1. Table 1 shows the *quick-sort* algorithm.

In parts (a), (b), and (c) of this question, assume that step 1.1 is implemented by a partitioning algorithm that chooses *a*[*left*] as the pivot. The partitioning algorithm leaves the pivot in *a*[*p*]. You may make any reasonable assumption about the order of the values it leaves in *a*[*left*…*p*–1] and in *a*[*p*+1…*right*].

(a) Illustrate the quick-sort algorithm's behaviour as it sorts the following array of country-codes alphabetically. Your illustration should show the contents of the array and the value of *p*, after step 1.1, after step 1.2, and after step 1.3.

0	1	2	3	4	5	6	7	8	9	10	11
NL	BE	LU	DE	FR	IT	UK	DK	IE	ES	PT	GR

(3)

(b) Analyse the quick-sort algorithm's behaviour in the *best case*. Let *comps*(*n*) be the number of comparisons performed by the algorithm as it sorts a (sub)array of length *n*. Write down and explain equations of the form:

$$comps(n) = \ldots \quad \text{if } n \leq 1$$
$$comps(n) = \ldots \quad \text{if } n > 1$$

State (but do not derive) the resulting time complexity. When does the best case arise?

(4)

(c) Similarly analyse the quick-sort algorithm's behaviour in the *worst case*. Write down and explain the corresponding equations. State the resulting time complexity. When does the worst case arise?

The quick-sort algorithm's worst-case behaviour resembles the behaviour of another well-known sorting algorithm. Which one?

(5)

(d) Step 1.1 can be implemented by a partitioning algorithm that avoids this worst-case behaviour. *Outline* such an algorithm.

Hint: Your algorithm should not necessarily choose *a*[*left*] as the pivot. Details of your algorithm are not required, but the idea must be clear.

(8)

To sort $a[left \ldots right]$:

1. If $left < right$:
   1.1. Partition $a[left \ldots right]$ such that $a[left \ldots p-1]$ are all less than or equal to $a[p]$, and $a[p+1 \ldots right]$ are all greater than or equal to $a[p]$.
   1.2. Sort $a[left \ldots p-1]$.
   1.3. Sort $a[p+1 \ldots right]$.
2. Terminate.

Table 1 Quick-sort algorithm (Question 1).

2. Appendix 1 shows a contract for the `Queue` abstract data type, in the form of a Java interface.

(a) Using diagrams, describe an efficient representation of a *bounded* queue using an array.

(6)

(b) Consider a bounded queue whose capacity is 6. Show the queue representation:

(i) after adding the objects *A*, *B*, *C*, *D*, *E* (in that order);

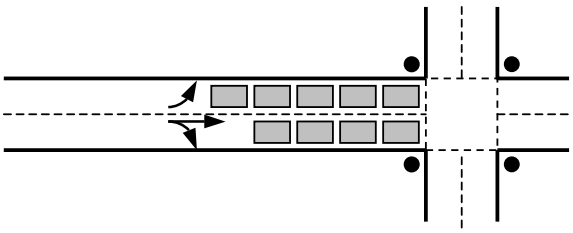
(ii) after removing the front object;

(iii) after adding the object *F*;

(iv) after adding the object *G*.

(4)

(c) Consider vehicles waiting at a traffic-signal on a road with two lanes (see below). The left lane is reserved for vehicles turning left; the right lane is reserved for vehicles going ahead or turning right. The traffic-signal changes through three states: *red* (all vehicles must wait), then *filter* (only left-turning vehicles may proceed), then *green* (all vehicles may proceed). The timing is such that, when the traffic-signal changes to the *filter* state, at most 10 waiting vehicles in the left lane actually proceed; and when the traffic-signal changes to the *green* state, at most 10 waiting vehicles in each lane actually proceed.



The following Java class is designed to simulate the traffic movements:

```
class Approach {

  // An Approach object simulates a traffic-signal on a road
  // with two lanes, and vehicles waiting at the signal.

  private …;

  public static final int     // traffic-signal states
      RED = 0, FILTER = 1, GREEN = 2;

  public static final int     // vehicle directions
      LEFT = 0, AHEAD = 1, RIGHT = 2;

  public Approach ()
  { … }
  // Construct an Approach object with no waiting vehicles
  // and with the traffic-signal at RED.

  public void arrive (Vehicle veh, int dir)
  { … }
  // Simulate the arrival of vehicle veh, intending to proceed
  // in direction dir (LEFT, AHEAD, or RIGHT).

  public void changeSignal ()
  { … }
  // Change the state of the traffic-signal (RED to FILTER,
  // FILTER to GREEN, or GREEN to RED).

}
```

Complete this class. Use the `Queue` interface. You may assume the existence of a class `ArrayQueue` that implements the `Queue` interface.

(10)

3. A *binary relation* is a set of ordered pairs (*x*, *y*). Neither *x* nor *y* is necessarily unique in a relation. The following is an example of a relation:

Languages = { (UK, “English”),
(FR, “French”),
(DE, “German”),
(BE, “French”),
(BE, “Flemish”) }

(a) Assume the following requirements for a `Relation` abstract data type:

1) It must be possible to make the relation empty.

2) It must be possible to add a given pair of objects (x, y) to the relation.

3) It must be possible to remove a given pair (x, y) from the relation.

4) It must be possible to test whether a given pair (x, y) is in the relation.

5) It must be possible, given x, to test whether there is at least one pair (x, y) in the relation.

6) It must be possible to iterate over all pairs in the relation.

7) It must be possible, given x, to iterate over all pairs (x, y) in the relation.

Design a contract that meets these requirements. Your contract must be in the form of a suitably commented Java interface.

Show that the operations in your contract are both sufficient and necessary to meet the above requirements.

You may assume the following class declaration:

```
class Pair {
  public Object x, y;
  …
}
```

(10)

(b) Using your interface, implement the following Java methods:

```
static Set getxs (Relation rel);
// Return the set of all x such that there is at least one pair (x, y)
// in rel.

static Set getys (Relation rel, Object x);
// Return the set of all y such that (x, y) is in rel.
```

(5)

(c) Briefly describe an efficient representation for relations. Illustrate your answer by showing how the above relation *Languages* would be represented.

(5)

4. **(a)** Explain the basic principles of *hash tables*. Distinguish between *closed-bucket hash tables* and *open-bucket hash tables*.

(5)

(b) Consider a map whose keys are course-codes and whose values are course descriptions. Assume that a course-code consists of two letters and three digits: the letters identify the department, the first digit is the level number (1–5), and the remaining digits are a serial number. For example, BI101 might be the course-code for a Biology level-1 course. Assume also that the map contains about 200 courses at any one time.

Suppose that a programmer decides to represent the map by a closed-bucket hash table with 26 buckets, with a hash function that simply uses the course-code's first letter. Explain clearly why this particular representation is unsuitable.

(2)

(c) Design a better hash-table representation for the map.

(3)

(d) Write down an algorithm to find the course description corresponding to a given course-code.

(3)

(e) What are the best-case and worst-case time complexities of your algorithm? Explain your answers.

(3)

(f) The following is a true story. An application program was written to maintain a large set of URLs. The program used Java's `String` class to represent each URL, and the `HashTable` class to represent the set of URLs. (*Note:* `HashTable` was a precursor of `HashMap`.)

At first, searching the set of URLs was found to be unexpectedly slow. Later, when a new version of the Java class library was installed, searching was found to be much faster. Neither the application code nor the size of the hash table had been changed.

What might account for this phenomenon?

(4)

Appendix 1 Contract for a Queue abstract data type

```
public interface Queue {

  // Each Queue object is a queue whose elements are objects.

  /////////////// Accessors ///////////////

  public boolean isEmpty ();
  // Return true if and only if this queue is empty.

  public int size ();
  // Return this queue's length.

  public Object getFirst ();
  // Return the element at the front of this queue.

    /////////////// Transformers ///////////////

  public void clear ();
  // Make this queue empty.

  public void addLast (Object elem);
  // Add elem as the rear element of this queue.

  public Object removeFirst ();
  // Remove and return the front element of this queue.

}
```