1. (a) What is meant by the *time complexity* of an algorithm? Give an example of the O-notation, and explain what it means.

(2)

(b) Table 1 shows the array quick-sort algorithm. Illustrate its behaviour as it sorts each of the following arrays of letters, assuming that step 1.1 selects $a[left]$ as the pivot. Each 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.

(i)

0	1	2	3	4	5	6	7
R	C	A	S	P	Z	T	M

(ii)

0	1	2	3	4	5	6	7
B	X	D	L	C	F	M	G

(4)

(c) In what circumstances does the quick-sort algorithm perform most efficiently?

Let $comps(n)$ be the number of comparisons performed by the algorithm as it sorts a (sub)array of length n. Assuming the best case, write down and explain equations of the form:

$$\begin{aligned} comps(n) &= \dots \qquad \text{if } n \le 1 \\ comps(n) &= \dots \qquad \text{if } n > 1 \end{aligned}$$

Assume that step 1.1 performs approximately n comparisons.

State (but do not derive) the quick-sort algorithm's best-case time complexity.

(3)

(d) In what circumstances does the quick-sort algorithm perform least efficiently? Write down and explain equations of the same form as in part (c). State (but do not derive) the algorithm's worst-case time complexity.

(3)

(e) Suggest how a partitioning algorithm might ensure that neither $a[left \dots p\text{–}1]$ nor $a[p\text{+}1 \dots right]$ is empty. You need not write out the partitioning algorithm in detailed steps, but your answer must make the idea clear.

(6)

(f) Repeat part (b)(ii), now assuming that step 1.1 uses the idea you suggested in part (e).

(2)

```
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) What is an *abstract data type* (*ADT*)?

(2)

(b) Design an ADT to meet the following requirements:

1. The values are to be *immutable* lists of any length.
2. It must be possible to test whether a list is empty.
3. It must be possible to determine the length of a list.
4. It must be possible to obtain the "head" of a non-empty list (i.e., the first element).
5. It must be possible to obtain the "tail" of a non-empty list (i.e., the list consisting of all elements except the first).
6. It must be possible to add a single element at the front of a list.

Express your design in the form of a Java interface.

(6)

(c) Outline a suitable data structure for representing immutable lists. Your answer must include diagrams showing: (i) the data structure's invariant; (ii) an empty list; (iii) a list containing the strings "alpha", "beta", and "gamma" in that order.

(4)

(d) Write a Java class that implements your interface.

For each operation of your ADT, briefly describe how it will be implemented, and state its time complexity.

(8)

3. A *bag* is a collection of members, some of which may be duplicates, but in which the order of the members is of no significance.

For example, the bags {apple, banana, apple} and {apple, apple, banana} are equal to each other, but unequal to {apple, banana} since the latter contains only one occurrence of "apple". The cardinality of {apple, apple, banana} is 3: the total number of members including duplicates.

Note: Bags resemble sets in that the order of the members is of no significance. Bags differ from sets in that bags may contain duplicate members.

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

- The values are to be bags of any cardinality.
- It must be possible to test whether a bag is empty.
- It must be possible to obtain the cardinality of a bag.
- It must be possible to determine whether a given value is a member of a bag.
- It must be possible to determine the number of occurrences of a given value in a bag.
- It must be possible to add a single occurrence of a given value to a bag. For example, adding "apple" to {apple, banana} should yield {apple, apple, banana}.
- It must be possible to remove a single occurrence of a given value from a bag. For example, removing "apple" from {apple, apple, banana} should yield {apple, banana}; removing "apple" again should yield {banana}.
- It must be possible to render a bag as a string, in a suitable format.

Express your design in the form of a Java interface.

(8)

(b) Outline an efficient representation of bags using hash tables.

Illustrate your answer with a diagram showing how the bag {apple, kiwi, apple, lime, lemon, date, lime, apple} would be represented. For the purposes of illustration you may use a hash function that simply takes the first letter of the word.

(5)

What hash function would you use in practice when the bag members are English-language words?

(1)

(c) Write application code that uses your bag abstract data type to count the frequency of words in a text document. You may assume that a class

`HashBag` implementing your interface of part (a) is available. You may also assume that the following method is available:

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

(6)

4. (a) What is meant by a *tree*? (*Note:* This question is about a tree abstract data type, not about search-tree data structures.)

(2)

(b) Suppose that a small university comprises Schools of Business, Science, and Technology. The School of Business comprises Departments of Management, Economics, and Law. The School of Science comprises Departments of Physics and Mathematics. The School of Technology comprises Departments of Computing and Electronic Engineering. Draw a tree that captures the structure of this university.

(2)

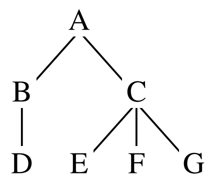
(c) Explain what are meant by *breadth-first traversal* and *depth-first traversal* of a tree (as opposed to a graph).

(2)

(d) Develop an algorithm that performs a depth-first traversal of a given tree.

(4)

(e) Table 2 shows the contract for a tree abstract data type. Assuming this contract, develop a Java method that traverses a given tree and prints out the tree elements indented to show their relationship. For example, the tree below left should be printed as shown below right:



```
A
   B
      D
   C
      E
      F
      G
```

(10)

```
public interface Tree {

   // Each Tree object is a tree whose elements are arbitrary objects.

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

   public Tree.Node root ();
   // Return the root node of this tree, or null if this tree is empty.

   public Tree.Node parent (Tree.Node node);
   // Return the parent of node in this tree, or null if node is the root node.

   public int childCount (Tree.Node node);
   // Return the number of children of node in this tree.

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

   public void makeRoot (Object elem);
   // Make this tree consist of just a root node containing element elem.

   public Tree.Node addChild (Tree.Node node,
                  Object elem);
   // Add a new node containing element elem as a child of node in this
   // tree, and return the new node. The new node has no children of its own.

   public void remove (Tree.Node node);
   // Remove node from this tree, together with all its descendants.

   //////////// Iterator ////////////

   public Iterator children (Tree.Node node);
   // Return an iterator that will visit all the children of node in this tree.

   //////////// Inner interface for tree nodes ////////////

   public interface Node {

      // Each Tree.Node object is a node of a tree, and contains a single
      // element.

      public Object getElement ();
      // Return the element contained in this node.

      public void setElement (Object elem);
      // Change the element contained in this node to be elem.

   }
}
```

Table 2 Contract for a tree abstract data type (Question 4).