# 1. Méthodes de programmation

## I - Quelques principes de base

### 1) Conception

- *Spécifications* : préciser le "cahier des charges" (données reçues en entrée, préconditions, résultats renvoyés en sortie, postconditions...).
- *Analyse descendante* : en cas de tâches complexes, procéder par raffinements successifs.
- *Modularité* : découper l'algorithme en "modules" aussi indépendants que possible les uns des autres (utiliser des fonctions, avec transmission des données sous forme de paramètres et création de variables locales, plutôt que des procédures agissant sur des variables globales).
- *Portabilité* : éviter d'abuser de particularités d'un langage précis ou d'une machine donnée.

### 2) Présentation

- *Indentation* : elle est obligatoire en Python, puisqu'elle structure le programme !
- *Identificateurs* : choisir autant que possible des *identificateurs explicites* pour les noms des objets utilisés.
- *Commentaires* : il peut être utile d'insérer dans le code de brefs commentaires (précédés de #), précisant par exemple les spécifications d'un module, des éléments de justification. Pour des commentaires de plusieurs lignes, sur une copie notamment, on les rédigera de préférence dans un paragraphe précédant ou suivant le code concerné. On peut toutefois transformer en commentaire une partie du code en l'entourant de *triples quotes*.

### 3) Preuve

Il faut justifier la *terminaison* de l'exécution du programme et sa *correction* (le fait qu'il fournit le résultat annoncé...).

### 4) Notion de complexité

Pour évaluer l'efficacité d'un algorithme, on s'intéresse notamment :

- à la quantité de mémoire nécessaire *(complexité spatiale)* ;
- au nombre d'opérations à effectuer *(complexité temporelle)*.

Pour cette dernière, on dénombre le plus souvent un seul type d'opération (en général la plus coûteuse) : il est bon de préciser *"complexité en nombre de comparaisons"*, *"complexité en nombre de multiplications"*...

Pour comparer des algorithmes s'appliquant à un certain type de données, de taille variable, on définit une *taille de données* $N$ (par exemple : valeur absolue d'un entier, longueur d'une liste, taille d'un tableau, degré d'un polynôme, ordre d'une matrice carrée...).

Si $C(d)$ désigne la complexité de l'algorithme étudié pour le traitement de la donnée $d$, élément de l'ensemble $\mathcal{D}(N)$ des données de taille $N$, on définit :

- *la complexité dans le pire des cas :* $T_{au\ pire}(N) = \sup\{C(d),\ d \in \mathcal{D}(N)\}$ ;
- *la complexité dans le meilleur des cas :* $T_{au\ mieux}(N) = \inf\{C(d),\ d \in \mathcal{D}(N)\}$ ;
- *la complexité en moyenne :* $T_{moyen}(N) = \sum_{d \in \mathcal{D}(N)} P(d)\,C(d)$ où $P(d)$ est la probabilité d'apparition de la donnée $d$, cela sous réserve de convergence...

  Lorsque $\mathcal{D}(N)$ est infini, on effectue souvent une "sommation par paquets" en regroupant les données selon leurs caractéristiques.

*Notation $\Theta(f(N))$* (complément hors programme)

$T$ et $f$ étant deux fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$, on écrit $T(N) = \Theta(f(N))$ si et seulement si l'on a simultanément $T(N) = O(f(N))$ et $f(N) = O(T(N))$ lorsque $N$ tend vers l'infini, autrement dit s'il existe $\alpha$ et $\beta$ positifs tels que, pour $N$ assez grand, on ait :

$$\alpha f(N) \leq T(N) \leq \beta f(N)$$

(*i.e.* $T(N)$ et $f(N)$ sont asymptotiquement "*du même ordre de grandeur*").

La hiérarchie des ordres de grandeur permet effectivement une *comparaison des algorithmes* indépendamment de la machine utilisée, car les temps de calcul $T_1(N)$ et $T_2(N)$ sur deux machines de performances différentes sont *grosso modo* des fonctions proportionnelles de $N$ (le facteur de proportionnalité mesurant la différence de rapidité des deux machines).

*Exemples de temps d'exécution*

À titre indicatif, le tableau ci-dessous donne des valeurs approchées du temps de calcul nécessaire à une machine, capable d'exécuter $10^9$ opérations par seconde, pour mettre en œuvre des algorithmes de complexités variées (les unités j, a, u désignant respectivement le jour, l'année et l'âge estimé de l'univers).

| $N$ | $\log_2 N$ | $N$ | $N \log_2 N$ | $N^2$ | $N^3$ | $2^N$ | $N!$ |
|---|---|---|---|---|---|---|---|
| 5 | $2.10^{-9}$ s | $5.10^{-9}$ s | $1.10^{-8}$ s | $3.10^{-8}$ s | $1.10^{-7}$ s | $3.10^{-8}$ s | $1.10^{-7}$ s |
| 10 | $3.10^{-9}$ s | $1.10^{-8}$ s | $3.10^{-8}$ s | $1.10^{-7}$ s | $1.10^{-6}$ s | $1.10^{-6}$ s | $4.10^{-3}$ s |
| 20 | $4.10^{-9}$ s | $2.10^{-8}$ s | $9.10^{-8}$ s | $4.10^{-7}$ s | $8.10^{-6}$ s | $1.10^{-3}$ s | 77 a |
| 50 | $6.10^{-9}$ s | $5.10^{-8}$ s | $3.10^{-7}$ s | $3.10^{-6}$ s | $1.10^{-4}$ s | 13 j | $7.10^{37}$ u |
| 100 | $7.10^{-9}$ s | $1.10^{-7}$ s | $7.10^{-7}$ s | $1.10^{-5}$ s | $1.10^{-3}$ s | $3.10^{3}$ u | |
| 500 | $9.10^{-9}$ s | $5.10^{-7}$ s | $5.10^{-6}$ s | $3.10^{-4}$ s | 0.1 s | | |
| 1 000 | $1.10^{-8}$ s | $1.10^{-6}$ s | $1.10^{-5}$ s | $1.10^{-3}$ s | 1 s | | |
| 5 000 | $1.10^{-8}$ s | $5.10^{-6}$ s | $6.10^{-5}$ s | 0.03 s | 13 s | | |
| 10 000 | $1.10^{-8}$ s | $1.10^{-5}$ s | $1.10^{-4}$ s | 0.1 s | 17 min | | |
| 50 000 | $2.10^{-8}$ s | $5.10^{-5}$ s | $7.10^{-4}$ s | 2.5 s | 35 h | | |
| 100 000 | $2.10^{-8}$ s | $1.10^{-4}$ s | $2.10^{-3}$ s | 10 s | 12 j | | |

# II - Itération et récursivité

## 1) Structures de boucle

### a) Boucles conditionnelles

Il s'agit du cas où la répétition est conditionnée par l'évaluation d'une expression booléenne (en Python, instruction `while`).

L'utilisation de telles boucles nécessite une justification claire de la terminaison !

### b) Boucles inconditionnelles

Elles sont utilisées lorsqu'on connaît *à l'entrée dans la boucle* la liste (finie !) des valeurs à utiliser durant la boucle (en Python, instruction `for in`).

**Attention !** La terminaison d'une boucle `for` est assurée par la nature même de la boucle, **à condition** que la liste de valeurs ne soit pas modifiée en cours de boucle (c'est possible en Python, mais formellement déconseillé !).

### c) Notion d'invariant de boucle

Dans le cas d'algorithmes non triviaux, qui nécessitent une justification du résultat obtenu, il peut être judicieux de mettre en évidence un "*invariant de boucle*", c'est-à-dire une propriété qui reste vraie après chaque exécution du corps de la boucle **et qui permet de justifier la correction de l'algorithme !** Pour cela, il est souvent commode de nommer **mathématiquement** (par exemple à l'aide d'indices) les valeurs successives contenues dans les variables **informatiques** utilisées.

*Exemple* : *exponentiation rapide itérative*

```
def exprap(x,n):
    result=1
    while n!=0:
        if n%2==1:
            result*=x
        x*=x
        n//=2
    return result
```

La terminaison de la boucle `while` est justifiée par la décroissance stricte de l'entier contenu dans `n`.

Il est clair que la complexité de cette fonction est un $\Theta(\log_2 n)$, d'où son nom, par comparaison avec l'exponentiation naïve qui est en $\Theta(n)$.

En revanche, il n'est pas clair à l'œil nu que cette fonction renvoie $x^n$... Pour le justifier, on peut démontrer que `result*x^n` n'est pas modifié par un passage dans la boucle. Il en résulte par récurrence que cette expression a la même valeur à la sortie de la boucle qu'à l'entrée, d'où le résultat !
Pour rédiger une preuve, il pourra être plus clair de noter $r_k$, $x_k$ et $n_k$ les valeurs contenues respectivement dans les variables `result`, `x` et `n` après le $k$-ième passage dans la boucle
On montre alors par récurrence sur $k$ que, pour tout $k$, $r_k \cdot x_k^{n_k} = x^n$...

## 2) Récursivité

La *programmation récursive* (*i.e.* utilisant une fonction ou une procédure qui s'appelle elle-même) est souvent élégante, naturellement adaptée aux suites définies par récurrence (cf. $n!$, $x^n$, $f^{(n)}$...) et à la stratégie "*diviser pour régner*" (cf. § **III**). Elle permet parfois d'éviter le recours à des variables locales et à des boucles. Elle est en revanche exigeante en ressources machine (*sauvegarde et restauration du contexte*) et peut conduire à des temps de calculs prohibitifs en cas d'utilisation irréfléchie (cf. suite de Fibonacci au § **3)c)**).
Tout algorithme récursif **doit** être soigneusement justifié :

- *preuve de la terminaison de l'algorithme* : les appels récursifs **doivent** être soumis à une condition et il faut montrer qu'elle devient fausse au bout d'un nombre fini d'appels imbriqués. Typiquement, il suffit d'associer aux paramètres transmis lors de l'appel récursif un entier naturel qui décroît *strictement* lors de chaque nouvel appel récursif, ledit appel étant soumis à un test vérifiant que l'entier en question est supérieur à une certaine valeur ;
- *preuve de la validité du résultat* : en général par récurrence ;
- *calcul de complexité* : souvent à l'aide d'une relation de récurrence !

## 3) Exemples

### a) Factorielle

*Version itérative*

```
def fact(n):
    result=1
    for k in range(2,n+1):
        result*=k
    return result
```

*Version récursive*

```
def fact(n):
    if n<2:
        return 1
    else:
        return n*fact(n-1)
```

### b) Exponentiation naïve

*Version itérative*

```
def puiss(x,n):
    result=1
    for k in range(n):
        result*=x
    return result
```

*Version récursive*

```
def puiss(x,n):
    if n==0:
        return 1
    else:
        return x*puiss(x,n-1)
```

### c) Suite de Fibonacci

Rappelons qu'elle est définie — $a$ et $b$ étant deux réels donnés — par

$$u_0 = a, \quad u_1 = b \quad \text{et} \quad \forall n \geq 2 \quad u_n = u_{n-1} + u_{n-2}.$$

*Version itérative*

```
def fib(n,a,b):
    if n==0:
        return a
    elif n==1:
        return b
    else:
        for k in range(n-1):
            a,b=b,a+b
        return b
```

*Version récursive*

```
def fib(n,a,b):
    if n==0:
        return a
    elif n==1:
        return b
    else:
        return fib(n-1,a,b)+fib(n-2,a,b)
```

Les justifications sont banales, l'évaluation de la complexité est édifiante. Déterminons le nombre d'additions effectuées *sur les termes de la suite* (donc sans compter la gestion des indices ; ce choix se justifie par le fait que les additions sur les réels ou les grands entiers sont plus coûteuses que celles sur les indices).
Pour le calcul de `fib`$(n)$, la version itérative effectue $n-1$ additions, pour tout $n \geq 2$, ce qui est raisonnable.
Par contre, si je note $C(n)$ le nombre d'additions effectuées par la version récursive ci-dessus, pour le calcul de `fib`$(n)$, il apparaît que $C(n)$ est défini par

$$C(0) = C(1) = 0 \quad \text{et} \quad \forall n \geq 2 \quad C(n) = C(n-1) + C(n-2) + 1.$$

La suite de terme général $C(n) + 1$ est donc la suite de Fibonacci de premiers termes 1 et 1 !

On en déduit classiquement que $C(n)$ est de l'ordre de $\phi^n$, où $\phi = \frac{1+\sqrt{5}}{2} \approx 1,618$ (*nombre d'or*).

Cette complexité exponentielle est rédhibitoire. On constate expérimentalement que le calcul de `fib`$(30)$ prend déjà plusieurs secondes, même sur une machine rapide... Cela provient des calculs redondants impliqués par le double appel récursif.

Il est possible de se ramener à un seul appel récursif en remarquant que

$$\forall n \geq 1 \quad \texttt{fib}(n, a, b) = \texttt{fib}(n-1, b, a+b)$$

d'où la fonction suivante, qui effectue bien $n-1$ additions pour $n \geq 2$.

```
def fib(n,a,b):
    if n==0:
        return a
    elif n==1:
        return b
    else:
        return fib(n-1,b,a+b)
```

On parle dans ce cas de *récursivité terminale*, car l'appel récursif constitue une expression à lui tout seul, ce qui simplifie le "dépilage" des calculs en attente...

*Mémoïsation* (complément hors programme)
Une autre idée pour éviter les appels redondants est de mémoriser au fur et à mesure (dans une variable globale !) les valeurs prises par la fonction programmée. Idée à manipuler avec précaution puisqu'elle a un coût en espace mémoire...

Avec Python, la structure naturelle de stockage de valeurs associées à une *clé* est le *dictionnaire* (cf. la documentation de Python). Lorsque les clés sont les entiers naturels successifs, on peut aussi utiliser une liste...

Au moment de calculer $f(n)$ on commence par vérifier s'il existe dans le dictionnaire une entrée de la forme $n : f(n)$ ; si oui, on renvoie directement la valeur stockée, sinon on calcule récursivement $f(n)$, sans oublier de stocker cette nouvelle valeur dans le dictionnaire !

Voici un exemple de mise en œuvre, où je déclare une fonction locale récursive `fibo` avec pour seul paramètre `n`.

```
def fib_mem(n,a,b):
    table={0:a,1:b}

    def fibo(n):
        if not(n in table):
            table[n]=fibo(n-1)+fibo(n-2)
        return table[n]

    return fibo(n)
```

Noter le test `n in table`, qui renvoie vrai ou faux selon que `n` est ou non la clé de l'une des entrées du dictionnaire, tandis que l'affectation `table[n]=...` crée une nouvelle entrée si `n` ne figure pas encore parmi les clés, modifie la valeur associée à la clé `n` si elle existe déjà.

### d) Recherche dichotomique dans un tableau trié

La recherche linéaire dans un tableau quelconque a une complexité linéaire mais, lorsque le tableau est préalablement trié, on peut (il faut !) en profiter pour accélérer la recherche : l'idée est de comparer la valeur cherchée à la valeur médiane et de relancer la recherche dans la "bonne" moitié du tableau, tant que le tableau comporte au moins deux valeurs (avec au plus une valeur la "recherche" est vite faite !). Comme on divise par deux (à peu près...) la taille du tableau à chaque étape, la complexité est de l'ordre de $\log_2 n$ où $n$ est la taille du tableau.

L'algorithme se prête aussi bien à une programmation itérative que récursive. Il faut juste prendre garde à couper le tableau en deux parties de taille **strictement** inférieure à $n$.
On vérifie que c'est bien le cas lorsque, partant de la tranche $t[i:j]$ (avec $i < j-1$), on pose $m = (i+j)//2$ et l'on considère les deux tranches $t[i:m]$ et $t[m:j]$. Dans ce cas $m$ sera l'indice de la première valeur de la tranche de droite.

**Attention** aux algorithmes qui ne se terminent pas, avec un autre choix des deux "moitiés" qui partagerait 2 en $2+0$ ou $0+2$...

Voici les traductions en Python, deux programmes recevant $x$ et $t$ et renvoyant `True` ou `False` selon que $x$ est présent ou non dans $t$ (*précondition* : $t$ est non vide et trié en ordre croissant) :

*Version itérative*

```
def Cherche(x,t):
    i=0
    j=len(t)
    while i<j-1:
        m=(i+j)//2
        if t[m]<=x:
            i=m
        else:
            j=m
    return t[i]==x
```

*Version récursive*

```
def Cherche(x,t):
    #fonction auxiliaire récursive
    def DansTranche(i,j):
        if i==j-1:
            return t[i]==x
        else:
            m=(i+j)//2
            if t[m]<=x:
                return DansTranche(m,j)
            else:
                return DansTranche(i,m)
    #fonction principale triviale
    return DansTranche(0,len(t))
```

Remarquer l'utilisation des *expressions booléennes* : l'évaluation de `t[0]==x` renvoie `True` ou `False`, pas besoin de test pléonastique !

**Attention** aux "pythonneries" du genre `Cherche(x,t[:m])` qui fournissent une version récursive séduisante, mais recèlent un *coût caché* linéaire, car Python doit recopier les valeurs de la tranche considérée...

# III - "Diviser pour régner"

## 1) Principes généraux

Pour traiter un "problème" de taille $N$, il est parfois efficace de :

**a) partager le problème en $k$ sous-problèmes de taille $\lfloor N/2 \rfloor$ ou $\lceil N/2 \rceil$ ($k \in \mathbb{N}^*$)**

**b) résoudre les $k$ problèmes ainsi obtenus *(souvent récursivement...)***

**c) fusionner les résultats ainsi obtenus pour fournir la solution**

*Analyse de la complexité* (complément hors programme)

Il existe diverses versions de ce que l'on appelle dans la littérature le "*master theorem*", permettant d'obtenir une estimation asymptotique du coût de la résolution d'un problème de taille $N$ selon ce principe.

Ledit coût $C(N)$ vérifie une relation de récurrence de la forme :

$$C(N) = k \cdot C(N/2) + f(N)$$

où $f(N)$ représente le coût total de partage et fusion.

Pour simplifier, on convient d'écrire $k \cdot C(N/2)$ au lieu de l'expression exacte $a \cdot C(\lfloor N/2 \rfloor) + b \cdot C(\lceil N/2 \rceil)$ (où $a+b=k$). On peut montrer en effet que le résultat ne dépend pas du couple $(a,b)$ tel que $a+b=k$.

On a par exemple le résultat suivant, avec des hypothèses simples sur $f$ (mais pas toujours satisfaites...).

Si $k > 1$, $f$ croissante et $f(N) = \Theta(N^p)$ pour un certain $p$ de $\mathbb{N}$, alors, en posant $\omega = \log_2 k$, on a :

- si $2^p < k$ (*i.e.* $p < \omega$), alors $C(N) = \Theta(N^\omega)$ ;
- si $2^p = k$ (*i.e.* $p = \omega$), alors $C(N) = \Theta(N^\omega \log_2 N)$ ;
- si $2^p > k$ (*i.e.* $p > \omega$), alors $C(N) = \Theta(N^p)$.

*Remarques*

**1)** On peut remplacer $\Theta$ par $O$.

**2)** On peut généraliser le résultat au cas du partage en sous-problèmes de taille $N/d$, où $d$ est un entier au moins égal à 3.

**3)** Pour des résultats plus généraux, tapez *master theorem* dans votre moteur de recherche préféré !

## 2) Exemples

### a) Exponentiation rapide

L'algorithme récursif est le suivant, pour le calcul de $x^N$ : $x^0 = 1$ et

si $N$ est pair, alors $x^N = (x * x)^{\lfloor N/2 \rfloor}$, sinon $x^N = x * (x * x)^{\lfloor N/2 \rfloor}$,

Déterminons le type de relation de récurrence vérifiée par le nombre $C(N)$ de multiplications de réels nécessaires au calcul de $x^N$ : ici le "partage" (calcul de $x * x$ avant l'appel récursif) nécessite une multiplication, la "fusion" des résultats zéro ou une, selon la parité de $N$ ; ainsi :

$$C(N) = C(\lfloor N/2 \rfloor) + \Theta(1)$$

Noter qu'ici la version ci-dessus du *master theorem* ne s'applique pas ($f$ n'est pas croissante...), mais on constate que, si l'écriture en base 2 de $N$ est

$$N = [b_n \ldots, b_0]_2 \quad (\textit{i.e. } N = \sum_{j=0}^{n} b_j 2^j \text{ avec } b_n = 1 \text{ et } b_j \in \{0,1\} \text{ pour tout } j),$$

alors celle de $\lfloor N/2 \rfloor$ est $[b_n \ldots, b_1]_2$ tandis que $b_0 = N \bmod 2$.

On en déduit facilement que $C(N) = n + \sum_{j=0}^{n-1} b_j$, or $n = \lfloor \log_2 N \rfloor$ d'où

$$C(N) = \Theta(\log_2 N) \text{ avec précisément } \lfloor \log_2 N \rfloor \leq C(N) \leq 2 \lfloor \log_2 N \rfloor .$$

L'implémentation en Python est naturelle (ne pas oublier la condition d'arrêt !) :

```
def exprap(x,n):
    if n==0:
        return 1
    elif n%2==0:
        return exprap(x*x,n//2)
    else:
        return x*exprap(x*x,n//2)
```

### b) Produit de polynômes par la méthode de Karatsuba *(hors programme)*

Étant donnés deux polynômes $P(X)$ et $Q(X)$ de degré au plus $N = 2^n$, on écrit :

$$P = A + X^{N/2} \cdot B \quad \text{et} \quad Q = C + X^{N/2} \cdot D \quad \text{où} \quad A, B, C, D \text{ sont de degré au plus } N/2$$

et l'on développe :

$$P \cdot Q = A \cdot C + X^{N/2} \cdot (A \cdot D + B \cdot C) + X^N \cdot (B \cdot D).$$

On peut se contenter de trois produits de polynômes de degré au plus $N/2$, à savoir

$$A \cdot C,\ B \cdot D,\ (A + B) \cdot (C + D)$$

en remarquant que

$$A \cdot D + B \cdot C = (A + B) \cdot (C + D) - A \cdot C - B \cdot D.$$

Ainsi, le coût $C(N)$ en nombre de multiplications vérifie la relation de récurrence :

$$C(N) = 3 \cdot C(N/2)$$

et l'on en déduit :

$$C(N) = \Theta(N^\omega) \quad \text{où} \quad \omega = \log_2 3 \approx 1,585.$$

*Remarques* :

- Avec l'algorithme naïf, $C(N) = \Theta\left(N^2\right)$ ; on a par exemple $N^\omega < N^2/2$ pour $N \geq 6$ et $N^\omega < N^2/10$ pour $N \geq 257$.
- La *transformée de Fourier rapide* fournit un algorithme de calcul **approché** en $\Theta(N \log_2 N)$.

### c) Produit de matrices carrées par la méthode de Strassen *(hors programme)*

De même, pour effectuer le produit de deux matrices carrées d'ordre $N = 2^n$, en les découpant en quatre blocs d'ordre $N/2$, on peut se contenter de sept produits de matrices d'ordre $N/2$. Ainsi, le coût $C(N)$ en nombre de multiplications vérifie la relation de récurrence :

$$C(N) = 7 \cdot C(N/2)$$

et l'on en déduit :

$$C(N) = \Theta(N^\omega) \quad \text{où} \quad \omega = \log_2 7 \approx 2,807.$$

Ici, le coût de l'application naïve de la définition du produit matriciel est un $\Theta\left(N^3\right)$.

Pour les détails, afin de calculer :

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} \times \begin{pmatrix} E & F \\ G & H \end{pmatrix} = \begin{pmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{pmatrix},$$

on calcule les sept produits :

$$\begin{aligned} M_1 &= (B - D) \times (G + H)\ ;\ M_2 = (A + D) \times (E + H)\ ;\ M_3 = (A - C) \times (E + F)\ ; \\ M_4 &= (A + B) \times H\ ;\ M_5 = A \times (F - H)\ ;\ M_6 = D \times (G - E)\ ;\ M_7 = (C + D) \times E \end{aligned}$$

et l'on remarque (habilement !) que :

$$\begin{aligned} AE + BG &= M_1 + M_2 - M_4 + M_6 \\ AF + BH &= M_4 + M_5 \\ CE + DG &= M_6 + M_7 \\ CF + DH &= M_2 - M_3 + M_5 - M_7 \end{aligned}$$