

Récurtivité

Principe
Utilisation
Exemples

Introduction

La récursivité est un concept fondamental, utilisé absolument partout. Ça paraît compliqué au début, mais en fait c'est très simple

C'est quoi la récursivité ?

La récursivité c'est quand une fonction s'appelle elle-même jusqu'à atteindre une condition d'arrêt. Elle arrête alors de s'appeler elle-même. Le résultat de chaque fonction enfant est retourné dans les fonctions parent, jusqu'à retourner à la fonction originale. Cette explication est peut-être pas tout à fait claire tout de suite, mais ça va le devenir!

Le principe de récursivité

- ❖ Tout objet est dit récursif s'il se définit à partir de lui-même
- ❖ Ainsi, une fonction est dite récursive si elle comporte, dans son corps, au moins un appel à elle-même
- ❖ De même, une structure est récursive si un de ses attributs en est une autre instance

Correspondance mathématique

- ❖ Principe de récurrence
- ❖ Exemple : définition des entiers (Peano)
 - 0 est un entier
 - Si n est un entier, alors $n+1$ est un entier

Exemples de fonctions récursives

- ❖ Calcul de la somme des entiers de 1 à n
 - On calcule la somme jusqu'à $n-1$
 - Puis on ajoute n
- ❖ Idem avec le produit (fonction factorielle)

Un peu de vocabulaire

- ❖ Pour une fonction récursive, on parlera :
 - De **récurtivité terminale** si aucune instruction n'est exécutée après l'appel de la fonction à elle-même
 - De **récurtivité non terminale** dans l'autre cas

Exemple

Terminale

```
void f(int n) {  
    if(n==0) System.out.println("Hello");  
    else f(n-1);  
}
```

Non terminale

```
void f(int n) {  
    if(n>0) f(n-1);  
    System.out.println("Hello");  
}
```

Réversivité directe

- ❖ Lorsque f s'appelle elle-même, on parle de **réversivité directe**
- ❖ Lorsque f appelle g qui appelle f , il s'agit aussi de réversivité
 - On l'appelle alors **indirecte**

Exemples de structures récursives

❖ Liste récursive

- Le premier élément
- Et le reste de la liste (qui est aussi une liste)

❖ Une expression arithmétique est :

- Soit une valeur
- Soit une expression, un opérateur et une autre expression

Implémentation

Comment programmer une fonction récursive ?
Quels sont les pièges à éviter ?

Programmer une fonction récursive

- ❖ Il suffit de la faire s'appeler elle-même

```
int f(int n){  
    return f(n-1);  
}
```

- ❖ La fonction f est récursive : elle s'appelle elle-même
- ❖ Voyez-vous un problème en f ?

Une obligation : s'arrêter

La fonction f telle qu'elle est écrite ne s'arrête pas :

- Appel : $f(2)$
- Appel : $f(1)$
- Appel : $f(0)$
- Appel : $f(-1)$
- Appel : $f(-2)$
- Etc...

Comment y parvenir

Première étape : la condition terminale

- *Obligatoirement au début de toute fonction réursive*
- Une condition : le cas particulier
- Pour ce cas, pas d'autre appel à la fonction : la chaîne d'appels s'arrête

Exemple

```
void f(int n) {  
    if (n==0)  
        System.out.println("Hello");  
    else f(n-1);  
}
```

- ❖ Ici quand n vaut 0, on s'arrête
- ❖ Problème : arrive-t-on à $n = 0$?

Terminaison de la fonction

- ❖ Il faut que la fonction s'arrête
- ❖ La condition terminale ne sert à rien si elle ne devient jamais vraie
- ❖ Exemple avec la fonction précédente :
 - `f(-2)` provoque une pile d'appels infinie
 - Probablement d'autres tests à faire :
Si $n < 0$, envoyer une exception par exemple

Une bonne solution

```
void f(int n) {  
    if(n<0) System.exit(-1);  
    if(n==0) {  
        System.out.println("Hello");  
    } else {  
        f(n-1);  
    }  
}
```


Pourquoi ça marche ?

- ❖ Si n est négatif : on s'arrête sur une exception
- ❖ Si n est nul : c'est le cas d'arrêt (« Hello »)
- ❖ Si n est positif : on appelle f avec la valeur $n-1$
 - Chaîne d'appels avec des valeurs entières strictement décroissantes de 1 en 1
 - On arrive forcément à 0
 - On affiche « Hello »
 - On remonte la pile des appels (sans rien faire, ici la récursivité est terminale)

Théorème de Gödel

Il n'existe pas de moyen automatique pour savoir si un programme termine ou pas.

Conclusion

- ❖ Il faut regarder cas par cas, et à la main
- ❖ Même si aucune méthode n'est générale, le principe de récurrence aide souvent

En résumé

- ❖ Une fonction récursive doit comporter :
 - Un cas d'arrêt dans lequel aucun autre appel n'est effectué
 - Un cas général dans lequel un ou plusieurs autres appels sont effectués
- ❖ *La chaîne d'appel doit conduire au critère d'arrêt*
 - Optionnellement, des cas impossibles ou incorrects à traiter par des exceptions

Quelques exemples

Réversivité obligatoire ?

Les boucles **for**

- ❖ Très bonne candidate
- ❖ Toute boucle **for** peut se transformer en une fonction récursive
- ❖ Principe :
 - Pour faire des choses pour un indice allant de 1 à n
 - On les fait de 1 à $n-1$ (même traitement avec une donnée différente)
 - Puis on les fait pour l'indice n (cas particulier)

Traduction

```
void f(int n) {
    for (int i=0; i<=n; i++)
        traiter(i);
}
```

```
void f(int n){
    if(n==0)
        traiter(0);
    else {
        f(n-1);
        traiter(n);
    }
}
```

Exemple : fonction factorielle

```
int fact(int n) { int
    res = 1;
    for (int i=1; i<=n;i++){
        res = res*i;
    }
    return res;
}
```

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

Appel de **fact** (5) récursif

❖ *Phase de descente récursive*

Appel à **fact** (5)

Appel à **fact** (4)

Appel à **fact** (3)

Appel à **fact** (2)

Appel à **fact** (1)

Appel à **fact** (0)

❖ *Condition terminale*

- Retour de la valeur 1

Suite

❖ *Phase de remontée* (après l'appel à $\text{fact}(n-1)$ on multiplie par n : la récursivité n'est pas terminale)

Retour de la valeur 1

Retour de la valeur 2

Retour de la valeur 6

Retour de la valeur 24

Retour de la valeur 120

Quelques conséquences

La plupart des traitements sur les tableaux peuvent se mettre sous forme récursive :

- Tris (sélection, insertion)
- Recherche séquentielle (attention: pas dichotomique)
- Inversion
- Problème des huit reines
- Etc...

Une constatation

L'écriture sous forme récursive est toujours plus simple que l'écriture sous forme itérative

Une question

- ❖ Une même fonction est-elle plus efficace sous forme récursive ou sous forme itérative ? Ou, sous une autre forme, y a-t-il un choix optimal généralisable ?
- ❖ La réponse est non. La réponse à la question inverse est non. Il n'y a pas de généralité

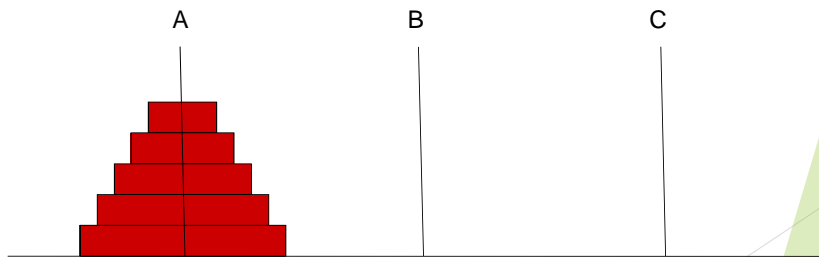
En revanche

- ❖ La plupart des traitements itératifs simples sont facilement traduisibles sous forme récursive, exemple du **for**
- ❖ L'inverse est faux
- ❖ Il arrive même qu'un problème ait une solution récursive triviale alors qu'il est très difficile d'en trouver une solution itérative (parcours arbres)

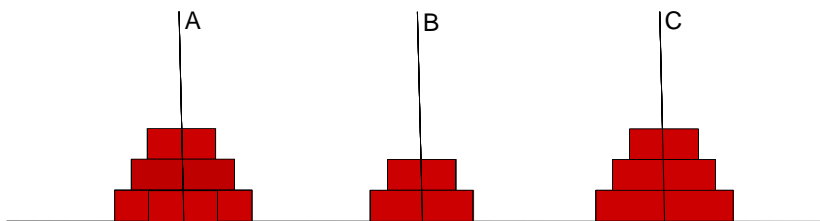
Les tours de Hanoï

❖ **Problème** : Transférer n disques de l'axe A à l'axe C, en utilisant B, de sorte que jamais un disque ne repose sur un disque de plus petit diamètre

❖ Ici : $n = 5$



Exemple pour $n=3$



Solution récursive

- ❖ On veut déplacer n disques de A vers C
 - Si n vaut 1, on déplace le disque
 - Sinon
 - On déplace d'abord $n-1$ disques de A vers B
 - On déplace le disque numéro n de A à C
 - On déplace $n-1$ disques de B à C
- ❖ On a respecté la consigne : le disque n n'est jamais au dessus d'un disque plus petit

Exercice :

Faire implantation dans un langage de votre choix.

En résumé

- ❖ On a transformé un problème de taille n en deux problèmes de taille $n-1$ et un problème de taille 1
- ❖ Tactique classique : « diviser pour régner »
- ❖ Essayez de trouver une solution itérative (bonne chance)

La fonction d'Ackermann

❖ $\text{Ack}(m, n)$ vaut :

- $n+1$ si $m=0$
- $\text{Ack}(m-1, 1)$ sinon et si $n=0$
- $\text{Ack}(m-1, \text{Ack}(m, n-1))$ autrement

❖ Remarque : on finit bien car $\max(m, n)$ est strictement décroissant sur les appels (à l'exception de $\text{Ack}(1, 0)$ qui finit trivialement)

Il y a trois grandes étapes pour chaque algorithme récursif.

La condition d'arrêt

La résolution du problème

L'appel récursif

Exemples à l'étude

1. exponentielle (boucle et récursif)
2. factoriel
3. suite de Fibonacci
4. simuler une boucle

Exemple : FACTORIEL


- Rappel : La fonction factorielle n , qui se note en mathématiques $n!$ a pour valeur :

$$n! = 1 \times 2 \times 3 \times \dots \times n$$

$$\text{Ex : } 4! = 1 \times 2 \times 3 \times 4 = 24$$

Pour programmer cette fonction, on peut :

- **soit utiliser une boucle**
- **Soit créer un algorithme récursif**

 Algorithme récursif pour calculer le $n^{\text{ième}}$ terme de la suite de Fibonacci

La suite de Fibonacci est définie par $(F_n)_{n \in \mathbb{N}}$:

$$\begin{cases} F_0 = 1 \\ F_1 = 1 \\ F_{n+2} = F_{n+1} + F_n, \forall n \in \mathbb{N} \end{cases}$$

Entrées : n entier, indice du terme de la suite de Fibonacci

Résultat : le terme de rang n de la suite de Fibonacci

```

1 fonction Fibb(n)
2 début
3   si n=0 alors
4     retourner 1
5   sinon si n=1 alors
6     retourner 1
7   sinon
8     retourner Fibb(n-1) + Fibb(n-2)
9 fin

```

Lorsque vous serez à l'aise avec la récursivité, on discutera de ce concept au parcours de structures de données.

Nous regarderons sur des structures simples et que vous connaissez mais aussi et sur des structures plus complexes comme les arbres.

Exercices

Faire l'implantation de ces algorithmes dans un langage de votre choix.