University of Glasgow

Friday, 11 May 2012
9.30 am – 11.30 am
(Duration: 2 hours)

DEGREES OF MSc in Information Technology
AND MSc in Software Development

ALGORITHMS & DATA STRUCTURES (M)

Answer any 3 out of 4 questions.

This examination paper is worth a total of 60 marks.

You must not leave the examination room within the first hour or the last half-hour of the examination.

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

[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).

[6]

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

[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.)

[3]

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

[6]

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

How are ADTs supported by Java?

[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.

[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.

[6]

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

[2]

```
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.

[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.)

[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"

[2]

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

[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.

[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"

[2]

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

[2]

```
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.

[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}
…	…	…	…	…	…
…	…	…	…	…	…

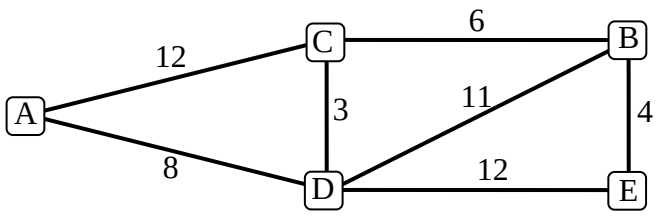
where the first line shows the values of the variables after step 2, and the remaining lines show their values after each iteration of the loop in step 3.

[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.)

[10]



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.