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

EXAMINATIONS – 2015

TRIMESTER 2

COMP 361

DESIGN AND ANALYSIS OF ALGORITHMS

Time Allowed: TWO HOURS

CLOSED BOOK

Permitted materials: Only silent non-programmable calculators or silent programmable calculators with their memories cleared are permitted in this examination.

Non-electronic foreign language to English dictionaries are permitted.

Instructions: The exam will be marked out of 120 marks.

Attempt ALL questions.

Questions	Marks
1. Divide and Conquer	[20]
2. Various Algorithms	[20]
3. Dynamic Programming	[30]
4. Approximation Algorithms	[20]
5. Chips but No Fish	[30]

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

Asymptotic notation:

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

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 $(\exists c < 1)(\mathbf{aa}\, n)[a.f(n/b) \le c.f(n)]$ 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. Divide and Conquer [20 marks]

(a) [10 marks] Use the master method to give tight asymptotic bounds for the following recurrences:

1. $T(n) = 2T(n/4) + 1$ Case 1: alpha is 0.5 so O(sqrt n)
2. $T(n) = 2T(n/4) + \sqrt{n}$ Case 2: same alpha so O(sqrt n log n)

(b) [10 marks] Describe a typical structure of a Divide and Conquer algorithm and outline a typical approach to proving it correct as was described in the lectures and was used to prove the correctness of merge sort etc.

See lectures on D and C including slide on the outline and slide on proof pattern.

Question 2. Various Algorithms [20 marks]

(a) [10 marks] Pick your favourite lock-free data structure and describe how it works despite the lack of locks etc. Use diagrams as appropriate.

See description in CACM article and make sure to include CAS operation and what it does.

(b) [5 marks] Draw two Venn Diagrams (showing how sets overlap or intersect) representing the sets of P, NP, NP-Hard, and NPC problems. One diagram should assume $P = NP$ and the other diagram should assume $P \neq NP$.

See wikipedia page on this.

(c) [5 marks] What is the difference between Monte Carlo and Las Vegas probabilistic algorithms?

See relevant lecture slide.

Question 3. Dynamic Programming [30 marks]

Suppose you're running a lightweight consulting business — just you, two associates, and some rented equipment. Your clients are distributed between Hawkes Bay and New Plymouth, and this leads to the following question.

Each month, you can either run your business from an office in Hawkes Bay (HB) or from an office in New Plymouth (NP). In month i, you'll incur an *operating cost* of H_i if you run the business out of HB; you'll incur an operating cost of N_i if you run the business out of NP. The costs depends on the distribution of client demands for that month.

However, if you run the business out of one city in month i, and then out of the other city in month $i+1$, then you incur a fixed *moving cost* of M to switch base offices.

Given a sequence of n months, a *plan* is a sequence of n locations — each one equal to either HB or NP — such that the i^{th} location indicates the city in which you will be based in the i^{th} month. The *cost* of a plan is the sum of the operating costs for each of the n months, plus a moving cost of M for each time you switch cities. The plan can begin in either city.

The problem. Given a value for the moving cost M, and sequences of operating costs $H_1, \ldots, H_n$ and $N_1, \ldots, N_n$, find a plan of minimum cost. (Such a plan will be called *optimal*.)

Example. Suppose $n = 4$, $M = 10$, and the operating costs are given by the following table.

	Month 1	Month 2	Month 3	Month 4
H_i	1	3	20	30
N_i	50	20	2	4

Then the plan of minimum cost would be the sequence of locations $[HB, HB, NP, NP]$, with a total cost of $1 + 3 + 2 + 4 + 10 = 20$, where the final term of 10 is the moving cost of changing locations once.

(a) [5 marks] Show that the following algorithm does not correctly solve this problem, by giving an instance on which it does not return the correct answer. *Assume that $n = 4$ and $M = 10$ as it was in the example above.*

Give the optimal solution and its cost, as well as what the algorithm finds.

```
for i = 1 to n
    if Hi < Ni then
        output "HB in Month i"
    else
        output "NP in Month i"
end
```

(b) [5 marks] Give an example of an instance (again with $n = 4$ and $M = 10$) in which the optimal plan must move (i.e., change locations) at least three times. *Provide a brief explanation, saying why your example has this property.*

(c) [20 marks] Give a pseudocode for an efficient algorithm that takes values for n, M, and sequences of operating costs $H_1, \ldots, H_n$ and $N_1, \ldots, N_n$, and returns the *cost* of an optimal plan.

Hint: What is the table of intermediate results with optimal substructure property?

Replace the SF and NY with NP and HB.

(a) Suppose that $M = 10$, $\{N_1, N_2, N_3\} = \{1, 4, 1\}$, and $\{S_1, S_2, S_3\} = \{20, 1, 20\}$. Then the optimal plan would be $[NY, NY, NY]$, while this greedy algorithm would return $[NY, SF, NY]$.

(b) Suppose that $M = 10$, $\{N_1, N_2, N_3, N_4\} = \{1, 100, 1, 100\}$, and $\{S_1, S_2, S_3, S_4\} = \{100, 1, 100, 1\}$.

Explanation: The plan $[NY, SF, NY, SF]$ has cost 34, and it moves three times. Any other plan pays at least 100, and so is not optimal.

(c) The basic observation is: The optimal plan either ends in NY, or in SF. If it ends in NY, it will pay N_n plus one of the following two quantities:

- The cost of the optimal plan on $n-1$ months, ending in NY, or
- The cost of the optimal plan on $n-1$ months, ending in SF, plus a moving cost of M.

An analogous observation holds if the optimal plan ends in SF. Thus, if $OPT_N(j)$ denotes the minimum cost of a plan on months $1, \ldots, j$ ending in NY, and $OPT_S(j)$ denotes the minimum cost of a plan on months $1, \ldots, j$ ending in SF, then

$$OPT_N(n) = N_n + \min(OPT_N(n-1), M + OPT_S(n-1))$$

$$OPT_S(n) = S_n + \min(OPT_S(n-1), M + OPT_N(n-1))$$

This can be translated directly into an algorithm:

$OPT_N(0) = OPT_S(0) = 0$
`For` $i = 1, \ldots, n$
  $OPT_N(i) = N_i + \min(OPT_N(i-1), M + OPT_S(i-1))$
  $OPT_S(i) = S_i + \min(OPT_S(i-1), M + OPT_N(i-1))$
`End`
`Return the smaller of` $OPT_N(n)$ `and` $OPT_S(n)$

The algorithm has n iterations, and each takes constant time. Thus the running time is $O(n)$.

[1] ex786.93.190

Question 4. Approximation Algorithms [20 marks]

Suppose you are acting as a consultant for the Port Authority of a small Pacific Rim nation. Their revenue is constrained almost entirely by the rate at which they can unload ships that arrive in the port.

Here is a basic sort of problem they face. A ship arrives, with n containers of weight $w_1, \ldots, w_n$. Standing on the dock is a set of trucks, each of which can hold K units of weight. (You can assume that K and each w_i is an integer.) You can stack multiple containers in each truck, subject to the weight restriction of K; the goal is to minimise the number of trucks that are needed in order to carry all the containers. This problem is NP-complete (you don't have to prove this).

A greedy algorithm you might use for this is the following. Start with an empty truck, and begin piling containers $1, 2, 3, \ldots$ into it until you get to a container that would overflow the weight limit. Now declare this truck "loaded" and send it off; then continue the process with a fresh truck. This algorithm, by considering trucks one at a time, may not achieve the most efficient way to pack the full set of containers into an available collection of trucks.

(a) [5 marks] Give an example of a set of weights, and a value of K, where this algorithm does not use the minimum possible number of trucks.

(b) [15 marks] Shown that the number of trucks used by this algorithm is within a factor of 2 of the minimum possible number, for any set of weights and any value of K.

(a) Let $\{w_1, w_2, w_3\} = \{1, 2, 1\}$, and $K = 2$. Then the greedy algorithm here will use three trucks, whereas there is a way to use just two.

(b) Let $W = \sum_i w_i$. Note that in *any* solution, each truck holds at most K units of weight, so W/K is a lower bound on the number of trucks needed.

Suppose the number of trucks used by our greedy algorithm is an odd number $m = 2q+1$. (The case when m is even is essentially the same, but a little easier.) Divide the trucks used into consecutive groups of two, for a total of $q + 1$ groups. In each group but the last, the total weight of containers must be *strictly* greater than K (else, the second truck in the group would not have been started then) — thus, $W > qK$, and so $W/K > q$. It follows by our argument above that the optimum solution uses at least $q + 1$ trucks, which is within a factor of 2 of $m = 2q + 1$.

[1] ex667.592.236

Question 5. Chips but No Fish [30 marks]

(a) Professor Dale has n supposedly identical integrated-circuit chips that in principle are capable of testing each other. The professor's test jig accommodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the professor cannot trust the answer of a bad chip. Thus, the four possible outcomes of a test are as follows:

Chip A says	Chip B says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

(i) [10 marks] Show that if you know that *all the chips* potentially can be bad (but you do not know how many are good or bad), no matter what algorithm the professor will come up with that claims to correctly identify the good chips, there is going to be a way to get the bad chips to conspire to behave in such a way that the algorithm would not work (i.e. not be able to correctly identify the good chips). Explain clearly how you can make the chips behave to defeat any such algorithm.

(ii) [10 marks] Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good (or in other words definitely less than half of the chips are bad in any given input). Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of less than half the size.

Hint 1: If you put two chips on the jig and the result says that one of them is bad, what would happen if you throw both of the chips away?

Hint 2: If you have a set of pairs of chips each of which is either *both good* or *both bad* (you don't necessarily know which though), and you know that the majority of such pairs are *both good*, then how can you come up with a *smaller* set of chips with the same property of having majority being *good*?

You should be able to use the two hints above to come up with an algorithm.

(iii) [10 marks] Show that the good chips can be identified with $\Theta(n)$ pairwise tests, assuming that more than $n/2$ of the chips are good. Give and solve the recurrence that describes the number of tests.

Problem 4.5

Chip testing

Professor Diogenes has n supposedly identical integrated-circuit chips that in principle are capable of testing each other. The professor's test jig accomodates two chips at a time. When the jig is loaded, each chip tests the other and reports whether it is good or bad. A good chip always reports accurately whether the other chip is good or bad, but the professor cannot trust the answer of a bad chip. Thus, the four possible outcomes of a test are as follows.

Chip A says	Chip B says	Conclusion
B is good	A is good	both are good, or both are bad
B is good	A is bad	at least one is bad
B is bad	A is good	at least one is bad
B is bad	A is bad	at least one is bad

1. Show that if more than $n/2$ chips are bad, the professor cannot necessarily determine which chips are good using any strategy based on this kind of pairwise test. Assume that the bad chips can conspire to fool the professor.
2. Consider the problem of finding a single good chip from among n chips, assuming that more than $n/2$ of the chips are good. Show that $\lfloor n/2 \rfloor$ pairwise tests are sufficient to reduce the problem to one of nearly half the size.
3. Show that the good chips can be identified with $\Theta(n)$ pairwise tests, assuming that more than $n/2$ chips are good. Give and solve the recurrence that describes the number of tests.

If more than half are bad

Lets say that there are $g < n/2$ good chips. The same amount of the remaining bad chips can choose to act similar to good chips. That is, they can identify each other as good and all other as faulty. Since this is what the good chips would do, both groups are symmetric in regards to the operation of parwise comparison. No strategy can distinguish between the two groups.

Finding a single good chip in logarithmic time

We split the chips in groups of two and compare them. We can take one of the chips if the outcome is the first one (both are good or both are bad) and but both away otherwise. When putting away, we're removing at least one bad chip for every good one we remove. Out of the pairs we've chosen a chip from, there would be more good chips than bad chips (there would be more good pairs, because the good chips are more than the half). Now we have at most $n/2$ chips, where at least half of them are good.

Finding the good chips

The recurrence for finding at least one good chip is:

$$T(n) = T(n/2) + n/2$$

By the master method, this is $\Theta(n)$. After we've found one, we can compare it will all others, which is a $\Theta(n)$ operation.

Python code

* * * * * * * * * * * * * * * *