Université de Bejaia.

Faculté des Sciences Exactes.

Département Informatique

Cours
Programmation Avancée

Master 1 en Informatique

Options : RN (RS & SIA)

Prof. BOUALLOUCHE Louiza

Chap.2 Complexité et Optimalité

La complexité constitue un moyen de contrôler un algorithme. L'analyse de la complexité est l'évaluation du coût généré par son exécution permettant de mesurer sa performance. Il s'agit exactement de

- quantifier le temps total qu'aura consommé un algorithme ou un programme pour son exécution. On parle alors de la complexité temporelle.
- quantifier l'espace mémoire utilisé pour l'exécution de l'algorithme ou le programme. On parle alors de la complexité spatiale.

Une complexité élevée peut conduire à une latence inadmissible du temps d'exécution, pouvant même compromettre la validité d'un algorithme, ou une utilisation excessive de l'espace mémoire entrainant un débordement de mémoire pouvant geler l'ordinateur, voire le planter.

L’analyse de la complexité est l’évaluation du coût généré par son exécution.

Dans ce cours, on s’intéressera principalement à la complexité temporelle. On prendra comme modèle de machine sur lequel les programmes seront implémentés, un modèle de machine à accès aléatoire (RAM) monoprocesseur, où les instructions sont exécutées l’une après l’autre, sans opérations parallèles.

Le temps d’exécution d’un programme dépend non seulement de la taille des données mais aussi de leur nature. En effet, des entrées différentes peuvent donner des temps d’exécutions différents.

Définition 1. La complexité temporelle d'un algorithme est le temps d'exécution de l'algorithme en fonction de la taille *n* des données (jeu de données).

Principe. Il s'agit en général de relever le nombre d'opérations (affectations, opérations arithmétiques ou logiques, comparaisons, …) effectuées par l'algorithme; puis d'exprimer la complexité en nombre d'unités de temps, soit le temps d'exécution d'une opération élémentaire (affectation, addition, soustraction, …). En effet, on effectue un appel au calcul de la complexité pour comparer des algorithmes, et les règles de calcul doivent être les mêmes et donc indépendantes

- du langage de programmation utilisé ;
- du processeur de l'ordinateur sur lequel sera exécuté le code ;
- de l'éventuel compilateur employé.

Choix de l'unité de temps. Afin de calculer la complexité de manière précise, il est judicieux de choisir l'unité la plus élémentaire. On pourrait par exemple considérer l'addition de 2 chiffres, comme opération élémentaire. Ainsi, l'addition de deux nombres de *n* chiffres nous fera effectuer *n* additions à 1 chiffre, la complexité en nombre d'opérations sera donc de *n (*la complexité en temps d'exécution sera *T = n*tu, tu est le temps d'exécution de l'opération élémentaire)* . Aussi, la multiplication de ces deux mêmes nombres aura une complexité de l'ordre de n^2 (n^2**tu, en temps d'exécution)*

Exemples. Soient A et B et C, 3 chiffres. On admet que l'affectation, l'addition et la soustraction ont le même coût.

Le nombre d'opérations élémentaires de l'instruction X ← A + B ; est 2 (1 affectation et 1 addition).

Sa complexité temporelle est alors *T= 2*tu* où *tu* est le coût ou le temps d'exécution moyen de l'unité choisie.

Remarque. Cependant il n'est pas toujours aisé d'évaluer la complexité exacte, on s'intéresse alors à la complexité moyenne (sur toutes les exécutions du programme sur des données de taille *n*), au pire cas (le plus défavorable), ou au meilleur cas ou le plus favorable.

Définition 2. Complexité au meilleur cas : c'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données.

Définition 3. Complexité au pire cas : c'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données.

Définition 4. Complexité en moyenne : c'est la moyenne des complexités de l'algorithme sur des jeux de données.

Définition 5. Efficacité d'un algorithme
Un algorithme est plus efficace qu'un autre si sa complexité dans le pire cas a un ordre de grandeur inférieur.

Définition 6. Optimalité d'un algorithme
Un algorithme est dit optimal si sa complexité est la complexité minimale parmi tous les algorithmes de sa classe.

Notations de Landau

Lors de l'évaluation de la complexité d'un algorithme, on s'intéresse généralement à son ordre de grandeur et on utilise les notations de Landau suivantes :

O	:	$f = O(g)$	$\Leftrightarrow$	$\exists n_0, \exists c \geq 0, \forall n \geq n_0, f(n) \leq c*g(n)$
Ω	:	$f = \Omega(g)$	$\Leftrightarrow$	$g = O(f)$
o	:	$f = o(g)$	$\Leftrightarrow$	$\forall c \geq 0, \exists n_0, \forall n \geq n_0, f(n) \leq c*g(n)$
Θ	:	$f = \Theta(g)$	$\Leftrightarrow$	$f = O(g)$ et $g = O(f)$

Exemples,

O : $n = O(n)$, $2n = O(3n)$, $5n = O(n)$, $n+2 = O(n)$ (pour s'en convaincre, prendre $n_0 = 2$ et $c = 2$)
$\log(n) = O(n)$, $n = O(n^2)$, $n = O(n^3)$

o : $\log(n) = o(n)$, $n = o(n^2)$, $n = o(n)$ $\log(n) = o(n)$.

Θ $n = \Theta(n)$, $n + \log(n) = \Theta(n + \sqrt{n})$

Classes de complexité

- Les algorithmes constants (de complexité O(1))
- Les algorithmes sub-linéaires (de complexité O (log(n)), O ($n^{1/2}$), …)
- Les algorithmes linéaires (de complexité O(n))
- Les algorithmes quadratiques (de complexité O(n^2))
- Les algorithmes polynomiaux (de complexité O(n^k) pour k > 2)
- Les algorithmes exponentiels (de complexité O(e^n)).

Remarque

Les algorithmes *O(n)* et *O(nlog(n))* sont considérés rapides.
Les algorithmes polynomiaux sont considérés lents.
Les algorithmes exponentiels sont impraticables dès que la taille des données est ≻ à quelques dizaines d'unités.

Exercices d'application

Exercice 1. Fonction qui retourne le produit des éléments d'un tableau A[1..n]

a. La fonction algorithmique

Fonction Prod-Tab(A : Tab ; n : Int) : real

Var P : real;

Debut

$P \leftarrow 1$;

Pour *allant de 1* ***à*** *n* ***faire***

$P \leftarrow P * A[i]$;

Fin pour

Retourner P;

Fin.

a. Calcul de la complexité

Il est difficile d'exprimer le nombre d'opérations en fonction d'une unité de temps du fait que la complexité dépend du jeux de données; c'est pourquoi on associe un coût à chaque instruction.

- Soient Ci le coût ou le temps d'exécution de la $i^{ème}$ instruction, et
- Nbi le nombre d'itérations ou d'exécutions de la $i^{ème}$ instruction

Ex. la 2ème instruction (***Pour*** *i allant de 1* ***à*** *n* ***faire****)* est exécutée $Nb2 = \sum_{i=1}^{n+1} 1$ *= n+1 fois et on lui associe un coût = C2.* Sa complexité est alors *(n+1)*C2*

La complexité de cet algorithme est alors $T(n) = \sum_{i=1}^{m} Ci * Nbi$

m est le nombre d'instructions exécutables.

Fonction Prod-Table(A : Tab ; n : Int) : Real	Coût Nb
Var P: Real;	
Debut	
P ← 1 ;	C1 1
Pour i allant de 1 à n faire	C2 n+1
P ← $P * A[i]$;	C3 n
Fin pour	
Retourner P ;	C4 1
Fin	

La complexité de cet algorithme est alors $T(n) = \sum_{i=1}^{m} \text{Ci} * \text{Nbi}$

$$= C1 * 1 + C2 * (n+1) + C3 * n + C4 * 1$$

$$= (C2 + C3) * n + C1 + C2 + C4 = A * n + B = O(n)$$

La complexité est de type linéaire.

Exercice 2. Fonction qui calcule la somme des éléments d'une matrice carrée

a. La fonction algorithmique

Fonction Som-Matrice(A : Mat ; n : Int) : real

Var S: real;
Debut
S ← 0 ;
Pour *i allant de 1* ***à*** *n* ***faire***
Pour *j allant de 1* ***à*** *n* ***faire***
$S \leftarrow S + A[i, j]$;
Fin pour
Fin pour
Retourner *S ;*
Fin

b. Calcul de la complexité

Pour calculer la complexité, on procèdera de la même manière qu'à l'exemple 1.

$T(n) = \sum_{i=1}^{m} \text{Ci} * \text{Nbi}$

Ex. la 3ème instruction (***Pour*** *j allant de 1* ***à n faire***) *est exécutée*

$Nb3 = n * \sum_{i=1}^{n+1} 1 = \text{n} * (\text{n} + 1)$ *fois*

et on lui associe un coût = C3.

Fonction Som-Matrice(A : Mat ; n, l : Int) : real	Coût Nb
Debut	
Var S : real	
S ← 0 ;	C1 1
Pour i allant de 1 à n faire	C2 n+1
Pour j allant de 1 à n faire	C3 n(n+1)
S ← $S + A[i, \text{j}]$;	C4 n*n
Fin pour	
Fin pour	
Retourner S ;	C5 1
Fin	

La complexité de cet algorithme est alors

$$T(n) = \sum_{i=1}^{m} \text{Ci} * \text{Nbi}$$

$$= C1 * 1 + C2 * (n+1) + C3 * n * (n+1) + C4 \ * n * n + C5*1$$

$$= (C3 + C4) * n * n + (C2 + C4) * n + C1 + C2 + C5 = A * n * n + B * n + C$$

La complexité est quadratique, en effet,

$T(n) = A*n^2 + B*n + C = O(n^2)$

Reprendre l'exercice 2 avec une matrice de taille NxM

Exercice 3 : Fonction qui calcule le nombre d'éléments > 0 dans une matrice *carrée*

1. La fonction algorithmique

Fonction Nbre-Pos-Matrice (A : Mat ; n : Int) : Int

```
Var Np: Int;
Debut
  Np ← 1 ;
  Pour i allant de 1 à n faire
      Pour j allant de 1 à n faire
        Si A[i] > 0 Alors
            Np ← Np + 1 ;
         Fin si
      Fin pour
   Fin pour
   Retourner Np;
Fin
```

2. Calcul de la complexité

Fonction Nbre-Pos-Matrice(A : Mat ; n : Int) : Int Debut Np ← 0 ; Pour i allant de 1 à n faire Pour j allant de 1 à n faire Si $A[i, j] > 0$ ***Alors*** Np ← $Np + 1$; Fin si Fin pour Fin pour Retourner Np; Fin	Coût Nb C1 1 C2 n+1 C3 n*(n+1) C4 n*n C5 np C6 1

L'instruction $Np \leftarrow Np + 1$ est exécutée uniquement si A[i, j] est testé positif. La complexité dépend alors du nombre d'éléments positifs *np*.

La complexité de cet algorithme est $T(n) = \sum_{i=1}^{m} \text{Ci} * \text{Nbi}$

$$= C1 * 1 + C2 * (n + 1) + C3 * n * (n + 1) + C4 * n * n + C5 * np + C6 * 1$$

$$= (C3 + C4) * n^2 + (C2 + C3) * n + C1 + C2 + C6 + C5 * np$$

1. **La complexité au meilleur cas :** cas où l'instruction 5 n'est jamais exécutée, c'est-à-dire que tous les éléments du tableau sont $<= 0$, donc $np = 0$.

$T(n) = (C3+C4)*n^2 + (C2+C3)*n + C1 + C2 + C6 = A*n^2 + B*n + C$

La complexité est quadratique $= O(n^2)$

2. **La complexité au pire cas :** cas où l'instruction 5 est exécutée à chaque itération, c'est-à-dire que tous les éléments du tableau sont > 0, donc $np = n*n$.

$T(n) = (C3+C4+C5)*n^2 + (C2+C3)*n + C1 + C2 + C6 = A*n^2 + B*n + C = O(n^2)$

La complexité dans ce cas est supérieure à celle du cas précédent.

3. **La complexité au moyen cas :** cas où l'instruction 5 est exécutée à n*n/2 fois; c'est-à-dire que $n^2/2$ éléments du tableau sont > 0, donc $np = n^2/2$.

$T(n) = (C3+C4+0.5*C5)*n^2 + (C2+C3)*n + C1 + C2 + C6 = A*n^2 + B*n + C = O(n^2)$

La valeur de T(n) est différente dans chacun des cas, néanmoins son ordre de grandeur est le même, $T(n) = A*n^2 + B*n + C = O(n^2)$, donc de type quadratique.

Remarque. La complexité dépend du jeu de données.

Tri par insertion des éléments d'un tableau A de n éléments.

Principe. De manière répétée on retire un élément du tableau et on l'insère à la bonne place dans la séquence des nombres déjà triés.

Algorithme tri-insertion	Coût	Nb-Ex
Debut		
Pour j allant de 2 à n faire	c1	n
x← $A[j]$;	c2	n-1
i← $j-1$;	c3	n-1
Tant que i > 0 et A[i] > x faire	c4	$\sum_{j=2}^{n} tj$
A[i+1] ← A[i];	c5	$\sum_{j=2}^{n} (tj-1)$
i← i-1;		
Fin tant que	c6	$\sum_{j=2}^{n} (tj-1)$
A[i+1] ← x ;		
Fin pour	c7	n-1
Fin		

b. Calcul de la complexité

Pour chaque valeur de j ∈ [2, n], *tj* est le nombre d'exécutions de la boucle tant que pour cette valeur de *j* (*tj* dépend des données).

Complexité au meilleur cas (tableau trié)

Donc *tj = 1*

*T(n) = C1*n+C2*(n-1) + C3*(n-1) + C4*(n-1) + C7*(n-1)*

*= (C1 + C2 + C3 + C4 + C7)*n - (C2 + C3 + C4 + C7) = A*n + B = O(n).*

Complexité au pire cas (tableau trié dans l'ordre inverse)

Donc *tj = j*

$T(n)=C_1*n+C_2*(n-1)+C_3*(n-1)+C_4*(\frac{n*(n+1)}{2}-1)+C_5*(\frac{n*(n-1)}{2})+C_6*(\frac{n*(n-1)}{2})+C_7*(n-1) = A*n^2 + B*n + C = O(n^2)$

La complexité est de type quadratique.

Complexité au moyen cas *tj = j/2*

T(n)= O (n^2) (quadratique).