3.4.

Requirement 1: Explain how wraparound works.

This is the wraparound that exists in the QueueApp class:

```
        public void insert(long j)   // put item at rear of queue
        {
            if(rear == maxSize - 1)         // deal with wraparound
                rear = -1;
            queArray[++rear] = j;         // increment rear and insert
            nItems++;                     // one more item
        }
        //--------------------------------------------------------------
        public long remove()         // take item from front of queue
        {
            long temp = queArray[front++]; // get value and incr front
            if(front == maxSize)           // deal with wraparound
                front = 0;
            nItems--;                      // one less item
            return temp;
        }
```

How this code fragment works:

- Insertion (Enqueue): If the rear index reaches maxSize - 1 and there is space at the front of the array (because some numbers have been removed), the rear index is set to -1. The next element is then inserted at rear + 1, which is index 0, effectively wrapping around to the start of the array.
- Removal (Dequeue): If the front index reaches maxSize after an element is removed, it means we've reached the end of the array. The front index is then set to 0, allowing the queue to continue removing elements from the start of the array.

Requirement 4: Insert fewer items or remove fewer items and investigate what happens when the queue is empty or full.

I have modified my class to insert fewer items, remove fewer items, as follow:

```
    public static void main(String[] args) {
        Queue theQueue = new Queue(5);  // queue holds 5 items

        theQueue.insert(10);            // insert 3 items
        theQueue.insert(20);
        theQueue.insert(30);
                                          // remove the insertion of 40

        theQueue.displayAll();

        theQueue.remove();              // remove 2 items
        theQueue.remove();              //    (10, 20)

        theQueue.displayAll();

        theQueue.insert(40);            // insert 3 more items
        theQueue.insert(50);            //    (wraps around)
```

```
        theQueue.insert(60);
                                            // remove the insertion of 80

        theQueue.displayAll();

        System.out.print("Removed numbers from the queue: ");
        for (int i = 0; i < 4; i++) {
            long n = theQueue.remove();
            // check if the removal is successful
            if ( n != -1) {
                System.out.print(n + " ");
            }
        }

        System.out.println();
        theQueue.displayAll();
    }  // end main()
```

Output of the code:

```
Array: 10 20 30 0 0
Queue: 10 20 30
[Array] Front indice: 0; Rear indice: 2

Array: 10 20 30 0 0
Queue: 30
[Array] Front indice: 2; Rear indice: 2

Array: 60 20 30 40 50
Queue: 30 40 50 60
[Array] Front indice: 2; Rear indice: 0

Removed numbers from the queue: 30 40 50 60
Array: 60 20 30 40 50
Queue: empty
[Array] Front indice: 1; Rear indice: 0
```

Explanation:

```
Array: 10 20 30 0 0
Queue: 10 20 30
[Array] Front indice: 0; Rear indice: 2
```

This is the case when the queue is empty. Can see that initially, the queue is empty. I insert 3 numbers: 10, 20, 30 into the queue and it becomes [10, 20, 30]. The front indice corresponds to “10” in the array, which is 0; and the the rear indice corresponds to “30”, which is 2. This front indice and the rear indice illustrate the order of the inserted numbers.

```
Array: 10 20 30 0 0
Queue: 30
[Array] Front indice: 2; Rear indice: 2
```

In this step, I removed 10 and 20 from the queue. The front indice is updated to 2 since now there is just a number “30” in the queue, and the rear indice remains 2.

```
Array: 60 20 30 40 50
Queue: 30 40 50 60
[Array] Front indice: 2; Rear indice: 0
```

In this step, I inserted 3 more numbers: 40, 50, 60. The queue is now wrapped around. The front indice is 2 (“30”) and the rear indice becomes 0 (“60).

```
Removed numbers from the queue: 30 40 50 60
Array: 60 20 30 40 50
Queue: empty
[Array] Front indice: 1; Rear indice: 0
```

The display function prints the array [60, 20, 30, 40, 50], and since the queue is empty, it prints "Queue: empty". The front index is updated to 1 (“20), and the rear index remains at 0 (“60).

In my Problem_3_4thReq_full.java, I have modified my original class to handle the case that the queue is full, and the result is as follow:

```
Array: 10 20 30 0 0
Queue: 10 20 30
[Array] Front indice: 0; Rear indice: 2

Array: 10 20 30 0 0
Queue: 30
[Array] Front indice: 2; Rear indice: 2

Queue is full. Cannot insert 80
Array: 60 70 30 40 50
Queue: 30 40 50 60 70
[Array] Front indice: 2; Rear indice: 1

Removed numbers from the queue: 30 40 50 60
Array: 60 70 30 40 50
Queue: 70
[Array] Front indice: 1; Rear indice: 1
```

The first two steps of the output stay the same with the case that the queue is empty. Just the two final steps is different:

```
Queue is full. Cannot insert 80
Array: 60 70 30 40 50
Queue: 30 40 50 60 70
[Array] Front indice: 2; Rear indice: 1
```

I tried to insert 3 more numbers: 40, 50, 60, which exceeds the maximum size of the queue (5). So, the code prints a message "Queue is full. Cannot insert 80".

After that, the display function prints the updated array [60, 70, 30, 40, 50], showing the wraparound of the queue. The queue line displays all elements in the queue, which

are [30, 40, 50, 60, 70]. The front index is 2 (“60”), and the rear index is 1 (“70”), indicating the wraparound.

```
Removed numbers from the queue: 30 40 50 60
Array: 60 70 30 40 50
Queue: 70
[Array] Front indice: 1; Rear indice: 1
```

In this step, I removed 4 numbers from the queue: 30, 40, 50, 60. Then, the display function prints the array [60, 70, 30, 40, 50], and the queue line prints the remaining element (70). The front index is updated to 1 (“70”), and the rear index remains at 1 (“70”).

3.6. Compare PriorityQApp.java with QueueApp.java. Which one is more efficient?

Operation	PriorityQApp.java	QueueApp.java
Insertion	Inserts items at correct position to maintain sorted order, requiring shifting elements in the array. Complexity: O(N)	Inserts items at the rear of the array. Complexity: O(1)
Removal	Removes minimum item from the priority queue. Complexity: O(1)	Removes item from the front of the queue. Complexity: O(1)
IsEmpty	Checks if the priority queue is empty. Complexity: O(1)	Checks if the queue is empty. Complexity: O(1)

Conclusion:

- Insertion: QueueApp is more efficient because it inserts items at the rear of the array with constant complexity O(1), whereas PriorityQApp inserts items at the correct position to maintain sorted order, requiring shifting elements in the array, leading to a complexity of O(N), where N is the number of items in the priority queue.
- Removal: Both PriorityQApp and QueueApp perform removal operations in constant time O(1).
- IsEmpty: Both PriorityQApp and QueueApp check if the queue is empty in constant time O(1).

Which one is more efficient?
In my opinion, QueueApp.java is more efficient for insertion operations compared to PriorityQApp.java.
However, if priority order needs to be maintained, PriorityQApp.java is a better option.