# *Algorithmique avancée*

# I. Introduction

Quand on tente de résoudre un problème, la question se pose souvent du choix d'un algorithme. Quels critères peuvent guider ce choix ? Deux besoins contradictoires sont fréquemment en présence : l'algorithme doit :

1. Etre simple à comprendre, à mettre en œuvre, à mettre au point.
2. Mettre intelligemment à contribution des ressources de l'ordinateur et, plus précisément, il doit s'exécuter le plus rapidement possible.

Si un algorithme ne doit servir qu'un petit nombre de fois, le premier critère est le plus important. Par contre, s'il doit être employé souvent, le temps d'exécution risque d'être un facteur plus déterminant que le temps passé à l'écriture.

L'étude de la complexité des algorithmes a pour objectif l'estimation du coût d'un algorithme. Cette mesure permet donc la comparaison de deux algorithmes sans avoir à les programmer.

Si l'on prend en compte pour l'estimation de la complexité les ressources de la machine telles que la fréquence d'horloge, le nombre de processeurs, le temps d'accès disque etc., on se rend compte immédiatement de la complication voire l'impossibilité d'une telle tâche. Pour cela, on se contente souvent d'estimer la **relation entre la taille des données et le temps d'exécution**, et ceci **indépendamment de l'architecture utilisée**.

Il s'agit d'un modèle simplifié qui tient compte des ressources technologiques ainsi que leurs coûts associés. On prendra comme référence un modèle de machine à accès aléatoire et à processeur unique où les opérations sont exécutées l'une après l'autre sans opérations simultanées.

Dans ce modèle, on appelle **opérations élémentaires** les opérations suivantes :

- Un accès mémoire pour lire ou écrire la valeur d'une variable ou d'une case d'un tableau ;
- Une opération arithmétique entre entiers ou réels telle que l'addition, soustraction, multiplication, division ou calcul du reste d'une division entière ;
- Une comparaison entre deux entiers ou réels.

⇒ ***Exemple :***

L'instruction « c ← a + b ; » nécessite les quatre opérations élémentaires suivantes :

- Un accès mémoire pour la lecture de la valeur de a,
- Un accès mémoire pour la lecture de la valeur de b,
- Une addition de a et b,
- Un accès mémoire pour l'écriture de la nouvelle valeur de c.

On définit ainsi la complexité (temporelle) d'un algorithme comme étant la mesure du nombre d'opérations élémentaires **T (n)** qu'il effectue sur un jeu de données **n**. La complexité est exprimée comme une fonction de la taille du jeu de données.

Pour comparer des solutions entre-elles, deux méthodes peuvent être utilisées :

- Méthode empirique
- Méthode mathématique

## *1. Méthode empirique*

Elle consiste à coder et exécuter deux (ou plus) algorithmes sur une batterie de données générées d'une manière aléatoire

À chaque exécution, le temps d'exécution de chacun des algorithmes est mesuré.

Ensuite, une étude statistique est entreprise pour choisir le meilleur d'entre-eux à la lumière des résultats obtenus.

⇒ ***Problème !***

Ces résultats dépendent

- La machine utilisée ;
- Jeu d'instructions utilisées
- L'habileté du programmeur
- Jeu de données générées
- Compilateur choisi
- L'environnement dans lequel est exécuté les deux algorithmes (partagés ou non)

### 2. *Méthode mathématique*

Pour pallier à ces problèmes, une notion de complexité plus simple mais efficace a été proposée par les informaticiens.

Ainsi, pour mesurer cette complexité, la méthode mathématique consiste non pas à la mesurer en secondes, mais à faire le décompte des instructions de base exécutées par ces deux algorithmes.

Cette manière de procéder est justifiée par le fait que la complexité d'un algorithme est en grande partie induite par l'exécution des instructions qui le composent.

Cependant, pour avoir une idée plus précise de la performance d'un algorithme, il convient de signaler que la méthode expérimentale et mathématique sont en fait complémentaires.

## II. Complexité asymptotique

La complexité asymptotique d'un algorithme décrit le comportement de celui-ci quand la taille n des données du problème traité devient de plus en plus grande, plutôt qu'une mesure exacte du temps d'exécution.

Ainsi, si **T(N) = 3+2N+1**, alors on dira que la complexité de cet algorithme est tout simplement en **N**. On a éliminé tout constante, et on a supposé aussi que les opérations d'affectation, de test et d'addition ont des temps constants.

D'une façon formelle, on appelle complexité d'un algorithme **a** tout **ordre de grandeur** du nombre d'opérations élémentaires effectuées pendant le déroulement de **a**.

### 1. *Notation O (grand o)*

Lorsque la fonction f est bornée **uniquement supérieurement** par la fonction g on utilise la notation **O**. **O(g(n))** désigne donc l'ensemble des fonctions positives de la variable n, pour lesquelles il existe une constante c et un entier $n_0$, satisfaisant la relation :

$$\mathbf{0 \leq f(n) \leq c\, g(n) \quad \forall\, n \geq n_0}$$

La relation **f(n) = O(g(n))** indique que la fonction f est bornée supérieurement par la fonction g pour des valeurs suffisamment grandes de l'argument n. Donc **f(n) = Θ(g(n))** implique **que f(n) = O(g(n))** (par abus de notation). Formellement, on peut écrire **Θ(g(n)) ⊆ O(g(n)).**

⇒ ***Exemple :***

- $\mathbf{100n^2 + 10n = O(n^2)}$ mais on peut aussi écrire **100n = O($n^2$) !** Car ceci est équivalent à dire que **100n** est asymptotiquement bornée supérieurement par $\mathbf{n^2}$.
- **f(n) = n log n + 12 n + 888 = O(n log n)**
- $\mathbf{f(n)=1000n -n+12n+(2^n/100)=O(2^n)}$

### 2. Quelques propriétés de la notation O

- Les facteurs constants peuvent être ignorés (Ex : $3x^2 = O(x^2)$)
- Une grande puissance de n croît plus vite qu'une puissance inférieure ($n^r = O(n^s)$ si $0 \leq r \leq s$)
- Le taux de croissance d'une somme de termes est le taux de croissance du terme le plus rapide en croissance. (Ex : $an^3 + bn^2 = O(n^3)$)
- Les fonctions exponentielles sont plus rapides en croissance que les puissances (polynômes)
- Tous les logarithmes ont un même taux de croissance.

## III. Différentes formes de complexité

Il est évident de la remarque précédente que la complexité d'un algorithme peut ne pas être la même pour deux jeux de données différents. Ceci peut avoir des conséquences sur le choix du meilleur algorithme pour un problème donné. En pratique, on s'intéresse aux formes suivantes de la complexité :

### 1. Complexité dans le pire des cas (Worst case)

$$T_{\max}(n) = \max_{d \in D_n} C(d)$$

C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à $n$.

**Avantage :** il s'agit d'un maximum, et l'algorithme finira donc toujours avant d'avoir effectué $T\max(n)$ opérations.

**Inconvénient :** cette complexité peut ne pas refléter le comportement « usuel » de l'algorithme. Le pire cas ne peut se produire que très rarement, mais il n'est pas rare que le cas moyen soit aussi mauvais que le cas pire.

### 2. Complexité dans le meilleur des cas (Best case)

$$T_{\min}(n) = \min_{d \in D_n} C(d)$$

C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée, ici à $n$. C'est une borne inférieure de la complexité de l'algorithme sur un jeu de données de taille $n$.

### 3. Complexité en moyenne (Average case)

$$T_{moy} = \frac{\sum_{d \in D_n} C(d)}{|D_n|}$$

Il s'agit de calculer la moyenne (espérance mathématique) des nombres d'opérations élémentaires effectuées sur la totalité des instances. Ce calcul est généralement très difficile et souvent même délicat à mettre en œuvre car il faut connaître la probabilité de chacun des jeux de données pour pouvoir calculer la complexité en moyenne. Cette forme d'analyse fait actuellement l'objet de nombreux travaux de recherche et permet d'expliquer, d'une part le comportement de certains algorithmes en pratique ; et d'autre part, le choix d'algorithmes pour des problèmes ayant des tailles considérables tels que les problèmes d'apprentissage en intelligence artificielle.

**Avantage :** reflète le comportement « général » de l'algorithme si les cas extrêmes sont rares ou si la complexité varie peu en fonction des données.

**Inconvénient :** en pratique, la complexité sur un jeu de données particulier peut être nettement plus importante que la complexité en moyenne, dans ce cas la complexité en moyenne ne donnera pas une bonne indication du comportement de l'algorithme. En pratique, nous ne nous intéresserons qu'à la complexité au pire et à la complexité en moyenne.

## 4. Illustration : cas du tri par insertion

### 4.1. Problématique du tri

**Données** : une séquence de *n* nombres, $a1$, ..., $an$.

**Résultats** : une permutation a'$_i$ de la séquence d'entrée, telle que $a'_i \leq a'_{i+1} \ \forall i$

### 4.2. Principe du tri par insertion

De manière répétée, on retire un nombre de la séquence d'entrée et on l'insère à la bonne place dans la séquence des nombres déjà triés (ce principe est le même que celui utilisé pour trier une poignée de cartes).

### 4.3. Algorithme

```
def Tri_Insertion(L):
    for i in range(1,len(L)):
        m=L[i]
        j=i
        while j>0 and L[j-1]>m:
            L[j]=L[j-1]
            j-=1
        L[j]=m
    return L
```

⇒ ***Explication***

- On retire un nombre de la séquence d'entrée.
- Les i-1 premiers éléments de A sont déjà triés.
- Tant que l'on n'est pas arrivé au début du tableau, et que l'élément courant est plus grand que celui à insérer. On décale l'élément courant (on le met dans la place vide).
- Finalement, on a trouvé où insérer notre nombre.

### 4.4. Complexité

Attribuons un coût en temps à chaque instruction, et comptons le nombre d'exécutions de chacune des instructions. Pour chaque valeur de $j \in [2,n]$ , nous notons $t_j$ le nombre d'exécutions de la boucle « while » pour cette valeur de *j*. Il est à noter que la valeur de $t_j$ **dépend des données**...

```
def Tri_Insertion(L):                    // coût        Nombre de fois
    for i in range(1,len(L)):            // coût C1 ........ N
        m=L[i]                           // coût C2 ........ N-1
        j=i                              // coût C3 ........ N-1
        while j>0 and L[j-1]>m:          // coût C4 ........ 2+3+...+N
            L[j]=L[j-1]                  // coût C5 ........ 1+2+...N-1
            j-=1                         // coût C6 ........ 1+2+...N-1
        L[j]=m                           // coût C7 ........ N-1
    return L
```

Le temps d'exécution total de l'algorithme est alors la somme des coûts élémentaires :

$$T(n)=c_1 n + c_2(n-1) + c_3(n-1) + c4 \sum_{j=2}^{N} tj + c5 \sum_{j=1}^{N-1} tj + c5 \sum_{j=1}^{N-1} tj + + c7\,(n-1)$$

#### 4.4.1. Complexité au meilleur :

Le cas le plus favorable pour l'algorithme TRI-INSERTION est quand le tableau est déjà trié.

Dans ce cas $t_j = 1$ pour tout j.

$$T(n) = c1\ n+c2\ (n\text{-}1)+c3\ (n\text{-}1)+c4(n\text{-}1)+c7(n\text{-}1)$$
$$= (c1+c2+c3+c4+c7)\ n\text{-}(c2+c3+c4+c7)$$

$T(n)$ peut ici être écrit sous la forme $T(n) = an+b$, $a$ et $b$ étant des *constantes* indépendantes des entrées, et **$T(n)$ est donc une fonction linéaire de $n$.** Le plus souvent, comme c'est le cas ici, le temps d'exécution d'un algorithme est fixé pour une entrée donnée ; mais il existe des algorithmes « aléatoires » intéressants dont le comportement peut varier même pour une entrée fixée.

#### 4.4.2. Complexité au pire :

Le cas le plus défavorable pour l'algorithme TRI-INSERTION est quand le tableau est déjà trié dans l'ordre inverse. Dans ce cas $t_j = j$ pour tout $j$.

*Rappel :*

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \quad \sum_{i=1}^{n-1} i = \frac{n(n+1)}{2}\text{-}1 \quad \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

$$T(n)=C1n + C2(n-1) + C3(n-1) + c4\ (\frac{n(n+1)}{2} - 1\ ) + C5(\frac{n(n-1)}{2}) + C6(\frac{n(n-1)}{2}) + c7\ (n-1)$$

$$T(n)= (\frac{C_4 + C_5 + C_6}{2})N^2 + (C_1 + C_2 + C_3 + \frac{C_4 - C_5 - C_6}{2} + C_7)N - (C_2 + C_3 + C_4 + C_7)$$

$T(n)$ est donc de la forme $T(n)=an^2+bn+c$

$T(n)$ est donc une **fonction quadratique** de $n$.

#### 4.4.3. Complexité en moyenne

Supposons que l'on applique l'algorithme de tri par insertion à $n$ nombres choisis au hasard. Quelle sera la valeur de $tj$ ? C'est-à-dire, où devra-t-on insérer A[ $j$] dans le sous-tableau A[1.. $j$-1] ?

En moyenne, la moitié des éléments de A[1.. $j$-1] sont inférieurs à A[ $j$], et l'autre moitié sont supérieurs. Donc $t_j = j/2$. Si l'on reporte cette valeur dans l'équation définissant $T(n)$, on obtient, comme dans le cas pire, une fonction quadratique en $n$.

Ce qui nous intéresse vraiment, c'est l'ordre de grandeur du temps d'exécution. Seul le terme dominant de la formule exprimant la complexité nous importe, les termes d'ordres inférieurs n'étant pas significatifs quand $n$ devient grand. On ignore également le coefficient multiplicateur constant du terme dominant. On écrira donc, à propos de la complexité du tri par insertion :

Complexité au meilleur = $O(n)$.

Complexité au pire = $O(n^2)$.

Complexité en moyenne = $O(n^2)$.

*En général, on considère qu'un algorithme est plus efficace qu'un autre si sa complexité dans le cas pire a un ordre de grandeur inférieur.*

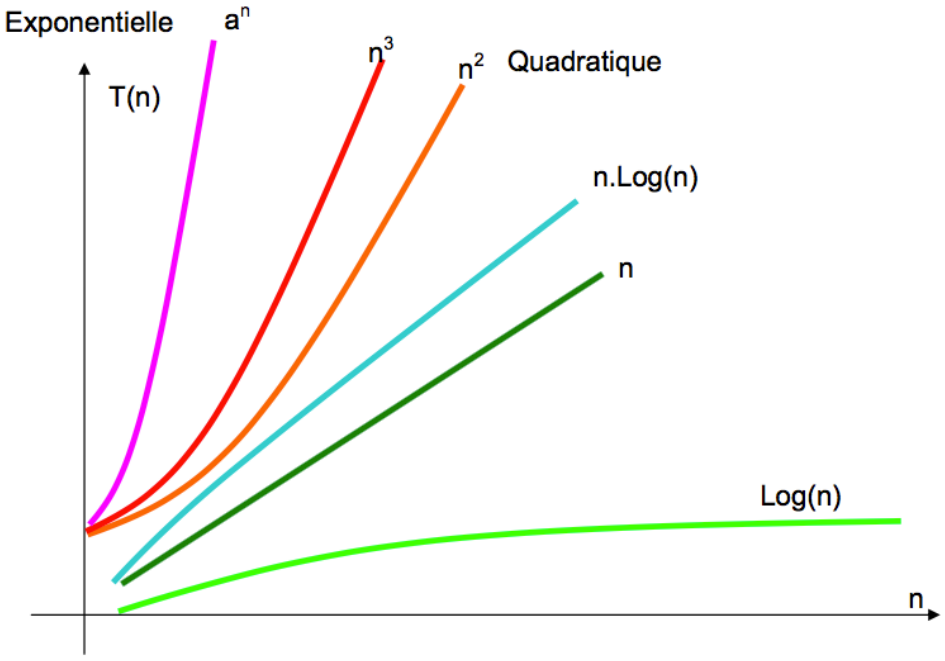
## 5. *Classes de complexité*

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

- ***O*(log*n*) :** Les algorithmes sub-linéaires dont la complexité est en général en $O(\log n)$.
- **O(n) :** Les algorithmes linéaires en complexité $O(n)$
- **O(nlogn) :** et ceux en complexité en $O(n\log n)$
- **O(nk) :** Les algorithmes polynomiaux en $O(n^k)$ pour $k > 3$
- **Exp(n) :** Les algorithmes exponentiels

Les trois premières classes sont considérées rapides alors que la quatrième est considérée lente et la cinquième classe est considérée impraticable.

| | $\log_2(n)$ | n | $n\log_2(n)$ | $n^2$ | $n^3$ | $n^4$ | exp(n) |
|---|---|---|---|---|---|---|---|
| 1 | 0,00 | 1 | 0,00 | 1 | 1 | 1 | 3 |
| 2 | 1,00 | 2 | 2,00 | 4 | 8 | 16 | 7 |
| 3 | 1,58 | 3 | 4,75 | 9 | 27 | 81 | 20 |
| 4 | 2,00 | 4 | 8,00 | 16 | 64 | 256 | 55 |
| 5 | 2,32 | 5 | 11,61 | 25 | 125 | 625 | 148 |
| 6 | 2,58 | 6 | 15,51 | 36 | 216 | 1296 | 403 |
| 7 | 2,81 | 7 | 19,65 | 49 | 343 | 2401 | 1097 |
| 8 | 3,00 | 8 | 24,00 | 64 | 512 | 4096 | 2981 |
| 9 | 3,17 | 9 | 28,53 | 81 | 729 | 6561 | 8103 |
| 10 | 3,32 | 10 | 33,22 | 100 | 1000 | 10000 | 22026 |
| 11 | 3,46 | 11 | 38,05 | 121 | 1331 | 14641 | 59874 |
| 12 | 3,58 | 12 | 43,02 | 144 | 1728 | 20736 | 162755 |
| 13 | 3,70 | 13 | 48,11 | 169 | 2197 | 28561 | 442413 |
| 14 | 3,81 | 14 | 53,30 | 196 | 2744 | 38416 | 1202604 |
| 15 | 3,91 | 15 | 58,60 | 225 | 3375 | 50625 | 3269017 |
| 16 | 4,00 | 16 | 64,00 | 256 | 4096 | 65536 | 8886111 |
| 17 | 4,09 | 17 | 69,49 | 289 | 4913 | 83521 | 24154953 |
| 18 | 4,17 | 18 | 75,06 | 324 | 5832 | 104976 | 65659969 |



# IV. La Récursivité, Souplesse et Complexité

## 1. Récursivité

De l'art et la manière d'élaborer des algorithmes pour résoudre des problèmes qu'on ne sait pas résoudre soi-même !

### 1.1. Définition

Une définition récursive est une définition dans laquelle intervient ce que l'on veut définir.

Un algorithme est dit récursif lorsqu'il est défini en fonction de lui-même.

### 1.2. Récursivité simple

Revenons à la fonction puissance $x \rightarrow x^n$. Cette fonction peut être définie récursivement :

$$X^n \begin{cases} 1 \ si \ n = 0 \\ X.X^{n-1} \ si \ n > 0 \end{cases}$$

L'algorithme correspondant s'écrit :

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

### 1.3. Récursivité multiple

Une définition récursive peut contenir plus d'un appel récursif.

On se propose de calculer le nombre de combinaisons $C_n^p$ *en se servant de la relation de Pascal :*

$$C_n^p = \begin{cases} 1 \ si \ p = 1 \ ou \ p = n \\ C_{n-1}^p + C_{n-1}^{p-1} \ sinon \end{cases}$$

L'algorithme correspondant s'écrit :

```
Combinaison (n, p) :
    if  p == 0 or p== n :
        return 1
    else :
        return (Combinaison (n-1, p) + Combinaison (n-1, p-1))
```

### 1.4. Récursivité mutuelle

Des définitions sont dites *mutuellement récursives* si elles dépendent les unes des autres. Ça peut être le cas pour la définition de la parité :

$$pair(n) \begin{cases} vrai \ \ si \ n = 0 \\ impair(n-1) \ sinon \end{cases} \qquad impair(n) \begin{cases} faux \ \ si \ n = 0 \\ pair(n-1) \ sinon \end{cases}$$

Les algorithmes correspondants s'écrivent :

```
def pair(n) :
    if n==0 :
        return True
    else :
        return impair(n-1)
```

```
def impair(n) :
    if n==0 :
        return False
    else :
        return pair(n-1)
```

### *1.5. Récursivité imbriquée*

La fonction d'Ackermann est définie comme suit :

$$A(m,n)\begin{cases} n+1\ si\ m=0 \\ A(m-1,1) si\ n=1\ et\ m>0 \\ A\big(m-1,A(m,n-1)\big) sinon \end{cases}$$

D'où l'algorithme :

```
def Ackermann (m, n) :
    if m==0 :
        return n+1
    elif n==1 :
        return Ackermann (m-1,1)
    else :
        return Ackermann (m-1, Ackermann (m,n-1))
```

### *1.6. Principe et dangers de la récursivité*

#### *1.6.1. Principe et intérêt :*

Ce sont les mêmes que ceux de la démonstration par récurrence en mathématiques. On doit avoir :

- Un certain nombre de cas dont la résolution est connue, ces «cas simples» formeront les cas d'arrêt de la récursivité
- Un moyen de se ramener d'un cas « compliqué » à un cas «plus simple». La récursivité permet d'écrire des algorithmes concis et élégants.

#### *1.6.2. Difficultés :*

La définition peut être dénuée de sens :

```
Algorithme A(n)
    renvoyer A(n)
```

Il faut être sûr qu'on retombera toujours sur un cas connu, c'est-à-dire sur un cas d'arrêt ; il nous faut nous assurer que la fonction est complètement définie, c'est-à-dire, qu'elle est définie sur tout son domaine d'applications.

**Moyen** : existence d'un ordre strict tel que la suite des valeurs successives des arguments invoqués par la définition soit strictement monotone et finit toujours par atteindre une valeur pour laquelle la solution est explicitement définie.

**Exemple :** L'algorithme ci-dessous teste si *a* est un diviseur de *b*.

```
def Diviseur (a,b) :
    If (a <=0) :
        return False
    elif a>=b :
        return (a=b)
    else :
        return Diviseur (a,b-a)
```

La suite des valeurs *b*, *b*-*a*, *b*-2*a*, etc. est strictement décroissante, car *a* est strictement positif, et on finit toujours par aboutir à un couple d'arguments (*a*,*b*) tel que *b*-*a* est négatif, cas défini explicitement.

### 1.6.3. Importance de l'ordre des appels récursifs

Fonction qui affiche les entiers par ordre décroissant, de *n* jusqu'à 1 :

```
def Décroissant (n)
    if (n = 0 ) :
        print(‘’)
    else :
        print(n)
        Décroissant (n-1)
```

Décroissant(2)-> affiche :2 suivi de 1

```
def Croissant (n)
    if (n = 0 ) :
        print(‘’)
    else :
        Croissant (n-1)
        print(n)
```

Croissant(2) -> affiche : 1 suivi de 2

## 2. Dérécursivation

Dérécursiver, c'est transformer un algorithme récursif en un algorithme équivalent ne contenant pas d'appels récursifs.

**Exemple 1 :** Est-ce que a est diviseur de b ?

| Version récursive | Version dérécursivée |
|---|---|
| def Diviseur (*a*,*b*) :<br>If (*a* <=0) :<br>return False<br>elif a>=b :<br>return (a=b)<br>else :<br>return Diviseur (*a*,*b*-*a*) | def Diviseur (*a*,*b*) :<br>If (*a* <=0) :<br>return False<br>while b>a :<br>b-=a<br>return (a=b) |

**Exemple 2 :** Factoriel (N) ?

| Version récursive | Version dérécursivée |
|---|---|
| def Factoriel (*N*) :<br>If N==0 :<br>return 1<br>else :<br>return N* Factoriel (*N-1*) | def Factoriel (*N*) :<br>F=1<br>for i in range(N,0,-1) :<br>F*=i<br>return F |

*Les programmes itératifs sont souvent plus efficaces, mais les programmes récursifs sont plus faciles à écrire. Il est toujours possible de dérécursiver un algorithme récursif.*

# V. Diviser pour régner

## 1. Principe

De nombreux algorithmes ont une structure récursive : pour résoudre un problème donné, ils s'appellent eux-mêmes récursivement une ou plusieurs fois sur des problèmes très similaires, mais de tailles moindres, résolvent les sous problèmes de manière récursive puis combinent les résultats pour trouver une solution au problème initial.

Le paradigme « diviser pour régner » donne lieu à trois étapes à chaque niveau de récursivité :

**Diviser :** le problème en un certain nombre de sous-problèmes ;

**Régner :** sur les sous-problèmes en les résolvant récursivement ou, si la taille d'un sous-problème est assez réduite, le résoudre directement ;

**Combiner :** les solutions des sous-problèmes en une solution complète du problème initial.

⇒ ***Exemple 1 : Recherche du maximum d'un tableau***

```
def Maximum(L) :
    if len(L)==1 :
        return L[0]
    else :
        m=len(L)//2 # division du problème en 2 sous-problèmes
        k1=Maximum(L[ :m]) // régner sur le 1er sous-problème
        k2=Maximum(L[m+1:]) // régner sur le 2ème sous-problème
        if k1>k2 : // combiner les solutions
            return k1
        else :
            return k2
```

⇒ ***Exemple 2 : multiplication naïve de matrices***

L'algorithme classique est le suivant : Soit $n$ la taille des matrices carrés $A$ et $B$

```
def produitMat(A,B) :
    n=len(A)
    C=[n*[0]]*n
    for i in range(0,n) :
        for j in range (0,n) :
            s=0
            for k in range (0,n) :
                s = s + A[i][k]*B[k][j]
            C[i][j]=s
    return C
```

Cet algorithme effectue $O(n^3)$ multiplications et autant d'additions.

- ***Algorithme « diviser pour régner » naïf***

Dans la suite nous supposerons que $n$ est une puissance de 2. Décomposons les matrices $A$, $B$ et $C$ en sous-matrices de taille $n/2$ x $n/2$. L'équation $C = AB$ peut alors se récrire :

$$\begin{Bmatrix} r & s \\ t & u \end{Bmatrix} = \begin{Bmatrix} a & b \\ c & d \end{Bmatrix} * \begin{Bmatrix} e & g \\ f & h \end{Bmatrix}$$

En développant cette équation, nous obtenons :

$$r = ae+bf\ ;\ s = ag+bh;\ t = ce+df \text{ et } u = cg+dh:$$

Chacune de ces quatre opérations correspond à :

- Deux multiplications de matrices carrées de taille $n/2$ -> 2T(n/2)
- Et une addition de telles matrices -> $O(n^2)$

A partir de ces équations on peut aisément dériver un algorithme « diviser pour régner » dont la complexité est donnée par la récurrence : $T(n) = 8T(n/2)+O(n^2)$

## 2. Analyse des algorithmes « diviser pour régner »

Lorsqu'un algorithme contient un appel récursif à lui-même, son temps d'exécution peut souvent être décrit par une équation de récurrence qui décrit le temps d'exécution global pour un problème de taille $n$ en fonction du temps d'exécution pour des entrées de taille moindre.

La récurrence définissant le temps d'exécution d'un algorithme « diviser pour régner » se décompose suivant les trois étapes du paradigme de base :

⇒ ***Si la taille du problème est suffisamment réduite, n <= c pour une certaine constante c, la résolution est directe et consomme un temps constant O(1).***

⇒ ***Sinon, on divise le problème en a sous-problèmes chacun de taille 1/b de la taille du problème initial. Le temps d'exécution total se décompose alors en trois parties :***

- $D(n)$ : le temps nécessaire à la division du problème en sous-problèmes.
- $aT(n/b)$ : le temps de résolution des $a$ sous-problèmes.
- $C(n)$ : le temps nécessaire pour construire la solution finale à partir des solutions aux sous-problèmes.

La relation de récurrence prend alors la forme :

$$T(n)\begin{cases} O(1)\ si\ n <= c \\ aT\left(\frac{n}{2}\right) + D(n) + C(n)\ sinon \end{cases}$$

# VI. Résolution des récurrences

## 1. *Équations de récurrence linéaires*

Exemple : La suite de Fibonacci est définie par :

$$Un\begin{cases} 1 \ si \ n < 2 \\ U_{n-1} + U_{n-2} \ sinon \end{cases}$$

Un algorithme récursif pour calculer le n$^{ème}$ terme de la suite est :

```
def Fib(n) :
    if n<2 :
        return 1
    else :
        return Fib(n-1)+Fib(n-2)
```

Soit T(n) le nombre d'additions effectuées par cet algorithme. T(n) vérifie les équations suivantes :

$$T(n) = \begin{cases} 0 \ si \ n = 0 \\ 0 \ si \ n = 1 \\ 1 + T(n-1) + T(n-2) \ si \ n > 1 \end{cases}$$

Cette équation est linéaire et d'ordre 2 car chaque terme de rang >1 dépend uniquement des deux termes qui le précèdent.

Définition : Une équation récurrente est dite linéaire d'ordre k si chaque terme s'exprime comme combinaison linéaire des k termes qui le précèdent plus une certaine fonction de n.

$$u_n = \alpha_1.u_{n-1} + \alpha_2.u_{n-2} + ... + \alpha_k.u_{n-k} + f(n)$$

Il faut bien sûr connaître les k premiers termes. A l'ordre 1, l'équation devient :

$$u_n = a.u_{n-1} + f(n)$$

Par itération et sommation on trouve :

$$u = a^n\left(u_0 + \sum_{i=1}^{n} \frac{f(i)}{a^i}\right)$$

## 2. *Résolution des récurrences « diviser pour régner »*

### 2.1. *Théorème 1 (Résolution des récurrences « diviser pour régner »).*

Soient a >=1 et b > 1 deux constantes, soit f (n) une fonction et soit T(n) une fonction définie pour les entiers positifs par la récurrence :

$$T(n) = a.T(n/b) + f(n) \ \text{ou} \ T(n) = a.T(n/b) + c.n^k$$

T(n) peut alors être bornée asymptotiquement comme suit :

| | Si on a : | Alors le coût est : |
|---|---|---|
| **1** | $a>b^k$ | $T(n)=O(n^{\log_b a})$ |
| **2** | $a=b^k$ | $T(n)=O(n^k \log n)$ |
| **3** | $a<b^k$ | $T(n) = O(f(n)) = O(n^k)$ |

⇒ ***Signification intuitive du théorème :***

Dans chaque cas, on compare $f(n)$ *avec* $n^{\log a}$ . La solution de la récurrence est déterminée par la plus grande des deux.

### *2.2. Exemples :*

#### *2.2.1. La multiplication de matrices :*

La relation de récurrence est : $T(n) = 8T(n/2)+O(n^2)$

Donc: $a$=8,b=2, k=2

1er cas $\log_b a = 3$ $T(n) = \Theta(n^3)$

#### *2.2.2. Exemple-2*

La relation de récurrence est : $T(n)=9T(\frac{n}{3}) + n$

On est dans le cas : a=9 , b=3 et f(n)=n

9>3 -> cas-1 $T(n)=O(n^{\log_b a})=O(n^{\log_3 9})=O(n^2)$

## *3. Autres récurrences :*

D'autres récurrences peuvent ne pas trouver de solution avec les techniques présentées. Dans ce cas, on peut procéder par itération :

- Calculer les quelques premières valeurs de la suite
- Chercher une régularité
- Poser une solution générale en hypothèse
- La prouver par induction

## *4. Pourquoi estimer la complexité d'un algorithme ?*

Considérons l'exemple de la suite de Fibonacci, et écrivons deux programmes différents : un récursif et l'autre non récursif.

| **Version récursive** | **Version itérative** |
|---|---|
| def Fib(n) :<br>  if n<2 :<br>  return 1<br>  else :<br>  return Fib(n-1)+Fib(n-2) | def Fib(n) :<br>  f1=f2=1<br>  for i in range(3,n) :<br>  f=f1+f2<br>  f1=f2<br>  f2=f<br>  print(f) |
| **T(n)=O($a^n$) -> exponentielle** | **T(n)= O(n)** |

Si on suppose qu'une opération d'addition nécessite 100ns, alors, pour calculer Fibo(n), on obtient les résultats suivants :

| | Nb-additions | | Nb-Secondes | | |
|---|---|---|---|---|---|
| n | Non Récursif | Récursif | Non Récursif | Récursif | Récursif |
| 20 | 19 | 10945 | 0,0000019 | 0,0010945 | 2*10-5 mn |
| 30 | 29 | 1346260 | 0,0000029 | 0,134626 | 2*10-3 mn |
| 40 | 39 | 165580140 | 0,0000039 | 16,558014 | 3*10-1 mn |
| 50 | 49 | 3185141889 | 0,0000049 | 318,514189 | 5 mn |
| 100 | | | | | 500 000 années |

Coût théorique = $(1+sqrt(5)/2)^n$

| n | Cout | sec | min | h | jour | an |
|---|---|---|---|---|---|---|
| 10 | 122,991869 | 1,23E-05 | 2,05E-07 | 3,42E-09 | 1,42E-10 | 3,90E-13 |
| 20 | 15126,9999 | 1,51E-03 | 2,52E-05 | 4,20E-07 | 1,75E-08 | 4,80E-11 |
| 30 | 1860498 | 1,86E-01 | 3,10E-03 | 5,17E-05 | 2,15E-06 | 5,90E-09 |
| 40 | 228826127 | 2,29E+01 | 3,81E-01 | 6,36E-03 | 2,65E-04 | 7,26E-07 |
| 50 | 2,8144E+10 | 2,81E+03 | 4,69E+01 | 7,82E-01 | 3,26E-02 | 8,92E-05 |
| 60 | 3,4615E+12 | 3,46E+05 | 5,77E+03 | 9,62E+01 | 4,01E+00 | 1,10E-02 |
| 70 | 4,2573E+14 | 4,26E+07 | 7,10E+05 | 1,18E+04 | 4,93E+02 | 1,35E+00 |
| 80 | 5,2361E+16 | 5,24E+09 | 8,73E+07 | 1,45E+06 | 6,06E+04 | 1,66E+02 |
| 90 | 6,44E+18 | 6,44E+11 | 1,07E+10 | 1,79E+08 | 7,45E+06 | 2,04E+04 |
| 100 | 7,9207E+20 | 7,92E+13 | 1,32E+12 | 2,20E+10 | 9,17E+08 | 2,51E+06 |

*Il ne faut pas toujours utiliser la récursivité, il faut estimer la complexité de son algorithme*