

## Degree Examination

## MX4028 / MX4528 Algorithms

Tuesday 30 May 2000

(3pm to 5pm)

Answer *THREE* questions

Calculators may be used *ONLY* for the arithmetic of real numbers or the numerical evaluation of trigonometric, logarithmic and exponential functions. Calculator memories must be clear at the start of the examination; in particular, the use of pre-stored programs is prohibited. Marks may be deducted for answers that do not show clearly how the solution is reached.

1. Write an efficient algorithm, using a pseudocode of your choice, to find the minimum element (**min**) and the second smallest element (**second**) of a given list of integers.

The algorithm is to be run on an abstract computer on which each basic operation (assignment, comparison between two numbers or incrementing a counter) takes one unit of time. Show that the time  $T(n)$  for the algorithm to run on a list of length  $n$  satisfies  $T(n) = O(n)$ .

Explain what assumptions you would make to get the expected value of this running time. Illustrate your answer by doing explicit calculations in the case where  $n = 4$ . [You are *not* required to compute the expected running time in general.]

2. Describe a *Huffman* code and give examples of circumstances when it (or a variant) might be used. Illustrate your answer by constructing a binary Huffman code for the string

A HUFFMAN EXAMPLE PLEASE

As well as deriving the coding tree, you should give your code for the word PLEASE. Note that the string contains 12 distinct symbols, include the “space” symbol, and 24 symbols in all.

An alphabet which contains  $2^n$  letters and their associated probabilities is given and the corresponding symbols are derived using the Huffman construction. How *short* can the *longest* of these symbols be? What property of the corresponding probabilities would ensure that this minimum length is attained. How *long* can the *longest* symbol be? Give restrictions on the corresponding probabilities to ensure that this maximum length occurs.

3. Consider the formal language  $\mathcal{L}$  given by the following grammar.

- The alphabet is  $A = \{a, b, c\}$ .
- The abstract alphabet is  $\mathcal{A} = \{\alpha, \beta, \gamma\}$ .
- The initial symbol is  $\alpha$ .
- The productions are
  1.  $\alpha \rightarrow a\alpha\beta\gamma$ ;
  2.  $\alpha \rightarrow a\beta\gamma$ ;
  3.  $\gamma\beta \rightarrow \beta\gamma$ ;
  4.  $a\beta \rightarrow ab$ ;
  5.  $b\beta \rightarrow bb$ ;
  6.  $b\gamma \rightarrow bc$ ;
  7.  $c\gamma \rightarrow cc$ .

(a) Prove that the string  $a^2b^2c^2$  is in  $\mathcal{L}$ .

(b) Prove that at every stage of the process, no member of the abstract alphabet appears to the left of any member of  $A$ .

(c) Prove that production (2) is always applied exactly once, and that after it has been applied the resulting string is of the form  $a^{n+1}\sigma$ , where  $n \geq 0$  and  $\sigma$  contains exactly  $n + 1$  copies of both  $\beta$  and  $\gamma$ , ordered in such a way that up to any point in  $\sigma$ , there are at least as many copies of  $\beta$  as of  $\gamma$ .

(d) Give a simple description of the strings in  $\mathcal{L}$  and give arguments to support your claim. [A formal proof is not expected.]

4. In this question, you are given a random number generator  $\text{RAND}()$  which produces random reals uniformly distributed in  $[0, 1)$ .

(a) You are required to choose  $k$  objects “at random” from a sequence of  $N$  objects — perhaps passing on a conveyor belt. You must decide whether a given object is to be selected *before* the next one becomes available. Give pseudocode for such an algorithm, and justify the assertion that it will produce exactly  $k$  items. [You are *not* asked to justify that the sample is a random sample.]

(b) Let  $T$  be the triangle with vertices at  $(0, 2)$ ,  $(-1, 0)$  and  $(1, 0)$ . Describe an algorithm to generate a sequence of points  $\{p_n\}$  in  $T$  “at random”. Your answer should produce an additional point  $p_n$  for each pair of calls to  $\text{RAND}()$ . Explain briefly why your algorithm does what is required.

1. Here is pseudocode for the minimum and second minimum element algorithm. I don't think it is in the spirit of this question to check that the list has at least three elements etc!

```

algorithm min_and_second(x)
  // To find the smallest and second smallest of x = (x(1)...x(n))
  begin
    if (x(1) < x(2)) then
      min = x(1)
      second = x(2)
    else
      min = x(2)
      second = x(1)
    endif
    for i = 3 to n begin
      if (x(i) < min) then
        second = min
        min = x(i)
      else if (x(i) < second) then
        second = x(i)
      endif
    else
      // do nothing
    endif
  end
  return (min AND second)
end

```

Let  $T(n)$  be the time taken to execute on a list of length  $n$ . The setup phase takes 1 test and 2 assignments. In the main loop, executed  $(n - 2)$  times, there is an initial assignment of  $i$ , a final test to see if  $(i == n)$  and either

- two tests and no assignments; or
- two tests and one assignment; or
- one test and two assignments.

giving either 4 or 5 operations in a given iteration. We ignore the cost of the return statement. Thus the total operation count satisfies

$$3 + 4(n - 2) \leq T(n) \leq 3 + 5(n - 2) \quad \text{or} \quad 4n - 5 \leq T(n) \leq 5n - 7.$$

In either case, and hence in general, the algorithm is  $O(n)$ .

We shall refer to either of the second two cases as involving a *detour*. Our aim is to find the “average” number of detours  $d$ ; since  $T(n) = 4n - 5 + d$ , this will give the “average” time to complete the algorithm.

Note that the actual numbers in the list are irrelevant. All that matters is their relative ordering. Thus we assume that we are dealing with lists of  $n$  distinct elements (we ignore the possibility that the lists with repeated elements). Then we may as well assume that the lists that we are dealing with are all the permutations of the list  $(1, 2, 3, 4, \dots, n)$ . In addition assume that *all* these permutations are *equally likely* to be presented to the algorithm. Thus we seek the average number of detours executed when applying the algorithm to the permutations of  $(1, 2, 3, \dots, n)$ , granted that all such permutations are equally probable.

We now compute the number of detours that occur when applying the algorithm to each permutation in  $S_4$ . Note that a detour occurs precisely when we encounter a new minimum element in either third or fourth place in the original list. The computation is given in Table 1 We see there are 28 detours in all so the expected number of detours is  $7/6$  and the average

$\pi$	$d$	$\pi$	$d$	$\pi$	$d$	$\pi$	$d$
(1,2,3,4)	0	(2,1,3,4)	0	(3,1,2,4)	1	(4,1,2,3)	1
(1,2,4,3)	0	(2,1,4,3)	0	(3,1,4,2)	1	(4,1,3,2)	2
(1,3,2,4)	1	(2,3,1,4)	1	(3,2,1,4)	1	(4,2,1,3)	1
(1,3,4,2)	1	(2,3,4,1)	1	(3,2,4,1)	1	(4,2,3,1)	2
(1,4,2,3)	1	(2,4,1,3)	1	(3,4,1,2)	2	(4,3,1,2)	2
(1,4,3,2)	2	(2,4,3,1)	2	(3,4,2,1)	2	(4,3,2,1)	2

Table 1: Counting detours for the elements of  $S_4$

time to sort a list of length 4 is  $13\frac{1}{3}$  units.

2. A Huffman code is used when the input is from a limited character set, and contains information that is redundant; a typical example is English text, in which the letter “e” occurs significantly more often than random. One such application is in the “gzip” utility which is based on the Huffman code.

We describe how to build the binary Huffman code by building the corresponding binary tree. We start by analysing the message to find the frequencies of each symbol that occurs in it. Our basic strategy will be to assign short codes to symbols that occur frequently, while still insisting that the code has the prefix property. Our example will be build around the message

A HUFFMAN EXAMPLE PLEASE

The corresponding frequencies are given in Table 1; we write the space symbol “ ”, in the table as  $\sqcup$ .

A	$\sqcup$	H	U	F	M	N	E	X	P	L	S
4	3	1	1	2	2	1	4	1	2	2	1

Table 2: Symbol frequencies used to build a Huffman Code.

Now begin with a collection (a forest) of very simple trees, one for each symbol to be coded, with each consisting of a single node, labelled by that symbol, and the frequency with which it

occurs in the string. The construction is recursive: at each stage the two trees which account for the least total frequency in their root nodes are selected, and used to produce a new binary tree. This has, as its children the two trees just chosen: the root is then labelled with the total frequency accounted for by both subtrees, and the original subtrees are removed from the forest. The construction continues in this way until only one tree remains; that is then the Huffman encoding tree.

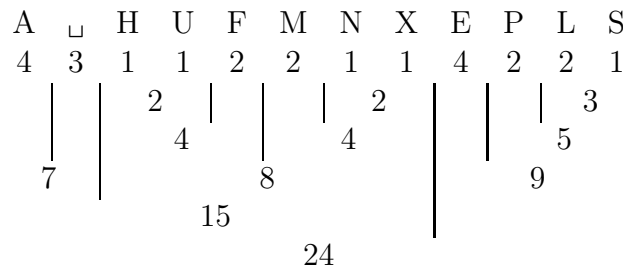


Table 3: Symbol frequencies used to build a Huffman Code.

[NB The particular order in which the symbols are displayed happens to give a table with no crossings when listed in the order in which the symbols are gathered. It is not necessary to give a presentation in this form; crossing are perfectly acceptable.]

We now code the word PLEASE using this coding tree. Although this is a prefix code and so spaces are *not* required, we include spaces between letters for clarity. We read up from the bottom of the table, assigning 0 to a choice of a left hand branch, and 1 to a choice of the right hand branch. The resulting code is then

$$110 \square 1110 \square 10 \square 000 \square 1111 \square 10$$

A binary tree with  $2^n$  nodes must have at least  $n$  levels, so the longest string must have at least  $n$  symbols. This bound is actually attained if the tree is a complete binary tree; one circumstance in which this will occur is if all the letters in the initial alphabet are equally probable.

To get the longest possible string, the tree must be completely unbalanced, with (for example) every left node being a leaf node, and with all growth taking place at the right. This means that apart from the final level, we have one symbol of each length, and so the longest symbol must have length  $2^n - 1$ . If the corresponding probabilities are  $p_0 \geq \dots \geq p_k$ , where  $k = 2^n$ , then to ensure the tree builds in the way specified, we must have

$$p_i \geq \sum_{j=i+1}^k p_j$$

so the probability of a given symbol must be at least as great as the sum of the probabilities of all the less likely symbols.

3. (a) Here is the initial derivation.

$$\alpha \xrightarrow{(1)} a\alpha\beta\gamma \xrightarrow{(2)} a^2\beta\gamma\beta\gamma \xrightarrow{(3)} a^2\beta\beta\gamma\gamma \xrightarrow{(4)} a^2b\beta\gamma\gamma \xrightarrow{(5)} a^2b^2\gamma\gamma \xrightarrow{(6)} a^2b^2c\gamma \xrightarrow{(7)} a^2b^2c^2$$

and so  $a^2b^2c^2$  is in  $\mathcal{L}$ .

- (b) It is trivial that for the initial symbol  $\alpha$ , our hypothesis  $H$  that no member of the abstract alphabet appears to the left of any member of  $A$ , is true. Assume inductively that  $H$  is true

for the first  $n$  steps in a derivation, and note that none of the 7 productions given destroys  $H$ ; it thus remains true after  $n + 1$  steps and the general result follows by induction.

(c) There is at most one  $\alpha$  in the string, and this always lies at the abstract concrete boundary until production (2) is used, at which point neither (1) nor (2) are relevant. So our initial productions are necessarily drawn from (1) and (3) until (2) is used, at which point, since each application of (1) introduces both an additional  $\alpha$  and an additional  $\beta\gamma$ , there are exactly  $n + 1$  each of  $a$ ,  $\beta$  and  $\gamma$ . The commutativity relation (3) can only move  $\beta$  to the left, so the result follows.

(d) We claim that  $\mathcal{L} = \{a^{n+1}b^{n+1}c^{n+1} \mid n \geq 0\}$ . One way is clear, since the following shows all the claimed strings lie in  $\mathcal{L}$ .

$$\begin{aligned} \alpha &\xrightarrow{(1)} a^n \alpha (\beta\gamma)^n \xrightarrow{(2)} a^{n+1} (\beta\gamma)^{n+1} \xrightarrow{(3)} a^{n+1} \beta^{n+1} \gamma^{n+1} \xrightarrow{(4)} a^{n+1} b \beta^n \gamma^{n+1} \\ &\xrightarrow{(5)} a^{n+1} b^{n+1} \gamma^{n+1} \xrightarrow{(6)} a^{n+1} b^{n+1} c \gamma^n \xrightarrow{(7)} a^{n+1} b^{n+1} c^{n+1}. \end{aligned}$$

Here we apply productions 1,3,5 and 7 a total of  $n$  times, and the remaining ones once.

Conversely we must show that these are the *only* strings that can be produced.

Note that the argument for the second part shows we necessarily apply (1) and (3) in any order before applying (2). Consider now the stage *after* production (2) has been applied; then only (3) and (4) are relevant initially. After using (3) at most  $n$  times we necessarily use (4), at which point we introduce  $b$  into the string. We now have  $n$  more  $b$ 's to make concrete, and this process can only occur at the single point where the left hand concrete symbols meet the abstract symbols on the right. Note also that as soon as we use production (6) we introduce a  $c$  at the abstract/concrete interface and so can only subsequently use production (7). We are thus forced to already have made all  $\beta$ 's concrete before applying (7). In other words we are forced to essentially imitate the sequence of productions at the start of this section except for production (3) which can be applied at any time, provided it has been applied  $n$  times before production (6) is used.

4. (a) We give the pseudocode for the Running Sample algorithm below:

```

algorithm runningsample(k, N)
// choose k items at random from N supplied
begin
  t = 0
  chosen = 0
  repeat begin
    if ((N-t)*rand() >= (k-chosen)) begin
      // pass over next item
      read in next item
      t = t + 1
    end
    else begin
      // accept next item
      read in next item
      print out next item
      t = t + 1
      chosen = chosen + 1
    end
  end // repeat

```

```

    until (chosen = k)
end.

```

The logic of this method is that if, at the  $i$ th item, we have so far chosen  $j$  items then we have  $k - j$  items left to choose from the remaining  $N - i + 1$  items (including the  $i$ th). So it seems reasonable to choose the  $i$ th item with probability

$$\frac{k - j}{n - i + 1}$$

and that's what the algorithm does.

We cannot end up with more than  $k$  items because we stop the loop if we have got  $k$ . It is thus enough to show that we cannot end up with *fewer* than  $k$ . Suppose conversely that we had ended up with  $j < k$  items and that the last item *not* chosen was the  $i$ th. Then, at that stage, we had already chosen  $j - (N - i)$  items. So the probability of choosing the  $i$ th was

$$\frac{k - (j - (N - i))}{N - i + 1} = \frac{N - i + (k - j)}{N - i + 1} \geq 1,$$

so we *must* have chosen it! Contradiction.

**(b)** Our algorithm identifies the half of the isosceles triangle to the left of the  $x$  - axis with the remaining half of the square

$$S = \{(x, y) \mid 0 \leq x \leq 1, 0 \leq y \leq 2\},$$

namely the half above the line  $y + 2x = 2$ . Our algorithm is then

```

x = RAND();
y = 2*RAND();
if ( y > 2 - 2*x) then
    x = -1 + x
    y = 2 - y
endif
return (x,y)

```

The pair  $(x, y)$  are chosen uniformly from a rectangle with the same area as the given triangle. Our algorithm arranges to use all the points produced, while the transformation which maps the part of the square outside the isosceles triangle to the “missing” part of the triangle is Euclidean and hence preserves areas.