Algorithms & Data Structures (M): Questions and Answers: Spring 2012
Duration: 120 minutes.
Rubric: Answer any three questions. Total 60 marks.

1. **(a)** What is meant by the *time complexity* of an algorithm?

[Notes]

The time complexity of an algorithm is the rate at which its time requirement grows as a function of its input data.

[2]

(b) Suppose that you are given two different algorithms that solve the same problem. Given n items of data, algorithm A takes $20n^2$ milliseconds, whilst algorithm B takes $(100\ n \log_2 n)$ milliseconds.

(i) What is each algorithm's time complexity?

(ii) *Without doing any calculations*, state which algorithm you would choose. Explain your answer.

(iii) Now calculate (to the nearest second) each algorithm's running times for n = 8, 16, 32, 64. Tabulate the results. Comment on whether the results confirm your answer to (b)(ii).

[Unseen problem]

(i) Algorithm A is $O(n^2)$. Algorithm B is $O(n \log n)$.

(ii) Algorithm B has the slower growth rate, so will be much faster than algorithm A for larger values of n. (Smaller values of n don't matter.)

(iii)

	n	8	16	32	64	
	algorithm A	1	5	20	82	secs
	algorithm B	2	6	16	38	secs

These results confirm that algorithm B is faster for n greater than about 16.

[2+2+2]

(c) Write down the array *merge-sort* algorithm. (*Note:* Code is not required.)

[Notes]

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

1. If $left < right$:
   1.1. Let m be an integer about halfway between $left$ and $right$.
   1.2. Sort $a[left \ldots m]$.
   1.3. Sort $a[m+1 \ldots right]$.
   1.4. Merge $a[left \ldots m]$ and $a[m+1 \ldots right]$ into auxiliary array b.

1.5. Copy *b* into *a*[*left*…*right*].
2. Terminate.

[3]

Illustrate the merge-sort algorithm's behaviour as it sorts the following array of words:

0	1	2	3	4	5	6	7	8	9
it	or	my	he	an	is	be	in	me	am

Your illustration must show the contents of the array after each step of the algorithm. (*Note:* If the algorithm calls itself or another algorithm, treat the call as a single step.)

[Unseen problem]

		0	1	2	3	4	5	6	7	8	9		
After step 1.1:	*a*	it	or	my	he	an	is	be	in	me	am	*m*	4
After step 1.2:	*a*	an	he	it	my	or	is	be	in	me	am	*m*	4
After step 1.3:	*a*	an	he	it	my	or	am	be	in	is	me	*m*	4
After step 1.4:	*b*	am	an	be	he	in	is	it	me	my	or	*m*	4
After step 1.5:	*a*	am	an	be	he	in	is	it	me	my	or	*m*	4

[3]

(d) In terms of time and space complexity, how does the merge-sort algorithm compare with the quick-sort algorithm?

[Notes]

Merge-sort is O(n log n) time and O(n) space.

Quick-sort's best case is O(n log n) time and O(log n) space. Its worst case is O(n^2) time and O(n) space.

So merge-sort tends to be less time-consuming but also more space-consuming.

[6]

[total 20]

2. **(a)** What is meant by an *abstract data type* (*ADT*)?

How are ADTs supported by Java?

[Notes]

An ADT is a type characterised by its values and operations only. Its data representation is hidden.

An ADT contract can be expressed as a Java interface, which specifies each operation (public method) to be provided. An ADT implementation can be expressed as a Java class, in which each operation is fully defined, and in which the data representation is defined by private instance variables.

[4]

(b) Box 2 shows a simplified contract for a homogeneous queue ADT.

Write a class `LinkedQueue<E>` that implements this contract using a *singly-linked-list* representation.

[Notes]

```
public class LinkedQueue<E> implements Queue<E> {

    private int n;
    private Node front, rear;

    public LinkedQueue<E> () {
        n = 0;
        front = rear = null;
    }

    public int size () {
        return n;
    }

    public E getFirst () {
        if (front == null) throw ...;
        return front.element;
    }

    public E removeFirst () {
        if (front == null) throw ...;
        E elem = front.element;
        front = front.succ;
        if (front == null) rear = null;
        n--;
        return elem;
    }

    public void addLast (E it) {
        Node newest = new Node(it);
        if (rear == null)
            front = newest;
```

```
            else
                rear.succ = newest;
            rear = newest;
            n++;
        }

        private static class Node {
            public E element;
            public Node succ;
            public Node (E it) {
                element = it; succ = null;
            }
        }

}
```

[8]

(c) A *deque* (or *double-ended queue*) is a sequence of elements with the property that elements can be added and removed at both ends.

Design a homogeneous deque ADT, whose elements are objects of type `E`. Your ADT must enable application programs to:

(1) determine the number of elements in the deque;

(2) add a given element at the front or rear of a deque;

(3) fetch and remove the element at the front or rear of a deque.

Express your design in the form of a Java generic interface. Each operation must be accompanied by a comment specifying the operation's observable behaviour.

[Seen problem]

```
public interface Deque<E> {

  // Each Deque<E> object is a deque whose elements are objects of type E.

  public boolean size ();
  // Return the number of elements in this deque.

  public void addFirst (E it);
  // Add it at the front of this deque.

  public void addLast (E it);
  // Add it at the rear of this deque.

  public E removeFirst ();
  // Remove and return the front element of this deque.

  public E removeLast ();
  // Remove and return the rear element of this deque.

}
```

[6]

(c) What data structure would you choose to represent an unbounded deque? Briefly explain your answer.

[Seen problem]

Choose a doubly-linked-list. This enables all operations including `addFirst` and `removeLast` to be O(1).

[2]

[total 20]

```
public interface Queue <E> {

      // Each Queue<E> object is a homogeneous queue whose elements are objects
      // of type E.

      public int size ();
      // Return the number of elements in this queue.

      public E getFirst ();
      // Return the frontmost element of this queue.

      public void addLast (E it);
      // Add it to the rear of this queue.

      public E removeFirst ();
      // Remove and return the frontmost element of this queue.

}
```

Box 2 A contract for homogeneous queues.

3. **(a)** Define what is meant by a *set*.

Explain clearly how sets differ from lists.

[Notes]

A set is a collection of distinct elements (members), in no particular order.

Sets differ from lists in that duplicate elements are not allowed and the order in which elements are added is of no significance.

[2]

Box 3 shows a simplified contract for a homogeneous `Set` abstract data type, expressed in the form of a Java interface `Set<E>`.

(b) Explain how a *bounded* set could be represented by an array. Briefly explain how each of the operations of Box 3 would be implemented. (*Note:* If you use standard algorithms, just identify them; you do not need to explain how the standard algorithms work.)

[Notes]

For a bounded set of capacity `m`, use an integer `n` and an array `a[0...m-1]`. Store the set elements in `a[0...n-1]`.

`contains` should use binary search.

`add` should use binary search followed (if necessary) by array insertion.

`remove` should use binary search followed (if necessary) by array deletion.

`addAll` should merge the two arrays, eliminating common elements.

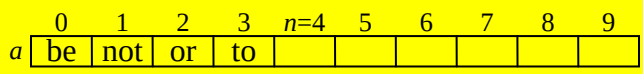
`equals` should compare the elements of the two arrays pairwise.

[5]

Illustrate your answer with a diagram showing the array after the following words have been added to an empty set:

"to", "be", "or", "not", "to", "be"

[Unseen problem]



[2]

State the time complexities of the `contains` and `add` operations.

[Notes]

`contains` is O(log *n*).

`add` is O(*n*).

[2]

(c) Describe a data structure that is more efficient than the array representation, and which could be used to represent unbounded sets. Briefly explain how each of the operations of Box 3 would be implemented.

[Notes + insight]

One possible data structure is a binary-search-tree (BST).

`contains` should use BST search.

`add` should use BST insertion.

`remove` should use BST deletion.

`addAll` should traverse the second BST, and add each element to the first BST using BST insertion.

`equals` should first compare cardinalities; then it should traverse one BST, and test each element's presence in the other BST using BST search.

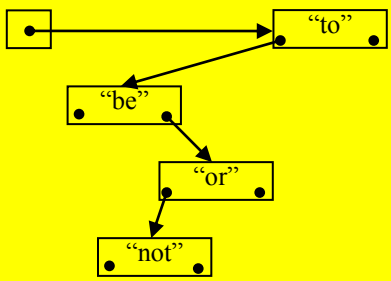
[Equally acceptable answer: Use a hash-table together with a hash function that tends to distribute the elements evenly among the buckets.]

[5]

Illustrate your answer with a diagram showing your data structure after the following words have been added to an empty set:

"to", "be", "or", "not", "to", "be"

[Unseen problem]



[2]

State the time complexities of the `contains` and `add` operations.

[Notes]

Both `contains` and `add` are O(log *n*) if the BST is well-balanced, O(*n*) otherwise.

[2]

[total 20]

```
public interface Set<E> {

    // Each Set<E> object is a homogeneous set whose members are objects
    // of type E.

    public boolean contains (E it);
    // Return true iff it is a member of this set.

    public void add (E it);
    // Add it as a member of this set.

    public void remove (E it);
    // Remove it from this set.

    public void addAll (Set<E> that);
    // Add all elements of that to this set.

    public boolean equals (Set<E> that);
    // Return true if this set is equal to that.

}
```

Box 3 A contract for homogeneous sets.

4. **(a)** Define the concept of a *graph*, and the concept of an *undirected graph*.

Box 4A shows a road network, in which A–E are towns and the numbers are road distances. Show that a road network such as this (where all roads are two-way) is an example of an undirected graph.

[Notes]

A graph is a collection of vertices connected by edges. Edges may have attributes.

An undirected graph is one in which the edges have no direction.

The road network is an undirected graph in which vertices correspond to towns, edges correspond to roads connecting these towns, and edge attributes are distances.

[4]

(b) Box 4B shows the *shortest-distances algorithm*, which determines the shortest distance from a given town, *start*, to every other town in a road network. For each town *t*, the variable $dist_t$ contains the shortest distance so far discovered from *start* to *t*.

Trace this algorithm as it finds the shortest distance from town A to every other town in the road network of Box 4A. Present your trace in tabular fashion as follows:

$dist_A$	$dist_B$	$dist_C$	$dist_D$	$dist_E$	*town-set*
0	∞	∞	∞	∞	{A, B, C, D, E}
…	…	…	…	…	…
…	…	…	…	…	…

where the first line shows the values of the variables after step 2, and the remaining lines show their values after each iteration of the loop in step 3.

[Unseen problem]

$dist_A$	$dist_B$	$dist_C$	$dist_D$	$dist_E$	*town-set*
0	∞	∞	∞	∞	{A, B, C, D, E}
0	∞	12	8	∞	{B, C, D, E}
0	19	11	8	20	{B, C, E}
0	17	11	8	20	{B, E}
0	17	11	8	20	{E}

[6]

(c) Now consider a related problem: to find the shortest path from town *start* to town *dest* in a road network.

Modify the shortest-distances algorithms to solve this problem. (*Hint:* You might find it helpful to write down your ideas before writing down the modified algorithm itself.)

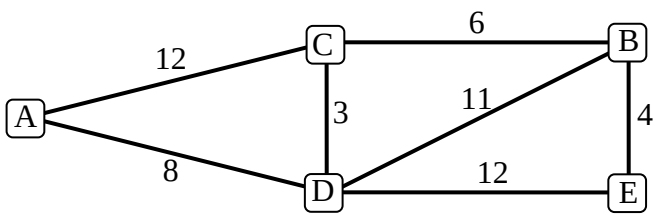
[Notes + seen problem]

Ideas: For each town t, let $pred_t$ be the predecessor of t on the shortest path from *start* to t. Update $pred_t$ whenever $dist_t$ is updated. Terminate when $t = dest$.

To find the shortest path from town *start* to town *dest* in a road network:

1. Make *town-set* contain all towns in the network.
2. Set $dist_{start}$ to 0, and set $dist_t$ to infinity for all other towns t.
3. Set $pred_t$ to null for all towns t.
4. While *town-set* is not empty, repeat:
   - 4.1. Remove from *town-set* the town t with least $dist_t$.
   - 4.2. If $t = dest$, terminate with the path defined by $pred_{dest}$, etc.
   - 4.3. For each road connecting t and another town u, such that u is in *town-set*, repeat:
     - 4.3.1. Let $d = dist_t$ + (distance along road from t to u).
     - 4.3.2. If $d < dist_u$, set $dist_u$ to d and set $pred_u$ to t.

[10]
[total 20]



Box 4A A road network.

To find the distance of the shortest path from town *start* to every other town in a road network:

1. Make *town-set* contain all towns in the network.
2. Set $dist_{start}$ to 0, and set $dist_t$ to infinity for all other towns t.
3. While *town-set* is not empty, repeat:
   - 3.1. Remove from *town-set* the town t with least $dist_t$.
   - 3.2. For each road connecting t and another town u, such that u is in *town-set*, repeat:
     - 3.2.1. Let $d = dist_t$ + (distance along road from t to u).
     - 3.2.2. If $d < dist_u$, set $dist_u$ to d.
4. Terminate with the distances $dist_t$.

Box 4B Shortest-distances algorithm.