**1.** **(a)** Explain what is meant by saying that an algorithm's ***time complexity*** is $O(n^2)$.

(2)

**(b)** Write down the time complexities of the following sorting algorithms:

(i) radix-sort

(ii) selection sort

(iii) merge-sort

stating what characteristic operations you are counting in each case. Briefly justify your answers. (*Note:* Intuitive explanations are sufficient; you are not required to derive your answers mathematically.)

(9)

**(c)** A company's internal telephone directory entries are kept in a serial file, each entry consisting of a name and number. The entries are kept sorted by name (each name being assumed to be unique).

Every month, a batch of new entries is assembled in a serial file (unsorted). If a new entry has the same name as an existing entry, the new entry should replace the existing entry. Otherwise the new entry should be added to the telephone directory.

Devise an efficient algorithm to update the telephone directory using a batch of new entries.

*Note:* You may use well-known algorithms as auxiliary algorithms. Do not reproduce them.

(6)

Let $n$ be the number of entries in the telephone directory, and let $m$ be the number of new entries. What is your algorithm's time complexity? Briefly justify your answer.

(3)

**2.** **(a)** What is the difference between ***singly-linked lists*** (SLLs) and ***doubly-linked lists*** (DLLs)?

(2)

Identify a basic operation on a DLL that is much more efficient than the corresponding operation on an SLL. Briefly explain your answer.

(2)

**(b)** Appendix 1 shows a contract for the `Queue` abstract data type (ADT).

Outline an *SLL* representation of queues. Illustrate your answer by showing the SLL representation of: (i) the queue containing Kenny, Kyle, and Stan (in that order); (ii) the queue after adding Cartman at the rear; (iii) the queue after removing the element at the front.

(3)

Write an implementation of the `addLast` method.

(3)

**(c)** Appendix 2 shows a contract for the `List` ADT.

Outline a *DLL* representation of lists. Illustrate your answer by showing the DLL representation of (i) the list containing Kenny, Kyle, and Stan (in that order); (ii) the list after adding Cartman at index 0; (iii) the list after removing the element at index 1.

(4)

Write an implementation of the second `add` method (the one that adds a given element after the last element of the list).

(4)

**(d)** SLLs make a perfectly adequate data representation for the `Queue` ADT, but DLLs are to be preferred for the `List` ADT. Explain.

(2)

**3.** A ***bag*** is a collection of members, which may contain duplicate members, but in which the order of members is of no significance. Here are three examples of bags:

(i) {apple, orange, apple}

(ii) {apple, apple, orange}

(iii) {apple, orange}

Bags (i) and (ii) are equal to each other, but bag (iii) is unequal to (i) or (ii) since it contains only one occurrence of 'apple'.

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

**(a)** Write a contract for a `Bag` abstract data type that meets the following requirements:

- It must be possible to make a bag empty.
- It must be possible to test whether a bag is empty.
- It must be possible to obtain the cardinality of a bag (i.e., the total number of members including duplicates).
- 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 member to a bag. For example, adding 'apple' to {apple, orange} should yield {apple, apple, orange}.
- It must be possible to remove (a single occurrence of) a member of a bag. For example, removing 'apple' from {apple, apple, orange} should yield {apple, orange}; removing 'apple' again should yield {orange}.
- It must be possible to test whether two bags are equal.
- It must be possible to obtain the set of (distinct) members of a bag.

Your contract must be in the form of a Java interface. Each operation must be accompanied by a comment specifying precisely (but concisely) the operation's observable behaviour. (*Note:* Appendices 1 and 2 illustrate a suitable style.)

(8)

**(b)** Outline an efficient data representation for a bag. Illustrate your answer by showing your representation of bag (i) above. Briefly explain how you would determine the number of occurrences of a given value in a bag.

(4)

**(c)** Using your `Bag` contract, write a piece of application code that reads a given document and produces a word frequency profile. Your code should print out all words that occur in the document together with each word's relative frequency (expressed as a percentage of the total number of words in the document). The words need not be printed in any particular order.

You may assume that the following method is available:

```
static String readWord (BufferedReader doc);
// Read the next word from doc, and return that word.
// Return null if all words have already been read.
```

(8)

**4.** Write an overview of the Java collections framework. Your answer should identify the principal interfaces and classes, explain the role of each, and explain the relationships among them. Illustrate your answer with appropriate class diagram(s).

*Note:* Do not describe the interfaces and classes in detail.

(20)

# 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.

}
```



continued overleaf/

# Appendix 2 Contract for a `List` abstract data type

```
public interface List {

  // Each List object is an indexed list whose elements are objects.

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

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

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

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

  …

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

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

  public void set (int i, Object elem);
  // Replace by elem the element at index i in this list.

  public void add (int i, Object elem);
  // Add elem as the element with index i in this list.

  public void add (Object elem);
  // Add elem after the last element of this list.

  public void addAll (List that);
  // Add all the elements of that after the last element of this list.

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

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

  public Iterator iterator ();
  // Return an iterator that will visit all elements of this list, in order
  // of increasing index.

}
```