Algorithmique

Récursivité

Florent Hivert

Mél : Florent.Hivert@lri.fr

Adresse universelle : http://www.lri.fr/~hivert

Récursivité et Récurrence

Deux notions très proche :

- mathématiques : récurrence
- informatique : récursivité

De nombreuses définitions mathématiques sont récursives :

Définition (Peano)

- *0 est un entier naturel.*
- *Tout entier n a un successeur unique Sn $(= n + 1)$;*
- *Tout entier sauf 0 est le successeur d'un unique entier ;*
- *Pour tout énoncé $P(n)$ si $P(0)$ est vrai et si pour tout n, $P(n)$ implique $P(Sn)$ alors l'énoncé $\forall n : P(n)$ est vrai.*

Définition

Moyen simple et élégant de résoudre certain problème.

Définition

On appelle récursive toute fonction ou procédure qui s'appelle elle même.

```
Algorithme Fact
Entrée : un entier positif N
Sortie : factorielle de N

si N = 0 retourner 1
sinon retourner N x Fact(N-1)
```

Exemple dans un vrai langage de programmation

```
unsigned int fact(unsigned int N)
{
   if (N == 0) return 1;
   else        return N*fact(N-1);
}
```

Exemple dans un vrai langage de programmation

```
unsigned int fact(unsigned int N)
{
   if (N == 0) return 1;
   else        return N*fact(N-1);
}
```

Ça marche ! ! !

Comment ça marche ?

```
Appel à fact(4)
.   4*fact(3) = ?
.   Appel à fact(3)
.   .   3*fact(2) = ?
.   .   Appel à fact(2)
.   .   .   2*fact(1) = ?
.   .   .   Appel à fact(1)
.   .   .   .   1*fact(0) = ?
.   .   .   .   Appel à fact(0)
.   .   .   .   Retour de la valeur 1
.   .   .   .   1*1
.   .   .   Retour de la valeur 1
.   .   .   2*1
.   .   Retour de la valeur 2
.   .   3*2
.   Retour de la valeur 6
.   4*6
Retour de la valeur 24
```

Notion de pile d'exécution

Définition (Pile d'exécution)

*La **Pile d'exécution** (call stack) du programme en cours est un emplacement mémoire destiner à mémoriser les paramètres, les variables locales ainsi que l'adresse de retour de chaque fonction en cours d'exécution.*

Elle fonctionne selon le principe LIFO (Last-In-First-Out) : dernier entré premier sorti.

Attention ! La pile à une taille fixée, une mauvaise utilisation de la récursivité peut entraîner un débordement de pile (*stack overflow*).

Notion de pile d'exécution

Définition (Pile d'exécution)

*La **Pile d'exécution** (call stack) du programme en cours est un emplacement mémoire destiner à mémoriser les paramètres, les variables locales ainsi que l'adresse de retour de chaque fonction en cours d'exécution.*

Elle fonctionne selon le principe LIFO (Last-In-First-Out) : dernier entré premier sorti.

Attention ! La pile à une taille fixée, une mauvaise utilisation de la récursivité peut entraîner un débordement de pile (*stack overflow*).

Notion de pile d'exécution

Définition (Pile d'exécution)

*La **Pile d'exécution** (call stack) du programme en cours est un emplacement mémoire destiner à mémoriser les paramètres, les variables locales ainsi que l'adresse de retour de chaque fonction en cours d'exécution.*

Elle fonctionne selon le principe LIFO (Last-In-First-Out) : dernier entré premier sorti.

Attention ! La pile à une taille fixée, une mauvaise utilisation de la récursivité peut entraîner un débordement de pile (*stack overflow*).

Point terminal

Retenir

Comme dans le cas d'une boucle, il faut un cas d'arrêt où l'on ne fait pas d'appel récursif.

```
procédure récursive(paramètres):
    si TEST_D'ARRET:
        instructions du point d'arrêt
    sinon
        instructions
        récursive(paramètres changés);  // appel récursif
        instructions
```

Détail d'un appel de fonction

PA = Programme Appelant F = Fonction appelée

1. Le PA réserve et initialise les paramètres sur la pile
2. transfert du contrôle du PA à F avec enregistrement de l'**adresse de retour** sur la pile
3. réservation sur la pile des variables locales de F

Retenir

*L'ensemble : paramètres + adresse de retour + variables locales constitue le **Tableau d'activation (Stack Frame)** de F*

4. exécution du code de F dans son TA jusqu'à `return`
5. désallocation du TA de F de la pile
6. retour au PA à l'adresse enregistrée à l'étape 2, avec transmission de la **valeur de retour** dans le cas échéant

La récursivité terminale

Définition

On dit qu'un fonction est récursive terminale, si tout appel récursif est de la forme `return f(...)`

Autrement dit, la valeur retournée est directement la valeur obtenue par un appel récursif, sans qu'il n'y ait aucune opération sur cette valeur. Il n'y a ainsi rien à retenir sur la pile.

```
Entrée : Entiers positifs n, a
Sortie : a*n!

    si n == 0 retourner a
    sinon     retourner Algo(n-1,n*a)
```

La récursivité terminale

Définition

On dit qu'un fonction est récursive terminale, si tout appel récursif est de la forme `return f(...)`

Autrement dit, la valeur retournée est directement la valeur obtenue par un appel récursif, sans qu'il n'y ait aucune opération sur cette valeur. Il n'y a ainsi rien à retenir sur la pile.

```
Entrée : Entiers positifs n, a
Sortie : a*n!

    si n == 0 retourner a
    sinon     retourner Algo(n-1,n*a)
```

La récursivité terminale (2)

Idée : Il n'y a rien à retenir sur la pile.

Retenir

Quand une fonction est récursive terminale, on peut transformer l'appel récursif en une boucle, sans utilisation de mémoire supplémentaire.

Attention ! cette optimisation n'est pas supportée par tous les compilateurs et est optionnelle (ex : -O3 avec gcc).

```
si n == 0 retourner a
sinon     retourner Algo(n-1,n*a)
```

Devient :

```
Tant que n <> 0:
    a <- n*a; n <- n-1
retourner a
```

La récursivité terminale (2)

Idée : Il n'y a rien à retenir sur la pile.

Retenir

Quand une fonction est récursive terminale, on peut transformer l'appel récursif en une boucle, sans utilisation de mémoire supplémentaire.

Attention ! cette optimisation n'est pas supportée par tous les compilateurs et est optionnelle (ex : `-O3` avec `gcc`).

```
si n == 0 retourner a
sinon     retourner Algo(n-1,n*a)
```

Devient :

```
Tant que n <> 0:
    a <- n*a; n <- n-1
retourner a
```

La récursivité terminale (3)

L'optimisation peut se faire appel par appel.

Retenir

Quand une appel est récursive terminal, on peut le transformer en un saut, sans utilisation de mémoire supplémentaire.

Exemple : le tri rapide

```
tri_rapide(tableau T, entier premier, entier dernier)
    si premier < dernier alors
        pivot := choix_pivot(T, premier, dernier)
        pivot := partitionner(T, premier, dernier, pivot)
        tri_rapide(T, premier, pivot-1)  // Non terminal
        tri_rapide(T, pivot+1, dernier)  // terminal
```