# Chapitre 2

# EVALUATION DES ALGORITHMES RÉCURSIFS, GRANDEUR DES FONCTIONS ET LA COMPLEXITÉ ASYMPTOTIQUE

## 1. Structures de données

Une structure de données indique la manière d'organisation des données dans la mémoire. Le choix d'une structure de données adéquate dépend généralement du problème à résoudre.
Deux types de structures de données :

1. Statiques : Les données peuvent être manipulées dans la mémoire dans un espace statique alloué dès le début de résolution du problème. Exemple : les tableaux
2. Dynamiques : On peut allouer de la mémoire pour y stocker des données au fur et à mesure des besoins de la résolution du problème. Exemple: liste chainée, pile, file,
    - Notion des pointeurs,
    - Nécessité de la gestion des liens entre les données d'un problème en mémoire.

## 2. Qu'est ce que la récursivité ?

**Qu'est-ce que la récursion?**
Le mot «récursion» vient du latin recursare qui signifie courir en arrière, revenir.
L'idée de réapparition, de retour, est donc étymologiquement liée à la récursivité.

**Définition 1.**
Lorsqu'un système contient une autoréférence (ou une copie de lui même), on dit que ce système est récursif**.**

**Définition 2.**
Un algorithme est dit récursif si l'expression qui le définit fait appel à lui-même. On qualifiera de récursif, un appel à une fonction f provoqué par l'évaluation d'un autre appel à f.

Une fonction récursive est caractérisée par une, ou plusieurs, *relations de récurrence***.**
La fonction récursive se rappelle jusqu'à arriver sur :
→ Cas d'arrêt (cas de base),
→ Quand on construit des fonctions récursives, le raisonnement adopté est en général:
- Comment traiter le cas le plus simple ?
- Comment me ramener d'un cas plus compliqué au cas simple ?

**Exemple 1 :** La fonction factorielle
$n > 1 \rightarrow f(n) = n*f(n-1)$
$n = 0 \rightarrow f(0) = 1$
Le cas de base est f(0) = 1.

## 3. Types de récursivité

**Récursivité simple**
Une récursivité simple contient un seul appel récursif à P dans le corps d'une procédure récursive P.

**Récursivité multiple**
Une récursivité est multiple si il y a plusieurs appels récursifs à P dans le corps d'une procédure récursive P.

**Récursivité mutuelle**
Une récursivité est mutuelle ou croisée quand une procédure P appelle une procédure Q qui déclenche un appel récursif à P.

**Récursivité imbriquée**
Une récursivité est imbriquée si une procédure récursive P contient un appel imbriqué.

**Récursivité terminale**
La récursivité est terminale si l'appel récursif est la dernière instruction et elle est isolée.

La récursivité n'est pas terminale si l'appel récursif n'est pas la dernière instruction et/ou elle n'est pas isolée.

## 4. Critères de terminaison pour un bon algorithme récursif

Deux règles sont à respecter impérativement.

**Règle 1 :** Un algorithme récursif doit être défini par une expression conditionnelle dont l'un au moins des cas mène à une expression évaluable sans appel récursif. Une telle expression est appelée condition d'arrêt ou test d'arrêt ou clause terminale ou cas de base ou encore cas trivial.

**Règle 2** : Il faut s'assurer que pour toute valeur du (ou des) paramètre(s), il suffira d'un nombre fini d'appels récursifs pour atteindre la condition d'arrêt.

## 5. Comment gérer la récursivité ?

Quand la fonction est récursive, l'ordinateur est bien obligé de *stocker les calculs quelque part*. Il utilise pour cela une pile; il s'agit d'un objet défini par deux opérations :

- Empiler (ajouter un élément au sommet),
- Dépiler (prendre l'élément qui est au sommet).

**Exemple 2 :**

**1.** Gestion de la pile pour la factorielle

Algorithme pour le calcul de la factorielle :

F(n) = F(m) (toujours une initialisation)

n > 1 → F(n) = n*F(n-1)

n = 0 → F(0) = 1

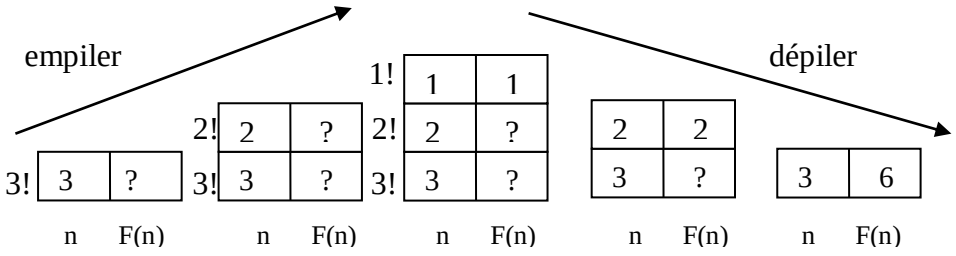


**Figure 1.** Calcul factorielle, f(3).

**2.** Une factorielle qui n'a pas besoin de pile

Algorithme pour la factorielle utilisant une variable acc, qui accumule :

F(n,acc) = F(m, 1)
n > 1 → F(n, acc) = F(n-1, n*acc)
n = 0 → F(n, acc) = acc

Il n'y a pas besoin ici de mettre des calculs en attente : on peut les enchaîner. Ainsi, on se passe de la pile.

### 5.1. Gestion de la récursivité sans la pile : Récursivité Terminale (RT)

La récursivité terminale est une notion qui peut améliorer nettement les performances de vos algorithmes. L'exécution d'un algorithme utilisant la récursivité terminale est transformée en général en algorithme itératif (plus rapide et moins gourmand en mémoire) par le compilateur.

**Définitions 3.**
Un algorithme est récursif terminal si son appel est le dernier. Une fonction est récursive terminale si elle renvoie, sans autre calcul, la valeur obtenue par son appel récursif.

Une fonction est de type récursivité terminale, si la fonction est du type
F(x1, …, xn) = G(y1, …, yn).

⇨ Il n'y a alors pas besoin de pile pour continuer le calcul.

### 5.2. Gestion de la récursivité avec la pile : Récursivité Enveloppée (RE)

**Définition 4.**
Une fonction est de type récursivité enveloppée, lorsqu'il y a quelque chose autour de la fonction G, Il y a des opérations qui enveloppent la fonction.

⇨ Il y a alors besoin de pile pour continuer le calcul.

## 6. Optimisation d'exécution de la récursivité

Optimisation d'exécution de la récursivité par la transformation de RE vers RT, En remplaçant son enregistrement au lieu d'en empiler un autre au-dessus de lui, l'utilisation de la pile est considérablement réduite, ce qui, en pratique, se traduit par de meilleures performances.

On doit donc transformer les fonctions récursives en fonctions récursives terminales à chaque fois que cela est possible.

Les langages exécutent un programme à récursivité terminale comme s'il était itératif (en espace constant). Sinon, il est facile de transformer une définition récursive terminale en itération pour optimiser l'exécution.

## 6.1. Comparaison d'algorithmes

**Exemple 3 :**

1. Gestion de la récursivité pour le calcul de la factorielle :

→ **RT :**

- F(n, acc) = F(n-1, n*acc), on peut identifier $y_1 = x_1 - 1$, $y_2 = n*x_2$ et G = F.
- F(1, acc) = acc. On peut identifier $y_1 = 0$, $y_2 = x_2$ et G = $y_2$.

→ **RE :**

- F(n) = n*F(n-1), la fonction est enveloppée par une multiplication avec n.

2. Gestion de la récursivité pour faire la somme des éléments d'un tableau

- On dispose d'un tableau T d'entiers, et on veut faire la somme de tous ses éléments.

→ **RE :** les règles trouvées

1. $\Sigma(T, n) = \Sigma(T, |T|)$
2. $n \geq 1 \rightarrow \Sigma(T, n) = T[n] + \Sigma(T, n-1)$
3. $n < 1 \rightarrow \Sigma(T, n) = 0$

**Problème :** On remarque à la ligne 2 que l'appel est enveloppé par l'addition avec T[n] : récursivité enveloppée → utiliser la pile.

**Solution** : traduire ceci en une récursivité terminale.
Le but maintenant est de faire rentrer l'addition avec T[n] dans la fonction.

⇨ Utiliser un accumulateur : au départ on le met à 0, et à chaque appel on y ajoute T[n]. A la fin, il suffira de rendre la valeur accumulée :

→ **RT** : les règles trouvées ≡ algorithme avec une boucle

1. $\Sigma(T, n, acc) = \Sigma(T, |T|, 0)$
2. $n \geq 1 \rightarrow \Sigma(T, n, acc) = \Sigma(T, n-1, acc + T[n])$
3. $n < 1 \rightarrow \Sigma(T, n, acc) = acc$

→ **Traduction d'un algorithme récursif vers un algorithme itératif**

⇨ Traduction d'un algorithme de RT vers un algorithme avec une boucle

**fonction** F(T : tableau[1..n] d'éléments entier; n, acc : entier) : entier
**Début**
**acc**← **0**;
**tant que** (n ≥ 1) **faire**

```
    debut
    acc ← acc + T[n];
    n ← n – 1;
    fin;
  retourner(acc) ;
fin.
```

3. Gestion de la récursivité pour trouver le maximum d'un tableau d'entiers :

→ **RT :**

```
 i←2 ;
 max← T[1] ;
Fonction Max(T : tableau [1..n] d'entier ; n,i,max: entier) : entier
Début
  Si (i ≤ n et T[i] > max) alors
    Max(T, n, i+1, T[i]) ;
  sinon
     Si (i ≤ n)  alors
        Max(T, n, i+1, max) ;
     sinon
        retourner(max) ;
     finsi ;
 finsi ;
Fin.
```

→ **Traduction de d'un algorithme récursif vers un algorithme itératif**

⇨ Traduction d'un algorithme de RT vers un algorithme avec une boucle

```
1.  fonction Max(T : tableau[1..n] d'élément entier ; n :entier)
2.  Début
3.       max ← T[1] ;
4.       pour i de 2 à n faire
5.           si (T[i] > max) alors
6.               max ← T[i] ;
7.       finpour ;
8.       Retourner (max) ;
9.  Fin.
```

→ **Calculer le coût de l'algorithme**

Pour compter le nombre exacts d'opérations de cet algorithme :
La condition dépend des données :

- Si le maximum est au début du tableau → on a juste une affectation (ligne 3) et n – 1 comparaisons soit un total de n opérations.
- Si le tableau est trié dans l'ordre croissant → (n – 1)*2 + 1 = 2n – 1 opérations.
- Si le tableau est aléatoire → on a (n – 1)*(1 + ½) + 1 = (3*n – 1)/2 opérations.
  - ⇨ Le coût de l'algorithme dépend de la taille (n) des données, mais peu aussi être dépendant de leur nature.

## 7. C'est quoi les notions de Landau ?

La plupart des programmes contient un nombre d'instructions très élevés, l'évaluation de la complexité de façon exact pour comparer deux algorithmes n'est pas facile parce que des critères matérielles sont introduite dans le calcul.

### 7.1. Classes de complexité

Les algorithmes usuels peuvent être classés en un certain nombre de grandes classes de complexité**.**

Quelques complexités de référence les plus utilisées: de la meilleure (algorithme le plus performant) à la pire (algorithme difficilement exploitable)**.**

| Complexités de référence | Notation | Fonction | Définition |
|---|---|---|---|
| **Complexité Constante** | **O(1)** | **f(n)=1** | Accéder au premier élément d'un ensemble de données. |
| **Complexité Logarithmique** | **O(logn)** | **f(n)=log(n)** | Couper un ensemble de données en deux parties égales, puis couper ces moitiés en deux parties égales. |
| **Complexité Linéaire** | **O(n)** | **f(n)=n** | Parcourir un ensemble de données. |
| **Complexité Quasi-linéaire** | **O(nlogn)** | **f(n)=nlog(n)** | Couper répétitivement un ensemble de données en deux et combiner les solutions partielles pour calculer la solution générale. |
| **Complexité Quadratique** | **$O(n^2)$** | **$f(n)= n^2$** | Parcourir un ensemble de données en utilisant deux boucles imbriquées. |
| **Complexité Cubique** | **$O(n^3)$** | **$f(n)= n^3$** | Parcourir un ensemble de données en utilisant trois boucles imbriquées. |
| **Complexité Polynomiale** | **$O(n^P)$** | **$f(n)= n^p$** | Parcourir un ensemble de données en utilisant P boucles imbriquées. |
| **Complexité Exponentielle** | **$O(2^n)$** | **$f(n)= 2^n$** | Générer tous les sous-ensembles possibles d'un ensemble de données. |

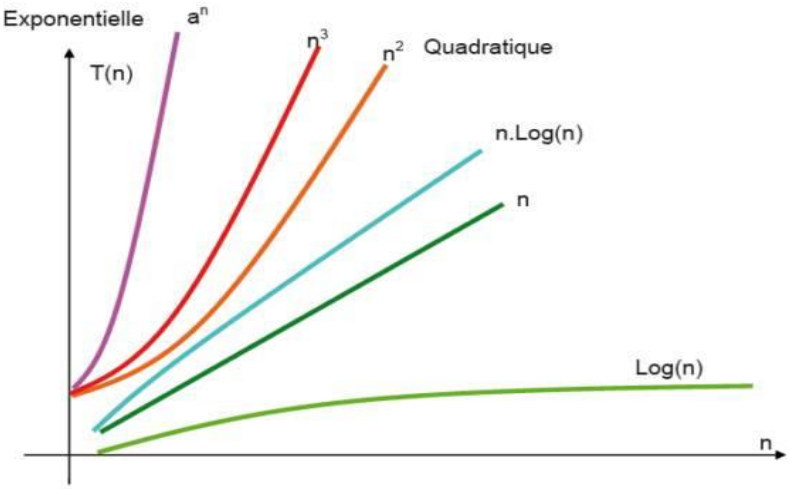


**Figure 2.** Classes de complexité

Algorithme par rapport à ces références : est-il au pire en $n^2$ ? Au mieux en $n^2$ ? Exactement en $n^2$ ? Pour cela, on a introduit et défini formellement les notations dites de **Landau** :

## 7.2. Notations asymptotique

### 7.2.1. Domination asymptotique

f et g étant des fonctions, f = O(g) s'il existe des constantes c>0 et n0 telles que
f(x) < c*g(x) pour tout x > n0

f = O(g) signifie que f est dominée (majorée) asymptotiquement par g.

**Définition 5.**
- Quand la complexité f(n) de l'algorithme est majorée par g(n), on dit qu'il est en O(g(n)).

$O(g(n)) = \{ f, \exists n0, \exists c>0, \forall n>n0, f(n) \leq c.g(n) \}$

Il existe un certain nombre à partir duquel f (n) est toujours inferieur à g(n)*constante.

La notation O, dite notation de Landau, vérifie les propriétés suivantes :
- si f=O(g) et g=O(h) alors f=O(h)
- si f=O(g) et k un nombre, alors k*f=O(g)
- si f1=O(g1) et f2=O(g2) alors f1+f2 = O(g2+g2)
- si f1=O(g1) et f2=O(g2) alors f1*f2 = O(g2*g2)

### 7.2.2. Notation Ω

f = Ω(g) s'il existe des constantes c>0 et n0 telles que f(x) ≥ c*g(x) pour tout x ≥ n0

**Définition 6.**
- Quand la complexité f(n) de l'algorithme est minorée par g(n), on dit qu'il est en Ω(g(n)).

$\Omega(g(n)) = \{ f, \exists n0, \exists c>0, \forall n>n0, f(n) \geq c.g(n) \}$

Il existe un certain nombre à partir duquel g(n) est toujours supérieur à f(n)*constante.

### 7.2.3. Equivalence asymptotique

f et g étant des fonctions, f = Θ(g) s'il existe des constantes c1, c2, strictement positives et n0 telles que $c1*g(x) \leq f(x) \leq c2*g(x)$ pour tout $x \geq n0$

**Définition 7.**
- Quand la complexité f(n) de l'algorithme est exactement en g(n), on dit qu'il est en Θ(g(n)).

S'il est exactement en Θ(g(n)), cela veut dire qu'il est borné par deux multiples de g(n). Il peut-être minoré et majoré par des multiples ; c'est donc l'intersection des classes de complexité précédentes : $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$

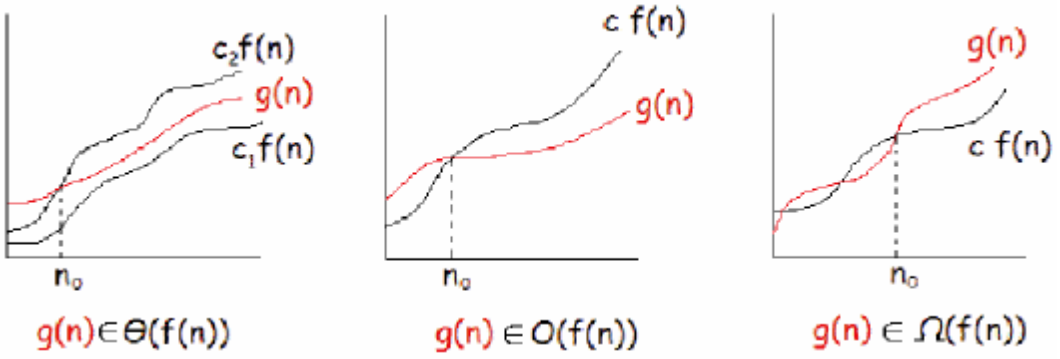


**Figure 3.** Les notations de Landau.

### 7.3. Calcul de complexité asymptotique

Nous concentrons sur le coût des actions résultant de l'exécution de l'algorithme, en fonction d'une "taille" n des données traitées. Ceci permet en particulier de comparer deux algorithmes traitant le même calcul. Nous sommes plus intéressés par un comportement asymptotique, que se passe -t'il quand n tend vers l'infini? que par un calcul exact pour n fixé. Le temps d'exécution dépend de la nature des données.

**Définitions 8.**
Borne supérieure non asymptotiquement approchée (négligeable)**.**
g(n) est une borne supérieure non asymtotiquement approchée de f(n) ou que f est négligeable devant g.
Si $\lim_{n\to\infty} f(n)/g(n) = 0$ alors On note $f(n) = o(g(n))$

**Propriétés**

- si $f(n) = \Theta(g(n))$ alors $f(n) = O(g(n))$
- si $f(n) = o(g(n))$ alors $f(n) = O(g(n))$
- $\Theta$ et O sont des relations réflexives
- et o sont des relations transitives
- $\Theta$ est une relation d'´equivalence
- pour tout $K \in IR$, $\Theta(K + f(n)) = \Theta(f(n))$
- pour tout $C \in IR$, $\Theta(C\, f(n)) = \Theta(f(n))$
- pour tout $j < k$, $\Theta(n^k + n^j) = \Theta(n^k)$

**Exemple 4 :**

```
{début}
K ← 0 ;
I ← 1 ;
{#1}
TantQue I ≤ N {#2} Faire
    R ← R+T[I]; {#3}
    I ← I+1;{#4}
FinTantQue;
{fin}.
```

Le temps d'exécution t(n) de cet algorithme en supposant que:

- N=n
- t1 est le temps d'exécution entre le début et {#1}
- t2 est le temps d'exécution de la comparaison {#2}
- t3 est le temps d'exécution de l'action {#3}
- t4 est le temps d'exécution de l'action {#4}
- t1, t2, t3, t4 sont des constantes (ne dépendent pas de n)

$t(n) = t1 + \sum_{i=1}^{n} (t2+t3+t4) + t2$
et en définissant le temps tit d'exécution d'une itération (condition comprise),
tit= (t2+t3+t4) d'où $t(n) = t1 + t2 + n \times tit$
Ce qui signifie que le temps d'exécution dépend linéairement ( plus précisément est une fonction affine) de la taille n.
Nous intéressions au comportement asymptotique:

Lim $_{n\to\infty}$ t(n)/tit×n =1 autrement dit t(n) est équivalent à l'infini à tit*n : t(n) ~$_{\infty}$ tit×n

⇨ L'algorithme est donc asymptotiquement linéaire en n.

Dans cet exemple simple l'évaluation en "pire des cas" est immédiate puisque t(n) ne dépend pas de la nature des données,