1. **(a)** Table 1 shows the array *quick-sort* algorithm. Illustrate its behaviour as it sorts the following array of country-codes:

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

Your illustration must show the contents of the array, and the value of *p*, after step 1.1, after step 1.2, and after step 1.3.

Assume that step 1.1 takes *a*[*left*] as the pivot, and does not reorder the components it puts into either sub-array.

[3]

(b) State the time complexity of the quick-sort algorithm (in terms of the number of comparisons performed). Informally justify your answer.

[6]

(c) Write down the array *merge-sort* algorithm. Assume that an array merging algorithm is already available.

[5]

(d) Illustrate the merge-sort algorithm's behaviour as it sorts the following array of country-codes:

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

Your illustration must show the contents of the array after each step of the algorithm.

[3]

(e) State the time complexity of the merge-sort algorithm (in terms of the number of comparisons performed). Informally justify your answer.

[3]

To sort *a*[*left*…*right*]:

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

Table 1 Array quick-sort algorithm (Question 1).

2. **(a)** Briefly explain what is meant by an abstract data type (ADT). Why are ADTs important?

[2]

(b) Explain what is meant by a *queue*.

Write down a design for a homogeneous queue ADT. Express your design in the form of a Java generic interface, with each operation accompanied by a comment specifying its behaviour. Your ADT must provide appropriate operations to add and remove elements, and to determine the length of the queue.

[5]

(c) Using your queue ADT, complete the following Java method:

```
static void printShortWords
                    (BufferedReader input);
// input contains a sequence of words of varying lengths.
// Read these words from input. Print out all the 1-letter words,
// followed by the 2-letter words, followed by the 3-letter words,
// without changing the order of the words within each group.
// Ignore all words longer than 3 letters.
```

You should use the auxiliary method of Table 2.

[6]

(d) Using diagrams, outline an efficient implementation of your queue ADT.

Explain briefly how each operation would be implemented. What is the time complexity of each operation?

Note: Where a standard algorithm can be used, you need only name the algorithm.

[7]

```
static String readWord (BufferedReader input);
// Read and return the next word from input, skipping any preceding
// spaces or punctuation. Return null if no word remains to be read.
```

Table 2 A Java auxiliary method (Questions 2 and 4).

3. **(a)** Explain the differences between *lists* and *sets*.

[2]

(b) Table 3 outlines a contract for a list ADT, expressed in the form of a Java interface `List<E>`.

Using a diagram, show how a list can be represented by an array.

How would the `addLast`, `add`, `remove`, and `get` operations be implemented? What is each operation's time complexity?

Note: In this part of the question, neglect the possibility that the `addLast` and `add` operations might exceed the array's capacity.

[6]

(c) Now consider the possibility that the `addLast` and `add` operations might exceed the array's capacity. In this situation, you are required to expand the array, so that the application code can continue normally.

Show how this can be done in such a way that each operation's *amortized* time complexity is as good as its best-case time complexity.

What is the `addLast` operation's time complexity: (i) in the best case, (ii) in the worst case, and (iii) in the average case? Informally justify your answer.

[6]

(d) Now consider the `iterator` operation of Table 3. How would the resulting iterator be represented?

[3]

(e) Show how application code could print all the elements of a list `xs` of type `List<T>`.

[3]

```
public interface List <E> {
  // A List<E> object is a homogeneous list whose elements are of
  // type E.

  public void addLast (E x);
  // Add x as the last element of this list.

  public void add (int i, E x);
  // Add x as the element at index i in this list.

  public E remove (int i);
  // Remove and return the element at index i in this list.

  public E get (int i);
  // Return the element at index i in this list.

  …

  public Iterator<E> iterator ();
  // Return an iterator that will visit the elements of this from left to right.

}
```

Table 3 Outline of a `List<E>` interface (Question 3).

4. A *bag* is a collection of elements, in no fixed order, in which each element may occur several times. For example, the bags {apple, banana, apple} and {apple, apple, banana} are equal to each other, but unequal to {apple, banana}.

Note: Bags resemble sets, except that each element of a set occurs just once.

(a) Design an abstract data type (ADT) to meet the following requirements:

- It must be possible to obtain the cardinality of a bag, counting all occurrences of all elements. For example, the cardinality of {apple, apple, banana} is 3.
- It must be possible to determine the number of occurrences of a given element in a bag. For example, the number of apples in {apple, apple, banana} is 2.
- It must be possible to add several occurrences of a given element to a bag. For example, adding 2 apples to {apple, banana} should yield {apple, banana, apple, apple}.
- It must be possible to remove several occurrences of a given element from a bag. For example, removing 3 apples from {apple, banana, apple, apple} should yield {banana}.
- It must be possible to render a bag as a string, in a suitable format.

Express your ADT design in the form of a Java interface with suitable comments.

[5]

(b) Outline an efficient representation of bags using binary-search-trees (BSTs). Explain briefly how you would implement the operations to add and remove occurrences.

Illustrate your answer with a diagram showing the BST that results if we start with an empty bag, then add 2 apples, 1 lime, 2 lemons, 1 banana, 3 apples, and 1 orange (in that order).

[5]

(c) Assuming your BST representation of a bag, write down an algorithm to render a bag as a string. Choose a suitable format such as "{apple, apple, apple, banana}" or "{apple 3, banana 1}". The elements must be in ascending order.

[5]

(d) Using your bag ADT, complete the following Java method:

```
static void countWords (BufferedReader doc);
// Count and print the frequency of words in the text document doc.
```

You should use the auxiliary method of Table 2.

[5]