5.4 Recursive Algorithms

A <u>recursive</u> <u>algorithm</u> 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! ("in 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 unabers.

The Fibonacci sequence is defined by:

Basis step $f_0 = 0$, $f_1 = 1$ Recursive step $f_{n+1} = f_n + f_{n-1}$ $n \ge 1$

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

procedure fibl(n: nonneg integer)

if n=0 or n=1 then return n

else return fibl(n-1) + fibl(n-2)

{output is fn}

procedure fib2 (n: nonneg integer)

if n=0 then return 0

else

X:=0, y:=1

for i:=1 to n-1

7:=X+y

X:=y

y:=7

return y

{ontput is fn}

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

Comparing fibl and fib2, we see that fibl 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 ged (30,40) = 10 because 10 is the largest druisor (factor) of both 30=3.10, 40=4.10

for another example gcd (24, 35) = 1 because I is the largest common backer.

We will use the notation b mod a to mean, the remainder when b is divided by a (so b mod a is 0,1,2, -- or .9-1)

eg 7 mod 3 = 1 24 mod 5 = 4.

Here is the recursive procedure for gcd.

procedure gcd(a,b:non neg integers a < b)if a = 0 then return belse return gcd(b nod a, a){ont put is gcd(a,b)}

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

Since a \$0 we need

gcd (40 rod 30, 30)

= gcd (10, 30)

new a new b

new a new b

The new a is \$0 so we go to ged (30 mod 10, 10)

= gcd (0,10) new a new b

This new a is zero, so we return b = 10. It seems to work correctly: ged(30, 40) = 10.

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

why does this algorithm work?

- A)well it is true that gcd(0,b) = bbecause every number is a factor of zero (they all go in zero times).
- (B) How about gcd (a, b) = gcd (b mod a, a)?

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

then brod 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|a,b (=) d| 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 {2,3,6,93. 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 then in the correct order {1} {0} {2} {7} {9} {4}

{0,13 {2,7} {4,93

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

This merging is done with the merge algorithm

procedure merge (L1, L2: sorted lists)

L:= empty list

while L, Lz both not empty

remove smaller of 1st elements of

Li, Lz and put at right of L

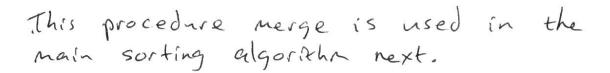
Put any remaining elements of

Li or Lz at right of L

Neturn L

EL is the merged, sorted list?

Example $L_1 = \{0,1,3\}$ $L_2 = \{2,4,7,9\}$ Merge creates $L = \{0,1,2,3,4,7,9\}$ Step (1) (2) (3) (4) (5)



procedure mergesort (L=a,,..,an)

If n>1 then

$$m:=\lfloor n/2\rfloor$$
 $L_1:=a_1,a_2,...,a_m$
 $L_2:=a_{m+1},...,a_n$
 $L:=merge(mergesort(L_1),mergesort(L_2))$

{L is now sorted}

Note that LXJ is the "floor" function which goes down to the next integer:

eg. L17.3] = 17 L7/2] = 3 L12] = 12.

Example Trace how mergesort works on L= {2,7,1}.

Solution Here $a_1=2$, $a_2=7$, $a_3=1$ and n=3

$$So M = L^{n/2}J = [3/2] = 1$$
 and $L_1 = \{23\}$, $L_2 = \{7, 1\}$

5

then new L = merge (mergesort (2), .

nergesort (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))

= merge ({7}; {1})

= {1,7}

Then our L is merge ({23, {1,73}})
= {1,2,7}.

Try tracing through mergesort on our original list

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