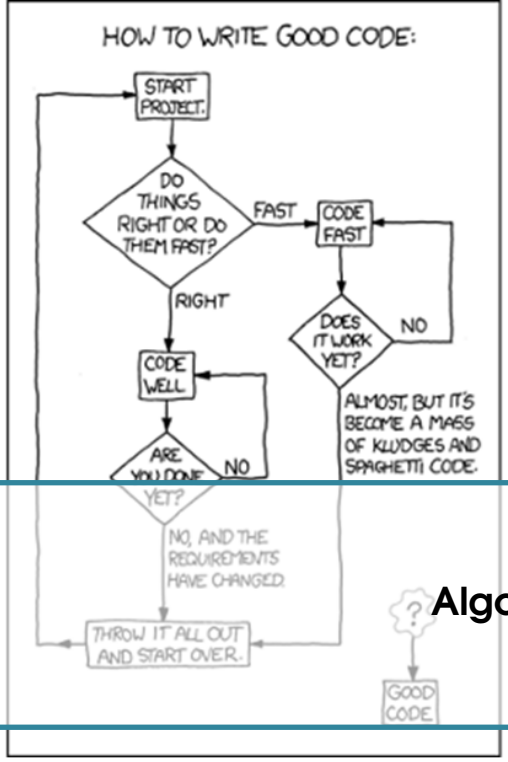
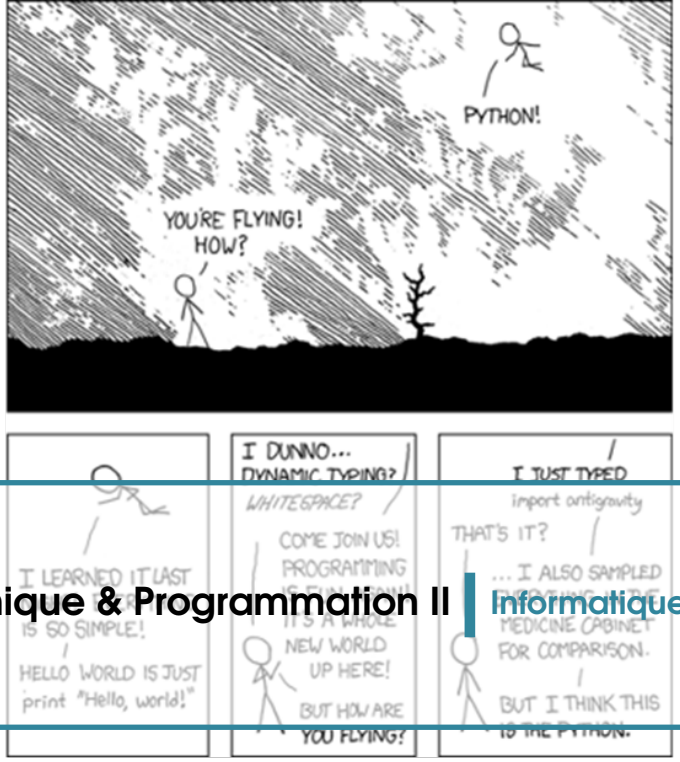




PSI ⋆

Partie 5

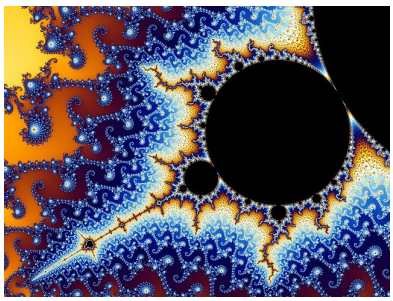
Algorithmique & Programmation II | Informatique

Cours

Chapitre 1
Programmation récursive

Savoirs et compétences :

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



Courbe fractale de Mandelbrot [2]

Poupées russes [3]

Xavier Pessoles
Patrick Beynet
PSI ⋆

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 appel 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 $2 \wedge n$

▪

R
- Ici la complexité algorithmique dépend de l'implémentation de $\wedge$ 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 ← 1
    tant que n > 0 faire :
        x ← 2 * x
        n ← 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

- ✔ Simplicité de l'écriture récursive dans certains cas : 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.
- ✗ Comme pour les algorithmes itératifs, il faut prêter attention à sa terminaison en utilisant 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.
- ✗ 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

Définition Rappel : Variant de boucle

Soit une condition booléenne (appelée condition d'arrêt) permettant de sortir d'une boucle constituée d'une comparaison entre une variable et une constante de types entiers positifs. La variable est un variant de boucle si :
- elle reste positive tout au long de l'algorithme ;
- elle décroît strictement à chaque itération de la boucle.

Ainsi, après un nombre fini d'itérations, on est sûr que la boucle se terminera.

Dans un algorithme récursif, il est nécessaire d'implémenter un cas terminal qui ne contient pas de nouvel appel à la fonction et qui permet la sortie de l'algorithme.

▪ **Exemple** Dans le cas de l'algorithme *P2_récursive*, n est un variant de boucle car :
- il est strictement positif en entrant dans la boucle ;
- il décroît à chaque appel récursif.

Le cas terminal a lieu lorsque $n = 0$. Dans ce cas, on sort de l'algorithme.
La terminaison de l'algorithme est bien prouvée.

▪

3.3 Correction des algorithmes récursifs – Invariant de boucle

Définition Rappel – Invariant de boucle

Un invariant de boucle est une propriété qui valider les points suivants :
1. la propriété doit être vraie en entrant dans la boucle ;
2. la propriété doit être vraie à chaque itération de la boucle ;
3. la propriété doit permettre de vérifier que que résultat donné est bien le résultat attendu.

▪ **Exemple** Dans le cas de l'algorithme *P2_récursive*, la propriété d'invariance peut être la suivante : $u_n = 2^n$ avec $u_0 = 1$.
La propriété est vrai au rang 0 : si $n = 0$, l'algorithme retourne 1. et on a bien $2^0 = 1$.
On suppose la propriété vraie au rang n. Au rang $n+1$, $n > 0$; donc d'après l'algorithme, $u_{n+1} = 2 \cdot u_n = 2 \cdot 2^n = 2^{n+1}$. La propriété est vraie au rang n+1.
Naturellement, on a donc au rang n, $u_n = 2^n$.
La propriété énoncée est donc un invariant de boucle. ▪

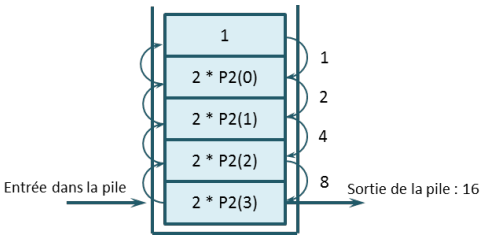
3.4 Notions de pile d'exécution

En informatique, les fonctions appelées sont gérées par une structure de données appelée **pile d'exécution** (*call stack*). Ainsi, lors de l'exécution d'un programme, les fonctions sont «agglutinées» sous forme de piles tant qu'elles ne fournissent pas de résultat. Dès qu'une fonction peut être exécutée, le résultat est stockée et elle sort de la pile. On parle de pile «dernier arrivé, premier sorti» (pile *LIFO : Last In First Out*).

▪ **Exemple**

Pour le calcul de 2^4, la pile d'exécution est donnée ci-contre.

R En python, la pile d'exécution limite à 1000 (988 ?) le nombre d'appels récursifs. En dépassant cette limite, l'exception *RuntimeError : maximum recursion depth exceeded in comparison* est levée.



▪

3.5 Calcul des puissances de 2 – Exponentiation rapide

Les algorithmes récursifs peuvent aussi être utilisés dans les algorithmes de type «Diviser pour régner» (*Divide and conquer*).

▪ **Exemple**

Le calcul des puissances de 2 peut s'effectuer par la méthode «d'exponentiation rapide» :

- si n est pair, $2^n = \left(2^2\right)^{\frac{n}{2}}$;
- si n est impair et $n > 1$, $2^n = 2 \cdot \left(2^2\right)^{\frac{n-1}{2}}$.

En conséquence plutôt que de réaliser le calcul $2^7 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2$, on a $2^7 = 2 \cdot 2^6 = 2 \cdot \left(2^2\right)^3 = 2 \cdot 2^2 \cdot \left(2^2\right)^2$.

Algorithme 4 : P2_rapide :
Calcul rapide de la nième puissance de 2

Entrée :
- n, int : un nombre entier

Sortie :
- un nombre entier.

```
P2_rapide(n)
    Si n==0 faire :
        Retourner 1
    Fin Si
    Si n%2==0 faire :
        tmp = P2_rapide(n/2)
        Retourner tmp*tmp
    Sinon faire :
        Retourner 2*P2_rapide(n-1)
    Fin Si Sinon
```

Suivant la puissance de 2 à calculer il est possible de représenter les étapes sous forme arborescente. On peut ainsi constater que, dans certain cas, des puissances «élevées» sont plus rapides à calculer que des puissances «moins élevées».

4 Exemples d'utilisation de la récursivité

4.1 Diviser pour régner

On rappelle l'algorithme itératif permettant de rechercher un nombre dans une liste triée.

Exemple

Algorithme : Recherche par dichotomie d'un nombre dans une liste triée

Données :
- nb, int : un entier
- tab, liste : une liste d'entiers triés

Résultat :
- m, int : l'index du nombre recherché
- None : cas où nb n'est pas dans tab

```
is_number_in_list_dicho(nb,tab) :
    g ← 0
    d ← longueur(tab)
    Tant que g < d alors :
        m ← (g+d) div 2
        Si tab(m)=nb alors :
            Retourne m
        Sinon si tab(m)<nb alors :
            g ← m+1
        Sinon, alors :
            d ← m-1
        Fin Si
    Fin Tant que
    Retourne None
Fin fonction
```

```
def is_number_in_list_dicho(nb,tab):
    """
    Recherche d'un nombre par dichotomie dans un
    tableau trié.
    Renvoie l'index si  le  nombre nb est dans la liste
    de nombres tab.
    Renvoie None sinon.
    Keyword arguments:
    nb,int —— nombre entier
    tab, list  —— liste  de nombres entiers triés
    """
    g, d = 0, len(tab)-1
    while g <= d:
        m = (g + d) // 2
        if  tab(m) == nb:
            return m
        if  tab(m) < nb:
            g = m+1
        else:
            d = m-1
    return None
```

Ce type d'algorithme peut être traité récursivement. En effet, la recherche du nombre se fait en divisant une liste en deux en regardant si le nombre recherché et dans une des deux parties du tableau. Ainsi à chaque itération, le recherche se réitère dans un des demi tableau.

L'algorithme récursif de la recherche d'un nombre dans une liste triée peut donc s'écrire ainsi :

Algorithme : Recherche par dichotomie d'un nombre dans une liste triée – méthode récursive

Données :
- nb, int : le nombre entier recherché
- tab, liste : une liste d'entiers triés, tab[0..max]
- m,n, int : deux entiers positifs tels que $0 \leq n \leq m \leq max$.

Résultat :
- m, int : l'index du nombre recherché
- None : cas où nb n'est pas dans tab

```
is_number_in_list_dicho_recursive(nb,tab,m,n) :
    Si m==n alors :
        Si tab(m)==x alors :
            Retourner m
        Sinon :
            Retourner None
        Fin Si
    Sinon
        k ← m+n div 2
        Si tab(k)<x alors :
            is_number_in_list_dicho_recursive(nb,tab,k+1,n)
        Sinon :
            is_number_in_list_dicho_recursive(nb,tab,m,k)
        Fin Si
    Fin Si
```

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.

[6] Jean-Pierre Becirspahic, Cours d'informatique du Lycée Louis le Grand, http://info-llg.fr/.