PC 3
Récursivité

- Appels récursifs
- Fractales
- Les tours de Hanoi
- Quicksort

Généralités

L'utilisation de la rcursivité permet d'écrire en général plus facilement les programmes : c'est une traduction directe des définitions par récurrence.

Cependant, les performances (en temps ou en place) des programmes ainsi écrits sont parfois beaucoup moins bonnes.

On peut dans certains cas commencer par écrire en récursif (prototypage) pour optimiser ensuite.

Exemples simples

Calcul de $n! = n \times (n-1)!$:

```
function fact( n : integer) : integer;
 begin
  if n = 0 then fact := 1
           else fact := n * fact(n-1)
 end;
```

Il faut faire attention au mode de transmission des paramètres. En général, le passage par valeur est préférable. Si on écrit la fonction précédente sous la forme

```
function fact( n : integer) : integer;
 begin
  if n = 0 then fact := 1
           else begin
            n:= n-1;
            fact := (n+1) * fact(n)
           end
 end
```

le passage de n par adresse est impossible.

Calcul du pgcd des entiers m et n: on pose

$$n = mq + r, \qquad 0 \leq q,\ 0 \leq r < m$$

Comme $pgcd(m, n) = pgcd(r, m)$ on obtient:

```
function pgcd (m, n: integer): integer;
begin
 if m=0 then pgcd := n
 else pgcd := pgcd(n mod m, m)
end;
```

L'algorithme converge car $n + m$ diminue.

On peut aussi calculer ainsi un couple (a, b) d'entiers tels que $am + bn = pgcd(m, n)$.

```
function pgcd (m, n: integer;
       var : a,b:integer): integer;
 var u,v: integer;
 begin
  if m=0 then begin
   pgcd := n;
   a:=0;b:=1
   end
  else begin
   pgcd := pgcd(n mod m, m,u,v);
   a:= v - u * (n mod m); b:= u;
  end
 end;
```

Comment ça marche?

La mémoire est gérée de façon *dynamique* : une nouvelle zone est utilisée pour chaque appel de la fonction dans la *pile d'exécution*. De cette façon, les anciennes valeurs ne sont pas écrasées.

Pour chaque appel, on reserve des places dans un *enregistrement d'activation* pour :

- Les valeurs des paramètres.
- La valeur rendue.
- L'adresse de retour
- Les variables locales

Récursivité terminale

Tout appel récursif figurant comme dernière instruction d'une procédure peut être éliminé. On peut remplacer:

```
procedure P ( x :  T);
   begin
   .
   .
   .
   P(e); {e :  expression}
   end;
```

par:

```
procedure P (x :  T);
   1 :  begin
   .
   .
   .
   x := e; goto 1
   end;
```

Suite de Fibonacci

Suite récurrente linéaire u_n définie par $u_0 = u_1 = 1$ et

$$u_n = u_{n-1} + u_{n-2}$$

Programme Pascal récursif:

```
function Fib(n:  integer):  integer;
   begin
      if n <= 1 then
         Fib := 1
      else
         Fib := Fib(n-1)+Fib(n-2)
   end;
```

Programme Pascal itératif:

```
function Fib(n:  integer):integer;
   var i, u, v :  integer;
   {u = u_{i-1}; v = u_i}
   begin
      u := 1; v := 1;
      for i := 2 to n do
      begin
         v := u + v;
         u:= v—u
         {ancienne valeur de v}
      end;
      Fib := v
end;
```

Forme matricielle:

$$\begin{bmatrix} u_n \\ u_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} u_{n-1} \\ u_{n-2} \end{bmatrix}$$

La fonction d'Ackerman

C'est la fonction définie par:

```
function Ack (m, n:  integer):  integer;
   begin
   if m := 0 then
        Ack := n+1
   else
        if n = 0 then
             Ack := Ack(m-1, 1)
        else
             Ack := Ack (m-1, Ack(m, n-1));
   end;
```

Premières valeurs:

$$
\begin{aligned}
Ack(1,n) &= n+2 \\
Ack(2,n) &= 2*n \\
Ack(3,n) &= 2^n \\
&\dots
\end{aligned}
$$

Fractales

Principe: objets définis par itération de transformations et donc aisément définissables récursivement.

Référence: G. Edgar, *Measure, Topology and Fractal Measure*, Springer, 1992.

Exemple: La courbe du dragon de Heighway est donnée par la procédure suivante:

```
procedure Dragon (n:  integer; x, y, z, t:  integer)
   var u, v:  integer;
   begin
   if n = 1 then
        begin
        MoveTo (x,y);
        LineTo (z, t);
        end
   else
        begin
        u := (x + z + t − y) div 2;
        v := (y + t − z + x) div 2;
        Dragon (n − 1, x, y, u, v);
        Dragon (n − 1, z, t, u, v);
        end;
   end;
```

Les tours de Hanoi

Problème: On dispose n plaques sur le piquet 1 et on a deux autres piquets vides. On veut amener toutes les plaques du piquet 1 au piquet 3 en respectant les règles suivantes:

1. Une plaque ne peut être placée que sur une plus grande.
2. On ne peut déplacer qu'une plaque à la fois.

Solution récursive:

```
procedure Hanoi (n : integer; i, j: integer);
   begin
   if n> 0 then
      begin
      Hanoi (n-1, i, 6 − (i+j));
      writeln(i: 2,'->', j: 2);
      Hanoi (n-1, 6 − (i+j), j)
      end;
   end;
```

Quicksort

C'est l'une des méthodes de tris les plus efficaces en pratique (tri interne standard d'Unix) avec les caractéristiques suivantes:

1. Temps $O(n \log n)$ en moyenne (mais pas dans le pire des cas) avec une petite constante.
2. Tri sur place (espace constant)

Principe:

```
procedure QSort (g, d:  integer);
   var i:  integer;
   begin
   if g < d then
      begin
      v := a[d];
      partitionner le tableau autour de v
      et mettre v a sa bonne position m
      QSort (g, i − 1); QSort (i + 1, d);
      end;
   end;
```

Le programme (version de R. Sedgewick):

```
procedure QSort (g, d: integer);
 var i, j, t, v: integer;
 begin
  if g < d then begin
   v := a[d]; i := g - 1; j := d;
   repeat
    repeat i := i + 1
    until a[i] >= v;
    repeat j := j - 1
    until a[j] <= v;
    t := a[i]; a[i] := a[j];
    a[j] := t;
   until j <= i;
   a[j] := a[i]; a[i] := a[d];
   a[d] := t;
   QSort(g, i - 1);
   QSort(i + 1,d);
  end;
 end;
```

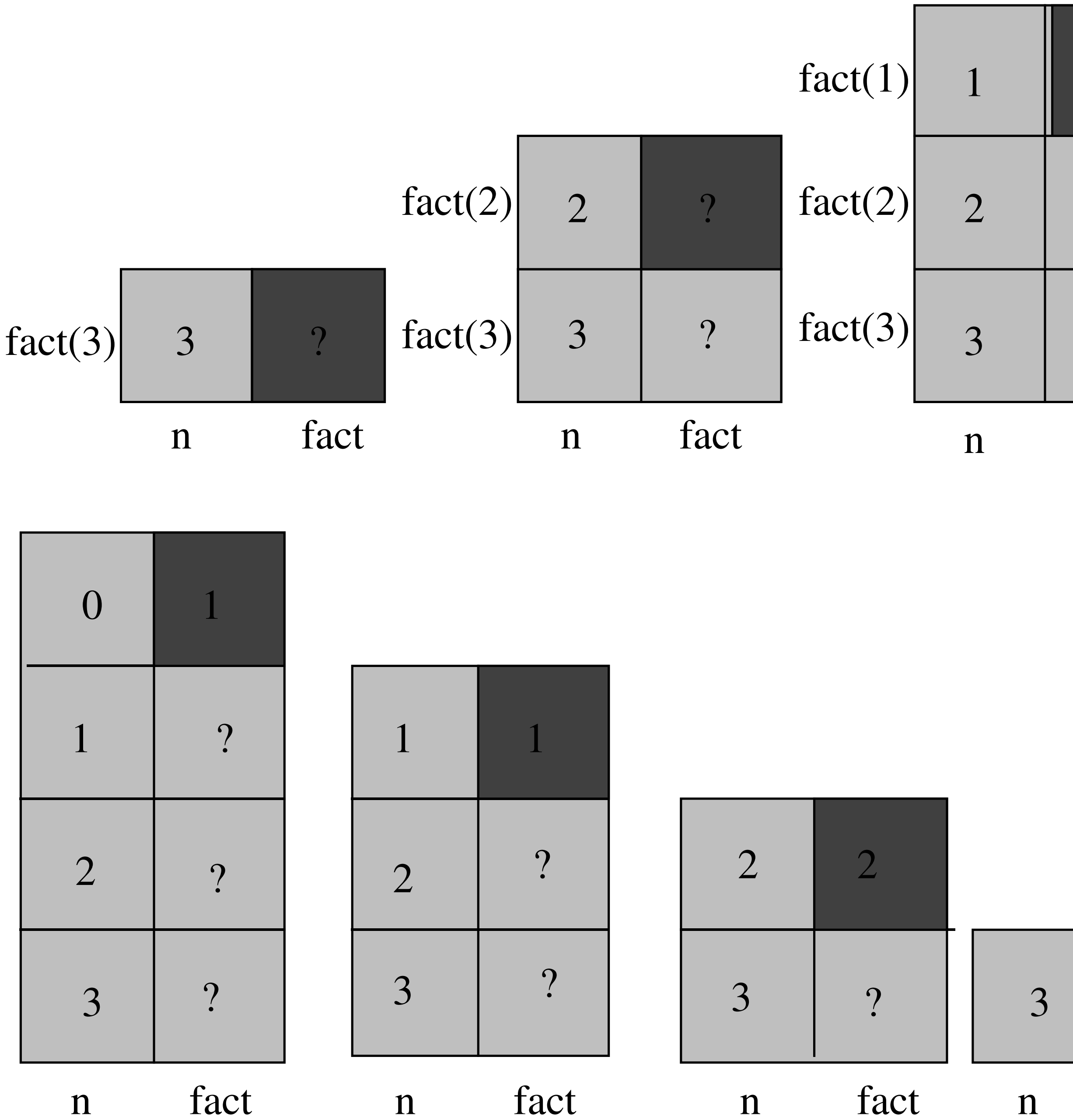

Figure 1: La pile d'execution

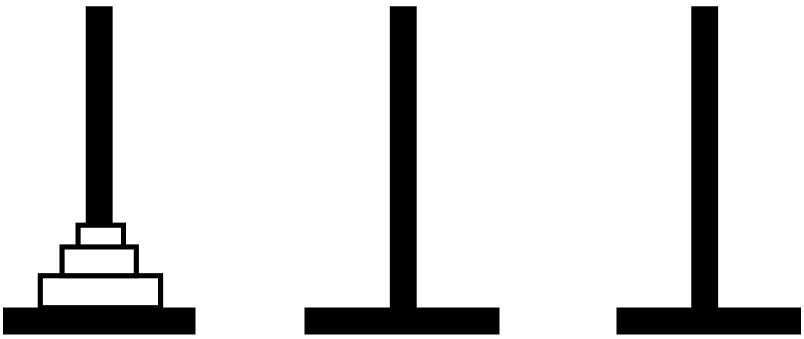

Figure 2: Le cas n=3