off
>>> ===== RESTART: C:/Users/harik/OneDrive/Documents/DAS3-109.PY =====
[1, 2, 3] How
>>> 11-
  3 is
  full
  idea
```