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

Input: s = "leetcode"

Output: 0

Input: s = "loveleetcode"

Output: 2

Input: s = "aabb"

Output: -1

Q. 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.

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

Output: 3

Explanation: Subarray [3] has maximum sum 3.

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

Output: 10

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

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

Output: -2

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

Q. 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.*

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.

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

Output: 3

Q. 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

Q. 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.

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.

Input: deck = [1,1000]

Output: [1,1000]

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].

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)

Q. 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.

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.

Answers: To solve the given questions, let's go through each one step by step:

1. Find the first non-repeating character in a string:

To find the first non-repeating character in a string, we can iterate through the string and keep track of the count of each character using a dictionary or an array. Once we have the count, we can iterate through the string again and return the index of the first character with a count of 1. If there are no non-repeating characters, we return -1.

Here's the Python code to solve this:

```python

def firstUniqChar(s):

char\_count = {}

for char in s:

char\_count[char] = char\_count.get(char, 0) + 1

for i, char in enumerate(s):

if char\_count[char] == 1:

return i

return -1

```

2. Find the maximum possible sum of a non-empty subarray in a circular integer array:

To find the maximum possible sum of a non-empty subarray in a circular integer array, we can use the Kadane's algorithm. The Kadane's algorithm is used to find the maximum subarray sum in a linear array, but in this case, we need to consider the possibility of wrapping around from the end of the array to the beginning.

Here's the Python code to solve this:

```python

def maxSubarraySumCircular(nums):

def maxSubarraySum(nums):

max\_sum = float('-inf')

curr\_sum = 0

for num in nums:

curr\_sum = max(curr\_sum + num, num)

max\_sum = max(max\_sum, curr\_sum)

return max\_sum

total\_sum = sum(nums)

max\_sum = max(maxSubarraySum(nums), total\_sum + maxSubarraySum([-num for num in nums]))

return max\_sum if max\_sum > 0 else max(nums)

```

3. Count the number of students unable to eat sandwiches:

To count the number of students unable to eat sandwiches, we can simulate the process described in the question. We can use a queue to represent the students and a stack to represent the sandwiches. We iterate through the students and check if the sandwich at the top of the stack matches the preference of the student. If it does, we remove the student from the queue and pop the sandwich from the stack. Otherwise, we move the student to the end of the queue. We continue this process until either the queue or the stack becomes empty.

Here's the Python code to solve this:

```python

from collections import deque

def countStudents(students, sandwiches):

students\_queue = deque(students)

sandwiches\_stack = sandwiches[::-1] # Reverse the sandwiches to simulate stack behavior

consecutive\_failures = 0

while students\_queue and sandwiches\_stack and consecutive\_failures < len(students\_queue):

if students\_queue[0] == sandwiches\_stack[-1]:

students\_queue.popleft()

sandwiches\_stack.pop()

consecutive\_failures = 0

else:

students\_queue.append(students\_queue.popleft())

consecutive\_failures += 1

return len(students\_queue)

```

4. Implement the RecentCounter class:

To implement the RecentCounter class, we can use a queue to store the timestamps of the requests. When a new request comes in, we add its timestamp to the queue. Then, we iterate through the queue from the front and remove any timestamps that are outside the range of the last 3000 milliseconds. Finally, we return the size of the queue, which represents the number of requests within the specified time frame.

Here's the Python code to implement the RecentCounter class:

```python

from collections import deque

class RecentCounter:

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

self.requests = deque()

def ping(self, t):

self.requests.append(t)

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

self.requests.popleft()

return len(self.requests)

```

5. Reveal cards in increasing order:

To reveal the cards in increasing order, we can start by sorting the deck in ascending order. Then, we can simulate the process described in the question by using a queue. We iterate through the sorted deck and alternate between adding the card at the front of the queue to the back and taking the front card and adding it to the result list. Finally, we return the result list.

Here's the Python code to solve this:

```python

from collections import deque

def deckRevealedIncreasing(deck):

deck.sort()

n = len(deck)

queue = deque(range(n))

result = [0] \* n

for card in deck:

result[queue.popleft()] = card

if queue:

queue.append(queue.popleft())

return result

```

6. Implement the FrontMiddleBackQueue class:

To implement the FrontMiddleBackQueue class, we can use two deques to represent the front and back halves of the queue. When pushing elements, we add them to the back deque. When popping elements, we follow the specified rules: for popFront, we check if the front deque is empty and return -1 if it is not, otherwise we pop from the front deque; for popMiddle, we check if the size difference between the deques is 0 or 1 and pop from the appropriate deque; for popBack, we check if the back deque is empty and return -1 if it is not, otherwise we pop from the back deque.

Here's the Python code to implement the FrontMiddleBackQueue class:

```python

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)

def pushBack(self, val):

self.back.append(val)

self.balance()

def popFront(self):

if self.front:

return self.front.popleft()

elif self.back:

return self.back.popleft()

else:

return -1

def popMiddle(self):

if len(self.front) == len(self.back):

return self.front.pop()

else:

return self.back.popleft()

def popBack(self):

if self.back:

return self.back.pop()

elif self.front:

return self.front.pop()

else:

return -1

def balance(self):

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

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

elif len(self.back) > len(self.front):

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

```

7. Implement the DataStream class:

To implement the DataStream class, we can use a queue to store the last k integers parsed in the stream. When a new integer is added to the stream, we push it into the queue and check if the queue is full. If it is, we compare the elements in the queue with the given value. If they are all equal, we return True; otherwise, we return False.

Here's the Python code to implement the DataStream class:

```python

from collections import deque

class DataStream:

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

self.stream = deque()

self.value = value

self.k = k

def consec(self, num):

self.stream.append(num)

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

return all(x == self.value for x in self.stream[-self.k:])

else:

return False

```

I hope this helps! Let me know if you have any further questions.