1. Définitions

- **Complexité des algorithmes :** C'est l'étude de l'efficacité comparée des algorithmes. On mesure le temps et/ou l'espace mémoire nécessaire à un algorithme pour résoudre un problème. Cela peut se faire de façon expérimentale (codage et exécution des algorithmes) ou formelle (analyse mathématique).
- **Complexité temporelle:** la complexité temporelle d'un algorithme est le temps mis par ce dernier pour transformer les données du problème considéré en un ensemble de résultats.
- **Complexité spatiale:** la complexité spatiale d'un algorithme est l'espace mémoire utilisé par ce dernier pour transformer les données du problème considéré en un ensemble de résultats.

2. Complexité temporelle d'un algorithme

2.1. Mesure de la complexité temporelle d'un algorithme

La complexité temporelle, notée **T(n),** d'un algorithme est le décompte du nombre d'opérations fondamentales qu'il effectue sur un jeu de données (une instance du problème). La complexité est exprimée comme une fonction de la taille du jeu de données (n). Il y a lieu de distinguer trois mesures de complexité:

- dans le meilleur cas (cas favorable)
- dans le pire cas (cas défavorable)
- dans le cas moyen

Soit Dn l'ensemble des jeux de données de taille n et $T_d(n)$ la complexité de l'algorithme sur un jeu de donnée $d \in Dn$.

- ✓ ***Complexité au meilleur $T_{meil}(n)$:*** C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée (*n*).

$$T_{meil}(n) = Min\ \{T_d(n)\}_{d \in Dn}$$

- ✓ ***Complexité au pire $T_{pire}(n)$***: C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille fixée (*n*).

$$T_{pire}(n) = Max\ \{T_d(n)\}_{d \in Dn}$$

- ✓ ***Complexité en moyenne $T_{moy}(n)$:*** C'est la moyenne des complexités de l'algorithme sur des jeux de données de taille *n* (en toute rigueur, il faut bien évidemment tenir compte de la probabilité d'apparition de chacun des jeux de données).

$$T_{moy}(n) = \sum_{d} P_d . T_d(n)$$

Avec P_d une loi de probabilité sur les jeux de données (fréquemment on utilise une loi de probabilité uniforme).

Remarque: Pour certains algorithmes, il n'y a pas lieu de distinguer entre ces trois mesures de complexité.

Exemple 1:

Calculer la complexité temporelle T(n) pour chacune des fonctions suivantes?

Fonction S1 (Val N : entier) : entier
Début
S1 ←N*(N+1)/2 **T(n)=1+1+1+1= 4**
Fin
Finfonction

Fonction S2 (Val N : entier) : entier
Variables S,nb : entier
Debut
S ← 0 ……………………..1 T1(n)=1
nb ←1 ……………………..2 T2(n)=1
Tant que nb ≤ N faire ……..3 T3(n)=1*(n+1) = n+1
S← S+ nb……………...4 T4(n)=2*n = 2n
nb ←nb+1 ……………5 T5(n)=2*n = 2n
Fintantque
S2←S ……………………6 T6(n)=1
Fin
Finfonction **T(n)= T1(n) + T2(n) + T3(n) + T4(n) + T5(n) + T6(n)= 5n+4**

Exemple 2:
Calculer la complexité temporelle T(n) en **nombre de comparaisons** pour un algorithme qui permet de rechercher l'existence d'une valeur X dans un tableau d'entiers.

Fonction RechercheSeq (Val X : entier, Val T : Tableau [1..N] d'entiers) : booléen
Variables
I: entier
Exist: booléen
Début
Exist ← Faux
I ←1
Tant que (I ≤ N) et (Non Exist) Faire
Si T[I] = X alors
Exist ← Vrai
Sinon
I ←I+1
Fsi
Fintantque
RechercheSeq ←Exist
Fin
Finfonction

- **Complexité au Meilleur:** X est la valeur du premier élément du tableau T

  $T_{Meil} = 1$

- **Complexité au Pire:** la valeur X n'existe pas dans le tableau T

  $T_{Pire} = N$

- **Complexité Moyenne:**

Pour simplifier le calcul de la complexité moyenne, on suppose que la valeur X existe dans le tableau T et on utilise une loi de probabilité uniforme.

$$T_{Moy} = \sum_{I=1}^{n} P_I * T_I(n) = \sum_{I=1}^{n} \frac{1}{n} * I = \frac{1}{n}(1+2+\ldots+n) = \frac{n+1}{2}$$

2.2. Taille du jeu de données

La définition de la taille du jeu de données dépend du problème. En pratique, on choisit comme taille la ou les dimensions les plus significatives.

Exemples : selon que le problème est modélisé par :

- des listes, tableaux, fichiers : nombre d'éléments.
- des mots : leur longueur
- des nombres : ces nombres
- des polynômes : le degré, le nombre de coefficients ≠ 0
- des graphes : ordre, taille, produit des deux

2.3. Repérage des opérations fondamentales

C'est la nature du problème qui fait que certaines opérations deviennent plus fondamentales que d'autres dans un algorithme. Leur nombre intervient alors principalement dans l'étude de la complexité de l'algorithme.

Exemples :

Problème	*Opérations fondamentales*
Recherche d'un élément *dans une liste, tableau, un arbre …*	***Comparaisons***
Tri *d'une liste, d'un tableau, d'un fichier …*	***Comparaisons*** ***Déplacements***
Multiplication *de polynômes, de matrices, de grands entiers…*	***Additions*** ***Multiplications***

3. Notations asymptotiques

Quand nous calculerons la complexité d'un algorithme, nous ne calculerons généralement pas sa complexité exacte, mais son ordre de grandeur. Pour ce faire, nous avons besoin de notations asymptotiques.

1. Définitions

Soient f et g deux fonctions de N dans R.

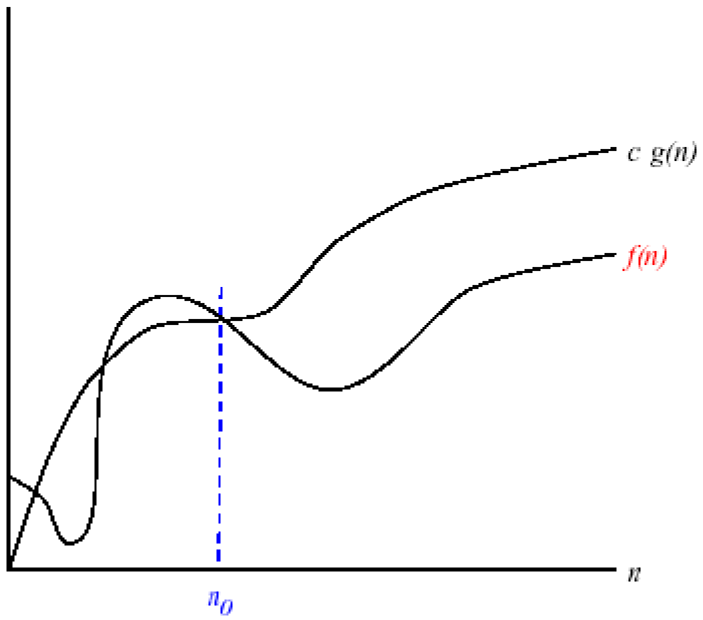
- **Définition 1 (notation grand O):**

On dit que $g(n)$ est **une *borne supérieure asymptotique*** pour $f(n)$ et l'on écrit $f(n) \in O(g(n))$ s'il existe des constantes ***strictement positive*** c et n_0 telles que, pour n assez grand, on ait $0 \leq f(n) \leq c \bullet g(n)$ $\forall n \geq n_0$.

Ceci revient à dire que $f(n)$ *est plus petit de* $g(n)$ à un facteur constant près pour n assez grand.

Pour indiquer que $f(n) \in O(g(n))$, on écrit $f(n) = O(g(n))$.

Illustration graphique



$f(n) = O(g(n))$

<u>Exemple:</u> montrer que si $f(n) = (n + 1)^2$ alors $f(n) = O(n^2)$.

On doit trouver c, $n_0 > 0$ telle que $(n + 1)^2 \leq c\, n^2$, pour $n \geq n_0$.

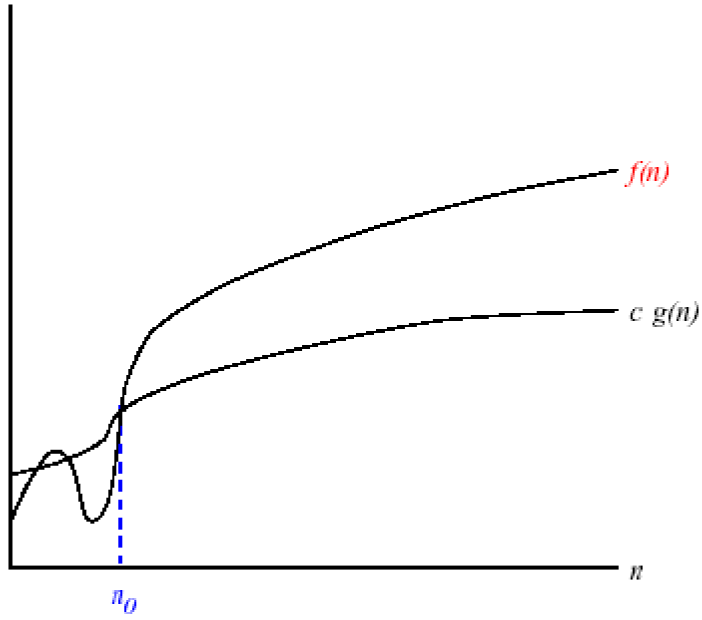
- **Définition 2 (notation grand Ω)**

On dit que $g(n)$ est **une *borne inférieur asymptotique*** pour $f(n)$ et l'on écrit $f(n) \in \Omega(g(n))$ s'il existe des constantes ***strictement positives*** c et n_0 telles que, pour n assez grand, on ait $c \bullet g(n) \leq f(n)$ $\forall n \geq n_0$.

Ceci revient à dire que $f(n)$ *est plus grand de* $g(n)$ à un facteur constant près pour n assez grand.

Pour indiquer que $f(n) \in \Omega(g(n))$, on écrit $f(n) = \Omega(g(n))$.

Illustration graphique



$f(n) = \Omega(g(n))$

Exemple: montrer que la fonction $3n^3 + 2n^2 = \mathbf{\Omega}(n^3)$.

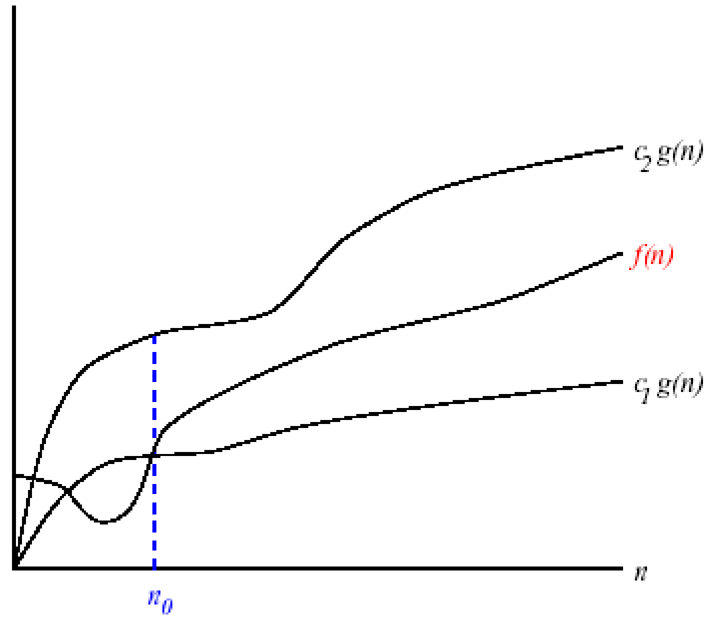
On doit trouver c, $n_0 > 0$ telle que $3n^3 + 2n^2 \geq c\, n^3$, pour $n \geq n_0$.

- **Définition 3 (notation grand θ):**

On dit que $g(n)$ est une ***borne approchée asymptotique*** pour $f(n)$ et l'on écrit $f(n) \in \theta(g(n))$ s'il existe des constantes ***strictement positives*** c_1, c_2 et n_0 telles que, pour n assez grand, on ait $0 \leq c_1 \bullet g(n) \leq f(n) \leq c_2 \bullet g(n)$ $\forall n \geq n_0$.

Ceci revient à dire que $f(n)$ *est égale* à $g(n)$ à un facteur constant près. Pour indiquer que $f(n) \in \theta(g(n))$, on écrit $f(n) = \theta(g(n))$.

Illustration graphique



$f(n) = \Theta\ (g(n))$

Exemple: montrer que $f(n) = n^2 - 3n = \theta\,(n^2)$.

On doit trouver c_1, c_2, $n_0 > 0$ telles que $c_1 \bullet n^2 \leq n^2 - 3n \leq c_2 \bullet n^2$ pour $n \geq n_0$.

2. Propriétés

- $f(n) = \theta\,(g(n))$ ***Si et seulement si*** $f(n) = O(g(n))$ et $f(n) = \Omega(g(n))$
- Toutes les notations (**θ, O,Ω**) sont ***transitives et réflexives***
- Seule la notation θ est **symétrique** $f(n) = \theta\,(g(n))$ ***Si et seulement si*** $g(n) = \theta(g(n))$
- de plus on a **la symétrie transposée suivante :** $f(n) \in O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

3. Quelques règles utiles pour simplifier les calculs (*pour O mais aussi pour θ et Ω*)

- Si f(n) = O(kg(n)) où k > 0, une constante **Alors**
f(n) = O(g(n)). Les constantes sont ignorées

- Si f1(n) = O(g1(n)) et f2(n) = O(g2(n)) **Alors**
f1(n) + f2(n) = O(max(g1(n), g2(n)))

- Si f1(n) = O(g1(n)) et f2(n) = O(g2(n)) **Alors**
f1(n)*f2(n) = O(g1(n)* g2(n))

Exemple:
Considérez l'algorithme suivant:

```
Début
   Partie1
   I ← 1
   Répéter
      Partie2
      I ← I + 1
   Jusqu'à (I > N)
Fin
```

Sachant que :

- ***Partie1*** à une complexité temporelle O(n)
- ***Partie2*** à une complexité temporelle $O(\log_2(n))$,

Déterminez la complexité de l'algorithme en notation asymptotique O.

4. Complexité des algorithmes récursifs

Pour calculer la complexité temporelle d'un algorithme récursif il faut:

- Exprimer la complexité temporelle sous forme d'une équation de récurrence
- Résoudre l'équation de récurrence

Exemple : calculer la complexité temporelle de la fonction suivante:

```
Fonction Fact (Val N : Entier) : Entier
 Début
      Si N = 0 Alors
           Fact ← 1
      Sinon
           Fact ← N  *  Fact( N-1)
      FSi
 Fin
FinFonction
```

- Expression de la complexité temporelle sous forme d'une équation de récurrence :

$$\mathbf{T(n)}=\begin{cases} \mathbf{2} & \boldsymbol{Si\ n = 0} \\ \mathbf{5 + T(n-1)} & \boldsymbol{Sinon} \end{cases}$$

- Résolution de l'équation de récurrence :

$$\begin{aligned} \mathbf{T(n)} &= \mathbf{5 * 1 + T(n-1) = 5*1 + 5 + T(n-2)} \\ &= \mathbf{5*2 + T(n-2) = 2*5 + 5 + T(n-3)} \\ &= \mathbf{5*3 + T(n-3)} \\ &\dots \\ &= \mathbf{5*n + T(0) = 5n + 2} \end{aligned}$$

- **Méthode générale**

La méthode générale donne une recette pour résoudre les récurrences de la forme :

$$\mathbf{T(n) = a\,T(n / b) + f(n)}.$$

La méthode générale s'appuie sur le Théorème suivant:

Théorème général: soient $a \geq 1$ et $b > 1$ deux constantes, f(n) une fonction asymptotiquement positive et T(n) définie pour les entiers naturels par la récurrence $T(n)= aT(n/b) + f(n)$. T(n) peut alors être bornée asymptotiquement de la façon suivante:

1. Si $f(n) = O(n^{\log_b a-\varepsilon})$ pour une certaine constante $\varepsilon>0$, alors $T(n)= \theta\,(n^{\log_b a})$
2. Si $f(n) = \theta\left(n^{\log_b a}\right)$, alors $T(n)= \theta\,(n^{\log_b a}\log(n))$
3. Si $f(n)= \Omega(n^{\log_b a+\varepsilon})$ pour une certaine constante $\varepsilon>0$, et si $a^* f(n/b) \leq c^* f(n)$ pour une certaine constante $c<1$ et pour tout n suffisamment grand, alors $T(n) = \theta\,(f(n))$

<u>Exemple:</u> résoudre les équations de récurrence suivantes:

- $T(n)= T(n/2) +1$
- $T(n)= 9T(n/3) + n$
- $T(n)= 2T(n/2) + n$

5. Classification des algorithmes

Les algorithmes habituellement rencontrés peuvent être classés dans les catégories suivantes:

- **<u>Complexité O(1):</u>** Complexité constante. On rencontre cette complexité quand toutes les instructions sont exécutées une seule fois qu'elle que soit la taille n du problème.
- **<u>Complexité O(log(n)):</u>** Complexité logarithmique. La durée d'exécution croit légèrement avec n. Ce cas de figure se rencontre quand la taille du problème est divisée par une entité constante à chaque itération.
- **<u>Complexité O(n):</u>** Complexité linéaire. C'est typiquement le cas d'un algorithme avec une boucle de 1 à n et le corps de la boucle effectue un travail de durée constante et indépendante de n.
- **<u>Complexité O(nlog(n)):</u>** Complexité n-logarithmique. Se rencontre dans les algorithmes où à chaque itération la taille du problème est divisée par une constante avec à chaque fois un parcours linéaire des données.
- **<u>Complexité $O(n^2)$:</u>** Complexité quadratique. Typiquement c'est le cas d'algorithmes avec deux boucles imbriquées chacune allant de 1 à n et avec le corps de la boucle interne qui est constant.
- **<u>Complexité $O(n^3)$:</u>** Complexité cubique. Typiquement c'est le cas d'algorithmes avec trois boucles imbriquées.
- **<u>Complexité $O(2^n)$:</u>** Complexité exponentielle. Les algorithmes de ce genre sont dits "naïfs" car ils sont inefficaces et inutilisable dès que n dépasse 50.

Remarques:

- On dit qu'un algorithme A est meilleur qu'un algorithme B si et seulement si:

  $\mathbf{T_A(n) = O(T_B(n))}$ Où $T_A(n)$ et $T_B(n)$ sont les complexités des algorithmes A et B.

- Un algorithme est dit efficace si sa complexité temporelle asymptotique est dans O(P(n)) où P(n) est un polynôme et n la taille des données du problème considéré.