CSL 102: Introduction to Computer Science

Major Wed 04 May 2005 VI 301(Gps 1-6)& VI 401(Gps 7-8) 10:30-12:30 Max Marks 60

1. Answer in the space provided on the question paper.
2. The answer booklet you have been given is for **rough work only**
3. The answer booklet will not be collected.

Q1	Q2	Q3	Q4	Q5	TOTAL
12	12	12	12	12	60

Note:

1. **You may use ML notation wherever necessary, in addition to the usual algorithmic notations that we use in lectures**
2. **Correctness is of paramount importance**
3. **The more time and space efficient your solution the more the marks you will be given**

1. Assume given an array A of length $n > 0$ such that each element of the array contains a copy of the element from the set $\{a, b, c\}$ $a < b < c$. It is possible to sort this array in time $O(n)$ (instead of the usual sorting algorithms which take either $O(n \log n)$ or $O(n^2)$ time).

   Write a **linear time** and **constant space** imperative ML program which sorts this array in-place such that the following restrictions are satisfied:

   (a) The program does not count the numbers of a's, b's and c's.

   (b) The program simply uses comparison and in-place swapping of values in the array. Assume the following code for swap:

```
fun swap (A, i, j) = let val t = sub (A, i)
                                 in ( update (A, i, sub (A, j));
                                      update (A, j, t)
                                    )
                                end
```

   (c) The program does not traverse the array more than once.

   (d) For each `while`-loop in your program clearly state:

      i. the **invariant** property which proves that your program is correct

      ii. the **bound function** which shows that your loop will eventually terminate.

   (6 + 4 + 2 = 12 marks)

Solution.

The strategy in this case is to keep three references `as`, `bs` and `cs` which divide the array into four parts by the intervals (as shown in the invariant given in the program) with the unsorted part of the array being the slice of the array indexed by the interval `[!bs, !cs]`. Note that if the array did not contain any occurrence of some element such as `b`, the program still works correctly, and maintains `!as` = `!bs`, and `swap (A, !as, !bs)` preserves correctness.

```
open Array;
   fun dnf A =
          let val n = length A; val as = ref 0;
              val bs = ref 0; val cs = ref (n-1)
          in  (* INV:: forall i: 0   <= i < !as: sub (A, i) = a,
                        forall j: !as <= j < !bs: sub (A, j) = b,
                        forall k: !cs <  k <   n: sub (A, k) = c

                   BF:: |[!bs, !cs]| i.e. !cs-!bs+1
              *)
              (* Iterate while BF > 0, i.e. !cs-!bs+1 > 0
                 i.e. !cs > !bs-1 i.e. !cs >= !bs
              *)
              (  while (!cs >= !bs) do
                 ( if sub(A, !bs) = a
                   then ( swap (A, !as, !bs); as := !as + 1; bs := !bs + 1)
                   else if sub(A, !bs) = c
                   then ( swap (A, !bs, !cs); bs := !bs + 1; cs := !cs - 1)
                   else bs := !bs + 1
                 );
              )
          end
```

2. Consider the datatype of propositions defined in ML by

```
datatype Prop =
   ATOM of string     |
   NOT of Prop        |
   AND of Prop * Prop |
   OR of Prop * Prop
```

Define

(a) an SML function `prop2dnf` which transforms each proposition into a proposition in *disjunctive normal form*, i.e. in *sum of products form*.

(b) a higher-order SML function `cnf2clauselist` that transforms each proposition in *conjunctive normal form* into a a list of clauses where each clause is represented as a list of literals.

(c) Suppose you are given a set of propositions and were asked to write a program to determine whether this set of propositions is inconsistent. Explain in words (not more than 10 lines), how you would design a such a system.

($4 \times 3 = 12$ marks)

Solution.

(a) Assume we have already programmed `nnf` and have converted the proposition into negation normal form.

```
(* Distribute AND over OR to get a NNF into DNF *)

fun distAND (P, OR (Q, R))  = OR (distAND (P, Q), distAND (P, R))
|   distAND (AND (Q, R), P) = OR (distAND (Q, P), distAND (R, P))
|   distAND (P, Q)          = AND (P, Q)

(* Now the DNF can be easily computed *)

fun d_o_c (AND (P, Q)) = distAND (d_o_c (P), d_o_c (Q))
|   d_o_c (OR (P, Q))  = OR (d_o_c (P), d_o_c (Q))
|   d_o_c (P)          = P
;
```

(b)

```
fun clause2litlist (ATOM a) = [ATOM a]
|   clause2litlist (OR(P, Q)) = clause2litlist(P) @ clause2litlist(Q)
;
fun cnf2clauselist (ATOM a)     = [[ATOM a]]
|   cnf2clauselist (OR (P, Q))  = [clause2litlist (OR(P, Q))]
|   cnf2clauselist (AND (P, Q)) = cnf2clauselist(P) @ cnf2clauselist(Q)
```

(c) Inconsistency of a set of propositions implies there exists a contradiction in the set. Hence first take the conjunction of the whole set of propositions and convert into disjunctive normal form. If there is a contradiction then every disjunct must have a contradiction. Hence we may separate each disjunct into positive and negative literals. and take the intersection. If every disjunct has a nonempty intersection of atoms that occur positively and negatively then the original set of propostions is indeed inconsistent.

3. Dates may be represented as positive integers in the form *ymmdd*, where *y* is any positive integer, *mm* denotes the two digit month and *dd* denotes the two digit date within the month. For example, the date *29 Jan 2000* would then be the integer 20000129. The first date in this representation is the integer 10101 (*1 Jan 1*). The date 4 May 2005 is represented as 20050504.

   Assuming that SML can represent any positive integer define a function `nextdate` : $\mathbb{N} \to \mathbb{N}$ which computes the integer representation of the next day. Be careful to raise the exception `invalid_date` whenever necessary.

   *(*12 *marks)*

   Solution.

```
exception invalid_date;
fun nextdate (k:int) =
    let val dd = k mod 100; val ymm = k div 100;
        val y = ymm div 100; val mm = ymm mod 100;
        fun days (month) = (case month of 1 => 31 | 2 => 28
            | 3 => 31 | 4 => 30 | 5 => 31 | 6 => 30
            | 7 => 31 | 8 => 31 | 9 => 30 | 10 => 31
            | 11 => 30 | 12 => 31);
        fun leapdays (m) = if (m = 2) then 29
                           else days(m);
        fun leapyear (y) =
            if (y mod 100 = 0) then
               if (y mod 400 = 0) then true else false
            else (y mod 4 = 0);
        fun valid (k) =
            if k < 10101 then false
            else if mm < 1 then false
            else if mm > 12 then false
            else if dd < 1 then false
            else if leapyear (y) then
                    if dd > leapdays(mm)
                    then false else true
            else if dd > days (mm)
     then false else true;
    in  if not(valid (k)) then raise invalid_date
        else let val succday = dd+1; val succmo  = mm+1;
                 val succyr = y+1;
             in  if leapyear (y) then
                    if (dd < leapdays(mm)) then
                         (y*100+mm)*100+succday
                    else if (mm < 12) then
                            (y*100+succmo)*100+1
                         else (succyr*100+1)*100+1
                 else if (dd < days(mm)) then
                            (y*100+mm)*100+succday
                      else if (mm < 12) then
                              (y*100+succmo)*100+1
                           else (succyr*100+1)*100+1
             end
    end;
```

4. Write a function `sortWihoutDuplicates` which takes a list `L` of elements of some type `'a` and an ordering relation R. The function `sortWihoutDuplicates` sorts the list and at the same time removes all duplicate copies of elements present in the original list. Note that your algorithm should not remove du[plicates after sorting the list. Further the algorithm should have a time complexity of $O(n \log_2 n)$ and a space complexity of $O(n)$.

(12 marks)

Solution.

Out of all the sorting algorithms we have studied, the only one that has a complexity of $O(n \log_2 n)$ is the merge sort algorithm. Also clearly it is easier to detect duplicates when merging sorted sub-lists than before they are sorted. Hence we proceed as follows:

```
local
fun mergeSortWD R [] = []
  | mergeSortWD R [h] = [h]
  | mergeSortWD R L = (* can't split a list unless it has > 1 element *)
       let fun split []  = ([], [])
           | split [h] = ([h], [])
           | split (h1::h2::t) =
                  let val (left, right) = split t;
                   in (h1::left, h2::right)
                  end;
           val (left, right) = split L;
           fun mergeWD (R, [], []) = []
             | mergeWD (R, [], L2) = L2
             | mergeWD (R, L1, []) = L1
             | mergeWD (R, (L1 as h1::t1), (L2 as h2::t2)) =
                       (* Remove duplicates *)
                       if (h1 = h2) then mergeWD (R, L1, t2)
                       else if R(h1, h2) then h1::(mergeWD (R, t1, L2))
                       else h2::(mergeWD (R, L1, t2));
           val sortedLeft  = mergeSortWD R left;
           val sortedRight = mergeSortWD R right;
       in  mergeWD (R, sortedLeft, sortedRight)
       end;
in fun sortWithoutDuplicates R L = mergeSortWD R L
end
```

The only difference from the `mergeSort` program given in class is in the function `mergeWD` which has the extra conditional

```
if (h1 = h2) then mergeWD (R, L1, t2)
```

Otherwise it is exactly the same.

5. Ash and Sal played the following game when there was a power-cut on the sets of *HDDCS*. The game was as follows:

   (a) Sal chooses a positive integer $k > 1$ and sends it to Ash.

   (b) Ash chooses a positive integer $n > k$ and sends it to Sal. It is then Sal's turn.

   (c) Then they proceed in turn sending numbers to each other while following the rules below:

      i. The player who receives a number $m > 1$ from the opponent chooses a number from the set $\{m - k + 1, \ldots, m - 1\}$ and sends it to the opponent. It is then the opponent's turn.

      ii. The player who receives a number $m \leq 1$ wins and the game stops.

   They played seven games and Ash won all of them. At the end of the day Ash was in seventh heaven and Sal was in love!

   (a) Ash won every game by sending back numbers which satisfied an invariant property. What is the invariant?

   (b) How many rounds will each game last with Ash's strategy?

   (c) Why can't Sal follow the same strategy and win?

   (d) What was Ash's algorithm?

   ($4 + 2 + 2 + 4 = 12$ *marks)*

   Solution.

   (a) She made sure that she always initially chose a number $n = k * p + 2$ and every time sent back a result of the form $m\_ash = k * q + 2$. The invariant:

   $$\exists q \geq 0[n \geq m\_ash = k * q + 2]$$

   $$(n \textbf{ mod } k = 2) \wedge (n \geq m\_ash) \wedge (m\_ash \textbf{ mod } k = 2) \wedge (m\_ash - k < m\_sal < m\_ash)$$

   where m_ash is the last value sent to Sal, and m_sal is the value received in response.

   (b) Clearly if $n = k * p + 2$ then in p rounds the number reduces to 2 which is sent by Ash to Sal, and of course he has lost the game.

   (c) For any number $m\_ash = k * q + 2$ that Sal received he was forced to send back a number from the set $\{m\_ash - k + 1, \ldots, m\_ash - 1\} = \{k * (q - 1) + 3, \ldots, k * q + 1\}$. None of the numbers in this set is of the form $k * r + 2$.

   (d) For each number m_ash that Ash sent, Sal had to send a number $m\_sal \in \{m\_ash - k + 1, \ldots, m\_ash - 1\}$, Whatever number, m_sal she received,

   $$m\_ash - k \in \{m\_sal - k + 1, \ldots, m\_sal - 1\}$$

   always holds. Clearly if m_ash **mod** $k = 2$, then $(m\_ash - k)$ **mod** $k = 2$. So she blindly executes the following algorithm.

   Algorithm.

```
Receive k > 1 from Sal;
Choose n = k*p + 2 (with p > 1);
m_ash := n;  Send m_ash to Sal;
Receive m_sal from Sal;
while (m_sal > 1) do
{ m_ash := m_ash - k;
  Send m_ash to Sal;
  Receive m_sal from Sal
}
```