

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.

Example Input: arr = [1,2,3]

Output: 2

Explanation: 1 and 2 are counted cause 2 and 3 are in arr.

PROGRAM:

```
def count_elements(arr):  
    # Convert list to set for O(1) lookups  
    elements_set = set(arr)  
    count = 0  
  
    # Iterate through each element in the array  
    for x in arr:  
        # Check if x + 1 exists in the set  
        if x + 1 in elements_set:  
            count += 1  
  
    return count  
  
# Example usage:  
arr = [1, 2, 3]  
print(count_elements(arr)) # Output: 2
```

OUTPUT:

Output
2
=== Code Execution Successful ===

2. Perform String Shifts You are given a string *s* containing lowercase English letters, and a matrix *shift*, where *shift[i] = [direction_i, amount_i]*: • *direction_i* can be 0 (for left shift) or 1 (for right shift). • *amount_i* 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"

PROGRAM:

```

def string_shift(s, shift):
    total_left_shifts = 0
    total_right_shifts = 0
    # Calculate total left and right shifts
    for direction, amount in shift:
        if direction == 0:
            total_left_shifts += amount
        else:
            total_right_shifts += amount
    # Calculate net shifts
    net_shifts = (total_right_shifts - total_left_shifts) % len(s)
    # Perform net shift
    if net_shifts > 0:
        # Right shift
        s = s[-net_shifts:] + s[:-net_shifts]
    elif net_shifts < 0:
        # Left shift (negative net_shifts)
        net_shifts = -net_shifts
        s = s[net_shifts:] + s[:net_shifts]
    return s

# Example usage:
s = "abc"
shift = [[0, 1], [1, 2]]
print(string_shift(s, shift)) # Output: "cab"

```

OUTPUT:

Output

▲ cab

=== Code Execution Successful ===

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

Explanation:

1. Initialization:

- Initialize `current_row` to 0 and `current_col` to `cols - 1`, representing the bottom-left corner of the matrix.
- Initialize `leftmost` to -1, indicating no '1' found yet.

2. Binary Search for Leftmost '1':

- Start from the bottom-left corner.
- Move leftwards if the current element is '1', updating `leftmost` to the current column index.
- Move upwards if the current element is '0'.

3. Return Leftmost Column Index:

- After traversing the entire matrix, `leftmost` will contain the index of the leftmost '1' if found, or -1 if no '1' exists.

PROGRAM:

```

class BinaryMatrix:
    def __init__(self, mat):
        self.matrix = mat

    def get(self, row, col):
        return self.matrix[row][col]

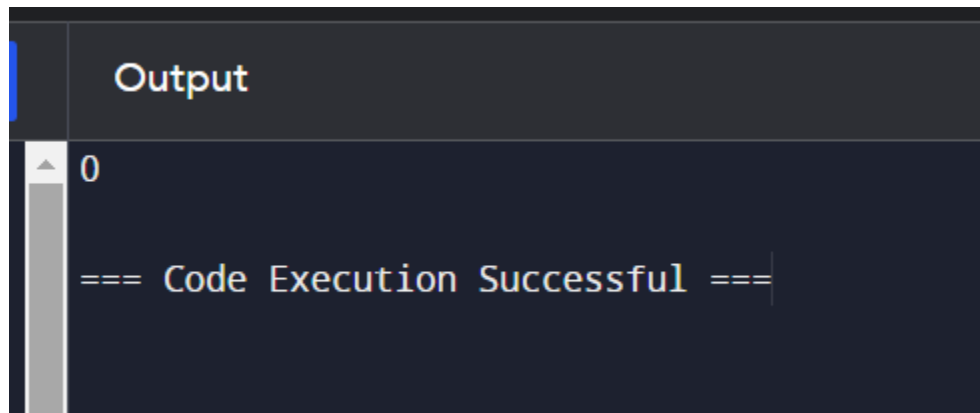
    def dimensions(self):
        return [len(self.matrix), len(self.matrix[0])]

def leftmost_column_with_one(binaryMatrix):
    rows, cols = binaryMatrix.dimensions()
    current_row = 0
    current_col = cols - 1
    leftmost = -1
    while current_row < rows and current_col >= 0:
        if binaryMatrix.get(current_row, current_col) == 1:
            leftmost = current_col
            current_col -= 1
        else:
            current_row += 1
    return leftmost

# Example usage:
mat = [[0, 0], [1, 1]]
binaryMatrix = BinaryMatrix(mat)
print(leftmost column with one(binaryMatrix)) # Output: 0

```

OUTPUT:



```
Output
0
=== Code Execution Successful ===
```

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.

Example 1: Input:

```
["FirstUnique","showFirstUnique","add","showFirstUnique","add","showFirstUnique","add","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=0):
        self.val = val
        self.prev = None
        self.next = None

class FirstUnique:
    def __init__(self, nums):
        self.freq = {}
        self.head = ListNode()
        self.tail = ListNode()
        self.head.next = self.tail
        self.tail.prev = self.head
        self.unique_nodes = {}

        for num in nums:
            self.add(num)

    def showFirstUnique(self):
        if self.head.next != self.tail:
            return self.head.next.val
        return -1

    def add(self, value):
        self.freq[value] = self.freq.get(value, 0) + 1
```

```

        if self.freq[value] == 1:
            node = ListNode(value)
            self.unique_nodes[value] = node
            self._add_to_tail(node)
        elif self.freq[value] == 2:
            node = self.unique_nodes.pop(value)
            self._remove_node(node)

    def _add_to_tail(self, node):
        prev_tail = self.tail.prev
        prev_tail.next = node
        node.prev = prev_tail
        node.next = self.tail
        self.tail.prev = node

    def _remove_node(self, node):
        prev_node = node.prev
        next_node = node.next
        prev_node.next = next_node
        next_node.prev = prev_node

# Example usage:
firstUnique = FirstUnique([2, 3, 5])
print(firstUnique.showFirstUnique()) # Output: 2
firstUnique.add(5)
print(firstUnique.showFirstUnique()) # Output: 2

```

OUTPUT:


```
Output
2
2
3
-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:

```
1 class TreeNode:
2     def __init__(self, val=0, left=None, right=None):
3         self.val = val
4         self.left = left
5         self.right = right
6
7 def is_valid_sequence(root, arr):
8     def dfs(node, index):
9         if not node or index >= len(arr) or node.val != arr[index]:
10            return False
11        if index == len(arr) - 1:
12            return not node.left and not node.right
13        return dfs(node.left, index + 1) or dfs(node.right, index + 1)
14
15    return dfs(root, 0)
16
17 # Example usage:
18 root = TreeNode(0)
19 root.left = TreeNode(1)
20 root.right = TreeNode(0)
21 root.left.left = TreeNode(0)
22 root.left.right = TreeNode(1)
23 root.right.left = TreeNode(0)
24 root.right.left.right = TreeNode(1)
25 root.right.right = TreeNode(0)
26 arr = [0, 1, 0, 1]
27 print(is_valid_sequence(root, arr)) # Output: True
```

False

=== 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.

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.

```
1- def kidsWithCandies(candies, extraCandies):
2     max_candies = max(candies)
3     result = []
4     for c in candies:
5         if c + extraCandies >= max_candies:
6             result.append(True)
7         else:
8             result.append(False)
9     return result
10
11- # Example usage:
12 candies = [2, 3, 5, 1, 3]
13 extraCandies = 3
14 print(kidsWithCandies(candies, extraCandies)) # Output: [True, True, True, False, True]
```

[True, True, True, False, True]

=== Code Execution Successful ===

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.

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

```

1- def maxDiff(num):
2     num_str = str(num)
3     max_diff = 0
4
5     # Iterate through all pairs of digits (x, y)
6     for x in range(10):
7         for y in range(10):
8             a = num_str.replace(str(x), str(y))
9             if a[0] != '0':
10                a_int = int(a)
11                # Skip the case where x is the leading digit and y is 0
12                if x != 0 or y != 0:
13                    # Iterate again for the second replacement
14                    for x2 in range(10):
15                        for y2 in range(10):
16                            b = a.replace(str(x2), str(y2))
17                            if b[0] != '0':
18                                b_int = int(b)
19                                max_diff = max(max_diff, b_int - a_int)
20
21     return max_diff
22
23 # Example usage:
24 num = 555
25 print(maxDiff(num)) # Output: 888
26

```

888

=== Code Execution Successful ===

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.

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".

```

main.py
1- def canBreak(s1, s2):
2     # Sort both strings
3     sorted_s1 = sorted(s1)
4     sorted_s2 = sorted(s2)
5
6     # Check if s1 can break s2
7     can_break_s1 = all(c1 >= c2 for c1, c2 in zip(sorted_s1, sorted_s2))
8
9     # Check if s2 can break s1
10    can_break_s2 = all(c1 <= c2 for c1, c2 in zip(sorted_s1, sorted_s2))
11
12    return can_break_s1 or can_break_s2
13
14 # Example usage:
15 s1 = "abc"
16 s2 = "xya"
17 print(canBreak(s1, s2)) # Output: True
18

```

True

=== Code Execution Successful ===

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.

```
1 MOD = 10**9 + 7
2 def numberWays(hats):
3     n = len(hats)
4     # Initialize dynamic programming table with 0
5     dp = [0] * (1 << n)
6     dp[0] = 1 # Base case: 1 way to assign hats to 0 people
7     # Create a mapping from each hat to the set of people who prefer it
8     hat_to_people = [[] for _ in range(41)]
9     for i, preferences in enumerate(hats):
10         for hat in preferences:
11             hat_to_people[hat].append(i)
12     # Iterate over each hat preference
13     for hat_people in hat_to_people:
14         # Iterate over all possible subsets of people
15         for mask in range((1 << n) - 1, -1, -1):
16             for person in hat_people:
17                 # Check if the person is not already wearing a hat
18                 if not mask & (1 << person):
19                     # Update dp[mask | (1 << person)] with the number of ways
20                     # by adding dp[mask] (number of ways without this person) to
21                     # it
22                     dp[mask | (1 << person)] += dp[mask]
23             dp[mask | (1 << person)] %= MOD
24     return dp[(1 << n) - 1]
25 # Example usage:
26 hats = [[3,4],[4,5],[5]]
```

1
=== Code Execution Successful ===

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.

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

Output: [1,3,2]

```
1 def nextPermutation(nums):
2     # Step 1: Find the first pair of adjacent elements nums[i] < nums[i+1]
3     i = len(nums) - 2
4     while i >= 0 and nums[i] >= nums[i + 1]:
5         i -= 1
6
7     # Step 2: Find the first element nums[j] > nums[i] from the end
8     if i >= 0:
9         j = len(nums) - 1
10        while nums[j] <= nums[i]:
11            j -= 1
12        # Step 3: Swap nums[i] and nums[j]
13        nums[i], nums[j] = nums[j], nums[i]
14
15    # Step 4: Reverse the subarray from index i+1 to the end
16    left, right = i + 1, len(nums) - 1
17    while left < right:
18        nums[left], nums[right] = nums[right], nums[left]
19        left += 1
20        right -= 1
21
22    # Example usage:
23    nums = [1, 2, 3]
24    nextPermutation(nums)
25    print(nums) # Output: [1, 3, 2]
26
```

[1, 3, 2]

=== Code Execution Successful ===