ASSIGNMENT 17

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

```
from collections import deque

def find_first_non_repeating(s):
    queue = deque()
    char_count = {}

    for index, char in enumerate(s):
        if char not in char_count:
            queue.append((char, index))
            char_count[char] = 1
        else:
            char_count[char] += 1

        while queue and char_count[queue[0][0]] > 1:
            queue.popleft()

    return queue[0][1] if queue else -1
```

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`.

```
from collections import deque

def max_subarray_sum(nums):
    max_sum = nums[0]
    current_sum = nums[0]
    queue = deque([nums[0]])

    for num in nums[1:]:
        current_sum += num

        while current_sum < 0 and queue:
            current_sum -= queue.popleft()

        queue.append(num)

        if current_sum > max_sum:
            max_sum = current_sum
```

```
    current_sum = 0
    queue.clear()

    for num in nums[1:]:
        current_sum += num

        while current_sum < 0 and queue:
            current_sum -= queue.pop()

        queue.appendleft(num)

        if current_sum > max_sum:
            max_sum = current_sum

    return max_sum
```

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.*

```
from collections import deque

def count_students_unable_to_eat(students, sandwiches):
    queue = deque(students)

    for sandwich in sandwiches:
        if not queue:
            return 0
        student = queue.popleft()
        while student != sandwich:
            queue.append(student)
            if not queue:
                return len(students)
            student = queue.popleft()

    return len(queue)
```

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.

```
from collections import deque

class RecentCounter:
   def __init__(self):
      self.queue = deque()

   def ping(self, t: int) -> int:
      self.queue.append(t)
      while self.queue[0] < t - 3000:
         self.queue.popleft()
      return len(self.queue)
```

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*.

```
def find_winner(n, k):
   circle = list(range(1, n + 1))
   current = 0

   while len(circle) > 1:
      current = (current + k - 1) % len(circle)
      circle.pop(current)
      if current == len(circle):
         current = 0

   return circle[0]
```

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.

```
from collections import deque

def deck_revealed_order(deck):
   deck.sort()
   order = deque(range(len(deck)))
   revealed = []

   while order:
      revealed.append(order.popleft())
      if order:
         order.append(order.popleft())

   result = []
   while revealed:
      result.append(deck[revealed.pop()])

   return result
```

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]`.

```
from collections import deque

class FrontMiddleBack:
    def __init__(self):
        self.front = deque()
        self.back = deque()

    def pushFront(self, val: int) -> None:
        self.front.appendleft(val)

    def pushMiddle(self, val: int) -> None:
        if len(self.front) > len(self.back):
            self.back.appendleft(self.front.pop())
        self.front.append(val)

    def pushBack(self, val: int) -> None:
        self.back.append(val)

    def popFront(self) -> int:
        if not self.front and not self.back:
            return 1
        if self.front:
            return self.front.popleft()
        self.front.append(self.back.popleft())
        return self.front.popleft()

    def popMiddle(self) -> int:
        if not self.front and not self.back:
            return 1
        if len(self.front) == len(self.back):
            return self.front.pop()
        return self.back.popleft()

    def popBack(self) -> int:
        if not self.front and not self.back:
            return 1
        if self.back:
            return self.back.pop()
        self.back.appendleft(self.front.pop())
        return self.back.pop()
```

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`.

```
from collections import deque

class DataStream:
    def __init__(self, value: int, k: int):
        self.stream = deque()
        self.value = value
        self.k = k

    def consec(self, num: int) -> bool:
        self.stream.append(num)
        if len(self.stream) < self.k:
            return False
        if len(self.stream) > self.k:
            self.stream.popleft()
        return all(x == self.value for x in self.stream)
```