VI Algorithmes récursifs

VI.1 Récursivité : Définition

La récursivité est la possibilité de faire figurer dans la définition d'un objet une référence à lui-même. En programmation, cela se traduit par la possibilité pour un sous-programme de s'appeler lui-même.

Définition : *Algorithme récursif*

Un algorithme est dit *récursif* lorsqu'il est défini en fonction de lui-même.

Soit la définition récursive du nombre d'embranchement :

Exemple : fonctions récursives : *puissance* et *factorielle*

La fonction puissance x^n peut se définir ***récursivement*** par :

$$x^n = \begin{cases} 1 & \text{si } n = 0 \\ x \,.\, x^{n-1} & \text{si } n \geqslant 1 \end{cases}$$

la fonction factorielle $x!$ peut se définir ***récursivement*** par :

$$x^n = \begin{cases} 1 & \text{si } x = 0 \\ x \,.\, (x-1)! & \text{si } x > 0 \end{cases}$$

Algorithme récursif et appel récursif

Définition : *Algorithme récursif*

Un algorithme calculant la solution d'un problème sur une donnée `x` est dit *récursif* si, parmi les instructions utilisées dans l'algorithme, on trouve l'expression du même problème sur une donnée `y`.

Définition : *Appel récursif*

Dans un algorithme récursif (une fonction récursive), on nomme *appel récursif* l'appel de la même fonction, c'est à dire l'instruction contenant l'expression du même problème sur une autre donnée.

Règle : *Terminaison*

Tout algorithme récursif doit distinguer plusieurs cas, dont l'un au moins ne doit pas comporter d'appel récursif.

Définition : *Condition de terminaison*

Les conditions, que doivent satisfaire les données pour déclencher l'exécution des instruction d'un cas ne comportant pas d'appel récursif, s'appellent les *conditions de terminaison*.

Théorème

Il n'existe pas de suite infinie strictement décroissante d'entiers positifs ou nuls.

Règle : *Croissance*

Tout appel récursif doit se faire avec des données plus « proches » des données satisfaisant une condition de terminaison.

Définition : *Récursivité simple*

Un algorithme *récursif* est *simple* si chaque cas qu'il distingue se résout en au plus un appel récursif.

VI.2 Récursivité multiple

Le calcul du nombre de combinaisons grâce à la relation de Pascal s'obtient par : $C_n^p = \begin{cases} 1 \text{ si } p = 0 \text{ ou } p = n \\ C_{n-1}^p + C_{n-1}^{p+1} \text{ sinon} \end{cases}$

Exemple : Algorithme de calcul du nombre de combinaisons

```
func combinaison(n, p: Int) -> Int
//calcule le nombre de combinaison de p valeurs parmi n selon
// la relation de Pascal
//données : n, p entier > 0
//pre : n ≥ p ≥ 0
//résultat : C_n^p
    if (p == 0) or (p == n) then return 1
    else return combinaison(n-1,p)+combinaison(n-1,p-1)
    endif
endfunc
```

VI.2 Récursivité multiple

Définition : *Récursivité multiple*

Un algorithme *récursif* est *multiple* si l'un des cas qu'il distingue se résout avec plusieurs appels récursifs.

Exemple : Algorithme de calcul du nombre de combinaisons

```
func combinaison(n, p: Int) -> Int
//calcule le nombre de combinaison de p valeurs parmi n selon
// la relation de Pascal
//données : n, p entier > 0
//pre : n ≥ p ≥ 0
//résultat : C_n^p
    if (p == 0) or (p == n) then return 1
    else return combinaison(n-1,p)+combinaison(n-1,p-1)
    endif
endfunc
```

VI.3 Récursivité mutuelle

Il y a d'autres formes encore plus compliquées de récursivité, par exemple la ***récursivité mutuelle***. On parle aussi de ***récursivité cachée***.

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

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

Définition : *Récursivité mutuelle*

Deux algorithmes sont *mutuellement récursifs* si l'un fait appel à l'autre, et l'autre fait appel à l'un.

VI.4 Récursivité imbriquée

Enfin, un appel récursif peut contenir un autre appel récursif. En effet, on peut, lors de l'appel à une fonction, donner un appel récursif comme paramètre d'un autre appel récursif.

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

Exemple : Algorithme de la fonction d'Ackerman

```
func ackermann(m, n : Int) -> Int
  if m == 0 then return n+1
  elsif n == 0 then return ackermann(m-1,1)
  else return ackerman(m-1,ackerman(m,n-1))
endfunc
```

VI.4 Récursivité imbriquée

Enfin, un appel récursif peut contenir un autre appel récursif. En effet, on peut, lors de l'appel à une fonction, donner un appel récursif comme paramètre d'un autre appel récursif.

Définition : *Récursivité imbriquée*

Lorsqu'un paramètre d'une fonction récursive est un appel récursif, on parle de *récursivité imbriquée*.

Exemple : Algorithme de la fonction d'Ackerman

```
func ackermann(m, n : Int) -> Int
  if m == 0 then return n+1
  elsif n == 0 then return ackermann(m-1,1)
  else return ackerman(m-1,ackerman(m,n-1))
endfunc
```

VI.5 Récursivité : Conclusion

- la récursivité est un moyen naturel de résolution de certains problèmes ;
- tout algorithme peut s'exprimer de manière récursive ;
- c'est un moyen de se ramener d'un cas « compliqué » à un cas « plus simple » ;
- la récursivité permet d'écrire des algorithmes concis et élégants.

Exercice

Exercice

Programmer un algorithme de résolution de sudoku