Chap-3 : La Récursivité

Année universitaire :2020/2021

1. Récursivité

Année universitaire :2017/2018

Récursivité

De l'art et la manière d'élaborer des algorithmes pour résoudre des problèmes qu'on ne sait pas résoudre soi-même !

1.1 Définition

- Une définition récursive est une définition dans laquelle intervient ce que l'on veut définir.

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

1.2 Récursivité simple

- Revenons à la fonction puissance $x \rightarrow x^n$. Cette fonction peut être définie récursivement :

$$x^n = \begin{cases} 1 & si \quad n = 0 \\ x.x^{n-1} & si \quad n \geq 1 \end{cases}$$

- L'algorithme correspondant s'écrit :

```
Puissance (x, n)
Begin
If (n = 0) then
    return 1
    Else
            return (x*Puissance (x, n-1))
End
```

1.3 Récursivité multiple

Une définition récursive peut contenir plus d'un appel récursif.

Exemple_1 : Nombre de Combinaisons

- On se propose de calculer le nombre de combinaisons *en se servant de la relation de Pascal :*

$$C_n^p = \begin{cases} 1 & si \quad p = 0 \quad ou \quad p = n \\ C_{n-1}^p + C_{n-1}^{p-1} & \sin on \end{cases}$$

- L'algorithme correspondant s'écrit :

```
Combinaison (n, p)
Begin
If ( p = 0 OR p = n) then
    return 1
    Else
            return (Combinaison (n-1, p) + Combinaison (n-1, p-1))
End
```

Exemple_2 : Suite de Fibonacci

```
Fibonacci ( n)
   If ( n=0  or n =1 )
            Return 1
            Else
            Return (Fibonacci (n-2) + Fibonacci (n-1) )
End_Fibonacci
```

1.4 Récursivité mutuelle

Des définitions sont dites *mutuellement récursives* si elles dépendent les unes des autres. Ça peut être le cas pour la définition de la parité :

$$pair(n) = \begin{cases} vrai & si \quad n = 0 \\ impair(n-1) & \sin on \end{cases}$$

et

$$impair(n) = \begin{cases} faux & si \quad n = 0 \\ pair(n-1) & \sin on \end{cases}$$

Les algorithmes correspondants s'écrivent :

Pair (*n*) Begin If (*n* = 0) Then return (vrai) Else return (impair(n-1)) End	Impair (*n*) Begin If (*n* = 0) Then return (faux) Else return (Pair (*n*-1)) End

1.5 Récursivité imbriquée

La fonction d'Ackermann est définie comme suit :

$$A(m,n)=\begin{cases} n+1 & si \quad m=0 \\ A(m-1,1) & si \quad m>0 \quad et \quad n=0 \\ A(m-1, A(m,n-1)) & \sin on \end{cases}$$

d'où l'algorithme :

```
Ackermann (m, n)
Begin
    If ( m = 0 ) Then
    return (n+1)
    else
        If (n=0) Then
                Return(Ackermann (m-1, 1))
                else
                        Return (  Ackermann (m-1, Ackermann (m, n-1)) )
End
```

1.6 Principe et dangers de la récursivité

Principe et intérêt

- Ce sont les mêmes que ceux de la démonstration par récurrence en mathématiques.
- On doit avoir :
  - un certain nombre de cas dont la résolution est connue, ces «cas simples» formeront les cas d'arrêt de la récursivité
  - un moyen de se ramener d'un cas « compliqué » à un cas «plus simple».
- La récursivité permet d'écrire des algorithmes concis et élégants.

1.6 Principe et dangers de la récursivité

Difficultés

- La définition peut être dénuée de sens :

  Algorithme A(n)

  renvoyer A(n)

- Il faut être sûr qu'on retombera toujours sur un cas connu, c'est-à-dire sur un cas d'arrêt ; il nous faut nous assurer que la fonction est complètement définie, c'est-à-dire, qu'elle est définie sur tout son domaine d'applications.

Moyen : existence d'un ordre strict tel que la suite des valeurs successives des arguments invoqués par la définition soit strictement monotone et finit toujours par atteindre une valeur pour laquelle la solution est explicitement définie.

```
Int Algo (int a, int b)
   If (a  <=0) then Erreur
            Else
            If (a>=b) return (a==b)
                        Else
                        Return (Algo (a,b-a) )
End
```

- La suite des valeurs *b*, *b*-*a*, *b*-2*a*, etc. est strictement décroissante, car *a* est strictement positif, et on finit toujours par aboutir à un couple d'arguments (*a*,*b*) tel que *b*-*a* est négatif, cas défini explicitement.

1.7 Importance de l’ordre des appels récursifs

Algo1 (*n*) If (*n* = 0)Then ne rien faire Else afficher *n* Algo1 (*n*-1) End_if End	Algo2 (*n*) If (*n* = 0) Then ne rien faire Else Algo2 (*n*-1) afficher *n* *Endif* End

1.8 Exemple d'algorithme récursif : les tours de Hanoï

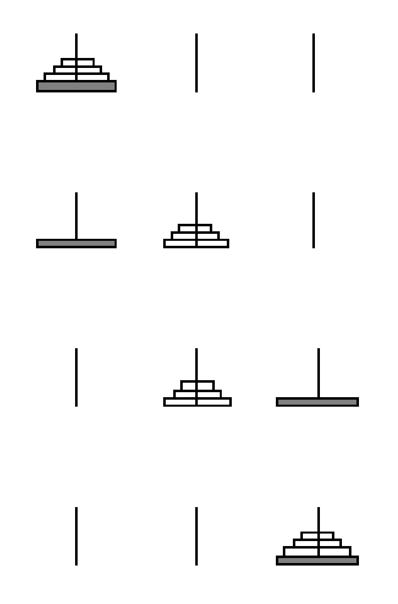
Le problème

- Le jeu est constitué d'une plaquette de bois où sont plantées trois tiges numérotées 1, 2 et 3. Sur ces tiges sont empilés des disques de diamètres tous différents. Les seules règles du jeu sont que

- l'on ne peut déplacer qu'un seul disque à la fois,

- et qu'il est interdit de poser un disque sur un disque plus petit.

- Au début, tous les disques sont sur la tige 1 (celle de gauche), et à la fin ils doivent être sur celle de droite.

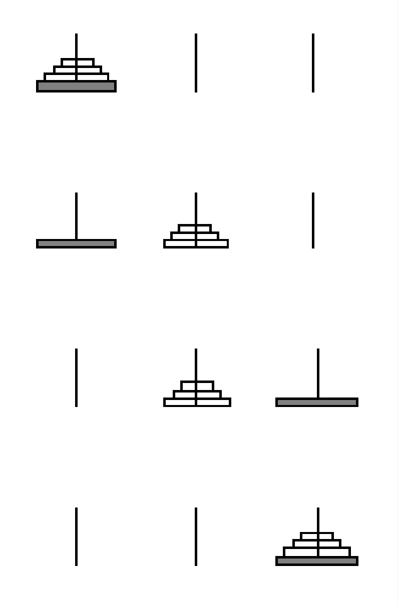


2

1.8 Exemple d'algorithme récursif : les tours de Hanoï

Résolution

- **Principe**
  - On suppose que l'on sait résoudre le problème pour (*n*-1) disques.
  - Pour déplacer *n* disques de la tige 1 vers la tige 3, on déplace les (*n*-1) plus petits disques de la tige 1 vers la tige 2,
  - puis on déplace le plus gros disque de la tige 1 vers la tige 3,
  - puis on déplace les (*n*-1) plus petits disques de la tige 2 vers la tige 3.



1.8 Exemple d'algorithme récursif : les tours de Hanoï

Validité

- il n'y a pas de viol des règles possible puisque le plus gros disque est toujours en « bas » d'une tige et que l'hypothèse (de récurrence) nous assure que nous savons déplacer le « bloc » de (*n*-1) disques en respectant les règles.

Algorithme
Hanoi (*n*, départ, intermédiaire, destination)
    If *n* > 0 **Then**
            Hanoi (*n*-1, départ, destination, intermédiaire)
            déplacer un disque de départ vers destination
            Hanoi (*n*-1, intermédiaire, départ, destination)
    Endif
End_Hanoi

1.8 Exemple d'algorithme récursif : les tours de Hanoï

L'appel à Hanoi(3,1,2,3) entraîne l'affichage de :

1. Déplace un disque de la tige 1 vers la tige 3
2. Déplace un disque de la tige 1 vers la tige 2
3. Déplace un disque de la tige 3 vers la tige 2
4. Déplace un disque de la tige 1 vers la tige 3
5. Déplace un disque de la tige 2 vers la tige 1
6. Déplace un disque de la tige 2 vers la tige 3
7. Déplace un disque de la tige 1 vers la tige 3

1.8 Exemple d'algorithme récursif : les tours de Hanoï

Complexité

- On compte le nombre de déplacements de disques effectués par l'algorithme Hanoi invoqué sur *n* disques.

$$C(n) = \begin{cases} 1 & si \quad n=1 \\ C(n-1)+1+C(n-1) & \sin on \end{cases} = \begin{cases} 1 & si \quad n=1 \\ 1+2C(n-1) & \sin on \end{cases}$$

- Et on en déduit facilement (démonstration par récurrence) que :

$$C(n) = 2^n - 1$$

- On a donc ici un algorithme de complexité exponentielle.
- En supposant que le déplacement d'un disque nécessite 1minute (il faut réfléchir et déplacer un disque qui peut être lourd puisqu'il est en or), et si on dispose de 64 disques, il faudrait :

$C(n) = 2^{64} - 1$ minutes = 3,50965 10^{13} = **35096,5 milliards d'années**

2. Dérécursivation

Année universitaire :2017/2018

Dérécursivation

- Dérécursiver, c’est transformer un algorithme récursif en un algorithme équivalent ne contenant pas d’appels récursifs.

Définition (Récursivité terminale)

- Un algorithme est dit récursif terminal s'il ne contient aucun traitement après un appel récursif.

Exemple

Algorithme P(U) If (*Condition(U)*) Then *Traitement_base (U)*; P(α(*U*)) ; Else *Traitement_terminaison(U);* *Endif* *End_Algorithme*	– *U* est la liste des paramètres ; – *C* est une condition portant sur *U* ; – α(*U*) représente la transformation des paramètres;

Algorithme dérécursivée

```
Algorithme P'(U)
While ( Condition(U) ) do
   Traitement_base(U);
   U ← a(U)
   End_while
Traitement_terminaison;
End_Algorithme
```

- L'algorithme P' non récursif équivaut à l'algorithme P.
- Remarquer la présence d'une boucle.

Exemple_1 : Est-ce que a est diviseur de b ?

Version récursive	
Diviseur (*a*,*b*) If (*a* <=0) then Erreur Else If (a>=b) return (a==b) Else Return (Diviseur (*a*,*b*-*a*)) End_Diviseur	

Exemple_1 : Est-ce que a est diviseur de b ?

Version récursive	Version dérécursivée
Diviseur (*a*,*b*) If (*a* <=0) then Erreur Else If (a>=b) return (a==b) Else Return (Diviseur (*a*,*b*-*a*)) End_Diviseur	Diviseur (*a*,*b*) If (*a* <=0) then Erreur While (a<b) do *b* ← *b*-*a* *End_while* return (a==b) *End_Algorithme*

Exemple_2 : Factoriel (N) ?

Version récursive	
Factoriel(N) If (N = 0) Return 1 ; Else Return N*Factoriel (N-1) ; End_Factoriel	

Exemple_2 : Factoriel (N) ?

Version récursive	Version Itérative
Factoriel(N) If (N = 0) Return 1 ; Else Return N*Factoriel (N-1) ; End_Factoriel	Factoriel (N) F=1; For i=N step -1 to 1 do F=F*i End_Factoriel

Récursivité non terminale

- Dans l'algorithme suivant la récursivité n'est pas terminale puisque l'appel récursif est suivi d'un traitement. Cela implique qu'il reste un traitement à reprendre ultérieurement. Il va falloir donc sauvegarder, sur une pile, le contexte de l'appel récursif, typiquement les paramètres de l'appel engendrant l'appel récursif.

Algorithme récursif

```
Algorithme Q(U)
If ( Condition(U) ) Then
    Traitement_A(U);
    Q(a(U));
    Traitement_B(U)
    Else
            Traitement_terminaison(U)
    Endif
End_Algorithme
```

Algorithme dérécursivé :

```
Algorithme  Q'(U)
    Pile.init() ;

    While (Condition (U)) do
        Traitement_A(U) ;
        Pile.push(U) ;
        U= a(U);
    Endwhile


    Traitement_terminaison (U) ;

    While (not Pile.empty()) do
        Pile.pop (U) ;
        Traitement_B(U) ;
    Endwhile

End_Algo
```

Exemple_3

Version récursive	Version dérécursivée
void recursif(int n) { if (n> 0) { printf(‘’%d\n’’,n) ; recursif(n-1); printf(‘’%d\n’’,n) ; } else printf(‘’FIN’’) ; }	recursif(int n) **Pile.init() ;** While (*Condition (U)*) do *Traitement_A(U) ;* **Pile.push(U) ;** U= α(*U*); Endwhile *Traitement_terminaison (U) ;* While (**not Pile.empty()**) do **Pile.pop (U) ;** *Traitement_B(U)* ; Endwhile End_Algo

Remarques

- Les programmes itératifs sont souvent plus efficaces, mais les programmes récursifs sont plus faciles à écrire.

- Il est toujours possible de dérécursiver un algorithme récursif.