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

Solution :

def firstUniqChar(s):

char\_count = {}

# Count the occurrences of each character

for char in s:

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

# Find the first non-repeating character

for i, char in enumerate(s):

if char\_count[char] == 1:

return i

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

return -1

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.

Solution :

def maxSubarraySumCircular(nums):

# Case 1: Maximum sum subarray within array bounds

def kadane(nums):

curr\_max = max\_sum = nums[0]

for i in range(1, len(nums)):

curr\_max = max(nums[i], curr\_max + nums[i])

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

return max\_sum

# Case 2: Maximum sum subarray wraps around circular array

def reverse\_kadane(nums):

curr\_min = min\_sum = nums[0]

for i in range(1, len(nums)):

curr\_min = min(nums[i], curr\_min + nums[i])

min\_sum = min(min\_sum, curr\_min)

# The maximum sum subarray wraps around circular array

# Subtract the minimum sum subarray from the total sum of all elements

circular\_sum = sum(nums) - min\_sum

return circular\_sum

# Find the maximum sum among the two cases

max\_sum\_within\_bounds = kadane(nums)

max\_sum\_circular = reverse\_kadane(nums)

if max\_sum\_within\_bounds < 0:

return max\_sum\_within\_bounds # All elements are negative, return maximum negative sum

else:

return max(max\_sum\_within\_bounds, max\_sum\_circular)

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.*

Solution :

from collections import deque

def countStudents(students, sandwiches):

count = 0

queue = deque(students)

for sandwich in sandwiches:

if queue[0] == sandwich:

queue.popleft()

count = 0

else:

queue.append(queue.popleft())

count += 1

# If all students have been through the queue without any sandwich match, exit the loop

if count == len(queue):

break

return len(queue)

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.

Solution :

from collections import deque

class RecentCounter:

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

self.requests = deque()

def ping(self, t):

self.requests.append(t)

# Remove requests that are outside the time frame

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

self.requests.popleft()

return len(self.requests)

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*.

Solution :

from collections import deque

def findTheWinner(n, k):

circle = deque(range(1, n + 1))

while len(circle) > 1:

# Move k-1 steps in the clockwise direction

for \_ in range(k - 1):

circle.append(circle.popleft())

# Remove the friend at the k-th position

circle.popleft()

return circle[0]

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.

Solution :

def deckRevealedIncreasing(deck):

deck.sort(reverse=True) # Sort the deck in descending order

result = []

while deck:

if result:

# Move the bottom card to the top

result.insert(0, result.pop())

result.insert(0, deck.pop())

return result

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].

Solution :

from collections import deque

class FrontMiddleBack:

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 self.front:

return self.front.pop()

elif self.back:

return self.back.popleft()

else:

return -1

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())

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.

Solution :

from collections import deque

class DataStream:

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

self.value = value

self.k = k

self.stream = deque()

def consec(self, num):

self.stream.append(num)

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

self.stream.popleft()

return len(self.stream) == self.k and all(x == self.value for x in self.stream)