Terminale Langages et algorithmique	La récursivité	NSI

Objectifs :

- Découvrir la programmation récursive ;
- Écrire un programme récursif ;
- Analyser le fonctionnement d'un programme récursif ;

Durée estimée : 8 h *(avec les exercices et le TP)*

I - Introduction

L'idée sous-jacente de la **récursivité** est que pour résoudre un problème ou effectuer un calcul, on se ramène à la résolution d'un problème similaire mais de complexité moindre. On recommence ainsi jusqu'à obtenir un problème élémentaire que l'on sait résoudre.

Exemple 1 - *La somme des éléments d'une liste*

Étant donné une liste d'entiers `L = [2, 12, 1, 8]`, calculer la somme des éléments de cette liste.
Comme les listes sont itérables, nous pouvons simplement résoudre ce problème avec l'un de ces algorithmes que l'on dit **itératif** :

```
Définition somme(liste : une liste d'entiers):
    S ← 0
    n ← longueur de la liste
    Pour i allant de 0 à n faire
        S ← S + liste[i]
    FinPour
    Retourner S
FinDéfinition
```

Algorithme 1 : Somme des éléments d'une liste

```
Définition somme(liste : une liste d'entiers):
    S ← 0
    Pour chaque element de liste faire
        S ← S + element
    FinPour
    Retourner S
FinDéfinition
```

Algorithme 2 : Somme des éléments d'une liste

Supposons maintenant que nous n'ayons pas la possibilité de faire de « boucles ».
On peut alors aborder le problème sous un autre angle : la somme des termes de cette liste est :

`2 + ( la somme des termes de [12, 1, 8])`

Soit :

`2 + (12 + (la somme des termes de [1, 8])`

et enfin

`2 + (12 + (1 + (la somme des termes de [8])))`

Il est clair que : `la somme des termes de [8] est 8`. **Au final** le calcul est : `2 + (12 + (1 + (8))) = 23`
Considérons alors une fonction `Somme()` et qui renvoie le résultat de la somme des éléments de la liste.
L'algorithme ci-dessous que l'on dit **récursif** réalise cette seconde approche :

```
Définition somme(liste : une liste d'entiers):
    Si longueur de liste est 1 alors
        Retourner liste[0]
    Sinon
        Retourner liste[0] + somme(liste[1 :])
    FinSi
FinDéfinition
```

Algorithme 3 : Somme des éléments d'un liste

Dans cette fonction `liste[1:]` désigne la liste `liste` tronquée de son premier élément.

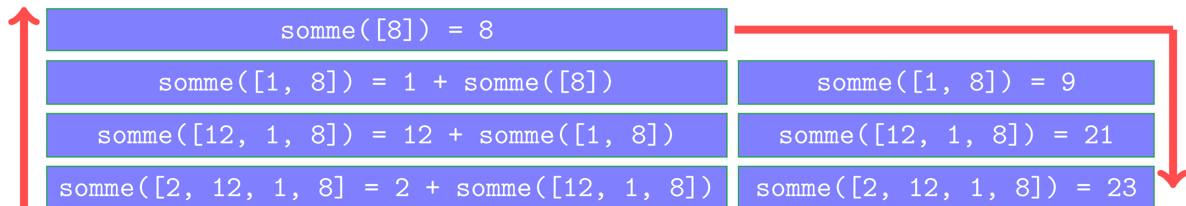
II - Fonctionnement des appels récursifs

La fonction va s'appeler elle-même avec un paramètre plus « petit ». Cet appel en induira un autre, puis un autre, etc. D'appel en appel, la taille du paramètre va ainsi diminuer. On s'arrêtera quand cette taille sera celle d'un problème immédiatement résolvable.

Les différents problèmes intermédiaires, ceux permettant de passer du problème initial au problème élémentaire, sont stockés successivement en mémoire dans ce que l'on appelle une **pile**. Il s'agit d'une structure de données dans laquelle on accède aux éléments dans l'ordre inverse de leur ajout en mémoire. Dans notre cas, on utilisera ainsi en premier le résultat du problème élémentaire, puis de proche en proche on arrivera à celui du problème initial.

Exemple 2 -

Exécution de la fonction de l'exemple précédent pour `liste = [2, 12, 1, 8]`



Comme expliqué précédemment, les différents appels récursifs sont empilés en mémoire jusqu'à ce que le paramètre d'appel est une longueur de 1 (condition d'arrêt). Ils sont ensuite dépilés jusqu'à l'appel initial.

Attention : Il est indispensable de prévoir une **condition d'arrêt** à la récursion sinon le sous-programme va s'appeler une infinité de fois.

Exemple 3 - ***Une erreur à ne pas commettre***

On a omis ici la condition d'arrêt, cette fonction ne se terminera en théorie jamais :

```
def factorielBad(n):
    return n*factorielBad(n-1)
```

En pratique, la pile où sont stockés les appels récursifs étant de taille finie, une fois qu'elle sera pleine le programme ne répondra plus.

III - Récursivité versus itération

Par opposition, on qualifiera d'**itératif** un sous-programme qui ne s'appelle pas

On peut démontrer qu'il est toujours possible de transformer un algorithme récursif en un algorithme itératif et inversement.

L'algorithme itératif sera plus rapide une fois implémenté dans un langage de programmation mais souvent plus complexe à écrire.

Exemple 4 - ***Implémentation itérative de la fonction factorielle***

Cette fois ci on utilise une structure itérative pour effectuer le calcul :

```
def factorielleIterative(n):
    resultat = 1
    for i in range(2, n+1):
        resultat = resultat*i
    return resultat
```

Il est manifeste que cette version est beaucoup moins élégante que la précédente.

Évoquons à présent quelques arguments en faveur puis en défaveur de l'usage de la récursivité.

Comme nous l'avons déjà mentionné, cette technique de programmation est très élégante et lisible. Elle évite en effet souvent le recours à de nombreuses structures itératives. Elle est d'autre part très utile voire indispensable pour concevoir des algorithmes sur des structures de données complexes comme les **listes**, les **arbres** et les **graphes**.

Nous reviendrons sur ces différentes structures de données dans les cours sur **les graphes** et d'algorithmique avancée.

L'inconvénient majeur de la récursivité est qu'une fois cette technique implémentée dans un langage de programmation, elle est très « gourmande » en mémoire. Rappelons en effet que l'on doit empiler tous les appels récursifs. Des débordements de capacité peuvent donc se produire s'il arrive que cette pile soit pleine.

Nous apprendrons à implémenter de façon plus satisfaisante des sous-programmes récursifs afin entre autres d'être beaucoup moins dépendant de la dimension de cette pile. Il s'agit d'une technique appelée **programmation dynamique**.