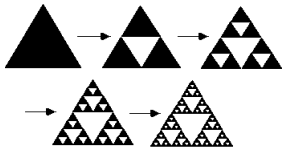
Cours 7 – Récursivité



I Introduction

Un algorithme récursif est un algorithme qui, pour parvenir au résultat voulu, se réemploie lui-même sur une instance plus petite du même problème

Exemple : Fonction factorielle

Commençons par écrire une version itérative de la fonction factorielle, *i.e.* une fonction définie uniquement à l'aide de boucles for ou while, qui ne s'appelle pas elle-même.

```
def factorielle(n):




.
```

Maintenant, en remarquant que $n! = n * (n-1)!$, écrivons une version récursive de la fonction factorielle, *i.e.* une fonction faisant appel à elle-même

```
def factorielle_rec(n):




.
```

Analysons les opérations effectuées lors de l'exécution de `factorielle_rec(4)`

Ainsi, chaque appel de la fonction récursive qui n'est pas un cas d'arrêt dépile le premier calcul, le résultat est utilisé dans le calcul suivant, et ainsi de suite.

Remarque : En Python, la pile d'appels d'une fonction récursive a une taille limitée à 1000. Si on fait plus d'appels, la fonction renvoie une erreur.

I.1 Conception d'une fonction récursive

Une fonction récursive

1. contient un cas de base (ou cas d'arrêt)
2. doit s'appeler elle-même sur un cas « plus petit » *(ou plus « proche » du cas de base)*

Exemple : Algorithme naïf d'exponentiation

On cherche à calculer x^n à l'aide d'un algorithme récursif simple, ne comportant que des multiplications.

Le cas de base est :

Pour obtenir x^n à partir de x^{n-1}, il suffit de faire :

On résume dans la fonction suivante :

```
def puissance(x,n):




.
```

II Exemples de fonctions récursives

Exemple 1 : Suite de Fibonacci

Ecrire une fonction récursive `fibo(n)` qui calcule le n-ième terme de la suite de Fibonacci définie par $\begin{cases} F_0 = F_1 = 1 \\ \forall n \in \mathbb{N}, F_{n+2} = F_n + F_{n+1} \end{cases}$

```
def fibo(n):




.
```

Exemple 2 : Exponentiations rapides

Ecrire deux fonctions récursives d'exponentation rapide (permettant de calculer x^n sans utiliser $**$)

La première version doit reposer sur les identités suivantes : $\begin{cases} x^{2k} = x^k \times x^k \\ x^{2k+1} = x \times x^k \times x^k \end{cases}$

```
def expo_rapide_1(x,n):
```

.

La seconde version doit reposer sur les identités suivantes : $\begin{cases} x^{2k} = (x \times x)^k \\ x^{2k+1} = x \times (x \times x)^k \end{cases}$

```
def expo_rapide_2(x,n):
```

.

III Complexité

III.1 Complexité et suites récurrentes

Que peut-on dire de la complexité asymptotique d'un algorithme dont la complexité $C(n)$ suit une relation de récurrence du type :

1. $C(n) = C(n-1) + R$:

2. $C(n) = qC(n-1)$:

3. $C(n) = aC(n-1) + b$

4. $C(n) = aC(n-1) + bC(n-2)$

5. $C(n) = C(n/2) + k$

III.2 Analyse de complexité de fonctions récursives

III.2.1 Un premier exemple

On considère la suite récurrente définie par $\begin{cases} u_0 = 2 \\ \forall n \in \mathbb{N}^*, u_n = \frac{1}{2}(u_{n-1} + \frac{3}{u_{n-1}}) \end{cases}$ qui converge vers $\sqrt{3}$ et permet donc d'en obtenir une approximation numérique.

Ecrire une fonction récursive permettant d'obtenir u_n puis estimer sa complexité en fonction de n.

```
def u(n):
```

.

III.2.2 Itératif ou récursif ? Exemple 1 : Factorielle

III.2.3 Itératif ou récursif ? Exemple 2 : Suite de Fibonacci

III.2.4 Conclusions

Les algorithmes récursifs présentent de nombreux avantages en termes de simplicité, lisibilité ou facilité à écrire et comprendre un programme.

En revanche, les algorithmes récursifs ne sont pas toujours efficaces, que ce soit en termes de complexité temporelle (quantité de calculs) ou spatiale (à cause de la pile d'appels)

III.2.5 Complexité des algorithmes d'exponentiation rapide