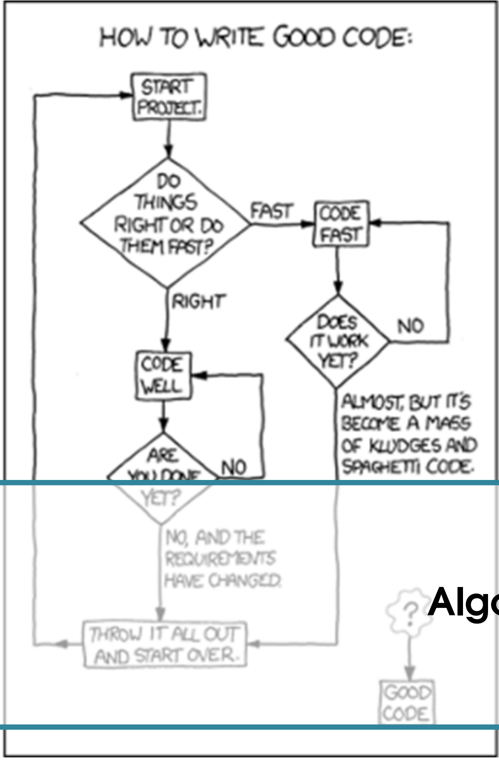
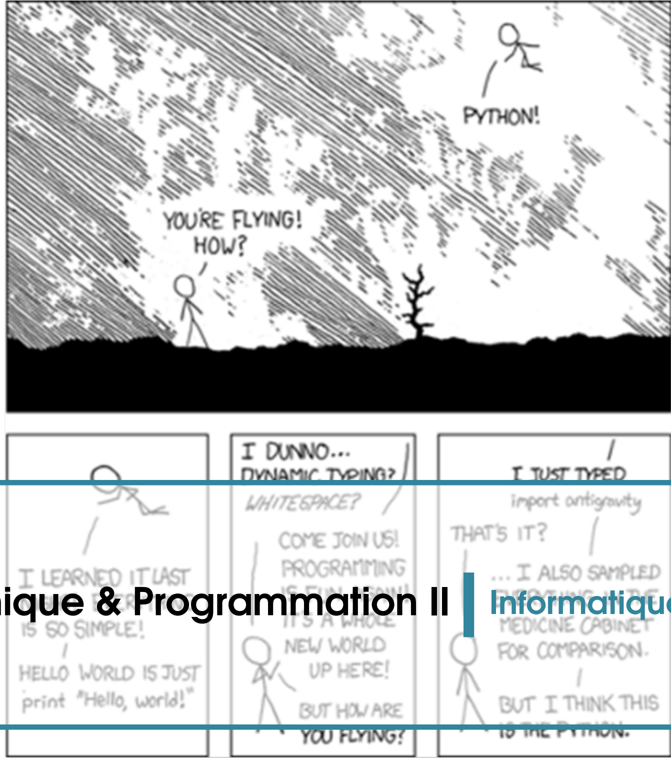




http://xkcd.com/353/

http://xkcd.com/844/

Partie 5

Algorithmique & Programmation II | Informatique

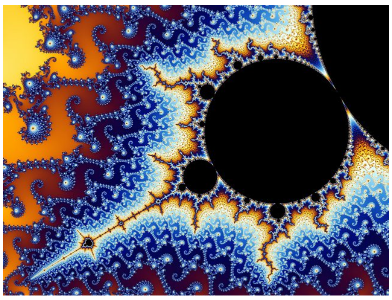
PT

Cours

Chapitre 1
Programmation récursive

Savoirs et compétences :

- *Alg – C15 : Récursivité : avantages et inconvénients.*



Courbe fractale de Mandelbrot [2]

Poupées russes [3]

Patrick Beynet
Xavier Pessoles
PT

1 Présentation

En mathématiques, en informatique, en biologie, mais aussi dans notre quotidien, nous faisons souvent face à des situations où un problème doit être résolu en utilisant une méthode de résolution qui est répétée plusieurs fois. Dans l'itération, cette méthode est appliquée par paliers de façon séquentielle. Dans la récursion, la méthode s'appelle elle-même. La récursion est si fondamentale qu'il n'est pas possible de l'éviter : l'auto-reproduction, qui constitue le fondement de toute vie, est un processus récursif.

*Arbre construit récursivement [**4**]*

Définition **Fonction récursive :** fonction faisant appelle à elle-même.

■ **Python**
Exemple de boucle sans fin :

```
def fonction_recursive():
    return fonction_recursive()
```

2 Exemple d'un algorithme récursif

2.1 Calcul explicite des puissances de 2

Dans certains cas une suite numérique peut être définie de manière explicite : $u_n = f(n)$. La détermination du nième terme est alors aisée. Il suffit d'évaluer $f(n)$.

■ **Exemple**

Puissances de 2 : *Comment évaluer le nombre 2 à la puissance n de manière explicite, $n \in \mathbb{N}$?* On définit de manière explicite la suite : $u_n = 2^n$.

■ **Pseudo Code**

Algorithme 1 : P2_explicite
Calcul explicite de la nième puissance de 2

Entrée :
- n, int : un nombre entier

Sortie :
- un nombre entier.

P2_explicite(n) :
    Retourner 2 ∧ n

■

(R)
- Ici la complexité algorithmique dépend de l'implémentation de ∧ dans le langage de programmation utilisé.
- Malheureusement, toutes les suites numériques ne peuvent pas être définies de manière explicite.

2.2 Calcul récursif des puissances de 2

Dans une suite définie de manière récurrente, il est possible de calculer le terme u_n de la suite en connaissant les termes précédents. Les égalités de la forme $u_n = f(u_{n-1})$, $u_n = f(u_{n-1}, u_{n-2})$, *etc.* s'appellent des relations de récurrence. Les égalités qui définissent les premiers termes d'une suite sont appelées des conditions de départ. Une suite recursive est donc définie par une relation de récurrence et une(des) condition(s) de départ.

■ **Exemple** On reprend l'exemple des puissances de 2. On définit de manière récursive la suite : $u_0 = 1$, $u_n = 2 \cdot u_{n-1}$. On propose un algorithme récursif proche de la définition et un algorithme itératif correspondant. ■

■ **Pseudo Code**

Algorithme 2 : P2_récursive :
Calcul récursif de la nième puissance de 2

Entrée :
- n, int : un nombre entier

Sortie :
- un nombre entier.

P2_recursive(n) :
  Si $n == 0$ ***alors :***
    Retourner 1
  Sinon :
    Retourner 2****P2_recursive***($n-1$)

Algorithme 3 : P2_iterative :
Calcul itératif de la nième puissance de 2

Entrée :
- n, int : un nombre entier

Sortie :
- un nombre entier.

P2_iterative(n)
  $x \leftarrow 1$
  tant que $n > 0$ ***faire :***
    $x \leftarrow 2 * x$
    $n \leftarrow n - 1$
  fin tant que
  Retourner x

Dans le premier cas, 2^4 se calcule sous la forme suivante :

$$2^4 = 2 \cdot P2(3) = 2 \cdot (2 \cdot P2(2)) = 2 \cdot (2 \cdot (2 \cdot P2(1))) = 2 \cdot (2 \cdot (2 \cdot (2 \cdot P2(0)))) = 2 \cdot (2 \cdot (2 \cdot (2 \cdot 1)))$$

Dans le second cas, on calcule itérativement les puissances de 2.

3 Analyse des algorithmes récursifs

3.1 Avantages et inconvénients

Le principal avantage d'une programmation récursive est, dans certains cas, la simplicité de son écriture. Comme l'exemple précédent le montre, dans le cas du calcul des puissances de 2, l'écriture récursive est plus naturelle que l'écriture itérative.

L'algorithme peut sembler alors plus aisé lors de sa lecture.

Par ailleurs, lors de l'écriture d'un programme récursif, il est nécessaire de porter attention à ce que l'algorithme se termine. Il pourra donc être nécessaire de vérifier la terminaison de l'algorithmique à l'aide d'un variant de boucle.

Comme pour les algorithmes itératifs, il est aussi nécessaire de vérifier la correction de l'algorithme en utilisant un invariant de boucle.

En outre, nous verrons de plus que la complexité algorithmique temporelle et spatiale d'un algorithme récursif peut être plus coûteuse qu'un algorithme itératif.

Enfin certains développeurs informatiques ont un avis tranché sur l'intérêt de la programmation récursive : *« (...) I don't believe in recursion as the basis of all programming. This is a fundamental belief of certain computer scientists, especially those who love Scheme and like to teach programming by starting with a "cons" cell and recursion. But to me, seeing recursion as the basis of everything else is just a nice theoretical approach to fundamental mathematics (turtles all the way down), not a day-to-day tool. (...)»*[**5**]

3.2 Terminaison des algorithmes récursifs – Variant de boucle

3.3 Correction des algorithmes récursifs – Invariant de boucle

3.4 Notions de piles d'exécution

3.5 Calcul des puissances de 2 – Exponentiation rapide

4 Exemples d'utilisation de la récursivité

4.1 Diviser pour régner

4.2 Courbes fractales – Courbe de Peano

Références

[1] Patrick Beynet, *Supports de cours de TSI 2*, Lycée Rouvière, Toulon.

[2] « Mandel zool 08 satellite antenna ». Sous licence CC BY-SA via Wikimedia Commons - `https://fr.wikipedia.org/wiki/Ensemble_de_Mandelbrot#/media/File:Mandel_zoom_08_satellite_antenna.jpg`

[3] `http://lestorytelling.com/blog/wp-content/uploads/2013/08/707px-matriochka.jpg`

[4] `http://www.obside.fr/fractales/pages/Recursif/`

[5] Guido van Rossum, `http://neopythonic.blogspot.fr/2009/04/tail-recursion-elimination.html`.