**Assignment Questions 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

**CODE:**

**def** firstUniqChar(s):

charCount **=** {}

*# Count the frequency of each character in the string*

**for** char **in** s:

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

*# Find the first character with frequency 1*

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

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

**return** i

*# If no non-repeating character found, return -1*

**return** **-**1

**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.

**CODE:**

**def** kadane(nums):

maxSum **=** float('-inf')

currSum **=** 0

**for** num **in** nums:

currSum **=** max(currSum **+** num, num)

maxSum **=** max(maxSum, currSum)

**return** maxSum

**def** maxSubarraySumCircular(nums):

n **=** len(nums)

*# Case 1: The maximum sum subarray is not circular*

maxSum1 **=** kadane(nums)

*# Case 2: The maximum sum subarray is circular*

*# Calculate the total sum of the array*

totalSum **=** sum(nums)

*# Invert the sign of each element and find the minimum sum subarray*

invertedNums **=** [**-**num **for** num **in** nums]

maxSum2 **=** totalSum **+** kadane(invertedNums)

*# If all elements in the array are negative, return the maximum sum from Case 1*

**if** maxSum2 **==** 0:

**return** maxSum1

*# Return the maximum of the two cases*

**return** max(maxSum1, maxSum2)

**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

**CODE:**

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

count **=** 0 *# Counter for students unable to eat*

n **=** len(students)

i **=** 0 *# Pointer for the front student*

**while** i **<** n:

**if** students[i] **==** sandwiches[0]: *# Front student prefers the top sandwich*

sandwiches**.**pop(0) *# Remove the top sandwich*

i **+=** 1 *# Move to the next student*

count **=** 0 *# Reset the count since a student was able to eat*

**else**:

count **+=** 1 *# Increment the count for students unable to eat*

i **+=** 1 *# Move to the next student*

**if** count **==** n: *# If all students have been checked without any able to eat*

**break**

**return** n **-** count *# Return the number of students able to eat*

**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

**CODE:**

**from** collections **import** deque

**class** RecentCounter:

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

self**.**requests **=** deque()

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

*# Remove requests that are outside the time frame*

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

self**.**requests**.**popleft()

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

**return** len(self**.**requests)

**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:**

!<https://assets.leetcode.com/uploads/2021/03/25/ic234-q2-ex11.png>

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.

**CODE:**

**from** collections **import** deque

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

friends **=** deque(range(1, n **+** 1))*# Create a deque of friends from 1 to n*

start **=** 0 *# Start index for counting*

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

count **=** (start **+** k **-** 1) **%** len(friends) *# Calculate the index of the friend to be counted out*

friends**.**rotate(**-**count) *# Rotate the deque to bring the friend to be counted out to the front*

friends**.**popleft() *# Remove the counted friend from the deque*

start **=** count **%** len(friends) *# Update the start index for the next count*

**return** friends[0] *# Return the 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]

**CODE:**

**from** collections **import** deque

**def** revealCardsInIncreasingOrder(deck):

*# Sort the deck in ascending order*

sorted\_deck **=** sorted(deck)

*# Initialize an empty queue*

queue **=** deque()

*# Iterate through the sorted deck*

**for** card **in** reversed(sorted\_deck):

*# If the queue is not empty, dequeue a card and enqueue it at the bottom*

**if** queue:

queue**.**appendleft(queue**.**pop())

*# Enqueue the current card*

queue**.**appendleft(card)

*# Convert the queue to a list and return the result*

**return** list(queue)

**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)

**CODE:**

**from** collections **import** deque

**class** FrontMiddleBackQueue:

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

self**.**front **=** deque()

self**.**back **=** deque()

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

self**.**front**.**appendleft(val)

self**.**\_balance()

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

**if** len(self**.**front) **>** len(self**.**back):

self**.**back**.**appendleft(self**.**front**.**pop())

self**.**front**.**append(val)

self**.**\_balance()

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

self**.**back**.**append(val)

self**.**\_balance()

**def** popFront(self):

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

**return** **-**1

val **=** self**.**front**.**popleft()

self**.**\_balance()

**return** val

**def** popMiddle(self):

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

**return** **-**1

val **=** self**.**front**.**pop()

self**.**\_balance()

**return** val

**def** popBack(self):

**if** **not** self**.**back:

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

**return** **-**1

**return** self**.**front**.**popleft()

**return** self**.**back**.**pop()

**def** \_balance(self):

**if** len(self**.**front) **>** len(self**.**back) **+** 1:

self**.**back**.**appendleft(self**.**front**.**pop())

**elif** len(self**.**front) **<** len(self**.**back):

self**.**front**.**append(self**.**back**.**popleft())

q **=** FrontMiddleBackQueue()

q**.**pushFront(1)

q**.**pushBack(2)

q**.**pushMiddle(3)

q**.**pushMiddle(4)

**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.

**CODE:**

**from** collections **import** deque

**class** DataStream:

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

self**.**value **=** value

self**.**k **=** k

self**.**stream **=** deque()

self**.**count **=** 0

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

**if** len(self**.**stream) **==** self**.**k:

removed\_num **=** self**.**stream**.**popleft()

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

self**.**count **-=** 1

self**.**stream**.**append(num)

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

self**.**count **+=** 1

**return** self**.**count **==** self**.**k