Master MCSC 1

Complexité

Réalisé par : SBAYTRI Youssef

I. Introduction à la compléxité

Ce que l'on entend par complexité des algorithmes est une évaluation du coût d'exécution d'un algorithme en termes de temps (**complexité en temps**) ou d'espace mémoire (**complexité en espace**). Ce qui suit traite de la complexité temporelle, mais les mêmes notions permettent de traiter de la complexité spatiale. Ce coût d'exécution dépend de la machine sur lequel s'exécute l'algorithme, de la traduction de l'algorithme en langage exécutable par la machine. Mais nous ferons ici abstraction de ces deux facteurs, pour nous concentrer 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 verrons également que nous sommes plus intéressés par un comportement asymptotique que par un calcul exact pour n fixé. Enfin, le temps d'exécution dépend de la nature des données, par exemple un algorithme de recherche d'une valeur dans un tableau peut s'arrêter dès qu'il a trouvé une occurrence de cette valeur. Si la valeur se trouve toujours au début du tableau, le temps d'exécution est plus faible que si elle se trouve toujours à la fin. Nous nous intéresserons d'abord dans ce qui suit à la complexité en "pire des cas", qui est une manière, pessimiste, d'ignorer cette dépendance (on évaluera le temps d'exécution, dans le cas évoqué ci-dessus, en supposant qu'il faut parcourir tout le tableau pour trouver la valeur cherchée).

1- Algorithme

Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein2, ont donné cette définition : "Procédure de calcul bien définie qui prend en entrée une valeur, ou un ensemble de valeurs, et qui donne en sortie une valeur, ou un ensemble de valeurs. Un algorithme est donc une séquence d'étapes de calcul qui transforment l'entrée en sortie."

Exemple.

Pour rechercher un nombre dans un tableau trié, le principe de dichotomie consiste à couper ce tableau en deux, à chercher dans une partie et si ce n'est dans celle-ci, alors nous cherchons dans l'autre et ainsi de suite jusqu'à la fin. Nous nous arrêtons quand la valeur a été trouvée ou bien lorsque nous sommes arrivés à la fin du tableau c.à.d. quand la valeur recherchée n'est pas dans le tableau.

2- Complexité d'un algorithme

La complexité d'un algorithme est le nombre d'opérations élémentaires qu'il doit effectuer pour mener à bien un calcul en fonction de la taille des données d'entrée. Pour Stockmeyer et Chandra, "l'efficacité d'un algorithme est mesurée par l'augmentation du temps de calcul en fonction du nombre des données." Nous avons donc deux éléments à prendre en compte :

- La taille des données ;
- Le temps de calcul.

3- La taille des données

La taille des données (ou des entrées) va dépendre du codage de ces entrées.

On choisit comme taille la ou les dimensions les plus significatives. Par exemple, en fonction du problème, les entrées et leur taille peuvent être :

- ✓ Des éléments : le nombre d'éléments ;
- ✓ Des nombres : nombre de bits nécessaires à la représentation de ceux-là ;
- ✓ Des polynômes : le degré, le nombre de coefficients non nuls ;
- ✓ Des matrices m × n : max(m,n), m.n, m + n ;
- ✓ Des graphes : nombre de sommets, nombre d'arcs, produit des deux ;
- ✓ Des listes, tableaux, fichiers : nombre de cases, d'éléments ;
- ✓ Des mots : leur longueur.

I- La complexité en temps

1- Le temps de calcul

Le temps de calcul d'un programme dépend de plusieurs éléments :

- ✓ la quantité de données et leur encodage ;
- ✓ la qualité du code engendré par le compilateur ;
- ✓ la nature et la rapidité des instructions du langage ;
- ✓ la qualité de la programmation ;
- ✓ l'efficacité de l'algorithme.

Nous ne voulons pas mesurer le temps de calcul par rapport à toutes ces variables. Mais nous cherchons à calculer la complexité des algorithmes qui ne dépendra ni de l'ordinateur, ni du langage utilisé, ni du programmeur, ni de l'implémentation. Pour cela, nous allons nous mettre dans le cas où nous utilisons un ordinateur RAM.

- ✓ ordinateur idéalisé ;
- ✓ mémoire infinie ;
- ✓ accès à la mémoire en temps constant ;
- ✓ généralement à processeur unique.

Pour connaître le temps de calcul, nous choisissons une opération fondamentale et nous calculons le nombre d'opérations fondamentales exécutées par l'algorithme.

2- Opération fondamentale

C'est la nature du problème qui fait que certaines opérations deviennent plus fondamentales que d'autres dans un algorithme.

Par exemple :

Problème	Opération fondamentale
Recherche d'un élément dans une liste	Comparaison
Tri d'une liste, d'un fichier, …	Comparaisons, déplacements
Multiplication des matrices réelles	Multiplications et additions
Addition des entiers binaires	Opération binaire

Tableau 1 : opérations fondamentales en fonction des problèmes

3- Coût des opérations

3.1– Coût de base

Pour la complexité en temps, il existe plusieurs possibilités :

- ✓ première solution : calculer (en fonction de n) le nombre d'opérations élémentaires (addition, comparaison, affectation, …) requises par l'exécution puis le multiplier par le temps moyen de chacune d'elle ;
- ✓ pour un algorithme avec essentiellement des calculs numériques, compter les opérations coûteuses (multiplications, racine, exponentielle, …) ;
- ✓ sinon compter le nombre d'appels à l'opération la plus fréquente (cf. tableau 1).

3.2- Coût en séquentiel

Séquence : $T_{séquence}(n)=\sum T_{éléments\ de\ la\ séquence}(n)$

Alternative : *si C alors J sinon K* $T(n)=T_C(n)+max\{T_J(n),T_K(n)\}$

Itération bornée : *pour i de j à k faire B* $T(n)=(k-j+1).(T_{entête}(n)+T_B(n))+T_{entête}(n)$. Dans l'en-tête est mis l'affectation de l'indice et le test de continuation.

Itération non bornée : *tant que C faire B* $T(n)=Nb_{boucles}.(T_B(n)+T_C(n))+T_C(n)$ avec $Nb_{boucles}$ le nombre de boucles qui s'évalue par méthode inductive.

Répéter B jusqu'à C $T(n)=Nb_{boucles}.(T_B(n)+T_C(n))$

Exemple 2

Fonction de multiplication de deux matrices[4]

```
MULTIPLICATIONMATRICES(A,B)
entrée : deux matrices A, B  n×n
sortie : matrice C  n×n
1  n ← ligne[A]
2  Soit C une matrice  n×n
3  pour i ← 1 à  n
4      faire pour j ← 1 à  n
5          faire  c_ij ← 0
6              pour k ← 1 à  n
7                  faire  c_ij ←  c_ij + a_ik . b_kj
8              fin pour
9      fin pour
10 fin pour
11 retourner C
```

3.3 – Coût en récursif

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

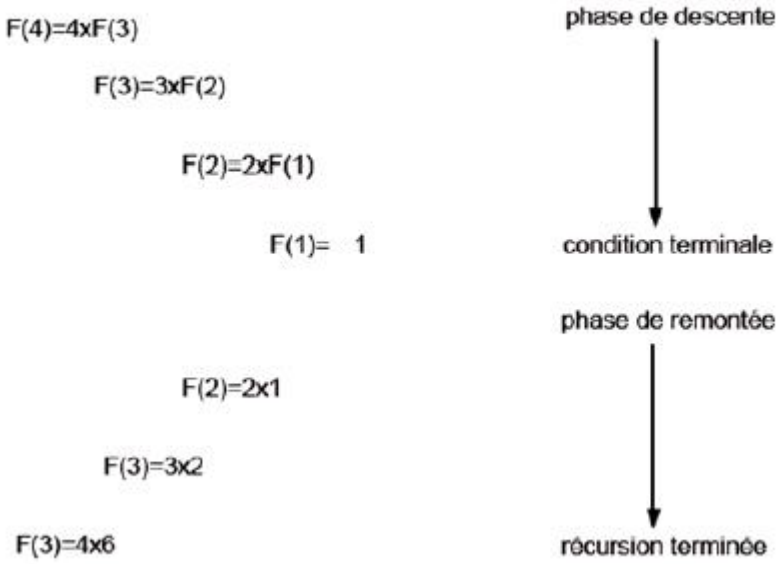
- ✓ La récursion est un principe puissant permettant de définir une entité à l'aide d'une partie de celle-ci.
- ✓ Chaque appel successif travaille sur un ensemble d'entrées toujours plus affinée, en se rapprochant de plus en plus de la solution d'un problème.
- ✓ L'exécution d'un appel récursif passe par deux phases, la phase de descente et la phase de la remontée :
  - Dans la phase de descente, chaque appel récursif fait à son tour un appel récursif. Cette phase se termine lorsque l'un des appels atteint une condition terminale. Condition pour laquelle la fonction doit retourner une valeur au lieu de faire un autre appel récursif.
  - Ensuite, on commence la phase de la remontée. Cette phase se poursuit jusqu'à ce que l'appel initial soit terminé, ce qui termine le processus récursif.

a- La récursivité simple

La fonction contient un seul appel récursif dans son corps. Exemple de la fonction factorielle.

```
FACTORIEL (n)
entrée : un entier  n
sortie :  n!
1 Si  n⩽1
2       alors retour ← 1
3       sinon retour ←  n ×FACTORIEL( n−1 )
4 fin si
5 retourne retour
```

Trace d'exécution de la fonction factorielle (calcul de la valeur de 4!)



b- La récursivité multiple.

Dans le cas de la récursivité multiple, la fonction contient plus d'un appel récursif dans son corps ;
Exemple : le calcul du nombre de combinaisons en se servant de la relation de Pascal :

$$C_n^p = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n; \\ C_{n-1}^p + C_{n-1}^{p-1} & \text{sinon.} \end{cases}$$

```
Fonction Combinaisons( n : entier, p : entier) : entier
    Si (p = 0 OU p = n) Alors
        Retourner 1 ;
    Fin Si
    Retourner Combinaisons (n − 1, p) + Combinaisons (n − 1, p − 1) ;
Fin
```

4- Les différentes mesures de complexité

Il existe trois mesures différentes, la complexité dans le meilleur des cas, la complexité en moyenne et la complexité dans le pire des cas.

Soit A un algorithme, n un entier, D_n l'ensemble des entrées de taille n et une entrée d ∈ D_n. Posons : $coût_A(d)$ le nombre d'opérations fondamentales effectuées par A avec l'entrée d .

4-1- La complexité dans le meilleur des cas

Elle est obtenue par :

$$Min_A(n) = min\{coût_A(d) / d \in D_n\}$$

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.

4-2- La complexité en moyenne

C'est le temps d'exécution moyen ou attendu d'un algorithme. La difficulté pour ce cas est de définir une entrée "moyenne" pour un problème particulier.

$Moy_A(n) = \sum_{d \in D_n} p(d).coût_A(d)$ avec $p(d)$ une loi de probabilité sur les entrées.

Il reste à définir $p(d)$. Une hypothèse est que toutes les entrées ayant une taille donnée sont équiprobables.

D'où : $Moyenne\ uniforme_A(n) = \frac{1}{card(D_n)} \sum_{d \in D_n} coût_A(d)$

D'autres hypothèses se basent sur l'analyse probabiliste[6].

4-3- La complexité dans le pire des cas

Elle est donnée par :

$$Max_A(n) = max\{coût_A(d) / d \in D_n\} .$$

C'est cette complexité qui est généralement calculée car c'est une borne supérieure du temps d'exécution associe à une entrée quelconque.

5- Des exemples de Calcul de complexité

5-1- Algorithme à complexité constante

Commençons par un exemple simple, soit celui d'une fonction prenant en paramètre le rayon d'une sphère, calculant le volume de cette sphère, et retournant cette valeur au sous-programme appelant. On aura le code proposé à ici :

```
public static double volume_sphere(double rayon) {
        final double PI = 3.14159; // (ae)
        double  volume = 4.0 / 3.0 * PI * rayon * rayon * rayon; // (oe)
        return volume;     }
```

L'instruction notée (ae) est l'affectation d'une valeur à une constante. On peut compter celle-ci comme étant une opération (certains l'omettront, ce qui importe peu, comme nous le verrons plus bas).

L'instruction notée (oe) est composée de cinq opérations arithmétiques et d'une affectation. On peut la compter comme six opérations ou comme une seule.

Si on additionne tout ça, on arrive à deux opérations, si on ne compte pas l'affectation d'une valeur à la constante – opération (ae) – et si on compte l'instruction (oe) comme une seule opération, et à huit opérations si on compte les instructions de manière plus rigide. L'algorithme sera donc T(n) = 8*cte ou T(n)=2*cte , tout dépendant de la manière de compter les opérations.

L'important ici n'est pas la valeur exacte de nombre d'instruction **T(n)**, mais le fait que cette valeur soit constante.

Lorsque le nombre d'instructions est dépend d'une constante, on dit que la complexité d'algorithme est constante et on la note **: O(1).**

Une complexité constante est la complexité algorithmique idéale, puisque peu importe la taille de l'échantillon à traiter, l'algorithme prendra toujours un nombre fixé à l'avance d'opérations pour réaliser sa tâche.

5-2- Algorithme à complexité linéaire

Par complexité linéaire, ou O(n), on dénotera des algorithmes pour lesquels le nombre d'instructions à effectuer variera en proportion directe de la taille de l'échantillon à traiter : si l'échantillon croît par un facteur de 100, la complexité sera accrue elle aussi par un facteur de 100.

Pensez à un algorithme qui parcourt chaque élément d'une liste chaînée, ou qui fait la somme des éléments d'un tableau.

```
public static int somme_elements(int tab[]) {
  int somme = 0;  //ae
  for(int compteur = 0; compteur < N; ++compteur) // ae + ce + oe
```

```
            somme += tab[compteur]; // oe
        return somme;
    }
```

Soit :

n = la taille du tableau,

ae = affectation des éléments,

ce = comparaison des éléments

oe = opération sur les éléments.

Donc

T(n) = ae + ae + n(ce + oe + oe)

= 2ae + n(ce + 2oe)

= cte1 + n*cte2

On voit bien ici que la complexité de cet algorithme ne dépend que de la taille du tableau, car les autres paramètres sont constants, on dit dans ce cas-là que la complexité est linéaire et on la note : **O(n).**

Exemple de recherche séquentielle

```
static int rechercheSequentielle( int[] tab, int Elt ) {
        int i = 0;
        while(i<tab.length-1){
            if(tab[i] == Elt) {
                return i;
            }
            i = i + 1;
        }
        return -1;
}
```

Complexité au pire (`Elt` n'est pas dans le tableau) : **ae+n*(2*ce+oe) = O(n)**

Complexité au mieux (`Elt` est dans la première case du tableau) : **ae+2*ce = O(1)**

Complexité en moyenne : considérons qu'on a 50% de chance que `Elt` soit dans le tableau, et 50% qu'il n'y soit pas, et, s'il y est sa position moyenne est au milieu. Le temps d'exécution est **[(ae+n*(2*ce+oe+ae)) + (ae+ (n/2)*(2*ce+oe+ae))] /2**, de la forme **a*n+b = O(n)**

En générale on tombe dans ce cas de complexité lorsque l'algorithme est contient une seul boucle.

5-3- Algorithme à complexité logarithmique

Les algorithmes de ce genre auront la propriété suivante : on leur donne un échantillon de taille n à traiter, et ils font en sorte (dans une répétitive) de diminuer (de moitié, par exemple) à chaque itération [4] la partie de l'échantillon qu'il vaut la peine de traiter. On peut penser, à tout hasard, à une recherche dichotomique.

Exemple de recherche dichotomique

Si t est un tableau d'entiers triés de taille n, on peut écrire une fonction qui cherche si un entier donné se trouve dans le tableau. Comme le tableau est trié, on peut procéder par dichotomie : cherchant à savoir si x est dans t[g..d], on calcule m = (g+d)/2 et on compare x à t[m]. Si x=t[m], on a gagné, sinon on réessaie avec t[g..m] si t[m] > x et dans `t[m+1..d]`.
Voici la fonction correspondante:

```
static int rechercheDichotomique( int[] tab, int Elt ) {
         int n = tab.length-1; //ae
         int g =  1, d =  n, m ;  //2ae
         int  Rang =  -1;  // ae
         do{
            m =  (g + d) / 2;  // oe
            if ( Elt == tab[m])  Rang =  m ; //ce
            else if  ( tab[m] < Elt )  g =  m + 1 ; //ce
            else d =  m-1 ; //oe
         }
         while ( ( Elt != tab[m] ) & ( g <= d ) );  //2ce
         return Rang;
}
```

Supposant tab[] ={3, 4, 6, 8, 17, 22, 199, 201, 202}

- Dans **le meilleur cas** (si **Elt** se trouve à l'élément 0 de **tab[]),** il nous faut seulement (4ae+oe + ce) opérations pour produire la solution, où :

  - ✓ 4ae :les quatre affectation au début.
  - ✓ oe : `m =  (g + d) / 2;`
  - ✓ ce: `if ( Elt == tab[m])  Rang =  m ;`

Donc la complexité en temps de l'algorithme de la recherche dichotomique dans le meilleur des cas est constante. O(1).

Dans **le pire cas** (si **Elt** se trouve à l'élément **N** de **tab[])**. Donc on doit continuer de faire des divisions par deux jusqu'à obtenir un tableau de taille 1. Cette méthode ne s'applique que sur un tableau trié. Supposons que le tableau est de taille n une puissance 2 :

$$(n = 2^q).$$

Avec **q** le nombre d'itérations nécessaires pour aboutir à un tableau de taille 1, donc la complexité de cet algorithme (nombre d'instructions) est dépend de nombre de division qu'on doit réaliser.

$$\begin{aligned} itration1 &\leftrightarrow \frac{n}{2} = \frac{n}{2^1} \\ itration2 &\leftrightarrow \frac{n}{4} = \frac{n}{2^2} \\ itration3 &\leftrightarrow \frac{n}{8} = \frac{n}{2^3} \\ itration4 &\leftrightarrow \frac{n}{16} = \frac{n}{2^4} \\ \ldots &\leftrightarrow \ldots \\ itrationq &\leftrightarrow \frac{n}{2^q} \end{aligned}$$

dernière itération ➔ taille tableau = 1

$$\begin{aligned} \frac{n}{2^q} &= 1 \\ 2^q &= n \\ q &= log_2(n) \end{aligned}$$

➔ La complexité = $\log_2(n)$

Donc : **`T(n) = cte1 + cte2*q = cte1 + cte2*log(n)` ➔ `O(log(n))`**

II. La complexité en espace

1- Complexité en espace

De la même façon qu'on définit la complexité temporelle d'un algorithme pour évaluer ses performances en temps de calcul, on peut définir sa complexité spatiale pour évaluer sa consommation en espace mémoire. Le principe est le même sauf qu'ici on cherche à évaluer l'ordre de grandeur du volume en mémoire utilisé : il ne s'agit pas d'évaluer précisément combien d'octets sont consommés par un algorithme mais de préciser son taux de croissance en fonction de la taille n de l'entrée. Cependant, on notera que la complexité spatiale est bien moins que la complexité temporelle, on dispose aujourd'hui le plus souvent d'une quantité pléthorique de mémoire vive, ce qui rend moins important la détermination de la complexité spatiale.

La mémoire est utilisée pour stocker l'ensemble des données plus ou moins structurées (nombres, vecteurs, matrices, listes...), donc pour calculer la complexité en espace on calcule l'ordre de grandeur du volume en mémoire utiliser pour stocker l'un de ces ensembles de donnés.

Le tableau ci-dessous donne la taille de stockage de chaque type en bytes de données de base C dans un processeur 16 bits

Type de variable	**Longeur en bytes (Java)**	**Longeur en bytes (C)**	**Longeur en bytes (Python)**
Byte	1 byte	**----**	1 bytes
char	2 bytes	1 bytes	1 bytes
short	2 bytes	2 bytes	2 bytes
Int	4 bytes	2 bytes	4 bytes
Float	4 bytes	4 bytes	4 bytes
Double	8 bytes	8 bytes	8 bytes
long	4 bytes	4 bytes	4 bytes
Long double	----	10 bytes	12 bytes

1bytes = 1octet = 8bits = 1/0

Example 1

```
Somme(a,b){
        Int a ,b ;  // 4octets + 4 octets
        Enter des valeurs a et b
        Return (a+b );  //4octets
}
```

S(n) = 4*3=12 octets espace mémoire pour un code comme ça.

➔ O(S(n)) = O(1) // complexité constant

```
Sum(A[1..n]){  //4n
        Int Somme=0 ; //4 octets réserver à  Somme
        For(i=1 ;i<=n ;i++){  //  4 octets réserver à  i ; 4 octets réserver à n
        Somme= Somme  +A[i] ;
        }
        Return Somme ;
}
```

S(n) = 4n + 3*4 ➔ O(S(n)) = O(1) + O(n) = O(n) // complexité linaires