

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`.

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`.

Constraints:

- `1 <= arr.length <= 1000`
- `0 <= arr[i] <= 1000`

```
def count_elements(arr):
    elements = set(arr)
    count = 0
    for num in arr:
        if num + 1 in elements:
            count += 1
    return count
arr = [1, 2, 3]
print(count_elements(arr))
output
2
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"

Constraints:

- $1 \leq s.length \leq 100$
- s only contains lower case English letters.
- $1 \leq shift.length \leq 100$

```
def string_shifts(s, shift):
```

```
    total_shift = sum(amount if direction == 1 else -amount for direction, amount in shift) %  
    len(s)
```

```
    return s[-total_shift:] + s[:-total_shift]
```

```
s = "abc"
```

```
shift = [[0, 1], [1, 2]]
```

```
print(string_shifts(s, shift))
```

output

cab

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:

0	0
1	1

Input: `mat = [[0,0],[1,1]]`

Output: 0

Example 2:

0	0
0	1

Input: `mat = [[0,0],[0,1]]`

Output: 1

Example 3:

0	0
0	0

Input: `mat = [[0,0],[0,0]]`

Output: -1

Constraints:

- `rows == mat.length`
- `cols == mat[i].length`
- `1 <= rows, cols <= 100`
- `mat[i][j]` is either 0 or 1.
- `mat[i]` is sorted in non-decreasing order.

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

```
mat = [[0, 0], [1, 1]]
binaryMatrix = BinaryMatrix(mat)
print(leftmost_column_with_one(binaryMatrix))
```

output

0

2

2

3

Time complexity

$O(\min(\text{rows}, \text{cols}))$

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
```

Example 2:

Input:

```
["FirstUnique","showFirstUnique","add","add","add","add","add","showFirstUnique"]
```

```
[[[7,7,7,7,7,7]],[],[7],[3],[3],[7],[17],[]]
```

Output:

```
[null,-1,null,null,null,null,null,17]
```

Explanation:

```
FirstUnique firstUnique = new FirstUnique([7,7,7,7,7,7]);
firstUnique.showFirstUnique(); // return -1
firstUnique.add(7);           // the queue is now [7,7,7,7,7,7]
firstUnique.add(3);           // the queue is now [7,7,7,7,7,7,3]
firstUnique.add(3);           // the queue is now [7,7,7,7,7,7,3,3]
firstUnique.add(7);           // the queue is now [7,7,7,7,7,7,3,3,7]
firstUnique.add(17);          // the queue is now [7,7,7,7,7,7,3,3,7,17]
firstUnique.showFirstUnique(); // return 17
```

Example 3:

Input:

```
["FirstUnique","showFirstUnique","add","showFirstUnique"]
```

```
[[[809]],[],[809],[]]
```

Output:

```
[null,809,null,-1]
```

Explanation:

```
FirstUnique firstUnique = new FirstUnique([809]);
firstUnique.showFirstUnique(); // return 809
firstUnique.add(809);          // the queue is now [809,809]
firstUnique.showFirstUnique(); // return -1
```

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $1 \leq \text{nums}[i] \leq 10^8$
- $1 \leq \text{value} \leq 10^8$

```
from collections import deque, Counter
class FirstUnique:
    def __init__(self, nums):
        self.queue = deque(nums)
        self.count = Counter(nums)

    def showFirstUnique(self):
        while self.queue and self.count[self.queue[0]] > 1:
            self.queue.popleft()
        return self.queue[0] if self.queue else -1

    def add(self, value):
        self.queue.append(value)
        self.count[value] += 1
```

```
firstUnique = FirstUnique([2, 3, 5])
print(firstUnique.showFirstUnique())
firstUnique.add(5)
print(firstUnique.showFirstUnique())
firstUnique.add(2)
print(firstUnique.showFirstUnique())
firstUnique.add(3)
print(firstUnique.showFirstUnique())
```

output

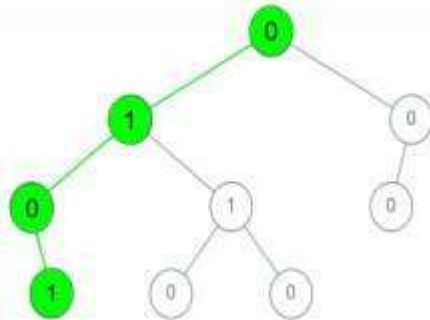
-1

Time 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.

Example 1:



Input: root = [0,1,0,0,1,0,null,null,1,0,0], arr = [0,1,0,1]

Output: true

Explanation:

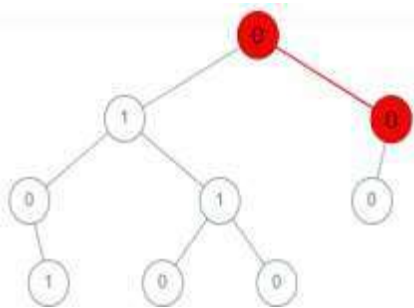
The path 0 -> 1 -> 0 -> 1 is a valid sequence (green color in the figure).

Other valid sequences are:

0 -> 1 -> 1 -> 0

0 -> 0 -> 0

Example 2:

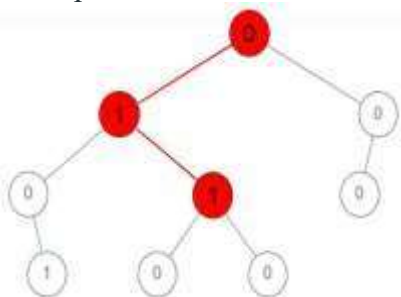


Input: root = [0,1,0,0,1,0,null,null,1,0,0], arr = [0,0,1]

Output: false

Explanation: The path 0 -> 0 -> 1 does not exist, therefore it is not even a sequence.

Example 3:



Input: root = [0,1,0,0,1,0,null,null,1,0,0], arr = [0,1,1]

Output: false

Explanation: The path 0 -> 1 -> 1 is a sequence, but it is not a valid sequence.

Constraints:

- $1 \leq \text{arr.length} \leq 5000$
- $0 \leq \text{arr}[i] \leq 9$
- Each node's value is between [0 - 9].

class TreeNode:

```
def __init__(self, val=0, left=None, right=None):  
    self.val = val  
    self.left = left  
    self.right = right
```

def is_valid_sequence(root, arr):

```
def dfs(node, i):  
    if not node or i >= len(arr) or node.val != arr[i]:  
        return False  
    if not node.left and not node.right:  
        return i == len(arr) - 1  
    return dfs(node.left, i + 1) or dfs(node.right, i + 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)  
arr = [0, 1, 0, 1]  
print(is_valid_sequence(root, arr))
```

output

True

Time complexity

$O(n * m)$

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.

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]`

Constraints:

- $n == \text{candies.length}$
- $2 \leq n \leq 100$
- $1 \leq \text{candies}[i] \leq 100$
- $1 \leq \text{extraCandies} \leq 50$

```
def kids_with_candies(candies, extraCandies):  
    max_candies = max(candies)  
    return [candy + extraCandies >= max_candies for candy in candies]
```

```
candies = [2, 3, 5, 1, 3]  
extraCandies = 3  
print(kids_with_candies(candies, extraCandies))
```

output

[True, True, 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` ($0 \leq x \leq 9$).
- 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

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

```
def max_diff(num):
    s = str(num)

    for digit in s:
        if digit != '9':
            a = int(s.replace(digit, '9'))
            break
    else:
        a = num

    if s[0] != '1':
        b = int(s.replace(s[0], '1'))
    else:
        for digit in s[1:]:
            if digit not in '01':
                b = int(s.replace(digit, '0'))
                break
        else:
            b = num

    return a - b
```

```
num = 555
print(max_diff(num))
```

output

888

Time 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`.

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 = "leetcode"`, `s2 = "interview"`

Output: true

Constraints:

- `s1.length == n`
- `s2.length == n`
- `1 <= n <= 10^5`
- All strings consist of lowercase English letters.

def can_break(s1, s2):

s1, s2 = sorted(s1), sorted(s2)

return all(c1 >= c2 for c1, c2 in zip(s1, s2)) or all(c2 >= c1 for c2, c1 in zip(s2, s1))

s1 = "abc"

s2 = "xya"

print(can_break(s1, s2))

output

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 `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 `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.

Constraints:

- $n == \text{hats.length}$
- $1 \leq n \leq 10$
- $1 \leq \text{hats}[i].\text{length} \leq 40$
- $1 \leq \text{hats}[i][j] \leq 40$

```
def number_ways(hats):
    from functools import lru_cache
    n = len(hats)
    hat_to_people = {i: [] for i in range(1, 41)}
    for i, hat_list in enumerate(hats):
        for hat in hat_list:
            hat_to_people[hat].append(i)

    @lru_cache(None)
    def dp(mask, hat):
        if mask == (1 << n) - 1:
            return 1
        if hat > 40:
            return 0
        result = dp(mask, hat + 1)
        for person in hat_to_people[hat]:
            if not mask & (1 << person):
                result += dp(mask | (1 << person), hat + 1)
        return result % (10**9 + 7)

    return dp(0, 1)

hats = [[3, 4], [4, 5], [5]]
print(number_ways(hats))
```

output

1

Time complexity

$O(n)$

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. 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]` Constraints:

- `1 <= nums.length <= 100`
- `0 <= nums[i] <= 100`

def next_permutation(nums):

i = j = len(nums) - 1

while i > 0 and nums[i - 1] >= nums[i]:

i -= 1

if i == 0:

nums.reverse()

return

k = i - 1

while nums[j] <= nums[k]:

j -= 1

nums[k], nums[j] = nums[j], nums[k]

l, r = k + 1, len(nums) - 1

while l < r:

nums[l], nums[r] = nums[r], nums[l]

l += 1

r -= 1

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

output

[1, 3, 2]

Time complexity

$O(n^2)$