Chapitre II : La complexité Algorithmique

Introduction : Soit à écrire un algorithme A pour trier d'un tableau de 100 cases, et un algorithme B pour trier un tableau de 10^6 cases (sachant qu'il existe plusieurs méthodes de tri). Quels sont les critères à prendre en compte pour l'évaluation des 2 algorithmes A et B ? Pour A, on prend uniquement le critère **résultat.** Par contre pour B, on prend, en plus du critère **résultat,** le critère **temps,** car la taille des données est grande.

L'évaluation du temps d'exécution d'un algorithme, manipulant une grande quantité de données, est un paramètre très important avant toute phase de développement ; car développer un algorithme fournissant de bons résultats mais qui met beaucoup de temps à s'exécuter est peut intéressant.

Temps d'exécution d'un algorithme : dépend des facteurs suivants :

- *Les données utilisées par le programme* (taille des données) ;
- La qualité du code généré par le compilateur (langage utilisé) ;
- La machine utilisée (vitesse, mémoire,. . .) ;
- La complexité de l'algorithme lui-même (nombre d'instructions).

Complexité algorithmique temporelle (en temps) d'un algorithme A : mesure le nombre d'opérations élémentaires effectuées par l'algorithme A en fonction de la taille des données n, indépendamment de la machine et du langage utilisés, exprimé par $C_A(n)$.

La taille des données n correspond à la quantité d'informations manipulée par l'algorithme. Par exemple, dans le cas des tableaux, n est égale au nombre D'éléments du tableau, dans le cas des graphes le nombre n est égal aux nombre de sommets plus le nombre d'arcs, etc.

Opérations élémentaires : affectations, comparaisons, opérations arithmétiques, branchement, R/W, appel de sous-programmes, etc. $C_{opérat\_élément}=1$.

Objectif de calcul de la complexité : Elle permet en particulier de comparer deux algorithmes résolvant le même problème.
Exemple : Soit l'algorithme A suivant. Calculer la complexité $C_A(n)$?

```
Algorithme A ;
Var T : tableau [1..n] d'entiers ; s : entier ;
Début
s←0 ;
        pour i allant de 1 à n faire
                s←s+T[i] ;
        FinPour
        écrire(s) ;
Fin.
```

$C_A(n)=1+2n+1=2n+2$.

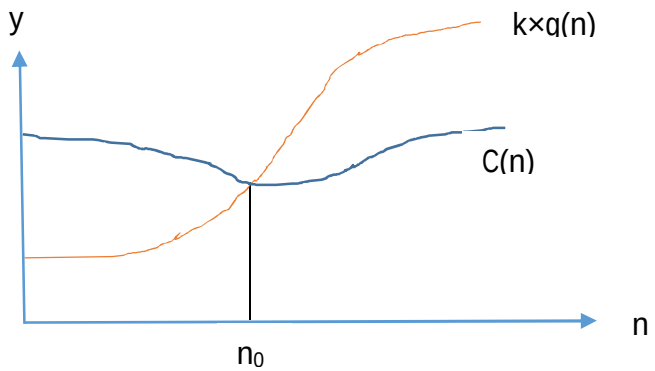
Remarque : en plus de la complexité temporelle, il existe la complexité spatiale (en espace), qui évalue l'espace mémoire occupé par l'exécution d'un algorithme. Dans l'exemple précédent, la complexité spatiale de l'algorithme A est égale à 4×n+4 octets. En pratique, on s'intéresse uniquement à la complexité en temps.

Types de complexité : Soit à écrire un algorithme qui recherche une valeur **val** dans un tableau **T** de taille **n**. Le principe est simple, il suffit de parcourir le tableau du début jusqu'à la fin. La valeur **val** peut être trouvée soit au début, soit au milieu, soit à la fin, soit à une position donnée. Le nombre d'opérations élémentaires effectuées (recherches) dépend donc non seulement de la taille du tableau T, mais également de la répartition de ses valeurs. Ce qui nous amène à distinguer trois types de complexité :

- la **complexité dans le meilleur des cas** : c'est la **situation la plus favorable**. C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme. **val** se trouve à la première position du tableau.
- la **complexité dans le pire des cas** : c'est la **situation la plus défavorable**. C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme. val se trouve à la dernière position du tableau.
- la **complexité en moyenne *(cas moyen)*** : c'est la **situation moyennement favorable**. C'est le nombre moyen d'opérations qu'aura à exécuter l'algorithme. val se trouve au milieu du tableau.

Remarque : En pratique, on calculera **le plus souvent** la complexité dans le pire des cas, car elle est **la plus pertinente**. Dans la plus part des situations, il vaut mieux en envisager le pire.

Calcul de la complexité C(n) en pratique.
En pratique, on ne calcule pas de façon exacte la complexité C(n), car ce calcul peut parfois être long et pénible, et de plus le degré de précision qu'il requière est souvent inutile. On se contente souvent d'une approximation par une analyse asymptotique (que se passe t'il quand n tend vers l'infini ?), c.-à-d. trouver une fonction d'ordre supérieure à C(n), g(n).
Définition : On dit que g(n) est une fonction d'ordre supérieure à C(n) ssi :
$\exists(k,n_0) \in N^2 / C(n) \leq k \times g(n)\ \forall n \geq n_0$, et on écrit **C(n)=O(g(n)).**
On dit que C(n) est **dominée asymptotiquement** par g(n), et signifie que C(n) est borné par k × g(n).
Notation de "grand" O (Landau)
La complexité est exprimée en termes de notation "grand" O (Landau), il explique formellement les performances d'un algorithme. On dit que A est en *O*(g(n)) et on écrit $C_A(n)= O(g(n))$.



Exemple : Soit l'exemple précédent. On a : $C_A(n)=2\times n+1$. g(n)= ?
Il faut chercher une fonction g(n) tel que : $\exists(k,n_0) \in N^{+2} / C(n) \leq k \times g(n)\ \forall n \geq n_0$

On a : $2\times n \leq 2\times n, \forall n$
$1 \leq n, \quad \forall n \geq 1$
alors $C_A(n) = 2\times n+1 \leq 3\times n, \forall n>=1$ ($g(n)=n$, $k=3$, $n_0=1$).
Donc **$C_A(n)=O(n)$,** on dit que l'algorithme A est en O(n).

Temps d'exécution d'un algorithme : il dépend du nombre d'opérations élémentaires exécutées en fonction de la taille des données et de la machine utilisée.
Exemple : Soit à calculer les temps d'exécution t_1, t_2 correspondant aux complexités O(n) et $O(n^2)$ sur des données de taille 10^6, avec un processeur ayant une puissance d'exécution 10^9 Flops.

$t_1=10^6/10^9=10^{-3}$s ; $t_2=10^{12}/10^9=10^3$s=3h.
On peut conclure que la complexité O(n) est bonne, par contre la complexité $O(n^2)$ est mauvaise.

Règles de calcul de la complexité d'un algorithme
a- Complexité d'une instruction élémentaire : (W/R, affectation, appel de sous-programme, etc.).
$C_{instr\_élément}(n) = 1 \leq 1\times n^0 = O(1)$
Exemple :
i←1 ;
$C(n) = 1 = 1\times n^0 = O(1)$.

b- Complexité d'un traitement conditionnel (pire cas)
si *condition* alors
  Traitement 1
sinon
  Traitement 2
finsi
$C_{trait\_cond}(n) = C_{condition}(n) + \max(C_{Traitement\ 1}(n), C_{Traitement\ 2}(n))$
Exemple :
si *i<10* alors
  s←s+i
sinon
  écrire(s) ;
finsi
C(n) = 1 + max(2, 1) = 3. On a : $C(n) = 3 \leq 3 \times 1$, alors C(n)=O(1)

c- Complexité de la Boucle Pour :

Pour i allant de indDeb à indFin faire
    Traitement
FinPour
$C_{pour}(n) = (indFin-indDeb) \times C_{Traitement}(n)$

Exemple
Pour i allant de 1 à n faire
    écrire(i, i+s)
FinPour
$C(n) = (n-1+1) \times (1+1+1)=3\times n \leq 3\times n=O(n)$ ($k=3$, $n_0=0$, $g(n)=n$)

d- Complexité de la Boucle Tant que :

Tant que i ≤ n faire
    Traitement
FinTq
$C_{Tant\ que}(n) = (n-i+2) \times C_{condition}(n) + (n-i+1) \times C_{Traitement}(n)$
Exemple
i←0
Tant que i ≤ n faire
    écrire('i', i) ;
    i←i+1
FinTq
$C(n) = (n-0+2)+1 \times 1 +(n-0+1)\times 4 = n+3+4n+4=5n-5 = 6n+7 \leq 13\times n = O(n)$ ($n_0=1, k=13, g(n)=n$)
Complexité des algorithmes récursifs
Pour calculer la complexité d'un algorithme récursif, il faut compter le nombre d'appels générés (exécutions). Ce calcul est souvent difficile et problématique.

Faire des appels récursifs en cascade peut amener à en générer un très grand nombre, par conséquent un algorithme récursive consomme beaucoup plus de mémoire que son équivalent itératif (recopie des paramètres et des variables locales à chaque appel dans la ple).

Exemple : Factoriel
fonction fact(n :entier) :entier ;
si (n=0) alors
    fact←1
sinon
    fact←n×fact(n-1) ;
finsi
La fonction fact est appelée n+1 fois.
$C(n)=(n+1)\times 1+1+n\times(4)=5n+2 \leq 7n, = O(n)$ ($n_0=1$, $k=7$, $g(n)=n$).

Classes de complexité algorithmique Il existe plusieurs classes de complexité prédéfinies, les plus importantes sont (par ordre croissant en termes de O) :
• **O(1) Complexité constante** : temps d'exécution indépendant de la taille des données n.

• **O(log(n)) Complexité logarithmique** : augmentation très faible du temps d'exécution quand le paramètre n croit. Couper répétitivement un ensemble de données en deux parties égales ;

• **O(n) Complexité linéaire** : augmentation linéaire du temps d'exécution quand le paramètre n croit. Parcourir un ensemble de données ;

O(nlog(n)) Complexité quasi-linéaire : augmentation un peu supérieure à O(n). Couper répétitivement un ensemble de données en deux et combiner les solutions partielles pour calculer la solution générale ;

• **O(n^2) Complexité quadratique** : quand le paramètre n double, le temps d'exécution est multiplié par quatre. Parcourir un ensemble de données en utilisant deux

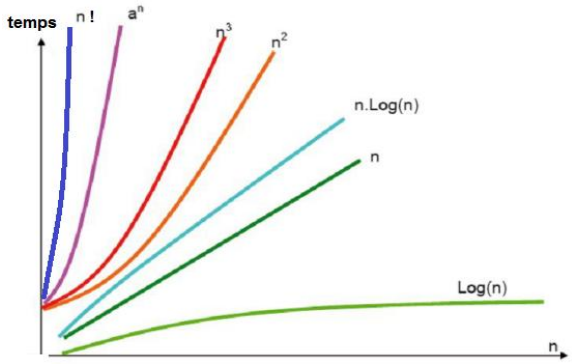
boucles imbriquées ;

• **O(n^3) Complexité cubique** : quand le paramètre n double, le temps d'exécution est multiplié par huit ;

• **O(n^p) Complexité polynomiale** : quand le paramètre n double, le temps d'exécution est multiplié par 2^p. Parcourir un ensemble de données en utilisant P boucles imbriquées ;

• **O(2^n) Complexité exponentielle** : quand le paramètre n double, le temps d'exécution est élevé à la puissance 2. Générer tous les sous-ensembles possibles d'un ensemble de données ;

• **O(n!) Complexité factorielle** : asymptotiquement équivalente à n^n.



Conclusion : Lorsqu'on développe un algorithme qui manipule une grande quantité de données (plus d'un million), il faut bien prendre en considération sa complexité (nombre instructions).

Exercice : Ecrire une fonction recherche qui (teste si une valeur val existe dans) permet de vérifier si une valeur val est dans un tableau T, et calculer sa complexité dans le meilleur des cas et dans le pire des cas.

```
Fonction recherche(T :tableau ; n, val :entier) :booléen ;
var trouve : booléen ; i : entier ;
début
        trouve←faux ; i←1 ;
        Tq non trouve et i≤n faire
            si T[i]=val alors
                trouve←vraie ;
            sinon
                i←i+1 ;
            finsi
        FinTq
        si trouve alors
            recherche←vraie ;
        sinon
            recherche←faux ;
        finsi
fin ;
```

UNIVERSITE ABDERRAHMANE MIRA DE BEJAIA
Faculté des Sciences Exactes
Département d'Informatique

Module : ASD
Niveau : 2e Année Licence
Année : 2019/2020

TD N° 1 : La complexité algorithmique

Exercice N° 1 : calculer la complexité de l'algorithme suivant :

```
Algorithme complexité1 ;
Type Tab = Tableau [1..100] de réel ;
Var A : Tab ;
        n, j : entier ;
        p, x :réel ;
Fonction puissance (x : réel ; p : entier) : réel ;
Var i : entier ;
        puis : réel ;
Début
        i←1 ;
        puis←1 ;
        Tq (i≤p) faire
                puis←puis*x ;
                i←i+1 ;
        FTq
        puissance←puis ;
Fin ;
Fonction polynome (T: Tab; x: réel ; n: entier): réel ;
Var i : entier ;
        poly : réel ;
Début
        poly←T[0] ;
        i←1 ;
        Tq (i≤n) faire
                poly←poly+T[i]*puissance(x,i) ;
                i←i+1 ;
        FTq
        polynome←poly ;
Fin ;
Début
        Lire(n) ;
        pour i allant de 0 à n faire
                lire(A[i]) ;
        FinPour ;
        Lire(x) ;
        p←plynome(A,x,n) ;
        Ecrire(p);
Fin.
```

Exercice N° 2 : soit l'algorithme suivant ci-dessous, avec A est un tableau trié par ordre croissant.

```
Algorithme complexité2 ;
Type Tab = Tableau [1..100] d'entiers;
Var A : Tab ;
        n, i, x : entier ;

Fonction dic (d, f, val : entier ; T : Tab) : booléen ;
Var m : entier ;
Début
        m←(d+f) div 2 ;
        si (val = T[m]) alors
                dic←vrai ;
        sinon si (val<T[m]) et (d<m)alors
                dic←dic(d, m-1, val, T)
        sinon  si (val>T[m]) et (f>m)alors
                dic←dic(m+1, f, val, T)
        sinon
                dic←faux ;
        finSi ;
Fin ;
Début
        Lire(n) ;
        pour i allant de 1 à n faire
                lire(A[i]) ;
        FinPour ;
        Lire(x) ;
        Ecrire(dic(1, n, x, A)) ;
Fin.
```

Q1 : que fait cet algorithme ?
Q2 : calculer la complexité de la fonction *dic* dans le meilleur des cas et dans le pire des cas.