Algorithms

Algorithms are usually written as procedures (functions):

input: **proc procname**(vars: type or **arrays$_{dim}$:** type)

output: return(vars or **array$_{dim}$**)

variables: single variables and array entries: x, a_i ; whole array: $\mathbf{a_n}$

assignment: :=

calculations: +, ─, • , / , ↓*↓(floor), ↑*↑(ceiling), etc.

controls: if (true)
then … (indentation demarcates the scope)
else if …
then …
else … .

loops while (true)
(indentation demarcates the scope)

until (true)
(indentation demarcates the scope)

for index := start to end …
(indentation demarcates the scope)

condition operators: = , ≠ , < , ≤ , > , ≥ , and , or

==========+==========

Algorithms are sometimes written as programs:

input: read vars

output: write vars

Example: Linear search an array of numbers, $\mathbf{a_n}$, for a specific number, **x** :

```
proc        linsrch( x: integer, a_n: integers, n>1 )
i: = 1
loc := 0
while ( i ≤ n  and  x ≠ a_i )
        i :=  i+1
if ( i ≤ n )
then   loc := i
return( loc )
/* The number of times thru the white-loop is ≤ n  and on average  n/2 . */
```

==========+==========

Example: Binary search an array of numbers, $\mathbf{a_n}$, for a specific number, **x** :

```
proc        binsrch( x: integers , a_n: integers increasing, n>1)
i: = 1                  /* left end */
j := n                  /* right end */
loc := 0
while ( i < j )
        m :=  floor( (i+j)/2 )          /* find middle */
        if ( x > a_m )                  /* which half is x in */
        then   i := m+1
        else   j := m
if ( a_i = x )
then loc := i
return( loc )
/* The number of times thru the while-loop is ≤ log2(n) + 1. */
```

==========+==========

Example: Bubble-Sort an array of numbers, $\mathbf{a_n}$:

```
proc        bubsort( a_n: integers, n≥2)
for i: = 1 to n – 1          /* at end of each i-loop, largest i items at end */
        for j := 1 to n – i
                if ( a_j > a_j+1 )
                then   x := a_j          /* swap a_j & a_j+1 */
                       a_j := a_j+1
                       a_j+1 := x
return( a_n )
/* The number of times thru both for-loops combined is:   n•(n-1)/2 . */
```

==========+==========

Example: Insertion-Sort an array of numbers, $\mathbf{a_n}$:

```
proc        inssort( a_n: integers, n≥2)
for j: = 2 to n
        i := 1
        while ( a_j > a_i )
                i := i+1
        m := a_j
        for k := 0 to j-i-1
                a_{j-k} := a_{j-k-1}        /* shifts these forward and then inserts*/
        a_i := m                            /* m, old a_j, where it belongs. */
return( a_n )
```

/*	j	i	m	k	a_n =						
	2	1,1	2	0,0		2	7	5	12	4	9
	3	1,2	5	0,0		2	5	7	12	4	9
	4	1,4	12	0,-1		2	5	7	12	4	9
	5	1,2	4	0,2		2	4	5	7	12	9
	6	1,5	9	0,0		2	4	5	7	9	12 */

Example: Merge without duplicates two pre-sorted arrays, $\mathbf{a_n}$, $\mathbf{b_n}$:

```
proc        merge( a_n, b_m: integers, n,m≥1)
I := 1
J := 1
k := 1
while ( i ≤ n  and  j ≤ m )
        if ( a_i < b_j )
        then   c_k := a_i
                i := i+1
        else if ( a_i > b_j )
        then   c_k := b_j
                j := j+1
        else            /*  ( a_i = b_j ) : this eliminates duplicates */
                c_k := b_j
                i := i+1
                j := j+1
        k := k+1
return( c_k )
/* The number of times thru the while loop is at most n+m */
```

Example: Recursive Merge-Sort an array of numbers, $\mathbf{a_n}$:

proc **mergesort**($\mathbf{a_n}$: integers, n≥2)
if n > 1
then m := floor(n/2)
/* assume $\mathbf{a_n}$ may be split and truncated to $\mathbf{a_{1\rightarrow m}}$ and $\mathbf{a_{m+1\rightarrow n}}$ */
/* by merely changing subscript (perhaps using address pointers). */
return(**merge(mergesort($\mathbf{a_{1\rightarrow m}}$), mergesort($\mathbf{a_{m+1\rightarrow n}}$))**)
else return($\mathbf{a_n}$)
/* The number of times deep thru the recursive call loop is: **$\log_2$(n+1)** . */
/* The number of times thru implicit double-loops is: **n•$\log_2$(n+1)** . */

==========+==========

Example: Add two matrices of numbers, $\mathbf{a_{nm}}$ & $\mathbf{b_{nm}}$;

proc **matrixadd**($\mathbf{a_{nm}}$, $\mathbf{b_{nm}}$: reals, n,m≥1)
for i: = 1 to n
for j := 1 to m
$c_{ij} := a_{ij} + b_{ij}$
return($\mathbf{c_{nm}}$)
/* The number of times thru both all-loops combined is: **n•m** . */
==========+==========

Example: Multiply two matrices of numbers, $\mathbf{a_{np}}$ & $\mathbf{b_{pm}}$;

proc **matrixmult**($\mathbf{a_{np}}$, $\mathbf{b_{pm}}$: reals, n,p,m≥2)
for i: = 1 to n
for j := 1 to m
$c_{ij} := 0$
for k:= 1 to p
$c_{ij} := c_{ij} + a_{ik} \bullet b_{kj}$
return($\mathbf{c_{nm}}$)
/* The number of times thru both all-loops combined is: **n•m•p** . */
==========+==========

Example: Omit k^{th} row and l^{th} column of a matrix, $\mathbf{a_{nm}}$;

proc **matrixomit**(k, l : integers , $\mathbf{a_{nm}}$: reals, n,m≥2)
for i: = 1 to n
    if (i < k)
    then r := i
    else if (i > k)
    then r := i – 1
    else r := 0
    for j := 1 to m
        if (j < l)
        then s := j
        else if (j > l)
        then s := j – 1
        else s := 0
    if (r ≠ 0 and s ≠ 0)
        then $b_{rs} := a_{ij}$
return($\mathbf{b_{rs}}$)
/* The number of times thru both for-loops combined is: **n•m** . */

==========+==========

Example: Recursive calculation of the determinant of a matrix, $\mathbf{a_{nn}}$;

proc **determinant**($\mathbf{a_{nn}}$: reals, n≥2)
d := 0
s := 1
if (n > 2)
then m := n – 1
    for i: = 1 to n
        d := d + s•**determinant**(**omitmatrix**(i,1,$\mathbf{a_{nn}}$))
        s := s•(-1)
else if (n = 2)
then $d := a_{11}a_{22} - a_{12}a_{21}$
else $d := a_{11}$
return(d)
/* The number of times thru all for-loops combined is: ?? . */

==========+==========

Example: Calculate ‘q = a div d’ and ‘r = a mod d’;

```
proc        divmod( a: int,  d: int > 0 )
r := a      /* a is dividend, d is divisor, q is quotient, r is remainder */
q := 0
if  (r ≥ 0)
then   while ( r ≥ d )
               r := r – d
               q := q + 1
else   while ( r < 0 )
               r := r + d
               q := q – 1
return(q,r)
/* The number of times thru all loops combined is:   ceiling(|a/d|)  */
/* In the following, split this into two procs:  mod(a,d) & div(a,d) */
```

Example: Calculate the prime factors of n > 1 and return in an array, $\mathbf{f_m}$;

```
proc        primefact(n: int > 1 )
j := 0
d := 2
while ( d ≤ n )
      while ( mod(n,d) = 0 )
            j := j + 1      /* really must do “malloc” to enlarge  f_m */
            f_j := d
            n := div(n,d)
      d := d + 1
      m := j
return( f_m )
/* The number of times thru all loops combined is exponential order */
```