**DSA – ASSIGNMENT 17**

💡 **Question 1** Given a string s, *find the first non-repeating character in it and return its index*. If it does not exist, return -1.

**Example 1:**

Input: s = "leetcode"

Output: 0

**Example 2:**

Input: s = "loveleetcode"

Output: 2

**Example 3:**

Input: s = "aabb"

Output: -1

**Solution. :-**

* Create an empty dictionary to store the frequency of each character.
* Iterate through the string and update the frequency of each character in the dictionary.
* Iterate through the string again and check the frequency of each character in the dictionary. Return the index of the first character with a frequency of 1.
* If no such character is found, return -1.

**def firstUniqChar(s):**

**charFreq = {}**

**# Update character frequency**

**for char in s:**

**charFreq[char] = charFreq.get(char, 0) + 1**

**# Find the first non-repeating character**

**for i in range(len(s)):**

**if charFreq[s[i]] == 1:**

**return i**

**return -1**

**s = "leetcode"**

**index = firstUniqChar(s)**

**print(index)**

💡 **Question 2** Given a **circular integer array** nums of length n, return *the maximum possible sum of a non-empty****subarray****of* nums.

A **circular array** means the end of the array connects to the beginning of the array. Formally, the next element of nums[i] is nums[(i + 1) % n] and the previous element of nums[i] is nums[(i - 1 + n) % n].

A **subarray** may only include each element of the fixed buffer nums at most once. Formally, for a subarray nums[i], nums[i + 1], ..., nums[j], there does not exist i <= k1, k2 <= j with k1 % n == k2 % n.

**Example 1:**

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

Output: 3

Explanation: Subarray [3] has maximum sum 3.

**Example 2:**

Input: nums = [5,-3,5]

Output: 10

Explanation: Subarray [5,5] has maximum sum 5 + 5 = 10.

**Example 3:**

Input: nums = [-3,-2,-3]

Output: -2

Explanation: Subarray [-2] has maximum sum -2.

**Solution. :-**

* Initialize two variables, maxSum and minSum, to the first element of the array nums.
* Initialize two variables, currentMax and currentMin, to the first element of the array nums.
* Iterate through the array from the second element:
  + Update currentMax by taking the maximum of nums[i] and currentMax + nums[i].
  + Update maxSum by taking the maximum of maxSum and currentMax.
  + Update currentMin by taking the minimum of nums[i] and currentMin + nums[i].
  + Update minSum by taking the minimum of minSum and currentMin.
* If minSum is equal to the sum of all elements in the array nums, return maxSum (this means that all elements in the array are negative, so we can't form a non-empty subarray with a positive sum).
* Otherwise, return the maximum of maxSum and the sum of all elements in the array nums minus minSum.

**def maxSubarraySumCircular(nums):**

**n = len(nums)**

**maxSum = nums[0]**

**minSum = nums[0]**

**currentMax = nums[0]**

**currentMin = nums[0]**

**totalSum = nums[0]**

**for i in range(1, n):**

**currentMax = max(nums[i], currentMax + nums[i])**

**maxSum = max(maxSum, currentMax)**

**currentMin = min(nums[i], currentMin + nums[i])**

**minSum = min(minSum, currentMin)**

**totalSum += nums[i]**

**if minSum == totalSum:**

**return maxSum**

**else:**

**return max(maxSum, totalSum - minSum)**

**nums = [1, -2, 3, -2]**

**maxSum = maxSubarraySumCircular(nums)**

**print(maxSum)**

💡 **Question 3** The school cafeteria offers circular and square sandwiches at lunch break, referred to by numbers 0 and 1 respectively. All students stand in a queue. Each student either prefers square or circular sandwiches.

The number of sandwiches in the cafeteria is equal to the number of students. The sandwiches are placed in a **stack**. At each step:

* If the student at the front of the queue **prefers** the sandwich on the top of the stack, they will **take it** and leave the queue.
* Otherwise, they will **leave it** and go to the queue's end.

This continues until none of the queue students want to take the top sandwich and are thus unable to eat.

You are given two integer arrays students and sandwiches where sandwiches[i] is the type of the ith sandwich in the stack (i = 0 is the top of the stack) and students[j] is the preference of the jth student in the initial queue (j = 0 is the front of the queue). Return *the number of students that are unable to eat.*

**Example 1:**

Input: students = [1,1,0,0], sandwiches = [0,1,0,1]

Output: 0

Explanation:

- Front student leaves the top sandwich and returns to the end of the line making students = [1,0,0,1].

- Front student leaves the top sandwich and returns to the end of the line making students = [0,0,1,1].

- Front student takes the top sandwich and leaves the line making students = [0,1,1] and sandwiches = [1,0,1].

- Front student leaves the top sandwich and returns to the end of the line making students = [1,1,0].

- Front student takes the top sandwich and leaves the line making students = [1,0] and sandwiches = [0,1].

- Front student leaves the top sandwich and returns to the end of the line making students = [0,1].

- Front student takes the top sandwich and leaves the line making students = [1] and sandwiches = [1].

- Front student takes the top sandwich and leaves the line making students = [] and sandwiches = [].

Hence all students are able to eat.

**Example 2:**

Input: students = [1,1,1,0,0,1], sandwiches = [1,0,0,0,1,1]

Output: 3

**Solution. :-**

* Initialize a queue and enqueue all the students' preferences.
* Iterate through the sandwiches in the stack.
* If the student at the front of the queue prefers the current sandwich, dequeue the student and move to the next sandwich.
* If the student at the front of the queue does not prefer the current sandwich, leave the student at the front of the queue and enqueue them at the end of the queue.
* Keep track of the number of students unable to eat (those left in the queue after all the sandwiches are processed).
* Return the count of students unable to eat.

**from collections import deque**

**def countStudents(students, sandwiches):**

**queue = deque(students)**

**unableToEat = 0**

**for sandwich in sandwiches:**

**if queue[0] == sandwich:**

**queue.popleft()**

**unableToEat = 0**

**else:**

**queue.append(queue.popleft())**

**unableToEat += 1**

**if unableToEat == len(queue):**

**break**

**return len(queue)**

**students = [1, 1, 0, 0]**

**sandwiches = [0, 1, 0, 1]**

**unableToEat = countStudents(students, sandwiches)**

**print(unableToEat)**

💡 **Question 4** You have a RecentCounter class which counts the number of recent requests within a certain time frame.

Implement the RecentCounter class:

* RecentCounter() Initializes the counter with zero recent requests.
* int ping(int t) Adds a new request at time t, where t represents some time in milliseconds, and returns the number of requests that has happened in the past 3000 milliseconds (including the new request). Specifically, return the number of requests that have happened in the inclusive range [t - 3000, t].

It is **guaranteed** that every call to ping uses a strictly larger value of t than the previous call.

**Example 1:**

Input

["RecentCounter", "ping", "ping", "ping", "ping"]

[[], [1], [100], [3001], [3002]]

Output

[null, 1, 2, 3, 3]

Explanation

RecentCounter recentCounter = new RecentCounter();

recentCounter.ping(1); // requests = [1], range is [-2999,1], return 1

recentCounter.ping(100); // requests = [1,100], range is [-2900,100], return 2

recentCounter.ping(3001); // requests = [1,100,3001], range is [1,3001], return 3

recentCounter.ping(3002); // requests = [1,100,3001,3002], range is [2,3002], return 3

**Solution. :-**

* Initialize a queue to store the timestamps of recent requests.
* Implement the constructor RecentCounter() to initialize the queue.
* Implement the ping(int t) method to add a new request at time t and return the number of requests within the past 3000 milliseconds.
  + Enqueue the current request's timestamp t into the queue.
  + Remove any outdated requests from the front of the queue. Iterate through the queue from the front and remove any timestamps that are less than t - 3000.
  + Return the size of the queue, which represents the number of recent requests within the past 3000 milliseconds.

**from collections import deque**

**class RecentCounter:**

**def \_\_init\_\_(self):**

**self.requests = deque()**

**def ping(self, t: int) -> int:**

**self.requests.append(t)**

**while self.requests[0] < t - 3000:**

**self.requests.popleft()**

**return len(self.requests)**

**counter = RecentCounter()**

**result = [None, counter.ping(1), counter.ping(100), counter.ping(3001), counter.ping(3002)]**

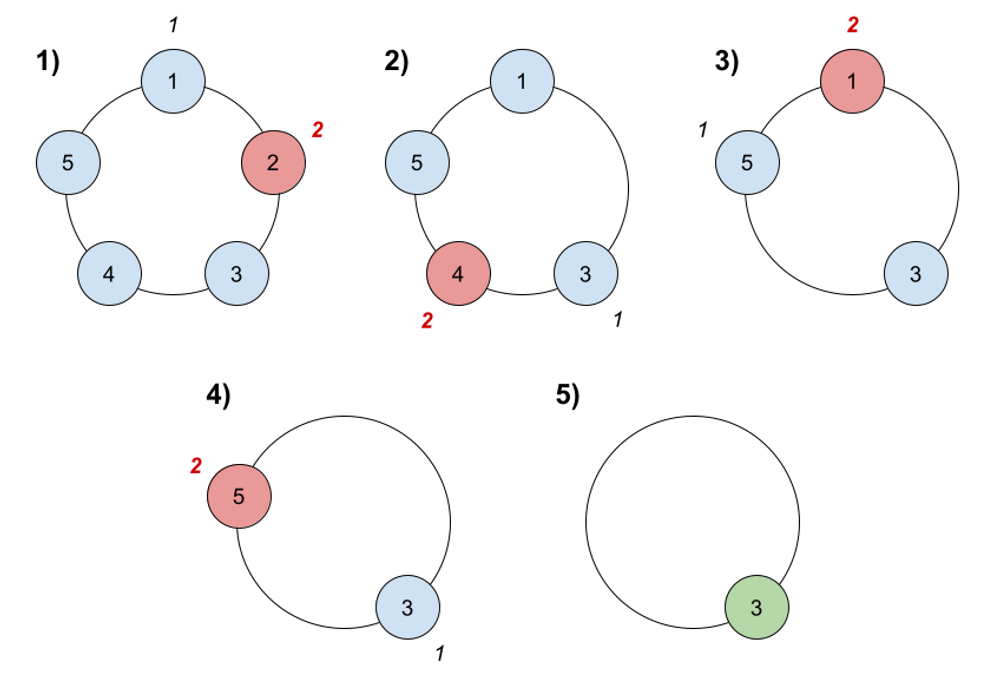
**print(result)**

💡 **Question 5** There are n friends that are playing a game. The friends are sitting in a circle and are numbered from 1 to n in **clockwise order**. More formally, moving clockwise from the ith friend brings you to the (i+1)th friend for 1 <= i < n, and moving clockwise from the nth friend brings you to the 1st friend.

The rules of the game are as follows:

1. **Start** at the 1st friend.
2. Count the next k friends in the clockwise direction **including** the friend you started at. The counting wraps around the circle and may count some friends more than once.
3. The last friend you counted leaves the circle and loses the game.
4. If there is still more than one friend in the circle, go back to step 2 **starting** from the friend **immediately clockwise** of the friend who just lost and repeat.
5. Else, the last friend in the circle wins the game.

Given the number of friends, n, and an integer k, return *the winner of the game*.



**Example 1:**

Input: n = 5, k = 2

Output: 3

Explanation: Here are the steps of the game:

1) Start at friend 1.

2) Count 2 friends clockwise, which are friends 1 and 2.

3) Friend 2 leaves the circle. Next start is friend 3.

4) Count 2 friends clockwise, which are friends 3 and 4.

5) Friend 4 leaves the circle. Next start is friend 5.

6) Count 2 friends clockwise, which are friends 5 and 1.

7) Friend 1 leaves the circle. Next start is friend 3.

8) Count 2 friends clockwise, which are friends 3 and 5.

9) Friend 5 leaves the circle. Only friend 3 is left, so they are the winner.

**Example 2:**

Input: n = 6, k = 5

Output: 1

Explanation: The friends leave in this order: 5, 4, 6, 2, 3. The winner is friend 1.

**Solution. :-**

* Create a deque to represent the circle of friends and initialize it with numbers from 1 to n.
* Start the game by setting the current position to 0 (representing the first friend).
* While the number of remaining friends is greater than 1, repeat steps 4 to 6.
* Increment the current position by k - 1 to count the next k friends.
* Remove the friend at the current position from the deque and update the remaining friends.
* If the current position reaches or exceeds the number of remaining friends, adjust it to wrap around the circle.
* Return the last remaining friend as the winner of the game.

**from collections import deque**

**def findWinner(n: int, k: int) -> int:**

**friends = deque(range(1, n + 1))**

**current = 0**

**while len(friends) > 1:**

**current = (current + k - 1) % len(friends)**

**friends.remove(friends[current])**

**current %= len(friends)**

**return friends[0]**

**n = 5**

**k = 2**

**winner = findWinner(n, k)**

**print(winner)**

💡 **Question 6** You are given an integer array deck. There is a deck of cards where every card has a unique integer. The integer on the ith card is deck[i].

You can order the deck in any order you want. Initially, all the cards start face down (unrevealed) in one deck.

You will do the following steps repeatedly until all cards are revealed:

1. Take the top card of the deck, reveal it, and take it out of the deck.
2. If there are still cards in the deck then put the next top card of the deck at the bottom of the deck.
3. If there are still unrevealed cards, go back to step 1. Otherwise, stop.

Return *an ordering of the deck that would reveal the cards in increasing order*.

**Note** that the first entry in the answer is considered to be the top of the deck.

**Example 1:**

Input: deck = [17,13,11,2,3,5,7]

Output: [2,13,3,11,5,17,7]

Explanation:

We get the deck in the order [17,13,11,2,3,5,7] (this order does not matter), and reorder it.

After reordering, the deck starts as [2,13,3,11,5,17,7], where 2 is the top of the deck.

We reveal 2, and move 13 to the bottom. The deck is now [3,11,5,17,7,13].

We reveal 3, and move 11 to the bottom. The deck is now [5,17,7,13,11].

We reveal 5, and move 17 to the bottom. The deck is now [7,13,11,17].

We reveal 7, and move 13 to the bottom. The deck is now [11,17,13].

We reveal 11, and move 17 to the bottom. The deck is now [13,17].

We reveal 13, and move 17 to the bottom. The deck is now [17].

We reveal 17.

Since all the cards revealed are in increasing order, the answer is correct.

**Example 2:**

Input: deck = [1,1000]

Output: [1,1000]

**Solution. :-**

* Sort the deck array in ascending order.
* Create an empty result array to store the ordered deck.
* Create a queue and initialize it with the indices from 0 to n-1, where n is the size of the deck.
* While the queue is not empty, repeat steps 5 and 6.
* Dequeue an index from the front of the queue and add the corresponding card from the sorted deck to the result array.
* If there are still indices in the queue, dequeue an index from the front of the queue and enqueue it to the back of the queue.
* Return the result array as the ordered deck.

**from collections import deque**

**def deckRevealedIncreasing(deck):**

**n = len(deck)**

**deck.sort() # Sort the deck in ascending order**

**result = [0] \* n**

**queue = deque(range(n)) # Initialize the queue with indices**

**for card in deck:**

**result[queue.popleft()] = card # Reveal the card and store it in the result array**

**if queue: # If there are remaining indices, move the next index to the back of the queue**

**queue.append(queue.popleft())**

**return result**

**deck = [17, 13, 11, 2, 3, 5, 7]**

**ordered\_deck = deckRevealedIncreasing(deck)**

**print(ordered\_deck)**

💡 **Question 7** Design a queue that supports push and pop operations in the front, middle, and back.

Implement the FrontMiddleBack class:

* FrontMiddleBack() Initializes the queue.
* void pushFront(int val) Adds val to the **front** of the queue.
* void pushMiddle(int val) Adds val to the **middle** of the queue.
* void pushBack(int val) Adds val to the **back** of the queue.
* int popFront() Removes the **front** element of the queue and returns it. If the queue is empty, return 1.
* int popMiddle() Removes the **middle** element of the queue and returns it. If the queue is empty, return 1.
* int popBack() Removes the **back** element of the queue and returns it. If the queue is empty, return 1.

**Notice** that when there are **two** middle position choices, the operation is performed on the **frontmost** middle position choice. For example:

* Pushing 6 into the middle of [1, 2, 3, 4, 5] results in [1, 2, 6, 3, 4, 5].
* Popping the middle from [1, 2, 3, 4, 5, 6] returns 3 and results in [1, 2, 4, 5, 6].

**Example 1:**

Input:

["FrontMiddleBackQueue", "pushFront", "pushBack", "pushMiddle", "pushMiddle", "popFront", "popMiddle", "popMiddle", "popBack", "popFront"]

[[], [1], [2], [3], [4], [], [], [], [], []]

Output:

[null, null, null, null, null, 1, 3, 4, 2, -1]

Explanation:

FrontMiddleBackQueue q = new FrontMiddleBackQueue();

q.pushFront(1); // [1]

q.pushBack(2); // [1,2]

q.pushMiddle(3); // [1,3, 2]

q.pushMiddle(4); // [1,4, 3, 2]

q.popFront(); // return 1 -> [4, 3, 2]

q.popMiddle(); // return 3 -> [4, 2]

q.popMiddle(); // return 4 -> [2]

q.popBack(); // return 2 -> []

q.popFront(); // return -1 -> [] (The queue is empty)

**Solution. :-**

* Define a class called Node to represent a node in the doubly linked list. Each node will have a value and references to its previous and next nodes.
* Define the FrontMiddleBack class.
* Initialize the FrontMiddleBack class with an empty doubly linked list.
* Implement the pushFront method:
  + Create a new node with the given value.
  + Set the next node of the new node as the current front node of the doubly linked list.
  + If the front node exists, set its previous node reference to the new node.
  + Set the new node as the front node of the doubly linked list.
* Implement the pushMiddle method:
  + Find the middle node of the doubly linked list.
  + If the middle node exists, insert the new node after the middle node.
  + If the middle node does not exist (even-sized list), insert the new node before the front node.
  + Update the references of the neighboring nodes accordingly.
* Implement the pushBack method:
  + Create a new node with the given value.
  + If the doubly linked list is empty, set the new node as both the front and back node.
  + If the doubly linked list is not empty, set the next node of the current back node as the new node.
  + Set the current back node as the previous node of the new node.
  + Set the new node as the back node of the doubly linked list.
* Implement the popFront method:
  + If the doubly linked list is empty, return 1.
  + Retrieve the value of the front node.
  + Set the next node of the front node as the new front node.
  + If the new front node exists, set its previous node reference to None.
  + Return the value of the front node.
* Implement the popMiddle method:
  + If the doubly linked list is empty, return 1.
  + Find the middle node of the doubly linked list.
  + Retrieve the value of the middle node.
  + Update the neighboring nodes to bypass the middle node.
  + Return the value of the middle node.
* Implement the popBack method:
  + If the doubly linked list is empty, return 1.
  + Retrieve the value of the back node.
  + Set the previous node of the back node as the new back node.
  + If the new back node exists, set its next node reference to None.
  + Return the value of the back node.

**class Node:**

**def \_\_init\_\_(self, val):**

**self.val = val**

**self.prev = None**

**self.next = None**

**class FrontMiddleBackQueue:**

**def \_\_init\_\_(self):**

**self.front = None**

**self.back = None**

**def pushFront(self, val):**

**new\_node = Node(val)**

**if not self.front:**

**self.front = self.back = new\_node**

**else:**

**new\_node.next = self.front**

**self.front.prev = new\_node**

**self.front = new\_node**

**def pushMiddle(self, val):**

**new\_node = Node(val)**

**if not self.front:**

**self.front = self.back = new\_node**

**elif not self.front.next:**

**new\_node.next = self.front**

**self.front.prev = new\_node**

**self.front = new\_node**

**else:**

**slow = self.front**

**fast = self.front.next**

**while fast and fast.next:**

**slow = slow.next**

**fast = fast.next.next**

**new\_node.prev = slow**

**new\_node.next = slow.next**

**slow.next = new\_node**

**if new\_node.next:**

**new\_node.next.prev = new\_node**

**def pushBack(self, val):**

**new\_node = Node(val)**

**if not self.front:**

**self.front = self.back = new\_node**

**else:**

**new\_node.prev = self.back**

**self.back.next = new\_node**

**self.back = new\_node**

**def popFront(self):**

**if not self.front:**

**return 1**

**val = self.front.val**

**if self.front == self.back:**

**self.front = self.back = None**

**else:**

**self.front = self.front.next**

**self.front.prev = None**

**return val**

**def popMiddle(self):**

**if not self.front:**

**return 1**

**if not self.front.next:**

**val = self.front.val**

**self.front = self.back = None**

**return val**

**slow = self.front**

**fast = self.front.next**

**while fast and fast.next:**

**slow = slow.next**

**fast = fast.next.next**

**val = slow.val**

**if slow.prev:**

**slow.prev.next = slow.next**

**if slow.next:**

**slow.next.prev = slow.prev**

**return val**

**def popBack(self):**

**if not self.back:**

**return 1**

**val = self.back.val**

**if self.front == self.back:**

**self.front = self.back = None**

**else:**

**self.back = self.back.prev**

**self.back.next = None**

**return val**

**queue = FrontMiddleBackQueue()**

**queue.pushFront(1)**

**queue.pushBack(2)**

**queue.pushMiddle(3)**

**queue.pushMiddle(4)**

**print(queue.popFront())**

**print(queue.popMiddle())**

**print(queue.popMiddle())**

**print(queue.popBack())**

**print(queue.popFront())**

💡 **Question 8** For a stream of integers, implement a data structure that checks if the last k integers parsed in the stream are **equal** to value.

Implement the **DataStream** class:

* DataStream(int value, int k) Initializes the object with an empty integer stream and the two integers value and k.
* boolean consec(int num) Adds num to the stream of integers. Returns true if the last k integers are equal to value, and false otherwise. If there are less than k integers, the condition does not hold true, so returns false.

**Example 1:**

Input

["DataStream", "consec", "consec", "consec", "consec"]

[[4, 3], [4], [4], [4], [3]]

Output

[null, false, false, true, false]

Explanation

DataStream dataStream = new DataStream(4, 3); //value = 4, k = 3

dataStream.consec(4); // Only 1 integer is parsed, so returns False.

dataStream.consec(4); // Only 2 integers are parsed.

// Since 2 is less than k, returns False.

dataStream.consec(4); // The 3 integers parsed are all equal to value, so returns True.

dataStream.consec(3); // The last k integers parsed in the stream are [4,4,3].

// Since 3 is not equal to value, it returns False.

**Solution. :-**

* Define the DataStream class.
* Initialize the DataStream class with an empty queue, the given value, and the value of k.
* Implement the consec method:
  + Add the new integer num to the queue.
  + If the size of the queue is greater than k, remove the oldest integer from the queue.
  + If the removed integer was equal to the given value, decrement the count of the given value.
  + If num is equal to the given value, increment the count of the given value.
  + If the size of the queue is equal to k and the count of the given value is equal to k, return True. Otherwise, return False.

**from collections import deque**

**class DataStream:**

**def \_\_init\_\_(self, value, k):**

**self.queue = deque()**

**self.value = value**

**self.k = k**

**self.count = 0**

**def consec(self, num):**

**self.queue.append(num)**

**if len(self.queue) > self.k:**

**oldest\_num = self.queue.popleft()**

**if oldest\_num == self.value:**

**self.count -= 1**

**if num == self.value:**

**self.count += 1**

**if len(self.queue) == self.k and self.count == self.k:**

**return True**

**return False**

**stream = DataStream(4, 3)**

**print(stream.consec(4))**

**print(stream.consec(4))**

**print(stream.consec(4))**

**print(stream.consec(3))**