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:-

```
class Solution {
    public int firstUniqChar(String s) {
        HashMap<Character,Integer> count= new HashMap<Character,Integer>();
        int n=s.length();
         // build hash map : character and how often it appears
        for(int i=0;i<n;i++){
            char c=s.charAt(i);
           count.put(c, count.getOrDefault(c, 0) + 1);
        }
        // find the index
        for(int i=0;i<n;i++){
           if (count.get(s.charAt(i)) == 1)
                return i;
        }
        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:-

```
class Solution {
    public int maxSubarraySumCircular(int[] array) {
        // variable to keep track of the total sum of the array
        int acc = 0;
        // variable to keep track of the maximum sum subarray using kadane's
algorithm
        int max1 = kadane(array);
        // iterate through the array and negate each element
        for(int i = 0; i < array.length; i++) {
            acc += array[i];
            array[i] = -array[i];
        }
        // variable to keep track of the minimum sum subarray using kadane's
algorithm on the negated array
        int min = kadane(array);
        // variable to keep track of the maximum sum subarray that can be formed by
wrapping around the array
        int max2 = acc + min;
        // if the maximum sum subarray that can be formed by wrapping around the
array is zero, return the maximum sum subarray using kadane's algorithm
        if(max2 == 0)  return max1;
        // return the maximum of the two maximum sum subarrays
        return Math.max(max1, max2);
    }
    // function to calculate the maximum sum subarray using kadane's algorithm
      public int kadane(int[] array) {
        // variable to keep track of the maximum sum subarray ending at current
index
        int maxSum = array[0];
        // variable to keep track of the overall maximum sum subarray
        int max = array[0];
        // iterate through the array starting from the second element
        for(int i = 1; i < array.length; i++) {
            // update the maximum sum subarray ending at current index
            // by taking the maximum between the current element and the sum of the
current element and the maximum sum subarray ending at the previous index
            maxSum = Math.max(maxSum+array[i], array[i]);
            // update the overall maximum sum subarray by taking the maximum
between the current maximum sum subarray ending at current index and the overall
maximum sum subarray
            max = Math.max(max, maxSum);
        }
        return max;
    }
}
```

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:-

```
class Solution {
    public int countStudents(int[] students, int[] sandwiches) {
        int ones = 0; //count of students who prefer type1
        int zeros = 0; //count of students who prefer type0

        for(int stud : students){
            if(stud == 0) zeros++;
            else ones++;
        }

        // for each sandwich in sandwiches
        for(int sandwich : sandwiches){
            if(sandwich == 0){  // if sandwich is of type0
                if(zeros == 0){ // if no student want a type0 sandwich
                    return ones;
                }
                zeros--;
            }
            else{  // if sandwich is of type1
                if(ones == 0){  // if no student want a type1 sandwich
                    return zeros;
                }
                ones--;
            }
        }
        return 0;
    }
```

```
}
```

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:-

```
class RecentCounter {
    ArrayList<Integer> calls ;
    public RecentCounter() {
        calls = new ArrayList<Integer>();
        }

    public int ping(int t) {
        calls.add(t);
        int count = 0;
        for(Integer call:calls){
            if( t-call<=3000)
            count++;
        }
        return count;

    }
}
```

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:
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:-

```
class Solution {
public:
    int findTheWinner(int n, int k) {
        int ans = 0;
        for(int i = 1; i <= n; i++)
        {
            ans = (ans + k) % i;
        }
        return ans + 1;
    }
};
```

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:-

```
      class Solution {
    public int[] deckRevealedIncreasing(int[] deck) {
        Arrays.sort(deck);
        Queue<Integer> queue = new LinkedList<>();
        int n = deck.length;
        for(int i = 0;i<n;i++){
            queue.add(i);
        }
        int[]res = new int[n];
        for(int i = 0;i<n;i++){
            res[queue.poll()] = deck[i];
            queue.add(queue.poll());
        }
        return res;
    }
}
```

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:-
class FrontMiddleBackQueue {
    private int size;
    private Node head, tail, mid;

    public FrontMiddleBackQueue() {
        size = 0;
        head = tail = new Node(-1);
        head.next = tail;
        tail.prev = head;
        mid = head;     // default mid position
    }

    public void pushFront(int val) {
        Node newNode = new Node(val);
        add(head, newNode); // new node is added right to head
        size++;

        if (size == 1) {    // if its the first ever node
            mid = head.next;// make it the mid
        }
        if (size % 2 == 0) {    // mid is the former of the two mids
            mid = mid.prev;
        }
    }

    public void pushMiddle(int val) {
        Node newNode = new Node(val);
        // if size is even, new mid will be at perfect centre
        // that is, right next to mid
        if (size % 2 == 0) {
            add(mid, newNode);  // goes right next to mid
            mid = mid.next;
        }
        // else new mid will go to the left of current mid
        // i.e. the right next position of mid's current prev
        else {
            add(mid.prev, newNode); // go next to mid's prev
            mid = mid.prev;
        }
        size++;
    }
```

```
public void pushBack(int val) {
    Node newNode = new Node(val);
    // gets added right before tail
    // i.e. right next to tail's current prev
    add(tail.prev, newNode);
    size++;
    // if size becomes odd, mid will move to perfect centre
    if (size % 2 == 1) {
        mid = mid.next;
    }
}

public int popFront() {
    if (size == 0) {
        return -1;
    }
    if (size == 1) {  // if list wil become empty
        mid = head;   // default mid is head
    } else if (size % 2 == 0) {
        // list will become odd, mid will move to next node
        mid = mid.next;
    }

    int val = head.next.value;
    delete(head.next);  // delete the node next to head
    size--;
    return val;
}

public int popMiddle() {
    if (size == 0) {
        return -1;
    }

    Node node = mid;
    if (size == 1) {  // list wil become empty
        mid = head;   // default mid is head
    } else {
        // if list will become odd, mid will go to next node
        // if list will become even, mid will go to prev node
        mid = (size % 2 == 0)? mid.next : mid.prev;
    }

    delete(node);   // delete the node (old mid)
    size--;
    return node.value;
}

public int popBack() {
    if (size == 0) {
        return -1;
    }
    if (size == 1) {  // list wil become empty
        mid = head;   // default mid is head
    } else if (size % 2 == 1) { // if list will become even
        mid = mid.prev;         // mid will move to prev node
    }
```

```
        int val = tail.prev.value;
        delete(tail.prev);  // delete the preceding node of tail
        size--;
        return val;
    }

    // Adds a node `newNode` right next to a reference node `ref`
    private void add(Node ref, Node newNode) {
        newNode.next = ref.next;
        ref.next = newNode;
        newNode.next.prev = newNode;
        newNode.prev = ref;
    }

    // Deletes the passed in node 'node'
    private void delete(Node node) {
        node.next.prev = node.prev;
        node.prev.next = node.next;
    }

    // Node class
    private class Node {
        int value;
        Node next, prev;
        private Node(int value) {
            this.value = value;
        }
    }
}
```

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`.

Example 1:

```
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.


code:-
class DataStream {
```

```
    private int count;
    private int value;
    private int k;

    public DataStream(int value, int k) {
        this.value = value;
        this.k = k;
        count = 0;
    }

    public boolean consec(int num) {
        if(num != value){
            count = 0;
            return false;
        }
        return ++count >= k;
    }
}
```