5.4 Recursive Algorithms

A recursive algorithm solves a problem by breaking it into smaller pieces and then using itself on those pieces.

We saw the next example already, that computes n! ("n factorial")

```
procedure factorial (n: nonnegative
                        integer)
if n=0 then return 1
else return n • factorial(n-1)

{output is n!}
```

then factorial(0) returns 1 $(0! = 1)$

and tracing through the procedure we find factorial(4) returns 24 which is 4! For this the procedure had to call itself 4 times.

An iterative algorithm works without calling itself and solves a problem step by step, often using loops.

We can give examples of the two types of algorithms in computing the Fibonacci numbers.

The Fibonacci sequence is defined by:

Basis step $f_0 = 0, \quad f_1 = 1$

Recursive step $f_{n+1} = f_n + f_{n-1} \qquad n \geq 1$

and we get the sequence 0, 1, 1, 2, 3, 5, 8, 13, ...

recursive

```
procedure fib1(n: nonneg integer)
if n=0 or n=1 then return n
else return fib1(n-1) + fib1(n-2)
{output is f_n}
```

iterative

```
procedure fib2(n: nonneg integer)
if n=0 then return 0
else
    x := 0, y := 1
    for i := 1 to n-1
        z := x + y
        x := y
        y := z
    return y
{output is f_n}
```

See section 3.1 in the book for the details of the pseudocode we are using. Note that $:=$ is the assignment operation with $x := 0$ meaning that the variable x is assigned the value 0.

Comparing fib1 and fib2, we see that fib1 is shorter and easier to understand while fib2 is more complicated. On the other hand fib2 will probably run faster and use less memory.

The Euclidean Algorithm is a very short recursive procedure to compute the greatest common divisor (gcd) of two integers.

For example $\gcd(30, 40) = 10$ because 10 is the largest divisor (factor) of both

$$30 = 3 \cdot \underline{10}, \quad 40 = 4 \cdot \underline{10}$$

For another example $\gcd(24, 35) = 1$ because 1 is the largest common factor.

We will use the notation $b \bmod a$ to mean the remainder when b is divided by a (so $b \bmod a$ is $0, 1, 2, \ldots$ or $\ldots a-1$)

eg $\quad 7 \bmod 3 = 1 \qquad 24 \bmod 5 = 4.$

Here is the recursive procedure for gcd.

```
procedure gcd(a, b : non neg integers
                            a < b)
if a = 0 then return b
else return gcd(b mod a, a)
{output is gcd(a,b)}
```

For example, we can trace what happens when $a = 30$, $b = 40$ are the inputs.

Since $a \neq 0$ we need

$$\gcd(40 \bmod 30,\ 30)$$

$$= \gcd(\underset{\uparrow\ \text{new } a}{10},\ \underset{\uparrow\ \text{new } b}{30})$$

The new a is $\neq 0$ so we go to

$$\gcd(30 \bmod 10,\ 10)$$

$$= \gcd(\underset{\uparrow\ \text{new } a}{0},\ \underset{\uparrow\ \text{new } b}{10})$$

This new a is zero, so we return $b = 10$.

It seems to work correctly:

$$\gcd(30, 40) = 10.$$

Check that for $a = 24$ and $b = 35$ it also gives the expected answer 1 for the gcd.

Why does this algorithm work?

(A) Well it is true that $\gcd(0, b) = b$ because every number is a factor of zero (they all go in zero times).

(B) How about $\gcd(a, b) = \gcd(b \bmod a, a)$?

Suppose that a goes into b q times with remainder r:

$$b = qa + r.$$

Then $b \bmod a = r$. It can be seen that d is a factor of a and b if and only if d is a factor of $b - qa$ and a

$$d \mid a, b \iff d \mid b - qa, a.$$

So these pairs have the same gcd. Also the second pair is smaller than the first, making the problem easier.

These two parts (A), (B) can be used in a (strong) induction proof of the algorithm.

Merge Sort

This is a recursive algorithm to sort a list of integers into increasing order. For example {3, 9, 2, 6} would get sorted to {2, 3, 6, 9}. We want an efficient way to do this, especially for very large lists, and Merge Sort gives the theoretically most efficient method.

How it works - for example with input

{3, 1, 0, 2, 7, 9, 4}

It starts by breaking list in two (almost) equal parts

{3, 1, 0} {2, 7, 9, 4}

repeat:

{3} {1, 0} {2, 7} {9, 4}

{1} {0} {2} {7} {9} {4}

Now the lists contain only one element it is easy to recombine them in the correct order

{1} {0} {2} {7} {9} {4}

{0, 1} {2, 7} {4, 9}

Then $\{3\}$ $\{0,1\}$ $\{2,7\}$ $\{4,9\}$

$$\{3\} \searrow \; \swarrow \{0,1\} \qquad \{2,7\} \searrow \; \swarrow \{4,9\}$$

$$\{0,1,3\} \qquad \{2,4,7,9\}$$

The point is that it is easier to combine two lists that are already ordered into an ordered list: you just have to compare the first elements in each list

$$\{0,1,3\} \searrow \quad \swarrow \{2,4,7,9\}$$

$$\{0,1,2,3,4,7,9\}$$

This merging is done with the <u>merge</u> algorithm

```
procedure merge(L1, L2 : sorted lists)

L := empty list
while L1, L2 both not empty
    remove smaller of 1st elements of
    L1, L2 and put at right of L
Put any remaining elements of
L1 or L2 at right of L
Return L
{L is the merged, sorted list}.
```

<u>Example</u> $L_1 = \{0,1,3\}$ $L_2 = \{2,4,7,9\}$

merge creates $L = \{0, 1, 2, 3, \underbrace{4, 7, 9}_{(5)}\}$

step (1) (2) (3) (4) (5)

This procedure merge is used in the main sorting algorithm next.

procedure mergesort $(L = a_1, \ldots, a_n)$

If $n > 1$ then

$m := \lfloor n/2 \rfloor$

$L_1 := a_1, a_2, \ldots, a_m$

$L_2 := a_{m+1}, \ldots, a_n$

$L := \text{merge}(\text{mergesort}(L_1), \text{mergesort}(L_2))$

$\{L \text{ is now sorted}\}$

Note that $\lfloor x \rfloor$ is the "floor" function which goes down to the next integer:

eg. $\lfloor 17.3 \rfloor = 17$ $\quad \lfloor 7/2 \rfloor = 3$ $\quad \lfloor 12 \rfloor = 12$.

<u>Example</u> Trace how mergesort works on $L = \{2, 7, 1\}$.

<u>Solution</u> Here $a_1 = 2$, $a_2 = 7$, $a_3 = 1$ and $n = 3$

So $m = \lfloor n/2 \rfloor = \lfloor 3/2 \rfloor = 1$ and

$L_1 = \{2\}$, $\quad L_2 = \{7, 1\}$

then new L = merge(mergesort(2),
mergesort(7,1))

Now mergesort(2) has $n=1$ and just returns 2

Mergesort(7,1) has $n=2$ and becomes
merge(mergesort(7), mergesort(1))

$= \text{merge}(\{7\}; \{1\})$

$= \{1,7\}$

Then our L is merge($\{2\}$, $\{1,7\}$)

$= \{1,2,7\}$.

Try tracing through mergesort on our original list

$$\{3,1,0,2,7,9,4\}.$$