Victoria University of Wellington

DEGREE EXAMINATIONS — 1996

(MID-YEAR)

COMP 303

DESIGN AND ANALYSIS OF ALGORITHMS

Time Allowed: THREE HOURS

Instructions:

- This exam consists of seven (7) questions of varying weights. Answer all questions.
- Read all questions before starting.
- The total marks for each question is given in square brackets "[]". Marks for each part of a question is given in parentheses "()".
- The exam is out of 180 marks so allocate about 1 minute for each mark.
- *Show all working details.* Credit will be given for *legible* but incomplete attempts, provided intent is clear.
- When you are asked to describe an algorithm, it is assumed that it is to be the most efficient (for time) algorithm possible, unless otherwise stated.

1. [30 marks]

(a) Consider the following definitions for asymptotic notation.

$$\Omega(g(n)) = \left\{ \begin{array}{l} h(n) | \exists c \text{ (positive real)}, n_0 \text{ (positive integer) such that,} \\ \forall n \geq n_0,\ 0 \leq c.g(n) \leq h(n) \end{array} \right\}$$

$$O(g(n)) = \left\{ \begin{array}{l} h(n) | \exists c \text{ (positive real)}, n_0 \text{ (positive integer) such that,} \\ \forall n \geq n_0,\ 0 \leq h(n) \leq c.g(n) \end{array} \right\}$$

$$\Theta(g(n)) = \left\{ \begin{array}{l} h(n) | \exists c_1, c_2 \text{ (positive reals)}, n_0 \text{ (positive integer) such that,} \\ \forall n \geq n_0,\ 0 \leq c_1.g(n) \leq h(n) \leq c_2.g(n) \end{array} \right\}$$

Explain why it is necessary to have all three definitions. (6 marks)

SOLUTION: *I'm looking for something to do with upper and lower bounds, and whether or not they match.*

(b) Consider the recurrence:

$$T(n) \leq \begin{cases} cn, & 0 \leq n < 50 \\ T(\lfloor \frac{n}{5} \rfloor) + T(\lfloor \frac{3n}{4} \rfloor) + cn, & n \geq 50 \end{cases}$$

i. Prove by mathematical induction that $T(n) \leq 20cn$. (8 marks)

SOLUTION:

Base case *Consider any $0 \leq n < 50$. Then $T(n) = cn \leq 20cn$.*

Induction Hypothesis *Assume that for $50 \leq n \leq k$, $T(n) \leq 20cn$.*

Induction Step *Consider the case $n = k + 1$. Then $\lfloor \frac{k+1}{5} \rfloor \leq k$ and $\lfloor \frac{3(k+1)}{4} \rfloor \leq k$ (this really should be proved but requires only straight-forward algebra.)*

$$\begin{array}{rcll} T(k+1) & = & T(\lfloor \frac{k+1}{5} \rfloor) + T(\lfloor \frac{3(k+1)}{4} \rfloor) + c(k+1) & \textit{By definition} \\ & \leq & 20c\lfloor \frac{k+1}{5} \rfloor + 20c\lfloor \frac{3(k+1)}{4} \rfloor + c(k+1) & \textit{By the induction hypothesis} \\ & \leq & 20c\frac{k+1}{5} + 20c\frac{3(k+1)}{4} + c(k+1) & \lfloor x \rfloor \leq x \\ & \leq & 20c(k+1) & \textit{as required.} \end{array}$$

ii. Prove that $T(n) \in O(n)$ using only the definition of "O" given above (that is, find appropriate constants and demonstrate that the appropriate relationships exist).

[Note: You may use part 1b(i) without having answered it.] (6 marks)

SOLUTION:

(c) A function $f(n)$ is *asymptotically non-decreasing* if there is a constant n_0 such that for every $n > n_0$, $f(n) \leq f(n+1)$.

Prove that, for any asymptotically non-decreasing function $f(n)$,

$$\lceil f(n) \rceil \in \Theta(f(n)).$$

[Hint: $f(n) \leq \lceil f(n) \rceil \leq f(n) + 1$.] (10 marks)

SOLUTION: *By the property of ceilings,* $f(n) \leq \lceil f(n) \rceil \leq f(n) + 1$.

Since $f(n)$ *is asymptotically non-decreasing there is a* n_{nd} *such that for every* $n > n_{nd}$, $f(n) \leq f(n+1)$.

Suppose there is a $n_1 \geq n_{nd}$ *such that* $f(n_1) \geq 1$. *Then* $\forall n \geq n_1$, $f(n) \geq 1$ *(since* f *is non-decreasing, once its value is larger than 1, it will remain larger than 1), that is* $\lceil f(n) \rceil \leq 2f(n)$. *So, for this case, choose* $n_0 = n_1$, $c_1 = 1$, *and* $c_2 = 2$.

Suppose there is no such n_1, *that is,* $\forall n \geq n_{nd}$, $f(n) \leq 1$. *Then* $\lceil f(n) \rceil = 1$. *Let* $c = \frac{1}{f(n_{nd})}$. *Because* f *is non-decreasing after* n_{nd}, $\forall n \geq n_{nd}$, $c \geq \frac{1}{f(n)}$, *and so* $\lceil f(n) \rceil = 1 \leq cf(n)$. *So, for this case, choose* $n_0 = n_{nd}$, $c_1 = 1$, *and* $c_2 = c$.

2. [30 marks]

(a) Sketch the basic structure of a general divide-and-conquer algorithm. (5 marks)

SOLUTION:

if input is small then
    solve directly
else
    divide into roughly equal subproblems
    recursively solve each subproblem
    combine the solutions to the subproblems

(b) Divide-and-conquer algorithms deal with subproblems. What *two* properties must these subproblems have for a divide-and-conquer algorithm to be efficient? (4 marks)

SOLUTION: *Subproblems must be independent and about equal size. Will also accept subproblems being a constant fraction of the original problem.*

(c) Give a *recurrence* that describes the running time of a typical divide-and-conquer algorithm. You must explain how each part of your recurrence refers to a divide-and-conquer algorithm. (6 marks)

SOLUTION:

$t_D(n)$	*time for solving by direct method*
$t_d(n)$	*overhead of divide phase*
$t_c(n)$	*overhead of combine phase*
k	*number of subproblems*
c	*size of each subproblem*
n_0	*"small"*

Time of a divide and conquer algorithm for problems of size n *is:*

$$T(n) \leq \begin{cases} t_D(n) & n \leq n_0 \\ kT(c) + t_d(n) + t_c(n) & n > n_0 \end{cases}$$

(d) Consider the following algorithm:

Quicksort(*list*, *start*, *end*) **returns** (*list*)
  if *start* < *end* **then**
    middle ← **Partition**(*list*, *start*, *end*)
    Quicksort(*list*, *start*, *middle*)
    Quicksort(*list*, *middle* + 1, *end*)
  endif

i. Give an informal argument that, for any n, there is an input of size n such that **Quicksort** takes time $\Omega(n^2)$. State any assumptions you are making about the behaviour of **Partition**. (5 marks)

SOLUTION: *Partition, on an ordered list of size n, will partition into parts of size 1 and $n-1$. So it will be called at least $n-1$ times, each time doing as many key comparisons as the sum of the two parts. This gives at least $1+2+3+\cdots+n-1$ key comparisons in total.*

ii. Prove the following theorem.

Theorem 1 *The worst-case running time for* **Quicksort** *is $\Theta(n^2)$ for a list of size n, assuming that* **Partition** *takes time $O(m)$ on a list of size m.*

[Note: You may use part 2d(i) without having answered it.] (10 marks)

SOLUTION:

- *Each call to* **Partition**, *partitions the list into sublists of size q and $n-q$, that is,*

$$T(n) = T(q) + T(n-q) + \Theta(n)$$

- *we want the worst case where $1 \leq q \leq n-1$, that is,*

$$T(n) = \max_{1 \leq q \leq n-1} (T(q) + T(n-q)) + \Theta(n)$$

- *"guess" that $T(n) \leq cn^2$ so*

$$T(n) \leq \max_{1 \leq q \leq n-1} (cq^2 + c(n-q)^2) + \Theta(n)$$

- *$cq^2 + c(n-q)^2$ is maximised when $q = 1$ or $q = n-1$*

$$\begin{aligned} T(n) &\leq c + c(n-1)^2 + \Theta(n) \\ &\leq cn^2 \text{ (for some } n_0\text{)} \end{aligned}$$

- *(Claim) there is an input of size n that requires $\Omega(n^2)$ running time.*

3. [30 marks]

A thief, known for his ability to steal gemstones, decides to change to art objects. He breaks into an Art Gallery and is once again confronted with more to steal than he can carry. Unlike the jeweler's shop, where all the gemstones were of the same weight (although of different value), the paintings he can steal here are of different weights as well as different value.

(a) Unfortunately, his greedy algorithm (shown below) that worked so well in the jeweler's shop doesn't work.

Greedy: Looking at each painting in decreasing order of value, take it only if it will fit in the bag.

Give a general explanation as to what goes wrong. Your explanation should include an example that demonstrates the problem, however your answer should not be limited to just the example. (5 marks)

SOLUTION: *Limit 7, Big painting W = 7, V = 1000, 7 little paintings, W = 1, V = 200*

(b) Since the greedy algorithm doesn't work in this case, the thief switches to Plan B: a *dynamic programming* algorithm.

Let the maximum capacity of the thief's sack be $\mathcal{M}$ kilogrammes, let V_i, $0 \le i \le n$ be the *value* of the i-th painting (listed in no particular order except that $V_0 = 0$), and let W_i, $0 \le i \le n$ be the *weight* of the i-th painting ($W_0 = 0$). (Assume that the weights of all the paintings are integers (kilogrammes) and their values are also integers (dollars).)

In order to develop an algorithm, the first step is to decide what a *subproblem* is. In this case, a subproblem consists of reducing the number of choices (paintings) available, and reducing the amount the thief is allowed to carry off. So, let c_{iw} be the value of the maximum haul that fits in a sack that can carry only $0 \le w \le \mathcal{M}$ kilogrammes when looking only at paintings $0, 1, 2, \ldots, i$. What we want to do is build up a recursive definition for c_{iw}. (Note that painting 0 is just there to make the algorithm a bit tidier.)

Suppose $W_i > w$. Then, the i-th painting cannot be carried if the thief is limited to only w kilogrammes. So it must be that $c_{iw} = c_{i-1,w}$, that is, the maximum haul when the limit is w does not include the i-th painting.

Now suppose $W_i \le w$. In this case, the i-th painting will fit — but does the thief want to take it? It depends on whether taking it will maximise his haul. So, $c_{iw} = \max(c_{i,w-W_i} + V_i,\ c_{i-1,w})$.

i. Give a *dynamic programming* algorithm for helping the thief solve his problem. If you use pseudo-code, you *must* also include an explanation of how your algorithm works. (10 marks)

SOLUTION:

- *$T[i]$ holds the value of the i-th coin.*
- *$C[i, j]$ will end up with the value c_{ij}.*

Make_Change

```
C[0,0] ← 0
0 < j ≤ V, C[i,j] ← +∞
for i ← 1 to n do
  for j ← 0 to V do
    if j ≥ T[i] then
      if C[i−1,j] > C[i,j−T[i]] + 1 then
        C[i,j] ← C[i,j−T[i]] + 1
      else
        C[i,j] ← C[i−1,j]
      endif
    else
      C[i,j] ← C[i−1,j]
    endif
  end for
end for
```

ii. Explain why your solution is a *dynamic programming* algorithm. (Some credit will be given for listing the properties of dynamic programming algorithms.) (5 marks)

SOLUTION: *Uses a table to looking up solutions to subproblems.*

SOLUTION: *(newpage)*

iii. Prove that your algorithm is correct. [Hint: You should prove it by induction on something like $i(\mathcal{M}+1)+w$, although it will depend on exactly how your algorithm works.] (10 marks)

SOLUTION: *It is important to have a clear idea of what it is that you are proving. The easiest way to do this is to write down what you are proving* in complete detail.

Claim 1 *$C[i,j]$ gets set to the minimum number of coins required to give j change using only coins $1,\ldots,i$, for all $0 \le i \le n$ and $0 \le j \le \mathcal{V}$.*

Proof by induction on $k = i(\mathcal{V}+1)+v$.

Base case *$k \le \mathcal{V}$. In this case, it must be that $i = 0$ and $v \le \mathcal{V}$. Now there are two cases. When $v = 0$, then we only need 0 coins of value 0 to give this change, which is what the initialisation does. When $v > 0$ then no amount of 0-valued coins can give change. This too is correctly handled by the initialisation.*

Induction Hypothesis *Assume the claim is true for $0 \le k < l \le n(\mathcal{V}+1)+\mathcal{V}$, that is, for i and j such that $k = i(\mathcal{V}+1)+j < l$, $C[i,j]$ gets set to the minimum number of coins required to give j change using only coins $1,\ldots,i$.*

Induction Step *Consider the case where $k = l$. In this case it must be that $i = \lfloor \frac{l}{\mathcal{V}+1} \rfloor$ and $j = l - i\lfloor \frac{l}{\mathcal{V}+1} \rfloor$. Because $(i-1)(\mathcal{V}+1)+j < l$ and $i(\mathcal{V}+1)+j-T[i] < l$, by the induction hypothesis, the values $C[i-1,j]$ and $C[i,j-T[i]]$ are correct. So, by the argument I used in question 1, $C[i,j]$ will be set to the correct value.*

Note: The answer to part (a)(i) includes the boundary conditions $c_{i0} = 0$, $0 \le i \le n$. These conditions (unlike the others) do not need to be part of the initialisation because of the nature of the algorithm — since it fills the array row by row, the value $C[i][0]$ is computed before it is needed.

4. [30 marks]

Below is **Dijkstra's algorithm**, for solving the **single-source shortest path** problem (find the shortest path between vertex 1 and every other vertex in a *connected undirected graph*).

$L[i,j] =$ length of edge between vertices i and j
(∞ if no edge exits)

SSSP(L)
$C \leftarrow \{2, 3, \ldots, n\}$
for $i \leftarrow 2$ **to** n **do**
$dist[i] \leftarrow L[1, i]$
end for
repeat $n - 2$ **times**
$v \leftarrow j \in C : dist[j] = \mathbf{min}_{i \in c} dist[i]$
$C \leftarrow C \setminus \{v\}$
for every $i \in C$ **do**
$dist[i] \leftarrow \mathbf{min}(dist[i], dist[v] + L[v, i])$
end for
end repeat

(a) After this algorithm has finished executing, $dist[i]$, $2 \leq i \leq n$ has the *length* of the shortest path between vertex 1 and vertex i. Explain how to modify this algorithm so as to be able to determine the actual path, not just the length of the path. (5 marks)

SOLUTION: *Something like whenever dist is updated, add the newly added node to the end of the path.*

(b) The most efficient implementation of Dijkstra's algorithm gives time $\Theta(e \log_2 n)$, where n is the number of vertices and e is the number of edges in the graph. Explain how to achieve this running time. (8 marks)

SOLUTION: *The important operations are to do with representing $\mathcal{C}$. Those operations are finding the minimum of $\mathcal{C}$, removing elements from $\mathcal{C}$, updating elements in $\mathcal{C}$, and traversing the elements of $\mathcal{C}$.*

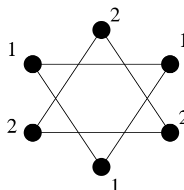
If $\mathcal{C}$ is implemented as a heap, we can do the first three operations in $\log_2 n$ *time (plus $O(n)$ overhead to build the heap).*

However to do the traversal efficiently, you really need an adjacency list *graph representation. Then the* **for every** *statement will take time linear in the number of edges incident on v that are in C (rather than liner in the total number of vertices) — provided there is a fast way to determine if a vertex is in $\mathcal{C}$. This means that an auxiliary array of boolean values for each vertex will be needed as well.*

(c) A *connected component* is defined as follows:

Definition 1 *A* connected component *of an undirected graph* $\mathcal{G} = (V, E)$ *is the largest subgraph* $S = (C, A)$ *(with* $C \subseteq V$ *and* $A \subseteq E$*) such that every vertex in* C *is reachable via a path of edges in* A *by every other vertex.*

i. Explain how to use Dijkstra's algorithm to label all the connected components in an *undirected* (but possible disconnected) graph with different values. Your algorithm should produce a labelling like that shown on the example below. (10 marks)



SOLUTION: *Modify Dijkstra to take a "component label" as an argument. Call it repeatedly on any unlabelled vertices, changing the label between each call, until they are all labelled.*

ii. What is the running time of your algorithm? You must explain any implementation details needed. You may use part 4b even if you haven't answered it. (7 marks)

SOLUTION: *The running time due to Dijkstra won't have changed (still depends on the number of edges. The real trick is keeping track of what's labelled and what isn't so that the next call to Dijkstra's can be done quickly.*

SOLUTION: *(newpage)*

5. [30 marks]

A *vertex cover* of a graph $\mathcal{G} = (V, E)$ is a set $V' \subseteq V$ such that for every edge $(u, v) \in E$, either $u \in V'$ or $v \in V'$, that is, one end of every edge is in V'.

(a) Design a linear-time *greedy* algorithm for finding the **minimal** vertex cover of a *binary tree*. [Hint: Your algorithm will need to check every vertex in the tree.] (10 marks)

SOLUTION: *In the following, assume that if a child does not exist then it is considered in the cover set.*

My algorithm is to have an array cover*:array[1..n] of* **boolean** *initialised to* **false**. *I then do a postorder traversal of the tree and whenever I visit a node, I check to see if both children are in the cover set (by checking* cover*). If at least one of the children is not in the cover set, then include the current vertex in the cover set by setting its* cover *value to* **true**. *(At last! An interesting use for postorder traversal.) The cover set can be read out of* cover.

(b) Explain why your algorithm is linear-time. (5 marks)

SOLUTION: *We know that postorder traversal takes time linear in the number of vertices and the use of* cover *means that any other operations only take constant time.*

SOLUTION: *(newpage)*

(c) Prove that your algorithm finds the minimal vertex cover for a binary tree. (15 marks)

SOLUTION: *Let* $V = \{v_1, v_2, \ldots, v_n\}$ *be the vertices listed in postorder order. Let* $\mathcal{C} = \{v_{i_1}, v_{i_2}, \ldots, v_{i_s}\}$ *be the cover produced by the algorithm, where the vertices are listed in the same order as in* V *(that is,* $i_j < i_{j+1}, \forall j$*).*

Now assume that the cover produced by the algorithm is not minimal. Let $\mathcal{C}' = \{v_{j_1}, v_{j_2}, \ldots, v_{j_t}\}$ *be a minimal cover that differs first a vertex* v_k*, that is,* $i_l = j_l$ *for any* $i_l < k$*. Choose* $\mathcal{C}'$ *to* maximise k*. Note that if* $k = n$ *then* $\mathcal{C} = \mathcal{C}'$ *so, by assumption,* $k < n$*.*

There are two cases: either v_k *is in* $\mathcal{C}$ *or it is in* $\mathcal{C}'$*.*

$v_k \in \mathcal{C}$ *That is,* $v_k = v_{i_l}$ *for some* l*.*

By the way the algorithm operates, this can only be if one of v_k*'s children is not in* $\mathcal{C}$*. Let that child be* x*. Since* x *occurs earlier in the postorder traversal than* v_k*, if* $x \notin \mathcal{C}$ *then* $x \notin \mathcal{C}'$ *otherwise that would contradict the choice of* k*. Since* $v_{i_l} \in \mathcal{C}$*, it must be that* $v_{i_l} \notin \mathcal{C}$ *by the choice of* k *so that means that the edge* (x, v_{i_l}) *is not covered by* $\mathcal{C}'$*. This contradicts that fact that* $\mathcal{C}'$ *is a vertex cover.*

$v_k \in \mathcal{C}'$ *That is,* $v_k = v_{j_m}$ *for some* m*.*

Since $v_{j_m} \in \mathcal{C}'$*, it must be that* $v_{j_m} \notin \mathcal{C}$ *and so, by the way the algorithm operates, both of* v_{j_m}*'s children must be in* $\mathcal{C}$*. By an argument similar to the above case, that means that those children are in* $\mathcal{C}'$ *as well.*

If v_{j_m} *is the root of the tree, then, because both its children are in* $\mathcal{C}'$*, those edges are covered, that is,* v_{j_m} *does not need to be in the cover. This contradicts the fact that* $\mathcal{C}'$ *is a minimal cover.*

So assume v_{j_m} *is not the root of the tree. Let* $\mathcal{C}'' = \mathcal{C}' \setminus \{v_{j_m}\} \cup \{\text{parent of } v_{j_m}\}$*. Then* $\mathcal{C}''$ *is still a cover but it differs from* $\mathcal{C}$ *at position* $v_{k'}$*, for* $k' > k$ *(because the parent of* v_{j_m} *appears in the postorder listing after* v_k*. This contradicts the fact that* $\mathcal{C}'$ *was chosen to maximise* k*.*

6. [15 marks]

Suppose you had n magnetic tapes containing $m_1, m_2, \ldots, m_n$ records on each tape in sorted order and you want to create a master tape that contains the contents of these tapes and is also in sorted order. Your system allows you to merge the records on two tapes containing p and q records respectively with $p + q$ record movements but you can only perform this operation on two tapes at a time to produce a third tape containing both sets of records (i.e., the primitive operation is *2-file merge*).

(a) Design an algorithm to determine the order to merge the n tapes that will **minimise** the number of record movements required to actually merge the tapes. You may assume as many extra tapes as you need. [Hint: How would you represent a series of tape-merges pictorially?] (8 marks)

SOLUTION: *This is basically* `min-wepl` *from class.*

(b) What is the asymptotic running time for your algorithm? Informally justify your answer. This must include any relevant implementation details. (7 marks)

SOLUTION: *They have to describe how to maintain the set of merged tapes so that finding the smallest tape and adding a new tape is efficient. The answer is* $\Theta(e \log_2 n)$

7. [15 marks]

This question concerns the following two decision problems.

Partition

INSTANCE A finite set A and a size $s(a)$ for each $a \in A$.
QUESTION Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A'} s(a) = \sum_{a \notin A'} s(a)?$$

Knapsack

INSTANCE A finite set U, a size $s(u)$ and a value $v(u)$ for each $u \in U$, a size constraint B, and a value goal K.
QUESTION Is there a subset $U' \subseteq U$ such that

$$\sum_{u \in U'} s(u) \leq B \text{ and } \sum_{u \in U'} v(u) \geq K?$$

Show that **Knapsack** is $\mathcal{NP}$-complete assuming that **Partition** is $\mathcal{NP}$-complete. You must give complete details. You will find the following theorem useful (explain how you use it).

Theorem 2 *If* $L_1 \leq_M^P L_2$, L_1 *is* $\mathcal{NP}$*-complete and* $L_2 \in \mathcal{NP}$ *then* L_2 *is* $\mathcal{NP}$*-complete*

SOLUTION: *They have to (i) show Knapsack is in NP, (ii) define a function* f *that transforms instances of Partition into instances of Knapsack, and (iii) show that an instance* I *of Partition is a "yes" instance if and only if* $f(I)$ *is a "yes" instance of Knapsack.*

Given an instance of Partition, define $v(a) = s(a)$ *and choose* $B = K = 1/2 \sum s(a)$

****