



Recursive



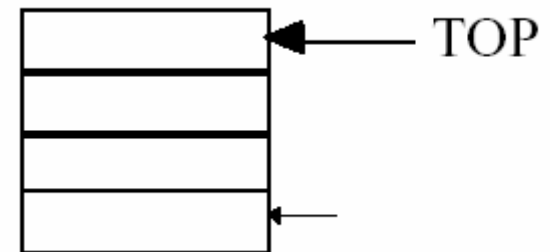
Section 3.5 Recursive Algorithms

Recursive Algorithms

- A recursive algorithm is one which calls itself to solve “smaller” versions of an input problem.
- Some algorithms are recursive by nature:
 - Binary search
 - Fibonacci sequence

How it works

- The current status of the algorithm is placed on a *stack*.
- A *stack* is a data structure from which entries can be added and deleted only from one end
 - like the plates in a cafeteria:
 - **PUSH**: put a 'plate' on the stack.
 - **POP**: take a 'plate' off the stack.
- When an algorithm calls itself, the current *activation* is suspended in time and its parameters are **PUSHed** on a stack.
- The set of parameters need to restore the algorithm to its current activation is called an *activation record*.



Example

■ Example : (classical)

procedure *factorial* (n)

/ make the procedure
idiot proof */*

if n < 0 **return** 'error'

if n = 0 **then return** 1

else

return (n * *factorial* (n-1))

The operating system supplies all the necessary facilities to produce:

factorial (3): **PUSH** 3 on stack and call

factorial (2): **PUSH** 2 on stack and call

factorial (1): **PUSH** 1 on stack and call

factorial (0): return 1

POP 1 from stack and return (1) (1)

POP 2 from the stack and return (2) [(1) (1)]

POP 3 from the stack and return (3) [(2) [(1) (1)]]

Example

- ***Complexity:***

Let $f(n)$ be the number of multiplications required to compute factorial (n).

$f(0) = 0$: the *initial condition*

$f(n) = 1 + f(n-1)$: the *recurrence equation*

Example

A recursive procedure to find the max of a nonvoid list.

Assume we have built-in functions called

- Length, which returns the number of elements in a list
- Max, which returns the larger of two values
- Listhead, which returns the first element in a list

Max requires one comparison.

Example

```
procedure maxlist (list)
/* strip off head of list and
   pass the remainder */
if Length(list) = 1 then
return Listhead(list)
else
return Max( Listhead(list),
             maxlist(remainder of
             list))
```

- Complexity: The recurrence equation for the number of comparisons required for a list of length n , $f(n)$, is
 - $f(1) = 0$
 - $f(n) = 1 + f(n-1)$

Example

■ Example

Find the maximum element from a list, assuming the length is a power of 2:

- We divide the list in half and find the maximum of each half.
- Then find the Max of the maximum of the two halves.

Example

```
procedure maxlist (list)
/* a divide and conquer
   approach */
if Length (list) = 1 then
    return Listhead(list)
else
    a = maxlist (fist half of list)
    b = maxlist (second half of
                 list)
return Max{a, b}
```

Recurrence equation for the number of comparisons required for a list of length n , $f(n)$, is

- $f(1) = 0$
- $f(n) = 2 f(n/2) + 1$
- There are two calls to *maxlist* each of which requires $f(n/2)$ operations to find the max.
- There is one comparison required by the Max function

Example

$$f(16) = 2 f(8) + 1$$

$$f(8) = 2 f(4) + 1$$

$$f(4) = 2 f(2) + 1$$

$$f(2) = 2 f(1) + 1$$

$$f(1) = 0$$

So

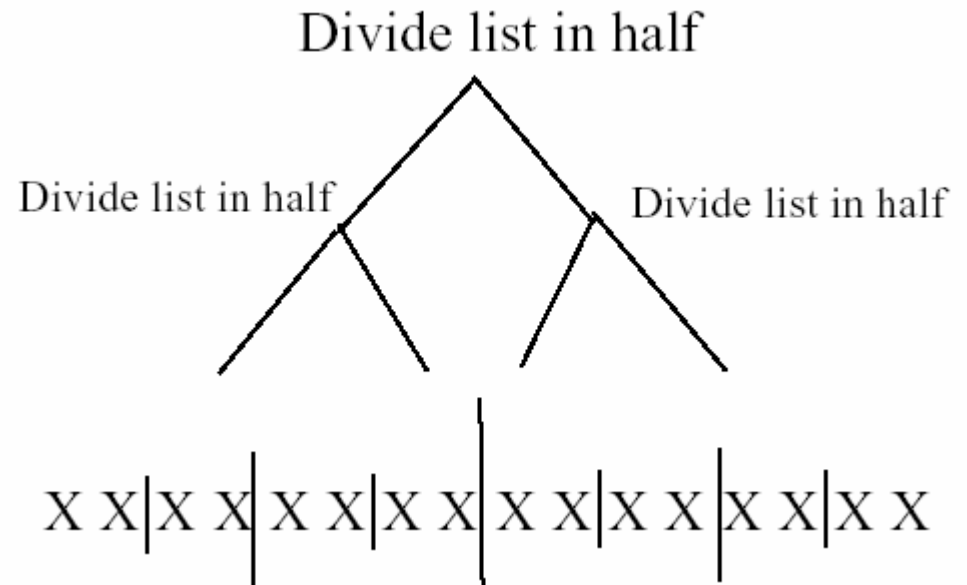
$$f(2) = 1,$$

$$f(4) = 2 (1) + 1 = 3$$

$$f(8) = 2 (3) + 1 = 7$$

$$f(16) = 2(7) + 1 = 15$$

$f(n)$?



Remark

- Although it is often easier to write a correct recursive algorithm for a function, iterative implementations typically run faster, because they avoid calling the stack.