Office Use Only

Monash University

Semester Two Examination Period 2009

Faculty of Information Technology

EXAM CODES:	**FIT 2004**
TITLE OF PAPER:	**Data Structures and Algorithms – Final Exam**
EXAM DURATION:	3 hours writing time
READING TIME:	10 minutes

THIS PAPER IS FOR STUDENTS STUDYING AT:(tick where applicable)

☐ Berwick	X Clayton	☐ Malaysia	☐ Off Campus Learning	☐ Open Learning
☐ Caulfield	☐ Gippsland	☐ Peninsula	☐ Enhancement Studies	☐ Sth Africa
☐ Pharmacy	☐ Other (specify)			

AUTHORISED MATERIALS

CALCULATORS	**NO**
OPEN BOOK	**NO**
SPECIFICALLY PERMITTED ITEMS **if yes, items permitted are:**	**NO**

Candidates must complete this section if required to write answers within this paper

STUDENT ID __ __ __ __ __ __ __ __ DESK NUMBER __ __ __ __

(Intentionally left blank)

Additional Instructions

1. All answers must be given in the exam booklet. If possible write your answer in the space directly below the question. You may use extra pages to develop your answers, but these pages will not be marked.
2. You must tick the boxes for all questions that you have attempted in the table below on this page.
3. Unless stated otherwise, you only need to give procedural pseudo-code for algorithms. Concrete program code is only required where this is explicitly stated.
4. Where the questions ask for pseudocode, you are allowed to use concrete program code if you wish, but this is more difficult and not recommended.
5. Where program code is requested, the questions ask for Java code. You are allowed to use 'C' or SML code instead.
6. 'C' and SML programmers please note that where the questions ask for *methods*, you have to substitute *functions*, where they ask for *classes* you have to substitute *structures* or *datatypes*, respectively.

Good Luck!

I declare that I have attempted the questions marked in the table below:

Question	attempted	Marks (office use only)
1		
2		
3		
4		
5		
6		
7		

(Extra space for working)

Question 1 [Short Answer] [10 × 2 marks = 20 marks]

For each of the following questions give a concise answer (in two sentences or less). Where required draw a diagram to explain your answer.

1. Draw a valid **min-heap** containing the key elements $\{8, 14, 3, 12, 21, 5, 7\}$. Draw your heap as a tree, not as an array.

2. What is the worst case run-time of inserting an element into a **skip list** with n nodes?

3. Explain what "amortized" refers to in the term **amortized runtime analysis**.

4. Two of the following algorithms are virtually identical. Which two? (a) Bellman-Ford's shortest path algorithm, (b) Dijkstra's shortest path algorithm, (c) Floyd-Warshall's shortest path algorithm, (d) Kruskal's minimum spanning tree algorithm, (e) Prim's minimum spanning tree algorithm, (f) Ford-Fulkerson's network flow algorithm.

5. The black depth of a **red-black tree** grows only in a *recoloring* (split) operation. Draw a diagram to illustrate the last step of a split that increases the height of a tree. (Draw only the top levels of the tree before and after the split).

6. Searching for a **longest path** in graph is in general an ill-defined problem. Why? For which type of graph does it make sense?

7. Give an upper bound for the total runtime spent on union operations in *Kruskal's minimum spanning tree algorithm* provided the **union-find** structure is implemented with path compression. What is the total complexity of Kruskal's algorithm when implemented in this way?

8. Explain what is wrong in the following statement: "It is not possible to write a **sorting algorithm** that has a better runtime bound than $O(n \cdot log\ n)$ steps for sorting n elements".

9. Is it possible to build a **2-4 tree** from an unsorted array of n items in $O(n)$ time? Justify your answer!

10. **Big-O notation**: is $log_a\ n^b$ in $O(log_b\ n^a)$ for constant a, b? Explain your answer.

Question 2 [Tree Structures] [18 marks]

1. Draw a compressed **suffix trie** for the text “sagemessages”. Use compressed paths, but you do not need to use the indexing of the compact representation. **[3 marks]**

2. In the above **suffix trie** indicate the matching path for the search string “agem”. **[1 mark]**

3. A **2-3 tree** is a data structure exactly like the 2-4 tree we discussed except for the fact that each node has at most 3 children. What is the runtime complexity of search in a 2-3 tree? **[2 marks]**

4. Construct an **AVL tree** from the item $\{2, 3, 1, 4, 12, 7, 9\}$ by inserting them in the given order. Draw the resulting tree after each insertion. **[4 marks]**

5. Write pseudo-code for an algorithm *avl_test* that tests whether a given binary tree is in **AVL balance**. You must assume that the nodes of the tree only store keys and children, but no information on the balance or depths of sub-trees. **[3 marks]**

6. Draw a valid **red-black tree** for the keys $3, 4, 8, 6, 12, 14, 18$. Indicate for each node whether it is red or black. **[3 marks]**

7. Which of the following operations in a **Red-black tree** can propagate through the tree when repairing a *double red*: re-coloring (=split), re-structuring? For operations that *cannot* propagate, explain why this cannot happen. **[2 marks]**

(Extra space for working)

Question 3 [Graphs and Graph Algorithms] [14 marks]

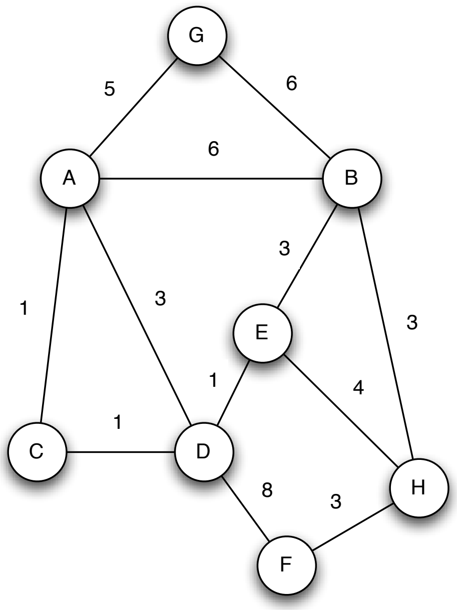

Figure 1: Example Graph

1. Write down the **adjacency list** representation of the graph given in Figure 1. For this subtask you may ignore the edge weights. You may restrict the adjacency list to the subgraph induced by the nodes $\{A, B, C, D\}$. **[2 marks]**

When you number the number the nodes in the graph to answer Subtasks 2 and 3 use the notation "(X/Y)", where X is the depth first number and Y is the breadth first level.

2. Perform a **depth first search** on the graph given in Figure 1 starting from node A. Whenever you have a choice between two nodes, break ties in ascending alphabetical order. Number the nodes in the diagram in the order in which they are visited. **[3 marks]**

3. Perform a **breadth first search** on the graph given in Figure 1 starting from node A. Clearly mark each node with its level number in the breadth first search, such that the level number reflects the order in which the levels are visited. **[3 marks]**

4. Find a **minimum spanning tree** of the graph given in Figure 1 using *Kruskal's algorithm* and highlight it in the diagram. The edge labels in the Figure indicate the edges' lengths. Write down the sequence of edges that are added to the spanning tree by Kruskal's algorithm (in the correct order). Denote edges as (u, v). **[1 + 2 = 3 marks]**

5. Find a **minimum spanning tree** of the graph given in Figure 1 using *Prim's algorithm.* Start from the node labelled "A". Give the complete state of the priority queue for each iteration of Prim's algorithm just before the next minimum element is retrieved from the priority queue. You do not need to indicate the resulting spannign tree in the diagram. To break ties, assume that elements are added to the queue in lexicographical order of node labels. **[3 marks]**

Question 4 [Dynamic Programming] [12 marks]

You have to write an algorithm to determine how to optimally insert line breaks into a text when typesetting a paragraph.

The text t is given as a list of n words, $t = \omega_1, \omega_2, \omega_3, \ldots, \omega_n$. It has to be partitioned into lines by inserting line breaks between successive words ω_i, ω_{i+1}. You are given a function *cost(i,j)* which returns a real value that measures how bad the words $\omega_i, \ldots, \omega_j$ look when typset as a single line. If these words don't fit on a single line at all, *cost(i,j)* returns $+\infty$.

cost(i,j) is the only measure for how good your paragraph looks, the total penalty for the paragraph thus is the sum of the penalties for all its lines. Your task is to compute the minimum total penalty with which the whole paragraph can be typeset. Your algorithm does not need to return the positions of the chosen line breaks.

A good approach to compute this is via dynamic programming. The core idea is this: in one iteration consider only the suffix $\Omega = \omega_i, \ldots, \omega_n$ (in order of decreasing i). To compute the total minimum penalty for Ω you can check all prefixes $\Omega_i = \omega_i, \ldots, \omega_k$ of Ω. If Ω_i fits on one line the total minimum penalty for Ω with a linebreak inserted after word k is the sum of the penalty for having Ω_i on one line plus the minimum total penalty for typesetting the rest of Ω as a paragraph.

1. Write down a recursive formula for computing the total minimum penalty $C(i)$ for typsettting the suffix $\Omega_i = \omega_i, \ldots, \omega_n$ for a given index i. **[2 marks]**

2. Give pseudocode for a dynamic programming algorithm that determines the total minimum penalty for typesetting the whole text t. **[6 marks]**

3. What is the runtime complexity of your dynamic programming algorithm? You may assume that *cost(i,j)* can be computed in $O(1)$. **[2 marks]**

4. Assume the paragraph has a fixed line length of L characters, and that you are using a monospaced font (all characters have the same width). Furthermore assume that each word contains at least one character and that the total number of characters in t is k. What can you say about the runtime complexity of your algorithm in terms of k? **[2 marks]**

(Extra space for working)

Question 5 [Runtime Recurrence Relations] [11 marks]

1. State the worst case runtime recurrence relations for the following algorithm which sums up the elements of an *array*. Solve the recurrence relation. **[2 + 2 = 4 marks]**

```
Alg sum(int[] A, int low, int high) {
    // a very weird way of summing up integers
    if (low=high) return A[low];
    if (low>high) return 0;
    else {
      mid = floor((low+high)/2);
      x=sum(A,low,mid);
      y=sum(A,mid+1,high);
      return x+y;
    }
}
```

2. State the worst case runtime recurrence relation for the following algorithm which sums up the elements of an *list*. Solve the recurrence relation. **[2 + 2 = 4 marks]**

```
Alg sum(list l0, int low, int high) {
    // an even weirder way of summing up integers
    if (low=high) return first(l0);
    if (low>high) return 0;
    else {
      mid = floor((low+high)/2);
      list l1 = take(l0,mid);
      list l2 = drop(l0,mid);
      return sum(l1)+sum(l2);
    }
}
```

This algorithm uses the following versions of *take* and *drop*:

```
Alg drop(list l, int n) {
    if (n<=0) return l;
   else return drop(rest(l), n-1);
}
```

```
Alg take(list l, int n) {
     if (n<=0) return nil;
     else return insert(first(l),take(rest(l),n-1));
}
```

3. Solve the two recurrence relations for T1, T2 defined below and compare them. Assuming they stand for runtimes, which one describes the faster algorithm? **[1 + 1 + 1 = 3 marks]**

$$\begin{aligned} T1(1) &= 1 \\ T1(n) &= 3T(n/4) + n^2 \end{aligned}$$

$$\begin{aligned} T2(1) &= 1 \\ T2(n) &= 4T(n/2) + n \end{aligned}$$

Question 6 [Algebras and Datatypes] [10 marks]

The appendix gives an algebra for integer lists very similar to the one discussed in the lectures.

Your task is to extend this algebra with an operation *summable*, which is supposed to take two arguments (an integer and an integer list) and decide whether the given integer can be expressed as a sum of elements of the integer list such that each element in the list is used at most once.

1. Extend the algebra by specifying the signature for *summable*. **[1 mark]**

2. Extend the algebra by specifying the function *summable* in the `functions` part. You can use other operations already defined in the algebra and you may assume Integer and Boolean types to be defined with the usual operations. **[3 marks]**

3. State the recurrence relation for the runtime complexity of your *summable* function under the assumption that it is implemented directly as in your answer above. Solve this recurrence relation. If you are using any of the other operations in the algebra, state explicitly what you assume for their runtime complexity. **[2 marks]**

4. Define a function

$$\textit{filter} : \textit{intlist} \times \textit{intlist} \rightarrow \textit{intlist}$$

that takes two integer lists as arguments and extracts from the first list those that can be expressed as a sum of elements of the second list. In standard set notation, *filter* can be defined as

$$filter(l1, l2) = \{x \in l1 \mid \textit{summable}(x, l2)\}$$

Using the definition of *summable*, give a recursive definition of *filter* that works with integer lists. **[2 marks]**

5. State the runtime complexity of your *filter* operation under the assumption that it is implemented directly as in your answer above. If you are using any of the other operations in the algebra, state explicitly what you assume for their runtime complexity. **[2 marks]**

Question 7 [Divide and Conquer] [15 marks]

You have to write an algorithm that solves the following problem: Given a collection of n screws and n corresponding nuts, each screw must be matched to its nut. This should be done as fast as possible. For simplicity, we will assume that the collection of nuts is given as a list of part numbers (implemented as a list of integers). The collection of screws is given in the same way. To compare the size of two parts you can assume a function *comp(x,y)* which returns 0 if the parts with numbers x and y have the same size, -1 if the part x is smaller than y and $+1$ otherwise. You may also assume an operation *sort(l)* that sorts a list of part numbers according to item size.

Note: If your answer to Part 3 of this question is correct you will automatically be awarded the marks for Part 1 and Part 2 of this question. Thus, if you are sure of your answer to Part 3, there is no need to answer the first two parts.

1. Write down pseudo-code for a simple algorithm $match(l1, l2)$ that performs the pairing. You may assume as given the usual algorithms on lists that we have defined in the lecture. Your solution will be marked on correctness and runtime efficiency. **[4 marks]**

2. Determine the worst-case runtime complexity of the algorithm you have defined above. **[1 mark]**

3. We now make the task more difficult, by disallowing the use of the *sort(l)* operation. Furthermore you cannot use *comp(x,y)* anymore to compare two nuts (or two screws) with each other. The only type of comparison allowed is to compare a nut with a screw. The task can still be solved efficiently and elegantly with a divide-and-conquer approach. Let us call this algorithm *match-dc*. *Hint:* The basic idea of *match-dc* is the same as that of quicksort. Define and use a partition function that splits a list of nuts using a screw as a pivot or vice versa. *... continued overleaf* ⇒

- Outline your core idea for an efficient divide-and-conquer version *match-dc* in *plain English*. **[3 marks]**

- Write down the pseudo-code for the modified *match-dc*(l_1, l_2) according to the core idea you have outlined above. **[3 marks]**

- What is the *average case runtime* of *match-dc*? **[2 marks]**

- What is the *worst-case runtime* complexity of *match-dc*? **[2 marks]**

Appendix: Algebra *integer list*

```
ALGEBRA integer list

sorts intlist, int, bool;

ops
        empty:                              -> intlist;
           (* returns an empty new list *)
        insert:      int x intlist          -> intlist;
           (* inserts an element at the front *)
        delete:      int x intlist          -> intlist;
           (* deletes an element *)
        first:       intlist                -> int;
           (* returns the first element in the list *)
        rest:        intlist                -> intlist;
           (* returns the list with the first element removed *)
        contains:    int x intlist          -> bool;
           (* tests whether an element is contained in the list *)
        isempty:     intlist                -> bool;
           (* tests whether the list is empty *)
        append:      intlist x intlist      -> intlist;
           (* appends the second list to the first *)
sets
        bool    = {true, false};
        int     = Z;
        intlist = nil | cons(e:int, s:intlist);
functions
        append(nil, l)              = l
        append(cons(e,l1), l2)      = cons(e, append(l1, l2))

        isempty(nil)                = true
        isempty(cons(e,s))          = false

        insert(e, s)                = cons(e,s)

        delete(e, cons(e,s))        = delete(e,s)
        delete(e, cons(f,s))        = cons(f, delete(e,s)) if not (e=f)
        delete(e, nil)              = nil

        contains(e, nil)            = false
        contains(e, cons(e,s))      = true
        contains(e, cons(f,s))      = contains(e,s) if not (e=f)

        first(cons(e,l))            = e
        rest(cons(e,l))             = l
```

*

END OF EXAM