Question 1

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

Soln:

```
def firstUniqChar(s):
    # Create a dictionary to store character frequencies
    freq = {}

    # Iterate through the string and count character frequencies
    for char in s:
        freq[char] = freq.get(char, 0) + 1

    # Iterate through the string again to find the first non-repeating character
    for i, char in enumerate(s):
        if freq[char] == 1:
            return i

    # If no non-repeating character is 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`.

Soln:

```
def maxSubarraySumCircular(nums):
    # Kadane's algorithm to find the maximum subarray sum without circular nature
    def kadane(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

    # Case 1: Maximum subarray sum lies within the circular array
    max_sum_linear = kadane(nums)

    # Case 2: Maximum subarray sum involves elements from both ends of the circular array
    total_sum = sum(nums)
    nums_inverted = [-num for num in nums]
    min_sum_linear = kadane(nums_inverted)
    max_sum_circular = total_sum - (-min_sum_linear)

    # Return the maximum of the two cases
    return max(max_sum_linear, max_sum_circular)
```

---

**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.*

**Soln:**

```
def countStudents(students, sandwiches):
    n = len(students)
    i = 0  # Index to iterate through the students
    j = 0  # Index to iterate through the sandwiches

    while i < n and j < len(sandwiches):
        if students[i] == sandwiches[j]:
            # Student takes the sandwich, move to the next student and sandwich
            i += 1
            j += 1
        else:
            # Student goes to the end of the queue
            i += 1

    # The remaining students in the queue are unable to eat
    return n - j
```

---

### 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.

**Soln:**

```
class RecentCounter:
    def __init__(self):
        self.requests = []

    def ping(self, t):
        # Add the new request to the list
        self.requests.append(t)

        # Remove requests that are older than t - 3000
        while self.requests[0] < t - 3000:
            self.requests.pop(0)
        # Return the number of requests in the past 3000 milliseconds
        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*.

**Soln:**

```
def findTheWinner(n, k):

    # Create a list to represent the circle of friends

    friends = list(range(1, n+1))



    # Start at the 1st friend

    index = 0



    while len(friends) > 1:

        # Count k friends in the clockwise direction

        index = (index + k - 1) % len(friends)
        # Eliminate the friend at the current index
        friends.pop(index)

    # Return the winner
    return friends[0]
```

---

**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

**Soln:**

```
import collections
import heapq


def deckRevealedIncreasing(deck):
    # Sort the deck in increasing order
    deck.sort()
```

```
# Initialize a queue to keep track of the order of revealed cards
queue = collections.deque()
for i in range(len(deck)):
    queue.append(i)

# Initialize a list to store the ordering of the deck
ordering = [0] * len(deck)

# Reveal the cards in increasing order
for card in deck:
    # Take the top card from the queue
    top_card = queue.popleft()

    # Assign the current card to the top card position in the ordering
    ordering[top_card] = card

    # If there are still cards in the queue, put the next top card at the bottom
    if queue:
        next_top_card = queue.popleft()
        queue.append(next_top_card)
return ordering
```

## 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]`.

**Soln:**

```
class ListNode:

    def __init__(self, val=0):

        self.val = val

        self.prev = None

        self.next = None




class FrontMiddleBack:

    def __init__(self):

        self.head = None
        self.middle = None
        self.size = 0

    def pushFront(self, val):
        new_node = ListNode(val)
        if self.size == 0:
            self.head = new_node
            self.middle = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
            if self.size % 2 == 0:
                self.middle = self.middle.prev
        self.size += 1

    def pushMiddle(self, val):
        new_node = ListNode(val)
        if self.size == 0:
            self.head = new_node
            self.middle = new_node
        elif self.size == 1:
            new_node.next = self.head
            self.head.prev = new_node
        self.head = new_node
        self.middle = self.head
    else:
        middle_prev = self.middle.prev
        new_node.next = self.middle
        new_node.prev = middle_prev
        self.middle.prev = new_node
        if self.size % 2 == 1:
            self.middle = middle_prev.next
        else:
            self.middle = new_node
            middle_prev.next = new_node
    self.size += 1

def pushBack(self, val):
    new_node = ListNode(val)
    if self.size == 0:
        self.head = new_node
        self.middle = new_node
    else:
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node
        new_node.prev = current
        if self.size % 2 == 0:
            self.middle = self.middle.next
    self.size += 1

def popFront(self):
    if self.size == 0:
        return -1
    front_val = self.head.val
    if self.size == 1:
        self.head = None
        self.middle = None
    else:
        self.head = self.head.next
        self.head.prev = None
        if self.size % 2 == 1:
            self.middle = self.middle.next
    self.size -= 1
    return front_val

def popMiddle(self):
    if self.size == 0:
        return -1
    middle_val = self.middle.val
```

```
        if self.size == 1:
            self.head = None
            self.middle = None
        elif self.size == 2:
            self.head.next = None
            self.middle = self.head
        else:
            middle_prev = self.middle.prev
            middle_next = self.middle.next
            middle_prev.next = middle_next
            middle_next.prev = middle_prev
            if self.size % 2 == 0:
                self.middle = middle_next
            else:
                self.middle = middle_prev
        self.size -= 1
        return middle_val

    def popBack(self):
        if self.size == 0:
            return -1
        if self.size == 1:
            back_val = self.head.val
            self.head = None
            self.middle = None
        else:
            current = self.head
            while current.next:
                current = current.next
            back_val = current.val
            current.prev.next = None
            if self.size % 2 == 1:
                self.middle = self.middle.prev
        self.size -= 1
        return back_val
```

---

**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`.

**Soln:**

```
import collections


class DataStream:
```

```
def __init__(self, value, k):
    self.value = value
    self.k = k
    self.stream = collections.deque()
    self.count = 0

def consec(self, num):
    # Add the new number to the stream
    self.stream.append(num)
    if num == self.value:
        self.count += 1

    # If the stream length exceeds k, remove the oldest number
    if len(self.stream) > self.k:
        oldest_num = self.stream.popleft()
        if oldest_num == self.value:
            self.count -= 1

    # Check if the last k integers are equal to value
    return self.count == self.k
```