Récursivité des actions [rc]
Support de Cours

Karine Zampieri, Stéphane Rivière

Unisciel algoprog Version 21 mai 2018

Table des matières

alg - Récursivité des actions (Cours)

Mots-Clés Récursivité des actions ■
Requis Algorithmes paramétrés, Preuve et Notations asymptotiques ■
Difficulté ●●○ (2 h) ■

Introduction

La **récursivité** est un des concepts de programmation les plus importants. Le principe de l'approche récursive est le suivant : ramener le problème à résoudre à un sous-problème correspondant à une instance « réduite » du problème lui-même. Approcher un problème de cette façon, un sous-programme s'appelant lui-même, requiert quelque expérience. Les approches récursives se trouvent dans les parcours d'arbres, les recherches en largeur d'abord et en profondeur d'abord dans les graphes et les tris.

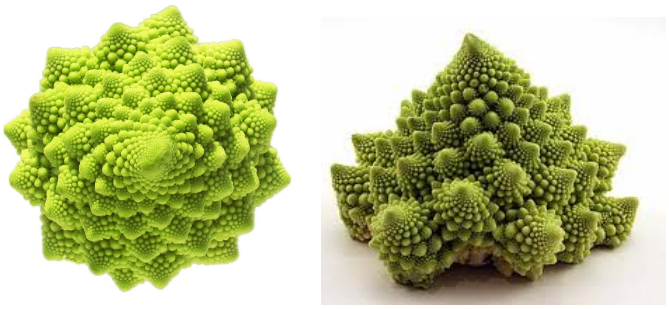
Ce module définit la récursion et les types de récursivité, donne les propriétés des algorithmes récursifs, présente des exemples classiques d'algorithmes récursifs, analyse le coût de la récursivité et décrit la récursivité terminale qui est une forme de récursion pour laquelle les compilateurs modernes savent la reconnaître et produire un code optimisé.

1 Définition et exemples

Éthymologie
Le mot « récursion » vient du latin *recursare* qui signifie *courir en arrière, revenir*. L'idée de réapparition, de retour, est donc étymologiquement liée à la récursivité.

Système récursif
Lorsqu'un système contient une **autoréférence** (ou une **copie de lui-même**), on dit que ce système est **récursif**.

Exemple
Dans le domaine végétal, le romanesco (hybride broccolo/choux-fleur) est très récursif.



Exemple
L'effet droste (image en abyme).
http://images.math.cnrs.fr/L-effet-Droste.html

Exemple
Les poupées russes ou *matriochkas* sont des séries de poupées de tailles décroissantes placées les unes à l'intérieur des autres. Une poupée russe est donc :

- Soit une poupée « pleine » (qu'on ne peut ouvrir).
- Soit une poupée « vide » contenant une poupée russe.

http://fr.wikipedia.org/wiki/Poup%C3%A9e_russe

Exemple

Une façon simple de définir la *relation de parenté* entre deux personnes est d'énoncer que x et y sont parents :

- Soit si y est père, mère, fils ou fille de x (ou époux si on inclut la parenté par alliance).
- Soit s'il existe un individu z tel que x est parent de z et z est parent de y.

Exprimer la même définition de façon non récursive obligerait à recourir à la notion de « chaîne de parenté » et à introduire dans la définition la longueur d'une telle chaîne.

Conclusion

La récursivité est un concept très puissant quand on sait décomposer un problème en un ou plusieurs sous-problèmes qui sont de même nature, mais qui s'appliquent à un nombre d'objets plus réduit. Il en est de même quand on sait décomposer un objet en groupes de composants qui présentent les mêmes propriétés que l'objet lui-même et qui ne contiennent qu'un sous-ensemble de l'objet.

Définition récursive, Algorithme récursif

Une **définition récursive** est une définition dans laquelle intervient ce que l'on veut définir. Un **algorithme récursif** est un algorithme défini en fonction de lui-même. Par conséquent :

Module récursif

Une procédure (ou fonction) `P` est dite **récursive** si son exécution peut provoquer un ou plusieurs appels (dits **récursif**) à `P`.

2 Types de récursivité

2.1 Récursivité simple

Récursivité simple
Contient un seul appel récursif à `P` dans le corps d'une procédure récursive `P`.

Exemple
Un escalier de hauteur h c'est :

- (Vision itérative) Une séquence de h marches
- (Vision récursive) Une marche suivie d'un escalier de hauteur $h-1$

Algorithme monterEscalier

```
Action monterEscalier ( h : Entier )
Début
  |  Si ( h > 0 ) Alors
  |    |  monterMarche ( )
  |    |  monterEscalier ( h - 1 )
  |  FinSi
Fin
```

2.2 Récursivité multiple

Récursivité multiple

Une **récursivité** est **multiple** si il y a plusieurs appels récursifs à `P` dans le corps d'une procédure récursive `P`.

Exemple

La suite de FIBONACCI est définie par (n entier naturel) :

$$\begin{cases} f_0 = 0; f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \text{ pour } n \geq 1 \end{cases}$$

((alg)) **Algorithme fib**

```
Fonction fib ( n : Entier ) : Entier
Début
  |  Si ( n = 0 )
  |    |  Retourner ( 0 )
  |  Sinon Si ( n = 1 )
  |    |  Retourner ( 1 )
  |  Sinon
  |    |  Retourner ( fib ( n - 1 ) + fib ( n - 2 ) )
  |  FinSi
Fin
```

Exécution de fib(3)

```
fib(3) = fib(2)+fib(1)
= (fib(1)+fib(0))+fib(1)
= (1+0)+1
= 2
```

2.3 Récursivité mutuelle

Récursivité mutuelle

Une **récursivité** est **mutuelle** ou *croisée* quand une procédure `P` appelle une procédure `Q` qui déclenche un appel récursif à `P`.

Remarque

La situation est obligatoirement symétrique, puisque `Q` déclenchera un appel de `P`, qui déclenchera à son tour un appel de `Q`.

Exemple

La parité d'un entier naturel n peut être définie par :

$$\text{pair}(n) = \begin{cases} vrai & \text{si } n = 0 \\ \text{impair}(n-1) & \text{sinon} \end{cases}$$

$$\text{impair}(n) = \begin{cases} faux & \text{si } n = 0 \\ \text{pair}(n-1) & \text{sinon} \end{cases}$$

Algorithmes pair, impair

```
Fonction pair ( n : Entier ) : Booléen
Début
  |  Si ( n = 0 ) Alors
  |    |  Retourner ( Vrai )
  |  Sinon
  |    |  Retourner ( impair ( n - 1 ) )
  |  FinSi
Fin

Fonction impair ( n : Entier ) : Booléen
Début
  |  Si ( n = 0 ) Alors
  |    |  Retourner ( Faux )
  |  Sinon
  |    |  Retourner ( pair ( n - 1 ) )
  |  FinSi
Fin
```

Exécution de pair(3)

Les fonctions `pair` et `impair` s'invoquent mutuellement :

```
pair(3)
-> impair(2)
-> pair(1)
-> impair(0)
```

La dernière invocation renvoie `Faux` :

```
impair(0)=Faux
-> pair(1)=Faux
-> impair(2)=Faux
-> pair(3)=Faux
-> Faux
```

2.4 Récursivité imbriquée

Récursivité imbriquée
Une **récursivité** est **imbriquée** si une procédure récursive `P` contient un appel imbriqué.

Exemple
La fonction d'Ackermann (n et p entiers naturels) est définie comme suit :

$$A(n,p) = \begin{cases} p+1 & \text{si } n = 0 \\ A(n-1,1) & \text{si } n > 0 \text{ et } p = 0 \\ A(n-1,A(n,p-1)) & \text{sinon} \end{cases}$$

Fonction ackermann

```
Fonction ackermann ( n , p : Entier ) : Entier
Début
  |  Si ( n = 0 ) Alors
  |    |  Retourner ( p + 1 )
  |  Sinon
  |    |  Si ( p = 0 ) Alors
  |    |    |  Retourner ( ackermann ( n - 1 , 1 ) )
  |    |  Sinon
  |    |    |  Retourner ( ackermann ( n - 1 , ackermann ( n , p - 1 ) )
  |    |  FinSi
  |  FinSi
Fin
```

@[rcackermannA1.alg]

2.5 Récursivité terminale

Récursivité terminale

La **récursivité** est **terminale** si l'appel récursif est la dernière instruction et elle est isolée.

Exemple

L'addition de deux entiers positifs ou nuls peut être définie comme suit :

((alg)) **Fonction plus**

```
Fonction plus ( a , b : Entier ) : Entier
Début
  |  Si ( b = 0 ) Alors
  |    |  Retourner ( a )
  |  Sinon
  |    |  Retourner ( plus ( a + 1 , b - 1 ) )
  |  FinSi
Fin
```

Exécution de plus(4,2)

```
plus(4,2)
= plus(5,1)
= plus(6,0)
= 6
```

Attention

La récursivité **n'est pas terminale** si l'appel récursif n'est pas la dernière instruction et/ou elle n'est pas isolée (c.-à-d. qu'elle fait partie d'une expression).

Exemple

Retour sur l'addition.

((alg)) **Fonction plus (non terminale)**

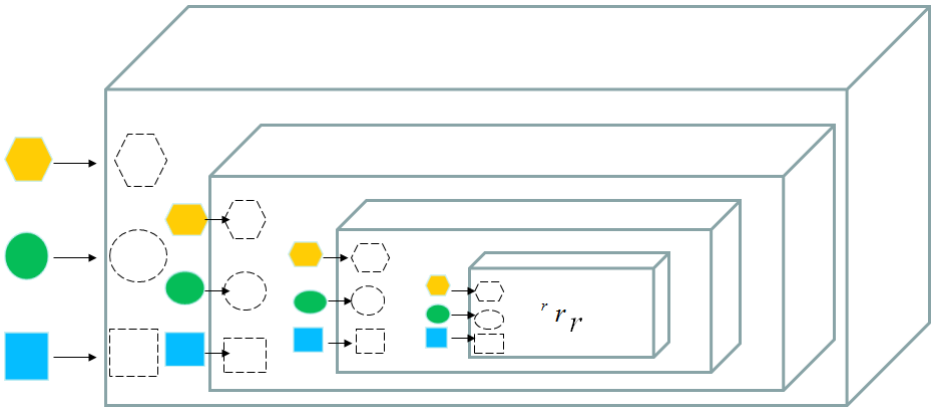
```
Fonction plus ( a , b : Entier ) : Entier
Début
  |  Si ( b = 0 )
  |    |  Retourner ( a )
  |  Sinon
  |    |  Retourner ( 1 + plus ( a , b - 1 ) )
  |  FinSi
Fin
```

```
plus(4,2) = 1+plus(4,1) = 1+1+plus(4,0) = 1+1+4=6
```

3 Algorithme récursif

Algorithme récursif

Un algorithme est dit **récursif** si l'expression qui le définit fait appel à lui-même. On qualifiera de récursif, un appel à une fonction f provoqué par l'évaluation d'un autre appel à f.



Critères pour un bon algorithme récursif

Deux règles sont à respecter impérativement :

Règle 1 : Un algorithme récursif doit être défini par une expression conditionnelle dont l'un au moins des cas mène à une expression évaluable sans appel récursif. Une telle expression est appelée **condition d'arrêt** ou **test d'arrêt** ou **clause terminale** ou **cas de base** ou encore **cas trivial**.

Règle 2 : Il faut s'assurer que pour toute valeur du (ou des) paramètre(s), il suffira d'un **nombre fini** d'appels récursifs pour atteindre la condition d'arrêt.

Remarque

Ces deux règles assurent la terminaison de tout appel à un algorithme récursif en empêchant la possibilité d'une infinité d'appels récursifs.

Schéma général d'un algorithme récursif

```
Action R( données X ; résultats Y )
Début
  Si terminaison( X ) Alors
    Y <-...
  Sinon
    ...
    R( entrée_réduite ,...)
    ...
  FinSi
Fin
```

Schéma général d'une fonction récursive

```
Fonction F( a1 : T1 ; a2 : T2...) : T
Variable rs : T
Début
  Si terminaison( a1 , a2 ,...) Alors
    rs <-...
  Sinon
    y1 <-...
    y2 <-...
    ...
    rs =...f( y1 , y2 ,...)
    ...
  FinSi
  Retourner ( rs )
Fin
```

4 Ordre des appels récursifs

Illustrons l'importance de l'ordre des appels récursifs en prenant l'affichage des entiers.

4.1 Entiers par ordre décroissant

Considérons la procédure suivante :

((alg)) **Procédure decroissant**

```
Action decroissant ( n : Entier )
Début
  |  Si ( n > 0 ) Alors
  |    |  Afficher ( n )
  |    |  decroissant ( n - 1 )
  |  FinSi
Fin
```

Exécution pour n=2

```
Appel de decroissant(2)
> affichage de 2
> appel de decroissant(1)
> > affichage de 1
> > appel de decroissant(0)
> > > l'algorithme ne fait rien
```

4.2 Entiers par ordre croissant

Invertissons l'ordre de l'affichage et de l'appel récursif :

((alg)) **Procédure croissant**

```
Action croissant ( n : Entier )
Début
  |  Si ( n > 0 ) Alors
  |    |  croissant ( n - 1 )
  |    |  Afficher ( n )
  |  FinSi
Fin
```

Exécution pour n=2

```
Appel de croissant(2)
> appel de croissant(1)
> > appel de croissant(0)
> > > l'algorithme ne fait rien
```

```
> > affichage de 1
> affichage de 2
```

4.3 Entiers par ordre croissant puis décroissant

La procédure suivante affiche les entiers par ordre croissant puis par ordre décroissant :

((alg)) **Procédure cdcroissant**

```
Action cdcroissant ( n : Entier )
Début
  |  croissant ( n )
  |  decroissant ( n )
Fin
```

Explication
Elle se contente d'appeler la procédure récursive `croissant(n)` suivie de la procédure récursive `decroissant(n)`.

4.4 Entiers par ordre décroissant puis croissant

De même, la procédure suivante affiche les entiers par ordre décroissant puis par ordre croissant :

((alg)) **Procédure dccroissant**

```
Action dccroissant ( n : Entier )
Début
  |  decroissant ( n )
  |  croissant ( n )
Fin
```

La procédure étant terminale, elle se simplifie en :

((alg)) **Procédure dccroissant (simplifiée)**

```
Action dccroissant0 ( n : Entier )
Début
  |  Si ( n > 0 ) Alors
  |     |  Afficher ( n )
  |     |  dccroissant0 ( n - 1 )
  |     |  Afficher ( n )
  |  FinSi
Fin
```

Exécution pour n=2

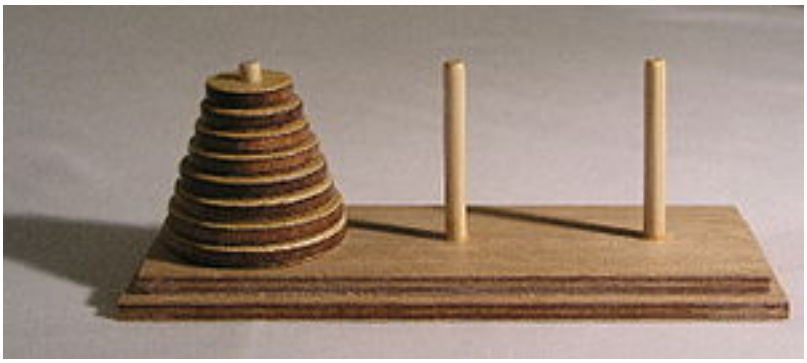
```
Appel de dccroissant(2)
> affichage de 2
> appel de dccroissant(1)
> > affichage de 1
> > appel de dccroissant(0)
> > > l'algorithme ne fait rien
> > affichage de 1
> affichage de 2
```

5 Exemple : Les tours de Hanoï

5.1 Le problème

Contrairement à ce que son nom suggère, le casse-tête appelé les **tours de Hanoï**[1] n'est pas d'origine asiatique. Il a été inventé à la fin du XIX^{e} siècle, en 1883 exactement, par Edouard Lucas, un mathématicien français spécialiste des jeux.

Des disques percés sont empilés sur un premier piquet (par exemple ici le premier, celui le plus à gauche). Ils sont placés dans l'ordre des diamètres croissants depuis le haut jusqu'au bas. On doit les enlever un à un pour les replacer dans la même position sur un second piquet (par exemple le dernier, celui le plus à droite). Pour cela, on dispose d'un troisième piquet auxiliaire (celui du centre) qui peut recevoir provisoirement les disques.



Le transfert des disques doit respecter les trois règles suivantes :

- On ne peut déplacer que le disque se trouvant au sommet d'un piquet.
- On ne peut déplacer qu'un seul disque à la fois.
- Lors d'un déplacement, il est interdit de poser un disque sur un disque plus petit.

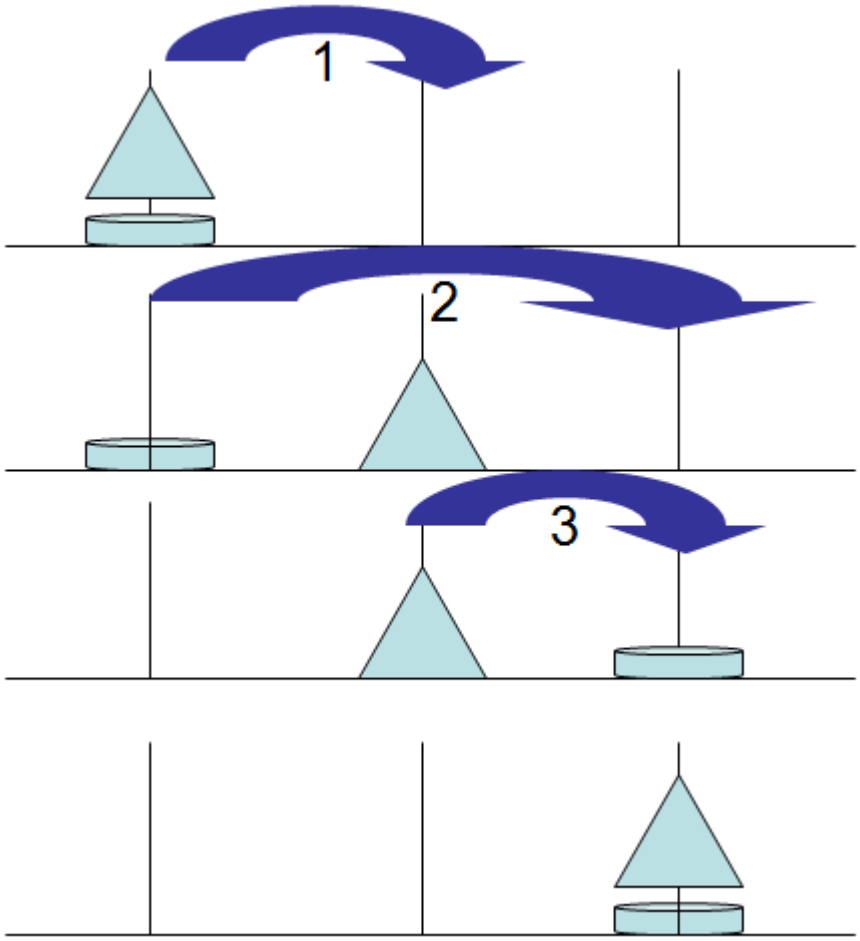
5.2 Résolution

Hypothèse
On suppose que l'on sait résoudre le problème pour $n-1$ disques.

Principe
Pour déplacer n disques de la tige A vers la tige C :

1. On déplace les $(n-1)$ plus petits de la tige A vers la tige B.
2. On déplace le plus gros disque de la tige A vers la tige C.
3. On déplace les $(n-1)$ plus petits de la tige B vers la tige C.

1. Voir l'article de Wikipedia, « Tours de Hanoï »



Validité

Les règles sont respectées puisque le plus gros disque est toujours en « bas » d'une tige et que l'hypothèse (de récurrence) nous assure que nous pouvons déplacer la « pile » de $(n-1)$ disques en respectant les règles.

5.3 Algorithme et Complexité

L'algorithme suivant déplace `n` disques de la tour `orig`(ine) vers la tour `dest`(ination) en passant par la tour `inter`(médiaire).

((alg)) **Algorithme hanoi**

```
Action hanoi ( n : Entier ; orig , dest , inter : Chaîne )
Début
  |  Si ( n > 0 ) Alors
  |     |  hanoi ( n - 1 , orig , inter , dest )
  |     |  deplacer ( orig , dest )
  |     |  hanoi ( n - 1 , inter , dest , orig )
  |  FinSi
Fin
```

Complexité

On compte le nombre de déplacements de disques effectués par l'algorithme invoqué sur n disques :

$$C(n) = \begin{cases} 0 & \text{si } n = 0 \\ C(n-1) + 1 + C(n-1) = 2C(n-1) + 1 & \text{sinon} \end{cases}$$

d'où :

$$\begin{aligned} C(n) &= 2C(n-1) + 1 \\ &= 2(2C(n-2) + 1) + 1 = 2^2 C(n-2) + (2+1) \\ &= 2^2(2C(n-3) + 1) + (2+1) = 2^3 C(n-3) + (2^2 + 2 + 1) \\ &\vdots \\ &= 2^n C(0) + (2^{n-1} + 2^{n-2} + \ldots + 2 + 1) \\ &= 2^n - 1 \end{aligned}$$

((alg)) **Algorithme deplacer**

```
Action deplacer ( pieu1 , pieu2 : Chaîne )
Début
  |  ncoups <- ncoups + 1
  |  Afficher ( ncoups , " -> deplacer de " , pieu1 , " sur " , pieu2 )
Fin
```

5.4 Programme principal

L'algorithme principal est le suivant :

((alg)) **Algorithme**

```
Variable ncoups : Entier
Algorithme pghanoi
Variable n : Entier
Début
  |  Afficher ( "Nombre de disques? " )
  |  Saisir ( n )
  |  ncoups <- 0
  |  hanoi ( n , "A" , "C" , "B" )
Fin
```

Résultat d'exécution

(Avec trois disques)

```
Nombre de disques? 3
1 -> deplacer de A sur C
2 -> deplacer de A sur B
```

```
3 -> deplacer de C sur B
4 -> deplacer de A sur C
5 -> deplacer de B sur A
6 -> deplacer de B sur C
7 -> deplacer de A sur C
```

6 Coût de la récursivité

6.1 Arborescences d'appels

Arbre d'exécution

Soit une procédure récursive `P` ne possédant que des appels récursifs simples, en nombre `n`, de la forme suivante :

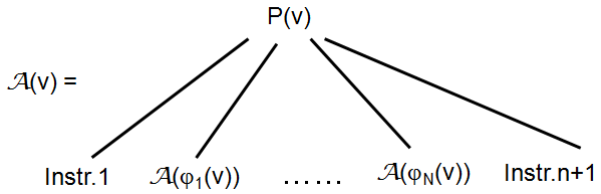
```
Action P ( v : T )
Début
  Si terminaison ( v ) Alors
    Instructions0
  Sinon
    Instructions1
    P ( phi1 ( v ) )
    ...
    P ( phiN ( v ) )
    InstructionsN + 1
  FinSi
Fin
```

A tout appel `P(v)| de \lstinline`P@, on peut associer un arbre $\mathcal{A}(v)$ appelé, **arbre d'exécution** de `P` pour l'appel `P(v)`, défini récursivement comme suit :

- Si `terminaison(v)` est vrai, $\mathcal{A}(v)$ est un arbre réduit à sa racine :

$$\mathcal{A}(v) = \text{Instructions0}$$

- Si `terminaison(v)` n'est pas vérifiée, $\mathcal{A}(v)$ est égal à :



Remarque

Si une procédure récursive `P` présente une récursivité croisée avec une procédure récursive `Q`, il faut définir récursivement et en parallèle, les arbres d'exécution associés à `P` et `Q` pour des appels donnés.

Exemple

L'exemple classique est celui de la définition récursive de la suite de Fibonacci :

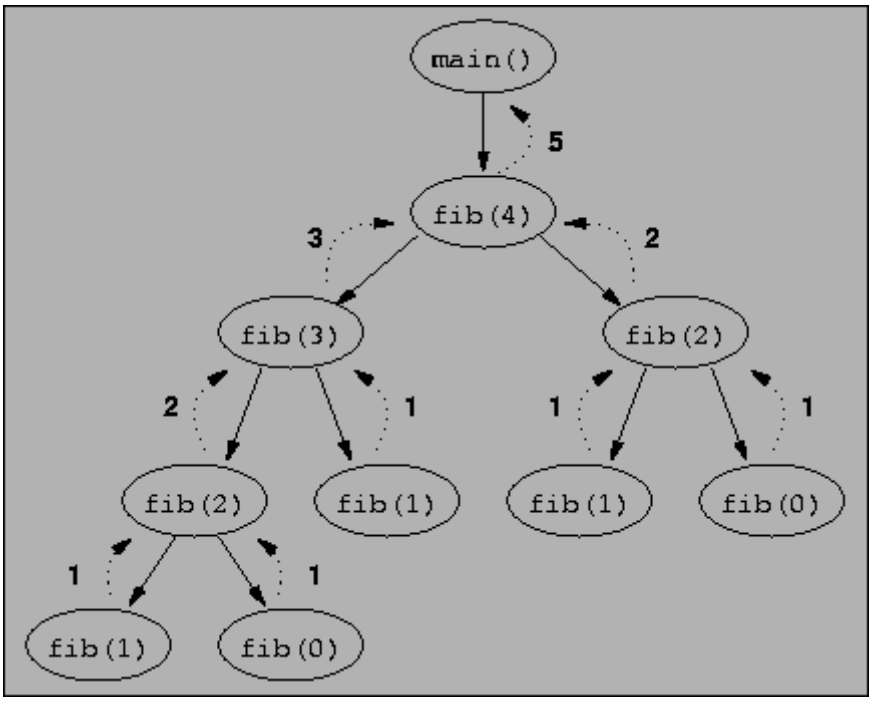
Fonction fib

```
Fonction fib ( n : Entier ) : Entier
Début
  |  Si ( n = 0 )
  |    |  Retourner ( 0 )
```

```
  |  Sinon Si ( n = 1 )
  |    |  Retourner ( 1 )
  |  Sinon
  |    |  Retourner ( fib ( n - 1 ) + fib ( n - 2 ) )
  |  FinSi
Fin
```

Arbre d'invocation de fib(4)

La figure représente l'arbre des invocations de `fib(4)`.



6.2 Coût de la récursivité

Le coût de la récursivité est lié à sa réalisation pratique.

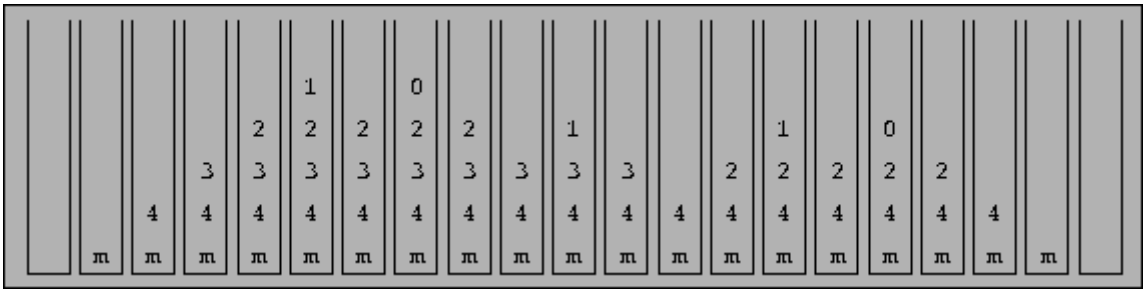
Dans le module @[Algorithmes paramétrés], nous avons vu comment se fait l'allocation de mémoire pour chaque appel de procédure ou de fonction : chaque appel entraîne la création d'une « assiette », on dit aussi *changements de contexte* – qui décrit les objets locaux et les paramètres de la procédure ou fonction –, qui est liée à cet appel et qui ne cesse d'exister que quand cet appel particulier se termine.

Si une procédure est appelée récursivement, cela signifie qu'un nouvel appel se produit avant le retour du précédent. Il peut donc exister simultanément plusieurs « assiettes » différentes pour la même procédure et par conséquent plusieurs « instances » (« incarnations ») différentes du même modèle d'« assiette ».

Il n'y a aucun rapport, ni aucune communication entre les « assiettes » en dehors du passage de paramètres : ceci signifie que la désignation d'un objet local à une procédure ou fonction récursive n'est pas ambiguë.

Quand un appel récursif se termine, les objets correspondants à l'appel précédent redeviennent accessibles, mais ils n'ont pas cessé d'exister entre temps car ils ont été empilés.

Exemple
La figure représente les états successifs de la pile d'exécution, pour une invocation de `fib(4)` dans le programme principal `main` (pour alléger, les cadres d'invocation de `main`, `fib(4)`, `fib(3)`, etc., y sont désignés par m, 4, 3, etc.). On remarquera que certaines invocations sont exécutées plusieurs fois : `fib(2)` est exécutée deux fois, `fib(1)` trois fois, `fib(0)` deux fois.



Remarque
On doit sauvegarder momentanément une « assiette » puis la restaurer. La restauration se fait dans l'ordre inverse de la sauvegarde.

Remarque
Dans une pile, les objets sont dépilés dans l'ordre inverse de celui où ils ont été empilés. C'est pourquoi on utilise la pile pour gérer les algorithmes récursifs.

Remarque
Le coût de la récursivité est lié au coût de cet empilement et dépilement des « assiettes ».

Conclusion

Cette analyse montre l'intérêt d'avoir des algorithmes récursifs avec :

- une profondeur de récursivité (hauteur de pile) faible (si elle est en n^2, l'algorithme n'est pas utilisable de façon pratique),
- un nombre d'appels raisonnable.

7 Récursivité terminale

7.1 Rappel de définition

La récursivité terminale est une notion qui peut améliorer nettement les performances de vos algorithmes. En effet, l'exécution d'un algorithme utilisant la récursivité terminale est transformée en général en algorithme itératif (plus rapide et moins gourmand en mémoire) par le compilateur.

Algorithme récursif terminal
Un algorithme est **récursif terminal** si son appel est le dernier.
Une fonction est **récursive terminale** si elle renvoie, sans autre calcul, la valeur obtenue par son appel récursif.

Autrement dit, la valeur retournée est directement la valeur obtenue par l'invocation récursive, sans qu'il n'y ait d'opération sur cette valeur.

7.2 Exemple : Factoriel récursif terminal

Considérons le calcul récursif du factoriel :

((alg)) **Fonction factoriel** (Factoriel récursif)

```
Fonction factoriel ( n : Entier ) : Entier
Début
  |  Si ( n <= 0 ) Alors
  |    |  Retourner ( 1 )
  |  Sinon
  |    |  Retourner ( n * factoriel ( n - 1 ) )
  |  FinSi
Fin
```

Explication
Cette invocation **n'est pas terminale**, puisqu'il y a multiplication par `n` avant de retourner. Par contre, l'invocation récursive suivante l'est :

((alg)) **Fonction factorielRt** (Factoriel récursif terminal)

```
Fonction factorielRt ( n : Entier ) : Entier
Début
  |  Retourner ( facRt ( n , 1 ) )
Fin
Fonction facRt ( n : Entier ; a : Entier ) : Entier
Début
  |  Si ( n <= 0 ) Alors
  |    |  Retourner ( a )
  |  Sinon
  |    |  Retourner ( facRt ( n - 1 , a * n ) )
  |  FinSi
Fin
```

Explication
Dans cette version, le deuxième paramètre `a`, qui vaut initialement 1, joue le rôle d'un accumulateur. L'évaluation de `facRt(5,1)` conduit à la suite d'invocations :

```
facRt(5,1) -> facRt(4,5) -> facRt(3,20) -> facRt(2,60) -> facRt(1,120)
```

dont la suite de retours :

```
facRt(1,120)=120
-> facRt(2,60)=120
-> facRt(3,20)=120
-> facRt(4,5)=120
-> facRt(5,1)=120
-> 120
```

est en fait une suite d'égalités :

```
facRt(5,1) = facRt(4,5) = facRt(3,20) = facRt(2,60) = facRt(1,120) = 120
```

7.3 Fonctionnement de la récursion terminale

Lorsqu'un compilateur détecte un appel récursif terminal, il réutilise le contexte d'exécution courant au lieu d'en empiler un nouveau. Cela est rendu possible par le fait que l'appel récursif est la dernière instruction exécutée dans l'activation courante et qu'il ne reste donc rien à faire dans celle-ci après le retour de cet appel : par conséquent, il n'y a pas de raison de conserver l'activation courante.

En remplaçant son enregistrement au lieu d'en empiler un autre au-dessus de lui, l'utilisation de la pile est considérablement réduite, ce qui, en pratique, se traduit par de meilleures performances.

Conclusion
On doit donc transformer les fonctions récursives en fonctions récursives terminales à chaque fois que cela est possible.

Remarque
La plupart des langages actuels exécutent un programme à récursivité terminale comme s'il était itératif, c'est-à-dire en espace constant. Sinon, il est facile de transformer une définition récursive terminale en itération pour optimiser l'exécution. La fonction dérécursivée du factoriel est :

((alg)) **Factoriel itératif**

```
Fonction facIter ( n : Entier ) : Entier
Variable a : Entier
Début
  |  a <- 1
  |  TantQue ( n > 0 ) Faire
  |     |  a <- n * a
  |     |  n <- n - 1
  |  FinTantQue
  |  Retourner ( a )
Fin
```

8 Compléments

Tous les algorithmes récursifs ont en commun un certain nombre de caractéristiques qu'ils partagent d'ailleurs avec les définitions récursives et les structures de données récursives.

8.1 Récursivité directe ou indirecte

La récursivité peut être directe ou indirecte. Les déclarations suivantes comprennent une récursivité indirecte :

```
Action P( x :...)
Début
  |  ...Q( f( x ) )...
Fin
Action Q( y :...)
Début
  |  ...P( g( y ) )...
Fin
```

Il faut en retenir que le fait qu'un algorithme soit récursif n'est pas évident à la lecture du texte de l'algorithme.

8.2 Terminaison

Les traitements engendrés par une définition récursive doivent être finis pour que le calcul puisse se terminer. Un algorithme `P` récursif doit prendre la forme générale suivante :

```
Action P ( x :...)
Début
  Si B Alors
    C
  Sinon
    D ( P )
  FinSi
Fin
```

La condition `B` et les instructions `C` sont évalués directement, sans récursivité.

Remarque

De même que pour un schéma itératif, on peut démontrer l'achèvement d'un schéma récursif en définissant, comme quantité de contrôle, une fonction entière N de l'ensemble des variables, et en montrant que chaque exécution de `P` fait décroître N. Ceci revient à montrer que les appels récursifs tendent vers un certain but. La façon la plus commode, quand elle est possible, consiste à donner à `P` un paramètre entier `n`, appeler récursivement `P` avec la valeur `n-1` comme paramètre effectif et à utiliser pour `B` la condition `n>0`. Il vient alors :

```
Action P ( n ,...)
Début
  Si ( n > 0 ) Alors
    C
  Sinon
    D ( P ( n - 1 ,...) )
  FinSi
Fin
```

La valeur initiale de `n` détermine alors la **profondeur** de la récursivité, ce qui correspond au nombre maximum d'appels récursifs du sous-programme.

Remarque

Dans le cas général, la situation n'est pas aussi favorable et il n'est pas toujours facile d'exhiber une quantité de contrôle associée au sous-programme.

Exemple

L'algorithme suivant, nommée aussi fonction de Syracuse, est bien défini et vaut 1 sur les entiers naturels $\mathbb{N}$.

Fonction collatz

```
Fonction collatz ( n : Entier ) : Entier
Début
  |  Si ( n <= 1 ) Alors
  |    |  Retourner ( 1 )
```

```
  |  Sinon
  |    |  Si ( Modulo ( n , 2 ) = 1 ) Alors
  |    |    |  Retourner ( collatz ( 3 * n + 1 ) )
  |    |  Sinon
  |    |    |  Retourner ( collatz ( DivEnt ( n , 2 ) ) )
  |    |  FinSi
  |  FinSi
Fin
```

8.3 Non décidabilité de la terminaison

Peut-on écrire un programme qui vérifie automatiquement si un programme donné `P` termine quand il est exécuté sur un jeu de données `D` ?

Entrée Un programme `P` et un jeu de données `D`

Sortie `Vrai` si le programme `P` termine sur le jeu de données `D`, et `Faux` sinon

Preuve : (de la non décidabilité) Supposons qu'il existe un tel programme, nommé `termine`, de vérification de la terminaison. A partir de ce programme, on conçoit le programme `Q` suivant :

```
Algorithme Q
Variable rs : Booléen
Début
  |  rs <- termine ( Q )
  |  TantQue ( rs ) Faire
  |    |  attendre une seconde
  |    |  rs <- termine ( Q )
  |  FinTantQue
  |  Retourner ( rs )
Fin
```

Supposons que le programme `Q` – qui ne prend pas de paramètre – termine. Donc `termine(Q)` renvoie `Vrai`, la deuxième instruction de `Q` boucle indéfiniment et `Q` ne termine pas : contradiction et donc `Q` termine.

Donc `termine(Q)| renvoie \lstinline`Faux@, la deuxième instruction de `Q` ne boucle pas et `Q` termine normalement : contradiction.

Par conséquent, il n'existe pas de programme tel que `termine`, c.-à-d. qui vérifie qu'un programme termine ou non sur un jeu de données. **Le problème de la terminaison est indécidable.** ■

8.4 Résumé : recette de récursivité

La recette de récursivité est la suivante :

1. Assurez-vous que le problème `P` peut se décomposer en un ou plusieurs **sous-problèmes de même nature**.
2. Identifiez le(s) **cas de base** qui est le plus petit problème qui ne se décompose pas en sous-problèmes.
3. `Résoudre(P)` =
   - Si `P` est un cas de base, résolvez-le directement
   - Sinon
     — Décomposez `P` en sous-problèmes `P1`, `P2`...
     — Résolvez récursivement `P1`, `P2`...
     — Combinez les résultats pour obtenir la solution pour `P`

9 Conclusion

La récursivité est une technique dont la mise en oeuvre sur des problèmes de nature récursive aboutit à des solutions élégantes et simples.

La conception d'un algorithme récursif implique la définition du cas général, du ou des cas triviaux et du ou des paramètres de la récursivité. Au fur et à mesure des appels récursifs, le ou les paramètres récursifs doivent s'approcher des valeurs définissant le ou les cas triviaux.

Une solution itérative (simple) est plus efficace qu'une solution récursive équivalente.

S'il y a plusieurs appels récursifs non consécutifs (tours de Hanoï par exemple), la dérécursification (version itérative équivalente) devient difficile.

Il est souvent plus fiable de spécifier une solution récursive, mais par souci d'efficacité, la solution récursive doit être écartée au profit d'une solution itérative dans les cas suivants :

- la solution itérative est évidente ;
- une étude de la solution récursive montre que la profondeur de récursivité est d'un ordre supérieur à $O(n \lg n)$, et on sait qu'il existe une solution itérative.

On utilise donc la récursivité :

- quand l'exposé du problème ou la structure de donnés sont récursifs,
- quand il n'y a pas de solution itérative évidente ou meilleure.