Algorithms & Data Structures (M): Questions and Answers: Spring 2014
Duration: 120 minutes.
Rubric: Answer any three questions. Total 60 marks.

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

[2]

[Notes]

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

(b) Suppose that you have to choose between two alternative algorithms to solve the same problem. Given n items of data, algorithm A take $(10n \log_2 n)$ milliseconds, whilst algorithm B takes $(2n^2 + 5n)$ 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 and tabulate both algorithms' running times for n = 10, 20, 30, 40. Do your calculations confirm your answer to (b)(ii)?

(*Note:* $\log_2 10 \approx 3.3$; $\log_2 20 \approx 4.3$; $\log_2 30 \approx 4.9$; $\log_2 40 \approx 5.3$.)

[7]

[Similar to seen problem]

(i) Algorithm A is O(n log n), algorithm B is O(n^2). [2]

(ii) Algorithm A has the slower growth rate, so Algorithm A will be much faster for larger values of n. (Smaller values of n don't matter.) [2]

(iii)

	n	10	20	30	40	
Algorithm A		330	860	1470	2120	ms
Algorithm B		250	900	1950	3400	ms

This confirms that Algorithm A is faster for n >= 20. [3]

(c) Box 1 shows an algorithm that takes an array a and searches it for a subarray that matches a shorter array b. This uses an auxiliary algorithm that compares two subarrays of equal length.

For example:

	0	1	2	3	4	5	6	7	8	9
a	Q	W	E	R	T	Y	W	E	R	E

	0	1	2
b	W	E	R

Here the answer should be 1 (the index of the leftmost matching subarray of a).

(i) What is the time complexity of the auxiliary algorithm? Informally justify your answer.

(ii) What is the time complexity of the main algorithm? Informally justify your answer.

[6]

[Similar to seen problem]

(i) The auxiliary algorithm is O(m), since it must perform up to m character comparisons. [2]

(ii) The main algorithm is O(m(n–m)), since it must call the auxiliary algorithm up to about n–m times. [4]

(d) Name a sorting algorithm that is faster than O(n^2). Write down your chosen algorithm. What is its time complexity?

[5]

[Notes]

Either merge-sort or quick-sort would be acceptable. Both are O(n log n).

Main algorithm:

To search the array $a[0\ldots n-1]$ for a subarray that matches $b[0\ldots m-1]$, where $m < n$:

1. For $i = 0, \ldots, n-m$, repeat:
   1.1. If $a[i\ldots i+m-1]$ matches $b[0\ldots m-1]$, terminate with answer i.
2. Terminate with answer *none*.

Auxiliary algorithm:

To test whether $a[i\ldots i+m-1]$ matches $b[0\ldots m-1]$:

1. For $j = 0, \ldots, m-1$, repeat:
   1.1. If $a[i+j] \neq b[j]$, terminate with answer *false*.
2. Terminate with answer *true*.

Box 1 An algorithm to search an array for a matching subarray.

2. **(a)** Define what is meant by a *queue*.

[2]

[Notes]

A queue is a first-in-first-out sequence of elements.

(b) Box 2 shows a contract for a homogeneous queue ADT (abstract data type), in the form of a Java interface `Queue<E>`.

Outline a class `ArrayQueue<E>` that implements the interface `Queue<E>`. Your implementation must represent a queue by a *circular array* with capacity 5. Your answer must include declarations of all necessary instance variables, and a full implementation of the `remove` and `add` methods. Do *not* implement the other methods.

[10]

[Notes]

```
class ArrayQueue <E> implements Queue<E> {

    private E[] elems;
    private int front, rear, size;                          [2]

    public ArrayQueue<E> () { … }

    public int size () { … }

    public E get () { … }

    public E remove () { … }
        if (size == 0) …  -- throw an exception
        E it = elems[front++]:
        if (front == elems.length) front = 0;
        size--;
        return it;
    }                                                       [4]

    public void add (E it) {
        if (size == elems.length) …  -- expand the array
        elems[rear++] = it;
        if (rear == elems.length) rear = 0;
        size++;
    }                                                       [4]

}
```

(c) Show what happens when the following operations are performed:

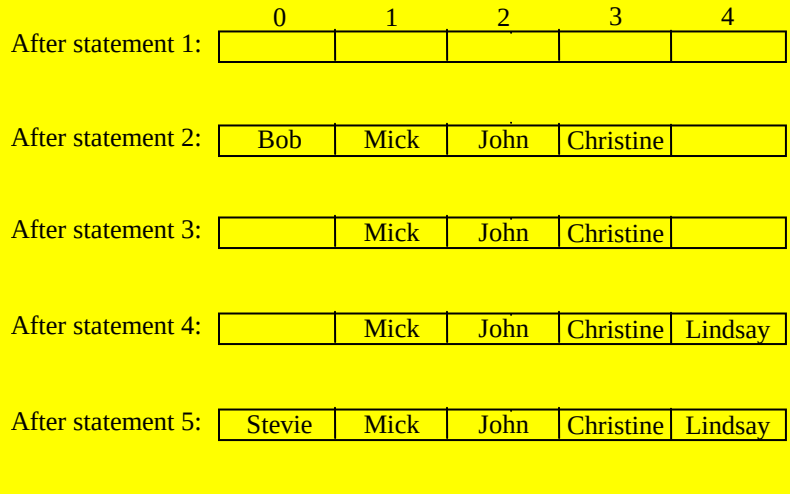
```
String s;
Queue<String> q = new ArrayQueue<String>();   --1
q.add("Bob");
q.add("Mick");
```

```
q.add("John");
q.add("Christine");   --2
s = q.remove();       --3
q.add("Lindsay");     --4
q.add("Stevie");      --5
```

Your answer should consist of diagrams showing the contents of the data structure immediately after each of the statements 1, 2, 3, 4, and 5.

[5]

[Unseen problem]



[1] [1] [1] [1] [1]

(d) Explain carefully why a circular array is better than an ordinary array for representing queues.

[3]

[Notes]

If an ordinary array is chosen, whenever the queue bumps into the right end of the array, it has to be shifted to the left end of the array. That makes the `add` operation O(*n*) instead of O(1).

```
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 get ();
    // Return the front element of this queue.

    public E remove ();
    // Remove and return the front element of this queue.

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

}
```

Box 2 A contract for homogeneous queues.

3. **(a)** Define what is meant by a *set*. Your definition must clearly distinguish sets from other types of collections (such as lists).

[2]

[Notes]

A set is a collection of elements (members), in no particular order, and in which no member is duplicated.

(b) Box 3 shows a simplified contract for a set ADT, in the form of a Java interface `Set<E>`.

Using this interface, implement the following Java method:

```
void printUnique (String[] words) { … }
```

The method must print all the *unique* words in the array, in lexicographic order. In other words, each *distinct* word in the array must be printed once only.

[4]

[Similar to seen problem]

```
void printUnique (String[] words) {
    Set<String> wordSet = new TreeSet<String>();
    for (String word: words)
        wordSet.add(word);
    for (String word: wordSet)
        System.out.println(word);
}
```

(c) Explain how a set may be efficiently represented by a *hash-table*. Explain the role of the *hash function*.

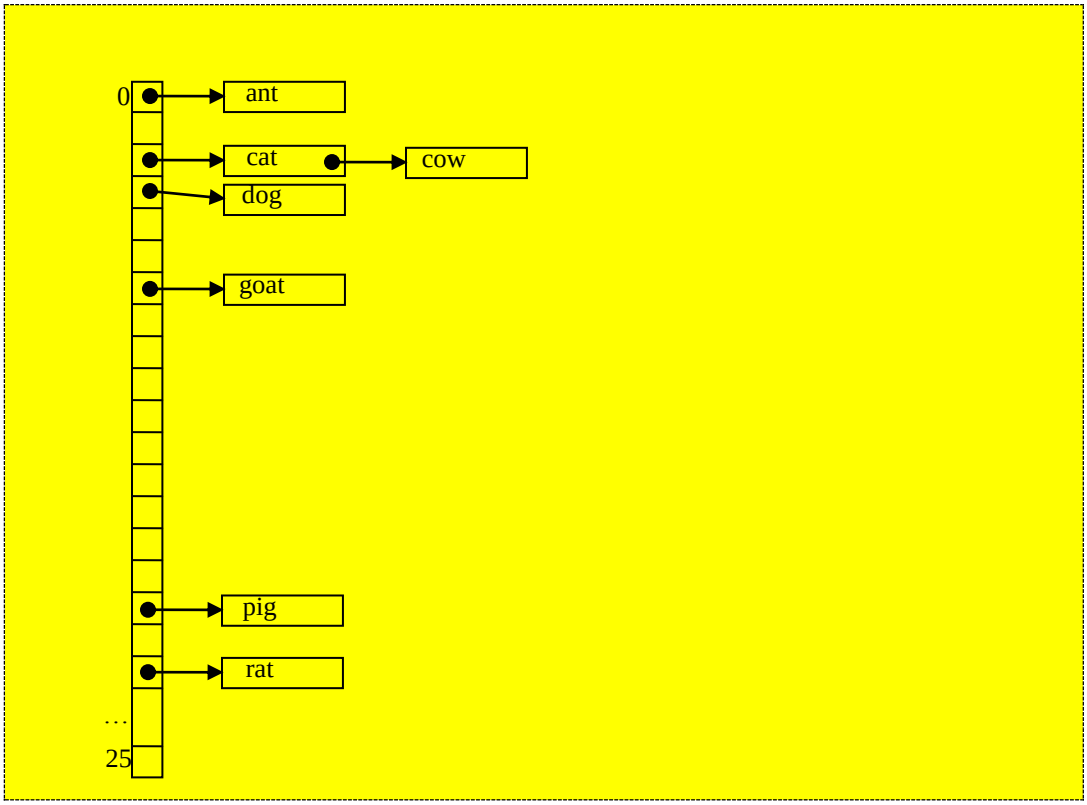
Illustrate your answer by showing a hash-table representation of the set of words { ‘cat’, ‘dog’, ‘cow’, ‘pig’, ‘goat’, ‘rat’, ‘ant’}. For simplicity, use a hash function based on the word’s initial letter only.

[6]

[Notes + unseen problem]

A (closed-bucket) hash-table is an array of buckets, where each bucket contains some of the set members in a linked-list. A hash function is chosen to translate each member to a bucket number, which determines that member’s “home bucket”. [3]

Illustration: [3]



(d) How should the hash function be chosen to ensure that the `Set` operations are efficient?

[2]

[Notes]

The hash function should be chosen to distribute the members thinly and evenly over the buckets.

(e) Compare and contrast the *hash-table* and *binary-search-tree* representations of sets. Your answer should cover the time complexities of the `add`, `remove`, and `contains` operations, and any other relevant issues.

[6]

[Notes + insight]

Hash-table: In the best case (when the members are distributed evenly and thinly), the `add`, `remove`, and `contains` operations are all O(1). In the worst case (when the members are all in one bucket), these operations are all O(*n*). [2]

BST: In the best case (when the tree is well-balanced), the `add`, `remove`, and `contains` operations are all O(log *n*). In the worst case (when the tree is ill-balanced), these operations are all O(*n*). [2]

So a hash-table is potentially faster, but only if the hash function is well chosen. BSTs do have the advantage of make it possible to iterate over the members in ascending order. [2]

[+1 for noting that a balancing insertion algorithm (AVL or red-black tree) can guarantee O(log *n*) performance.]

```
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 if and only if 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.

}
```

Box 3 A simplified contract for homogeneous sets.

4. **(a)** Define what is meant by a *graph*.

What is the difference between a *directed graph* and an *undirected graph*?

[4]

[Notes]

A graph is a set of vertices together with a set of edges, where each edge connects two vertices. [2]

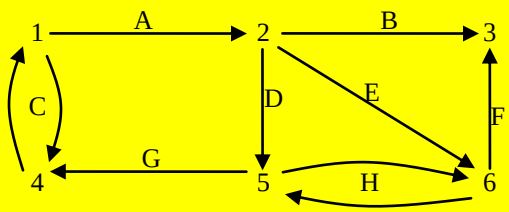
A directed graph is one in which every edge has a direction. An undirected graph is one in which no edge has a direction. [2]

(b) Box 4 shows a city-centre street network. The street intersections are numbered 1, 2, 3, 4, 5, and 6. The streets themselves are named A, B, C, D, E, F, G and H. Each street *with* an arrow is one-way; each street *without* an arrow is two-way.

Draw a graph suitable for modeling this street network. Have you chosen a directed or undirected graph? Briefly explain your decision.

[5]

[Unseen problem]



[4]

This is a directed graph. This enables a 1-way street to be modelled by a single edge, and a 2-way street to be modelled by a pair of edges. [1]

(c) Carefully describe a data structure suitable for representing such a graph.

Illustrate your answer by means of a diagram showing the representation of the graph you have drawn in part (b).

[7]

[Notes + unseen problem]

The most natural data structure is an adjacency matrix, with one row and one column for each intersection. If the graph contains an edge from intersection *i* to intersection *j*, the matrix cell [*i*,*j*] contains a description of that edge (the street name and the numbers of its end intersections). If the graph contains no such edge, the matrix cell [*i*,*j*] is empty. [3]

	1	2	3	4	5	6
1		A		C		
2			B		D	E
3						
4	C					
5				G		H
6			F		H	

[4]

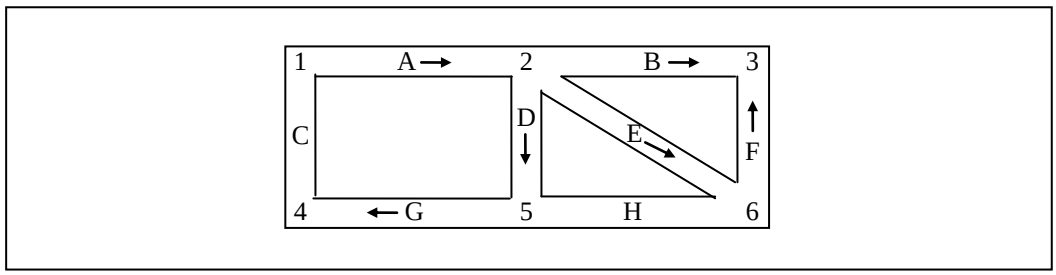
[Acceptable alternatives: the edge-set representation or the adjacency-set representation.]

(d) Consider the problem of finding the shortest route from intersection *start* to intersection *dest*. For simplicity, assume that all streets are of equal length. Identify a standard graph algorithm suitable for solving this problem.

[4]

[Notes]

Use the breadth-first search algorithm, starting at *start*, and terminating when it reaches *dest*.



Box 4 A city-centre street network.