TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS — 2008

END-YEAR

COMP303
Design and Analysis of Algorithms

Time Allowed: 3 Hours

Instructions:

- *Read each question carefully before attempting it.*
- This examination will be marked out of **180** marks.
- Answer all questions.
- You may answer the questions in any order. Make sure you clearly identify the question you are answering.
- Non-electronic foreign language-English dictionaries are permitted.
- Reference material, *calculators*, use of mobile phones, laptop computers, PDAs or other electronic devices is NOT PERMITTED.

Questions		Marks
1.	Basic Algorithm Analysis	[32]
2.	Divide and Conquer	[34]
3.	Knapsacks	[24]
4.	Dynamic Programming	[32]
5.	Graphs	[14]
6.	Computability and Complexity	[12]
7.	Randomised Algorithms	[8]
8.	Approximation Algorithms	[24]

The following definitions are provided for your convenience. You may find it useful to tear off this front page of the paper.

Asymptotic notation:

$$\begin{aligned} O(g(n)) &= \{f(n) \mid (\exists d)(\mathbf{aa}\, n)[0 \leq f(n) \leq d.g(n)]\} \\ \Omega(g(n)) &= \{f(n) \mid (\exists c > 0)(\mathbf{aa}\, n)[f(n) \geq c.g(n) \geq 0]\} \\ \Theta(g(n)) &= \{f(n) \mid (\exists c > 0, d)(\mathbf{aa}\, n)[0 \leq c.g(n) \leq f(n) \leq d.g(n)]\} \end{aligned}$$

Master Theorem: Let $T(n)$ be defined by the recurrence $T(n) = aT(n/b) + f(n)$.
Let $\alpha = \log_b a$.

1. If $(\exists \epsilon > 0)[f(n) \in O(n^{\alpha-\epsilon})]$ then $T(n) \in \Theta(n^{\alpha})$.
2. If $f(n) \in \Theta(n^{\alpha})$ then $T(n) \in \Theta(n^{\alpha} \log n)$.
3. If $(\exists \epsilon > 0)[f(n) \in \Omega(n^{\alpha+\epsilon})]$ and $(\exists c < 1)(\mathbf{aa}\, n)[a.f(n/b) \leq c.f(n)]$ then $T(n) \in \Theta(f(n))$.

Logarithms:

$$\log_a x = y \text{ if and only if } a^y = x$$
$$\log_a x = \log_b x \div \log_b a$$

Question 1. Basic Algorithm Analysis [32 marks]

(a) [2 marks] Describe what is generally meant by an *algorithm* in Computer Science.

(b) [6 marks] Explain in plain English the meaning and usage of the O, Ω, and Θ notations defined on page 2 of this paper.

(c) [4 marks] Explain why it is necessary to have all three (O, Ω, and Θ) definitions.

(d) Using the definitions for O, Ω, and Θ given on page 2 of this paper, show that:

(i) [4 marks] $\frac{1}{2}n(n-1) \in \Theta(n^2)$

(ii) [4 marks] $2n + n^3 \in O(n^4)$

(e) For each of the following recurrence relations, give the asymptotic complexity (Θ bound) of $T(n)$. Justify your answers using the Master Theorem (given on page 2 of the paper).

(i) [4 marks] $T(n) = \begin{cases} 1, & \text{if } n = 0 \\ 9T(\lceil \frac{n}{3} \rceil) + n^2, & \text{otherwise} \end{cases}$

(ii) [4 marks] $T(n) = \begin{cases} 1, & \text{if } n = 0 \\ 6T(\lceil \frac{n}{6} \rceil) + \log(n^2), & \text{otherwise} \end{cases}$

(iii) [4 marks] $T(n) = \begin{cases} 1, & \text{if } n = 0 \\ T(\lceil \frac{n}{2} \rceil) + T(\lceil \frac{n}{4} \rceil) + T(\lceil \frac{n}{8} \rceil) + n & \text{otherwise} \end{cases}$

Question 2. Divide and Conquer [34 marks]

(a) [6 marks] Write pseudocode to define the basic structure of a typical divide-and-conquer algorithm. Explain the components of your algorithm.

Consider the following algorithm:

```
STOOGE-SORT(A, i, j)

if A[i] > A[j]
    then exchange A[i] ↔ A[j]
if i + 1 ≥ j
    then return
k ← ⌊(j−i+1)/3⌋               // Round down.
STOOGE-SORT (A, i, j − k)      // First two-thirds.
STOOGE-SORT (A, i + k, j)      // Last two-thirds.
STOOGE-SORT (A, i, j − k)      // First two-thirds again.
```

(b) [12 marks] Give the general structure of the proof of correctness of a divide and conquer algorithm, and use it to show that the STOOGE-SORT algorithm above correctly sorts the input array.

(c) [8 marks] Give a recurrence relation for the worst-case running time of STOOGE-SORT and a tight asymptotic (Θ) bound on the worst-case running time.

(d) Compare the worst-case running time of STOOGE-SORT with that of:

(i) [2 marks] insertion sort,

(ii) [2 marks] mergesort,

(iii) [2 marks] heapsort, and

(iv) [2 marks] quicksort.

Question 3. Knapsacks [24 marks]

(a) Discuss the applicability of

(i) [6 marks] Divide and Conquer,

(ii) [6 marks] Dynamic Programming,

(iii) [6 marks] Greedy Algorithms,

(iv) [6 marks] and Graph Algorithms

to both *Fractional Knapsack* and *0-1 Knapsack Problems*.

Question 4. Dynamic Programming [32 marks]

Suppose you're managing a consulting team of expert computer hackers, and each week you have to choose a job for them to undertake. Now, as you can well imagine, the set of possible jobs is divided into those that are *low-stress* (e.g., setting up a Web site for a class at the local elementary school) and those that are *high-stress* (e.g., protecting the nation's most valuable secrets, or helping a desperate group of Victoria University students finish an Honours Project). The basic question, each week, is whether to take on a low-stress job or a high-stress job.

If you select a low-stress job for your team in week i, then you get a revenue of $l_i > 0$ dollars; if you select a high-stress job, you get a revenue of $h_i > 0$ dollars. The catch, however, is that in order for the team to take on a high-stress job in week i, it's required that they do no job (of either type) in week $i-1$; they need a full week of prep time to get ready for the crushing stress level. On the other hand, it's okay for them to take a low-stress job in week i even if they have done a job (of either type) in week $i-1$.

So, given a sequence of n weeks, a *plan* is specified by a choice of "low-stress", "high-stress", or "none" for each of the n weeks, with the property that if "high-stress" is chosen for week $i > 1$, then "none" has to be chosen for week $i-1$. (It's okay to choose a high-stress job in week 1.) The *value* of the plan is determined in the natural way: for each i, you add l_i to the value if you choose "low-stress" in week i, and you add h_i to the value if you choose "high-stress" in week i, and you add h_i to the value if you choose "high-stress" in week i. (You add 0 if you choose "none" in week i.)

The problem. Given sets of values $l_1, l_2, \ldots, l_n$ and $h_1, h_2, \ldots, h_n$, find a plan of maximum value. (Such a plan will be called *optimal*.)

Example. Suppose $n = 4$, and the values of l_i and h_i are given by the following table. Then the plan of maximum value would be to choose "none" in week 1, a high-stress job in week 2, and low-stress jobs in weeks 3 and 4. The value of this plan would be $0 + 50 + 10 + 10 = 70$.

	Week 1	**Week 2**	**Week 3**	**Week 4**
l	10	1	10	10
h	5	50	5	1

(a) [8 marks] Show, by giving an instance on which it does not return the correct answer, that the following algorithm does not solve this problem.

To avoid problems with overflowing array bounds, we define $h_i = l_i = 0$ when $i > n$.

In your example, say what the correct answer is and also what the algorithm finds.

```
For iterations i = 1 to n
  If h_{i+1} > l_i + l_{i+1} then
    Output "Choose no job in week i"
    Output "Choose a high-stress job in week i+1"
    Continue with iteration i+2
  Else
    Output "Choose a low-stress job in week i"
    Continue with iteration i+1
  Endif
End
```

(b) [8 marks] Show that the problem has the **optimal substructure** property.

(c) [8 marks] Describe an appropriate **dynamic programming** algorithm for solving the problem.

(d) [4 marks] Briefly outline a proof that your algorithm is correct.

(e) [4 marks] State the asymptotic complexity of your algorithm. **Justify your answer.**

Question 5. Graphs [14 marks]

(a) [14 marks] Below are two algorithms for finding a minimal spanning tree $G' = (V', E')$ of a graph $G = (V, E)$.

Prim(V, E) returns (V', E')
  $V' \leftarrow \{\}$
  $E' \leftarrow \{\}$
  while $V' \neq V$
    $e \leftarrow$ shortest edge $x \rightarrow y$
      such that $x \in V'$ but $y \notin V'$
    $E' \leftarrow E' \cup \{e\}$
    $V' \leftarrow V' \cup \{y\}$

Kruskal(V, E) returns (V', E')
  $V' \leftarrow V$
  $E' \leftarrow \{\}$
  while $\#E' < \#V - 1$
    $e \leftarrow$ shortest edge $x \rightarrow y$ in E
    $E \leftarrow E - \{e\}$
    if no path from x to y in E'
      $E' \leftarrow E' \cup \{e\}$

(i) Briefly explain how Kruskal's algorithm may be efficiently implemented using a **Union-Find** data structure to manage disjoint, non-empty sets of nodes.

(ii) Let $n = \#V$ be the number of nodes, and $a = \#E$ be the number of edges, in the graph G. Give asymptotic running times for the two algorithms: justify your answer.

Question 6. Computability and Complexity [12 marks]

(a) Define and explain in plain English the classes

(i) [2 marks] *P*,

(ii) [2 marks] *NP*,

(iii) [2 marks] *NP-Complete*, and

(iv) [2 marks] *NP-Hard*.

(b) You are given two problems A and B, and told that A is NP-complete. How would you:

(i) [2 marks] show that B is *NP-Hard*?

(ii) [2 marks] show that B is *NP-Complete*?

Question 7. Randomised Algorithms [8 marks]

(a) [4 marks] Describe a problem that would be suitable for a Monte Carlo algorithm and explain how randomness will help you make the algorithm more efficient.

(b) [4 marks] Describe a problem that would be suitable for a Las Vegas algorithm and explain how randomness will help you make the algorithm more efficient.

Question 8. Approximation Algorithms [24 marks]

Suppose you are given a set of positive integers $A = a_1, a_2, \ldots, a_n$ and a positive integer B. A subset $S \subseteq A$ is called *feasible* if the sum of the numbers in S does not exceed B:

$$\sum_{a_i \in S} a_i \leq B$$

The sum of the numbers in S will be called the *total sum* of S.

You would like to select a feasible subset S of A whose total sum is as large as possible.

Example. If $A = 8, 2, 4$ and $B = 11$, then the optimal solution is the subset $S = 8, 2$.

(a) [8 marks] Here is an algorithm for this problem.

```
Initially S = ∅
Define T = 0
For i = 1, 2, ..., n
  If T + a_i ≤ B then
    S ← S ∪ a_i
    T ← T + a_i
  Endif
Endfor
```

Give an instance in which the total sum of the set S returned by this algorithm is less than half the total sum of some other feasible subset of A.

(b) [16 marks] Give a polynomial-time approximation algorithm for this problem with the following guarantee:

- It returns a feasible set $S \subseteq A$ whose total sum is at least half as large as the maximum total sum of any feasible set $S' \subseteq A$.

Your algorithm should have a running time of at most $O(n\ logn)$ (note that *at most* means that a running time of $\Theta(n)$ is acceptable).

****