

CORE EXAMINATION

with Solutions

May 1995

Department of Computer Science
New York University

May 22, 1995

Programming Languages & Compilers

PL&C1. Consider the following program fragment:

```

var s:array[1..3] of char;
var i,j: integer;

procedure P(x:integer, y:char);
var j:integer;
begin
  j:=2;
  x := x+1;
  output(y);
  output(i);
end;

s[1]:='A'; s[2]:='B'; s[3]:='C';
i:=0; j:=1;
P(i,s[i+j]);
output(i);

```

What values are output under the following calling conventions:

- (a). (1 point) Call by value
- (b). (2 points) Call by value-result
- (c). (2 points) Call by reference
- (d). (5 points) Call by name

Solution:

- a. A, 0, 0
- b. A, 0, 1
- c. A, 1, 1
- d. B, 1, 1

PL&C2. Recall that a *container class* is an abstract type whose purpose is to represent a collection of values of some type. An iterator is an abstract type associated with a container class, by means of which one can apply some operation to all the values in the container.

- (a). In the language of your choice, write the complete specification for an iterator over a collection of integers. No implementation is required yet.

- (b). Using the parametrization mechanism of the language you chose, write an iterator that applies to homogeneous collections of any arbitrary type.
- (c). Assume that the collection is to be represented as a linked list. Write the implementation of the iterator operations.
- (d). Suppose that the collection is to be represented as a doubly-linked list. In the language you chose, can you use inheritance to define this new class in terms of the class defined in part (c) above?
- (e). In C++ it is often the case that an iterator class is defined as a friend class of the corresponding collection. Explain why.

Please remember: this question is about programming languages, so syntax matters. Your answer should not be an approximation in pseudo-code, but should be correct in a specific language.

Solution:

- a) There are three related types (or classes) to consider: some element, a container for such elements, and an iterator over these containers. The iterator has some internal state that indicates the current element of the container that is being processed. Given a container, we must be able to create an iterator over it. Given an iterator, we must be able to obtain the first and the next element, and determine whether there are further elements to process. This leads to the following Ada code:

```
package Containers is
  type Container is private;
  type Iterator is private;
  function Create (C : Container) return Iterator;
  function First (I : Iterator) return Integer;
  procedure Next (I : in out Iterator; Elmt : out Integer);
  function Done (I : Iterator) return Boolean;
private
  ...
end Containers;
```

with this definition, the elements of a container can be processed as follows:

```
It : Iterator := Create_Iterator (Cont);
E : Integer;

if not Done (It) then
  E := First (It);
  while not Done (It) then
    ... process E.
    Next (It, E);
  end while;
end loop;
```

- b) To parametrize this definition to handle containers of arbitrary elements, we make it into a generic package:

```
generic
  type Element is private;
package Containers is
  ...
```

and replace Integer throughout with the generic type Element.

- c) If the container is a list, we can declare it as follows (in the private part of the package):

```

type Node;
  type Link is access Node;
type Node is record
  Value : Element;
  Next  : Link;
end record;
type Container is record
  First  : Link;      -- pointer to head of list.
  Last   : Link;      -- pointer to tail of list.
end record;

```

An iterator holds a link (in this simple case it could just BE a link):

```

type Iterator is record
  Current : Link;
end record;

```

The package body will contain the following subprograms:

```

function Create_Iterator (C : Container) return Iterator is
begin
  return (Current => C.First);
end Create_Iterator;

function First (I : Iterator) return Element is
begin
  return I.Current.Value;
end First;

procedure Next (I : in out Iterator; Elmt : out Integer) is
begin
  I.Current := I.Current.Next;
  E := I.Current.Value;
end Next;

function Done (I : Iterator) return Boolean is
begin
  return (I.Current = Null);
end Done;

```

d) In Ada83 it is not possible to extend a type by inheritance. We can copy the specification of the package, and modify the private part to describe doubly-linked lists. In Ada95, as in C++, we can extend the container type (or class) by adding another data member to it (a backward pointer). The iterator functions are inherited without modification, but we can now add an operation Previous to traverse the list in both directions. The point of this exercise is to emphasize that the user code is not affected by this change of representation, and previous loops over containers continue working without modification, even if we choose more complex representations for containers (trees, hash-tables, etc).

e) Section 8.3.4 of "The C++ Programming Language", by B. Stroustrup, discusses iterators in great detail.

PL&C3. Suppose we want to add a "parallel assignment" construct to Pascal or Ada with the syntax:

```
(nam1, nam2, nam3 ...) := (expr1, expr2, expr3, ...);
```

where the names on the left are any legal left hand sides and the expressions on the right are any legal expressions of the corresponding types. The number of names must be the same as the number of expressions. The semantics is similar to:

```
nam1 := expr1;
nam2 := expr2;
nam3 := expr3;
...
```

except that all the expressions are evaluated before any of the assignments is done, so that for example, a swap can be written as:

```
(i, j) : (j, i);
```

It is possible to write a context free grammar for this construct:

```
PASSIGN ::= ( NAME MORE EXPRESSION )
MORE    ::= ) := (
          |   , NAME MORE EXPRESSION ,
```

(a). Verify that this grammar works by showing the parse tree for

```
(i, j) := (j, i).
```

(b). This grammar is unfortunately not at all convenient for the purposes of building a parser. Why?

(c). Describe how this construct would be handled in the parser, giving grammar rules appropriate for use with YACC or a similar tool. Briefly state what is done by the parser, and what is done during semantic analysis.

In the general case, e.g., in the swap example above, temporaries are required to keep the right semantics, but in other cases, such as $(a, b) := (c, d);$ a permissible optimization allows the assignments to be done directly without temporaries.

(d). Briefly state the exact conditions for enabling this optimization.

(e). In a sentence or two, state what techniques would be required in the compiler to determine whether these conditions were met.

Solution:

To be supplied by Professor Dewar.

Operating Systems

OS1 A page replacement algorithm should minimize the number of page faults. We can do this minimization by distributing heavily used pages evenly over all memory, rather than having them compete for a small number of page frames. We can associate with each page frame a counter of the number of pages that are associated with that frame. Then, to replace a page, we search for the page frame with the smallest counter.

(a). Define a page-replacement algorithm using this basic idea. Answer the following issues:

- 1) What is the initial value of the counter;
- 2) When is the counter increased and when is the counter is decreased; and
- 3) How is the page to be replaced selected?

(b). How many page faults occur for your algorithm for the following reference string, for four page-frames:

1,2,3,4,5,3,4,1,6,7,8,9,5,4,5,4,2.

(c). What is the minimum number of page faults for an **optimal** page-replacement strategy for the same reference string using four page-frames?

(d). What is Belady's anomaly?

(e). Does Belady's anomaly occur using our paging scheme for the reference string above, using:

- 1) 3 frames
- 2) 4 frames
- 3) 5 frames

Solution:

(a). Define a page replacement algorithm such that:

The initial value of the counter is zero;

The counters are increased whenever a new page is associated with that frame, and a counter is decreased whenever one of the pages associated with that frame is no longer needed;

A page to be replaced is selected as the frame with the smallest counter, with a FIFO queue to break ties.

(b). 14 page faults.

(c). 11 page faults.

(d) *To be supplied by Professor Palem.*

(e) *To be supplied by Professor Palem.*

OS2. Consider a file consisting initially of 100 blocks. Assume that the file control block (and the index block, in the case of indexed allocation) is already in memory. In cases a) through c) below, one block will be added. In cases d) through f), one block will be removed. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies in each of the cases a) through f). That is, fill in the table to the right (copied to your exam booklet). Note that:

- In the contiguous allocation case, assume that there is no room to grow in front of the beginning, but there is room to grow at the end;
- Assume that the block information to be added is stored in memory.

a) The block is added at the beginning

b) The block is added at the middle

a) The block is added at the end

b) The block is removed from the beginning

c) The block is removed from the middle

d) The block is removed from the end.

	Number of I/O Operations		
	Contiguous	Linked	Indexed
a)	201	1	1
b)	101	52	1
c)	1	3	1
d)	198	1	0
e)	98	52	0
f)	0	100	0

Solution:

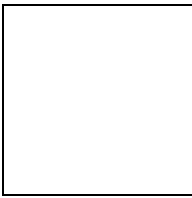
. ALGORITHMS

ALGS1. The following recursive algorithm computes the minimum value in an array $A[i..j]$:

```

function min(var A: array of integer; i, j : integer):integer
var min1, min2: integer;
begin
  if i = j then
    min := A[i];
  else
    begin
      min1 := min(A, i, (i+j) div 2);
      min2 := min(A, (i+j) div 2 + 1, j);
      if min1 < min2 then
        min := min1
      else
        min := min2
    end
  end
end

```

Calling a recursive algorithm (like `min`) generates a tree of recursive calls. For , define $T(i,j)$ to be the number of calls (including the original call) generated by the call `min(A, i, j)`. Write an exact recurrence equation for T . Treat the base case $i = j$ and the recursive case $i < j$ separately.

(b). Find an exact solution $T(i,i)$, $T(i,i+1)$, and $T(i,i+2)$. Then find the more general solution for $T(i,j)$ for every $i < j$. **Hint:** Use your recurrence from part (a).

Solution:

(a). If $i = j$, then $T(i, j) = 1$, since in that case there are no calls besides the original call. If $i < j$, then the two calls result in

$$T(i, j) = T\left(i, \left\lfloor \frac{i+j}{2} \right\rfloor\right) + T\left(\left\lfloor \frac{i+j}{2} \right\rfloor + 1, j\right) + 1.$$

The first term on the right is the number of calls generated by the recursive call `min(A, i, (i, j) div 2)`. The second term is the number of calls generated by the recursive call `min(A, (i, j) div 2 + 1, j)`. The last term 1 is for the original call itself.

(b). The base case above showed that $T(i, i) = 1$ (for every integer i). For the next case $j = i + 1$, we have

$$\begin{aligned} T(i, j) &= T\left(i, \left\lfloor \frac{i+i+1}{2} \right\rfloor\right) + T\left(\left\lfloor \frac{i+i+1}{2} \right\rfloor + 1, i+1\right) + 1 \\ &= T(i, i) + T(i+1, i+1) + 1 = 1 + 1 + 1 = 3. \end{aligned}$$

For the next case $j = i + 2$, we have

$$T(i, j) = T\left(i, \left\lfloor \frac{i+i+2}{2} \right\rfloor\right) + T\left(\left\lfloor \frac{i+i+2}{2} \right\rfloor + 1, i+2\right) + 1$$

$$= T(i, i+1) + T(i+2, i+2) + 1 = 3 + 1 + 1 = 5.$$

The pattern is now clear. For every $i \leq j$, we claim that $t(i, j) = 2(j - i) + 1$. We prove this by induction. Let $P(d)$ be the statement that for every integer i , $T(i, i + d) = 2d + 1$. We have already verified $P(0)$ above, and also $P(1)$ and $P(2)$. Suppose that $P(0), \dots, P(d - 1)$ are true for some $d \geq 1$; we will verify that $P(d)$ is true. We have:

$$\begin{aligned} T(i, j) &= T\left(i, \left\lfloor \frac{i+i+d}{2} \right\rfloor\right) + T\left(\left\lfloor \frac{i+i+d}{2} \right\rfloor + 1, i+d\right) + 1 \\ &= T\left(i, i + \left\lfloor \frac{d}{2} \right\rfloor\right) + T\left(i + \left\lfloor \frac{d}{2} \right\rfloor + 1, i+d\right) + 1 = \left[2\left\lfloor \frac{d}{2} \right\rfloor + 1\right] + \left[2\left(d - \left\lfloor \frac{d}{2} \right\rfloor - 1\right) + 1\right] + 1 = 2d + 1 \end{aligned}$$

ALGS2. Let $A[1..n]$, $B[1..n]$, $C[1..n]$, and $D[1..n]$ be four arrays of numbers.

(a). For this part only, suppose that A and B are sorted (in increasing order). Design an algorithm to decide whether there exist two indices i and j with $1 \leq i \leq n$ and $1 \leq j \leq n$ such that $A[i] + B[j] = 0$. Your algorithm should run in $O(n)$ time.

(b). Now A and B are general (not necessarily sorted). Design an algorithm to decide whether there exist two indices i and j with $1 \leq i \leq n$ and $1 \leq j \leq n$ such that $A[i] + B[j] = 0$. Your algorithm should run in $O(n \log n)$ time.

(c). Design an algorithm to decide whether there exist four indices i , j , k , and l in the range $1..n$ such that $A[i] + B[j] + C[k] + D[l] = 0$. Your algorithm should run in $O(n^2 \log n)$ time.

Solution:

(a). One way to solve this problem is to reduce it to merging. Namely, construct an array $Y[1..n]$ as follows:

$$Y[m] = -B[n+1-m] \quad (\text{for all } m \text{ in } 1..n).$$

That is, Y is the reversal of the negation of B . It is easy to see that Y is also sorted in increasing order. Now we just need to check if A and Y share any common elements. This task can be solved by applying the usual “two-finger” merging algorithm (i.e., the usual MERGE algorithm) to A and Y , looking for any equalities along the way. The running time is $O(n)$ to create the B array and $O(n)$ to perform the merge.

A different way to solve the problem is to write an explicit algorithm. Here is an algorithm that is similar to the MERGE algorithm above, but operates on the A and B arrays, walking left to right in A and right to left in B .

```

i := 1;
j := n;
while ( i <= n ) and ( j >= 1 ) and ( A[i]+B[j] <> 0 ) do
  if A[i]+B[j] < 0 then
    i := i + 1
  else
    j := j - 1
  if ( i <= n ) and ( j >= 1 ) then
    write('Yes')
  else
    write('No')
```

This algorithm runs in $O(n)$ time since the value of $j - i$ necessarily decreases by 1 in each iteration through the loop, and since it begins at $n - 1$ and cannot get smaller than $1 - n$, there are at most iterates.

The algorithm works by successively eliminating rows or columns in (i, j) space, so that if i and j are currently being considered, then the search for $A[i'] + B[j'] = 0$ must occur with $i' \geq i$ and $j' \leq j$.

(b). We first sort the elements of A and B , using an $O(n \log n)$ algorithm, such as MERGESORT, and then apply the $O(n)$ algorithm from part (a).

(c). Construct two dimensional arrays $E(1..n, 1..n)$ and $F(1..n, 1..n)$ given by:

$$E(i, j) = A(i) + B(j) \quad F(k, l) = C(k) + D(l).$$

Then flatten the arrays E and F to become one-dimensional arrays of length $1..n^2$ in any arbitrary fashion. (This construction takes time $O(n^2)$.) Finally, apply the algorithm from part (b) to these two one-dimensional arrays. The complexity is $O(n^2 \log(n^2)) = O(2n^2 \log n) = O(n^2 \log n)$.

ALGS3. Let $G=(V,E)$ be an undirected graph with no self-loops (that is, no edges from a vertex to itself). Let x be a vertex in V . Define the elimination graph $G[x]$ to be the graph obtained from G by removing the vertex x (and all edges containing x) and inserting edges between every pair of neighbors of x . More precisely, set $G[x] = (V', E')$ where $V' = V - \{x\}$ and where (for every two distinct vertices y and z in V') we have

$$\{y, z\} \in E' \text{ if and only if } \{y, z\} \in E \text{ or } (\{y, x\} \in E \text{ and } \{x, z\} \in E)$$

Let x_1, x_2, \dots, x_n be an ordering of all the vertices in V . Consider the sequence of graphs $G, G[x_1], G[x_1][x_2], \dots, G[x_1][x_2] \dots [x_n]$. (That is, the vertices are eliminated one at a time in the order x_1, x_2, \dots, x_n .) Define the *cost* of the ordering to be the maximum number of edges of any graph in this sequence. I.e.:

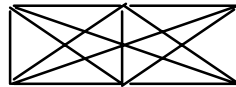
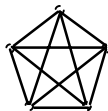
Definition: The cost of x_1, x_2, \dots, x_n is:

$$\max(\# \text{ edges in } G, \# \text{ edges in } G[x_1], \dots, \# \text{ edges in } G[x_1][x_2] \dots [x_n]).$$

(a). Find a graph G and two orderings of the vertices of G such that the two orderings have different costs.

(b). Find a planar graph G and two orderings of the vertices of G such that with the first ordering all of the graphs in the elimination sequence are planar, whereas in the second ordering not all of the graphs are planar.

Recall that a graph is planar if it can be drawn in the plane with no pair of edges crossing. The two graphs below are not planar.



Solution:

The following examples serve for both (a) and (b). Consider:



(a). In the left graph, in the first elimination step, $G[v_1]$, we obtain the complete graph of five nodes, which has 10 edges (see the pentagon example above). Successive eliminations reduce the number of edges, so the cost is 10. Using the ordering on the right, each elimination removes one edge, so that the total cost is the number of edges in the original graph, which is 5.

(b). There are many examples that work, but the two orderings of the graph from the solution in part (a) work. Notice that the graph is planar, since it is drawn in a plane. After performing the first elimination using the ordering on the left, planarity is lost, as the resulting graph is the complete graph on five vertices, which is not planar. On the other hand, each elimination step using the ordering on the right simply eliminates a single node and edge, and thus all graphs in the elimination sequence are planar.