PC 3
Récursivité

- Appels de fonctions en C
- Appels récursifs
- Fractales
- Les tours de Hanoi
- Quicksort

Généralités

L'utilisation de la récursivité 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 $n! = n \times (n-1)!$:

```
int fact(int n ) {
  return (n == 0)?  1 : n * fact(n-1);
}
```

Calcul du pgcd de m et n

```
int pgcd (int m, int n) {
  return (m == 0)?  n : pgcd(n % m, m);
}
```

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

L'écriture en binaire

Pour écrire un entier x en binaire, on peut utiiser la formule récursive

$$bin(x) = (bin(q), r)$$

avec $x = 2q + r$. On en déduit facilement

```
void binaire(int x) {
  if (x>= 2) binaire(x/2);
  printf("%d",x%2);
}
```

Suites de Farey

On cherche à donner dans l'ordre la suite $F(N)$ des nombres rationnels $0 \leq p/q \leq 1$ tels que $p \leq N$. On utilise le type

```
typedef struct {
  int num; int den;
} rat;
```

La fonction suivante affiche les éléments de $F(N)$ dans $]x, y[$ pour $x = p/q, y = r/s$ avec $qr - ps = 1$.

```
void Farey(rat x, rat y) {
  rat m;
  m.num= x.num + y.num;
  m.den= x.den + y.den;
  if (m.den <= N) {
    Farey(x,m);
    printf("%d/%d,",m.num,m.den);
    Farey(m,y);
  }
}
```

Appels de fonctions en C

Un appel de fonction comme

```
y= Truc(x);
```

execute la fonction avec la *valeur* de x. Si on veut modifier la valeur de x, il faut passer à la fonction l'*adresse* de x. On la note `&x`. Inversement, si p est l'adresse de x, `*p` a pour valeur x.

Ainsi, pour échanger les valeurs des variables x et y, on utilise

```
void echange (int *px, int *py) {
  int t;
  t= *px; *px= *py; *py= t;
}
```

et on appelle `echange(&x,&y);`

Exemple

On va calculer un couple (a, b) d'entiers tels que $p = pgcd(m, n)$ s'écrive $p = am + bn$. On pose

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

On obtient:

$$p = br + (a + bq)m$$

```
void pgcd (int m, int n, int *pp, int *pa, int *pb) {
  int u,v;
  if (m == 0) {
    *pa = 0; *pb = 1;
    *pp= n;
  }
  else {
    pgcd(n % m, m, pp, &u, &v);
    *pa = v - u * (n / m); *pb = u;
  }
}
```

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

Suite de Fibonacci

Suite récurrente linéaire F_n définie par $F_0 = F_1 = 1$ et

$$F_n = F_{n-1} + F_{n-2}$$

On a $F_n \simeq \varphi^n$ avec $\varphi = (1 + \sqrt{5})/2$. Calcul récursif:

```
int Fibo(int n) {
 return (n <= 1)?
    1 : Fibo(n-1)+Fibo(n-2);
}
```

Temps de calcul:

$$T(n) = T(n-1) + T(n-2)$$

d'où

$$T(n) \simeq F_n$$

Calcul itératif:

```
int Fibo(int n) {
  int i, u, v;
  u = 1; v = 1;
  for (i = 2; i<= n; ++i) {
    v = u + v;
    u = v - u;
  }
  return v;
}
```

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:

```
int Ack (int m, int n) {
  if (m  == 0) return n+1;
  else if (n == 0) return Ack(m-1, 1);
  else return Ack (m-1, Ack(m, n-1));
}
```

Premières valeurs:

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

Fractales

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

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

```
void Dragon (int n, int x, int y,
                    int z, int t) {
  int u, v;
  if (n == 1) {
    MoveTo (x,y)
    LineTo (z, t);
  } else {
    u = (x + z + t - y) / 2;
    v = (y + t - z + x) / 2;
    Dragon (n - 1, x, y, u, v);
    Dragon (n - 1, z, t, u, v);
  }
}
```

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:

```
void Hanoi (int n, int i, int j) {
  if (n > 0) {
    Hanoi (n-1, i, 6 - (i+j));
    printf("%d->%d",i, j);
    Hanoi (n-1, 6 - (i+j), j);
  }
}
```

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:

```
void QSort (int g, int d) {
  int i;
  if (g < d) {
    v = a[d];
  {\it partitionner le tableau autour de v}
  {\it et mettre $v$ a sa bonne position $i$}
  QSort (g, i - 1); QSort (i + 1, d);
  }
}
```

Le programme (version de R. Sedgewick):

```
void QSort (int g, int d) {
  int i, j, t, v;
  if (g < d) {
    v = a[d]; i = g - 1; j = d;
    do {
      do
        ++i;
      while (a[i] < v);
      do
        --j;
      while (a[j] > v);
      t = a[i]; a[i] = a[j]; a[j] = t;
    } while (j > i);
    a[j] = a[i]; a[i] = a[d]; a[d] = t;
    QSort(g, i - 1);
    QSort(i + 1,d);
  }
}
```

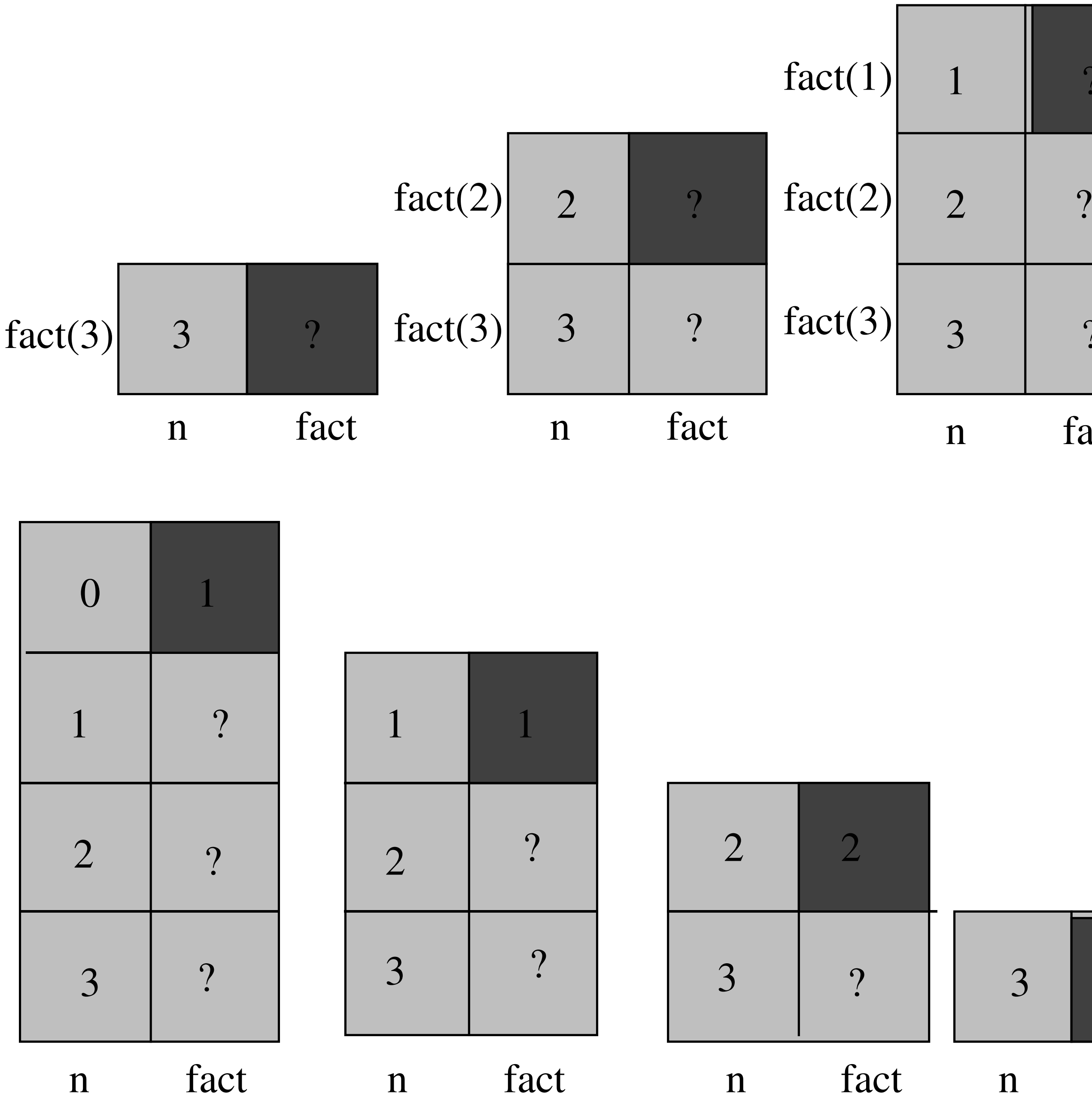

Figure 1: La pile d'execution

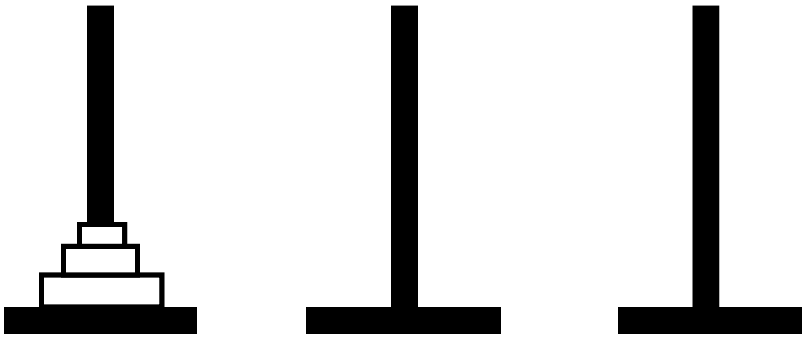

Figure 2: Le cas n=3