

# Récurtivité

La récursivité est un paradigme de programmation qui permet, entre autres, de simplifier la programmation de la résolution de certains algorithmes. Ce paradigme est notamment beaucoup utilisé lors du parcours d'arbres et de graphes qui sont des structures de données que vous découvrirez en deuxième année. L'objectif de ce cours est de vous faire découvrir et implémenter les quatre grands type de récursivité. Vous étudierez également les effets de ce paradigme de programmation sur la mémoire et les limites de la récursivité.

## 1 Introduction à la récursivité

**Définition 1 (Fonction récursive).** Une fonction *récursive* est une fonction qui s'appelle elle-même.

Toute fonction récursive contient obligatoirement les deux éléments suivants :

- au moins un appel récursif;
- une condition d'arrêt.

L'objectif d'une fonction récursive est donc de s'appeler elle-même en mettant à jour des variables pour calculer un résultat. Le retour de ce résultat se fait lorsque la fonction atteint la condition d'arrêt. Cela évite à la fonction de s'appeler indéfiniment.

Le développement d'une fonction récursive part de la définition récursive d'un problème. La plupart des problèmes étant posés de façon itérative, un gros travail de reformulation est nécessaire pour résoudre le même problème en récursif.


**Exemple 1.** Considérons l'exemple du calcul de la factorielle. La définition « classique », telle qu'on peut la retrouver par exemple sur Wikipedia, nous donne  $n! = 1 \times 2 \times \dots \times n$ . Une fonction itérative est alors utile pour boucler sur le produit des entiers à réaliser. Il existe une autre définition de la factorielle qui se rapproche de celle d'une suite : pour tout entier  $n > 0$ , on pose  $n! = n \times (n - 1)!$  et  $0! = 1$ . La définition est alors récursive et il possible d'implémenter simplement cela en C.

```
// Calcul itératif
int factIter ( int N )
{
    int res = 1, i;

    for (i=1; i<=N; i++)
        res*= i;
    return res;
}
```

```
// Calcul récursif
int factRec ( int N )
{
    // Condition d'arrêt
    if (N==0)
        return 1;
    // Retour du résultat
    return N * factRec(N-1);
}
```

La fonction récursive se compose donc d'une condition d'arrêt, équivalente au cas trivial de la définition mathématique du problème, et d'un appel récursif permettant de calculer le

résultat. Cet appel récursif étant précédé d'un symbole `*` décrivant un produit, le résultat ne peut pas être retourné directement. Il est nécessaire d'appeler à nouveau la fonction `FactRec` avant de faire le `return`. 

Visionnez maintenant la vidéo indiquée sur Moodle afin de voir un exemple de récursivité simple au niveau théorique.

**Définition 2 (Récursivité simple).** Une fonction récursive simple est une fonction récursive qui ne contient qu'un unique appel à elle-même.

La récursivité simple est le type le plus courant. Une première phase de descente d'indice permet de décrire l'opération à calculer au fur et à mesure des appels. La deuxième phase (remontée d'indice) permet d'effectuer le calcul et de retourner le résultat.

### TD Exercices 1 à 3.

Notons que si la rédaction d'une fonction récursive est plus simple lorsque le problème est lui-même formulé de façon récursive, cela peut entraîner une consommation de mémoire excessive. En effet, chaque appel de fonction entraîne une augmentation de la consommation de la mémoire, notamment en déclarant à nouveau les paramètres de la fonction. En comparaison, une fonction itérative n'a nul besoin de déclarer « plusieurs fois » ses paramètres. Ces points seront abordés plus tard dans le semestre dans une conférence thématique dédiée aux avantages et inconvénients de la récursivité.

## 2 Récursivité multiple

**Définition 3 (Récursivité multiple).** Une fonction récursive *multiple* est une fonction réalisant plusieurs appels récursifs.

Ce type de récursivité implique donc à chaque appel, au moins deux nouveaux appels de cette même fonction. Puis chacun de ces deux appels en génère lui-même deux, ce qui nous amène à quatre appels, puis huit etc. Chacun des appels est pourtant traité séquentiellement, les appels récursifs ne sont pas traités en parallèle.<sup>1</sup> Si la quantité de mémoire consommée est donc relativement modérée, son temps d'exécution en revanche, explose, du au grand nombre d'appels récursifs produits. Il faut donc prendre garde à la taille des données à traiter en entrée pour vérifier que le calcul pourra être réalisé en un temps raisonnable.

Visionnez maintenant la vidéo indiquée sur Moodle afin de voir un exemple de récursivité avec le calcul de la suite de Fibonacci. On rappelle que cette dernière est définie pour tout  $n \in \mathbb{N}$  par  $u_0 = 0$ ,  $u_1 = 1$  et  $u_n = u_{n-1} + u_{n-2}$ .

### TD Exercice 4.

1. Sauf programmation parallèle dédiée, mais c'est un autre problème.

## 3 Récursivité mutuelle

**Définition 4** (Récursivité mutuelle). Deux fonctions récursives s'appelant l'une l'autre sont dites récursives *mutuelles*.

On utilise ce type de récursivité en particulier dans les calculs de suite. L'étude d'évolution de populations interdépendantes est alors particulièrement simple à modéliser. Il s'agit d'un type de récursivité assez délicat à détecter ou à implémenter. La fonction ne s'appelle pas elle-même, mais en appelle une autre qui est susceptible de la rappeler. On retrouve alors un schéma de la forme **f1** appelle **f2**, qui appelle **f1**, qui appelle **f2** etc.

**TD** Exercices 5 et 6.

## 4 Pour aller plus loin

**TD** Exercices 7 à 9.