


TD3: Rappels sur la récursivité

Ce que vous avez vu en PeiP1 ...

La récursivité est un concept utilisé dans de nombreux domaines qui consiste à définir une entité à partir d'une entité de taille plus petite. Voici quelques exemples :

- Les entiers naturels : l'entier n est défini à partir de l'entier $n - 1$,
- Un dictionnaire : les mots sont définis à partir d'autres mots,
- Les images publicitaires, comme le paquet de *La vache qui rit* qui a des boucles d'oreilles étant elles-mêmes des paquets de vache qui rit dont les boucles d'oreilles ... 

La récursivité en informatique consiste à définir une fonction qui s'appelle elle-même. C'est un outil qui aide à raisonner, en particulier pour les problèmes qui ne peuvent être résolus qu'en énumérant toutes les possibilités.

Induction structurelle

L'induction structurelle consiste à définir un ensemble à partir d'une *base*, c'est à dire les éléments atomiques de l'ensemble, et de *règles de construction* qui définissent comment construire un élément à partir d'un autre.

Les entiers Les entiers naturels sont définis à partir de :

Base $\{0\}$

Règle n est un entier $\Rightarrow n + 1$ est un entier.

Quand vous faites une *preuve par récurrence* vous suivez ce schéma inductif.

Pour définir une *fonction récursive* sur les entiers, en suivant ce schéma inductif, il faudra décrire ce qu'il se passe pour 0 puis décrire ce qu'il se passe pour n en fonction du résultat calculé par la fonction pour $n - 1$.

Voici par exemple la fonction qui inverse une chaîne de caractères en Python :

```
def inverse(c):
    if c=="":
        return c
    return inverse(c[1:])+c[0]
```

Les listes Les listes sont définies à partir de :

Base $\{\{\}\}$

Règle l est une liste $\Rightarrow [x, l]$ est une liste où x est un élément.

Voici par exemple la fonction qui compte les éléments d'une liste en Python :

```
def count(l):
    if l==[]:
        return 0
    return 1 + count(l[1:])
```

B A BA de la récursivité

Le schéma de chacune ces fonctions récursives est le suivant:

- un cas d'arrêt : donne la valeur pour les éléments élémentaires (de la base)
- un appel récursif : rappelle la fonction elle-même sur une donnée plus petite. A chaque appel récursif, la donnée en paramètre diminue jusqu'à atteindre la valeur du cas d'arrêt (si cette valeur n'est pas atteinte, la fonction ne s'arrête jamais, problème !)

Secret: il faut raisonner en supposant que la fonction est correcte pour une donnée, et chercher ce qu'il faut faire pour calculer le résultat un cran plus loin. Par exemple, si je suppose que $f(n-1)$ calcule bien la factorielle, alors pour trouver la factorielle de n , il suffit de multiplier par n .

En PeiP1, vous avez écrit des algorithmes compliqués de retro-parcours qui explorent des arbres de jeux (par exemple pour le jeu des allumettes). Les exercices ici sont bien plus simples ... Récupérez la classe Recursive avec des méthodes statiques (exceptionnellement) pour écrire en java les petits exercices d'échauffement. Une fois complétée, déposez votre classe dans le test prévu à cet effet où il y a des exemples de résultats attendus.

1. **Factorielle.** Compléter la méthode **récursive** factorielle(int n).
2. **Somme des n premiers entiers.** Compléter la méthode **récursive** public int sommeCarreRec(int n) qui renvoie la somme des carrés des entiers de 1 à n .
3. **Nombre d'occurrences d'une lettre dans un mot.** Compléter la méthode **récursive** public static int compte(String s, char l). Nota: en java, `s.substring(1)` renvoie s privé de son 1er élément.

4. **Recherche dichotomique récursive.** Compléter la méthode **récursive** private `int rechercheViteRekursif(K x, int gauche, int droite)` de la classe `TableauGenerique` qui cherche x dans la partie du tableau comprise entre les indices *gauche* et *droite*. Cette méthode est appelée par la méthode publique `public int rechercheViteRekursif(K x)`. Ici, ce qui diminue et permet de stopper la récursion, c'est le nombre d'éléments du tableau dans lequel on est en train de chercher. Le cas d'arrêt est donc quand *gauche* == *droite*.

5. **Permutations d'une chaîne de caractères.** Ecrire la méthode `public static ArrayList<String> permute(String s)` qui prend une chaîne de caractères et renvoie une liste (`ArrayList`) avec toutes les permutations possibles. Par exemple, `permute("dur")` doit renvoyer `["dur", "udr", "urd", "dru", "rdu", "rud"]`.

Aide 1: si l'on a calculé les permutations possibles de `s.substring(1)` alors pour calculer les permutations de `s` il faut insérer `s.charAt(0)` à chaque position de chacune des permutations de `s.substring(1)`. Par exemple, les permutation de "ur" sont ["ur", "ru"]. Pour obtenir ["dur", "udr", "urd", "dru", "rdu", "rud"] on a inséré *d* dans chaque position de *ur* et *ru*.

Aide 2: le nombre de permutations d'une chaîne de caractères de longueur n est $n!$. En effet, pour 1 caractère, il y a 1 permutation, et il y a $i + 1$ positions possibles pour insérer un caractère dans une chaîne de longueur i . Il ne vous reste plus qu'à vérifier que la longueur de la liste calculée par `permute(s)` vaut bien `s.length()`!

Calcul de la complexité

Pour évaluer la complexité d'une fonction récursive, la première question est de caractériser de quelle grandeur entière dépend cette complexité. Par exemple, pour un entier, cela dépend de l'entier lui-même, pour un tableau, cela dépend du nombre d'éléments, pour une chaîne de caractères, de la longueur de cette chaîne.

On écrit ensuite une relation de récurrence dont le schéma général est le suivant :

- $C(0) = C_b$ si 0 est la taille d'un élément dans la base et que la complexité du calcul du cas d'arrêt est C_b ,
- $C(n) = a \times C(f(n)) + C_n$ si n est la taille d'un élément construit à partir d'une règle et d'un élément de taille $f(n)$, a est le nombre d'appels récursifs, et C_n est la complexité de l'étape de calcul.

Exemple: factorielle

```
public static int factorielle(int n) {  
    if (n==1)  
        return 1;  
    return factorielle(n-1)*n;  
}
```

Pour les entiers naturels, la base est de taille 0 et la règle est que si $n - 1$ est un entier naturel alors n l'est aussi.

- $C(0) = \theta(1)$ car “return 1” est une opération constante,
- $C(n) = 1 \times C(n-1) + \theta(1)$ car il y a un seul appel récursif sur $n-1$ et l’action de cette étape est une multiplication donc d’une complexité constante $\theta(1)$.

En calculant cette relation de récurrence on trouve $\theta(n)$.

6. **Complexité de la somme des n premiers entiers** Ecrivez la relation de récurrence comme pour la factorielle.
7. **Complexité du comptage des occurrences d’une lettre dans un mot** Ici la complexité dépend de la taille de la chaîne. $C(0)$ est pour la chaîne vide et $C(n)$ pour une chaîne de longueur n . L’appel récursif se fait sur la chaîne moins son 1er caractère donc d’une longueur $n-1$.
8. **Recherche dichotomique récursive** Ici, la complexité dépend du nombre d’éléments de la zone du tableau considérée. Donc de $droite - gauche + 1$ que l’on pourra noter n .