```
private static <T> void partition(Queue<T> q, T partitioner,
	Queue<T> front, Queue<T> back, Comparator<T> order) {
	Queue<T> nullQ = new Queue3<T>();
	while (!q.equals(nullQ)) {
		T next = q.dequeue();
		// if next greater than or equal to partitioner
		if (order.compare(next, partitioner) > 0) {
			back.enqueue(next);
		} else {
			front.enqueue(next);
		}
	}
}
```

```
public static <T> void sort(Queue<T> qthis, Comparator<T> order) {
	if (qthis.length() > 2) {
		Queue<T> temp = new Queue3<T>();
		/*
		* Dequeue the partitioning entry from this
		*/
		while (qthis.length() > temp.length()) {
			temp.enqueue(qthis.dequeue());
		}
		T partitioner = qthis.dequeue();
		temp.enqueue(partitioner);
		while (qthis.length() > 0) {
			temp.enqueue(qthis.dequeue());
		}
		/*
		* Partition this into two queues as discussed above
		* (you will need to declare and initialize two new queues)
		*/
		Queue<T> lower = new Queue3<T>();
		Queue<T> higher = new Queue3<T>();
		// partition queue
		while (temp.length() > 0) {
			T dequeue = temp.dequeue();
			if (order.compare(dequeue, partitioner) > 0) {
				higher.enqueue(dequeue);
```

```
                } else {
                        lower.enqueue(dequeue);
                }
        }
        /*
        * Recursively sort the two queues
        */
        // THIS SHIT DONT WORK!!!!!!!!!// THIS SHIT DONT WORK!!!!!!!!!// THIS
        SHIT DONT WORK!!!!!!!!!
        sort(lower, order);
        sort(higher, order);
        /*
        * Reconstruct this by combining the two sorted queues and the
        * partitioning entry in the proper order
        */
        qthis.append(lower);
        qthis.append(higher);
    }
}
```