Complexité des algorithmes

Plan

- Introduction
- Modèle de calcul de complexité
- Notations
- Classes de complexités
- Exercices d’applications

Introduction

- Un problème possède plusieurs méthodes de résolution
- Pour décrire ces méthodes, on doit écrire de algorithmes
- Un algorithme est donc une description d'une méthode de résolution du problème, qui prend en entrée un valeur ou un ensemble de valeurs et qui produit en sortie une valeur ou un ensemble de valeurs

Introduction

- Pour choisir un parmi plusieurs algorithmes du même problème, on doit les évaluer.
- Evaluer un algorithme revient à mesurer sa complexité on se basant sur deux principaux critères :
  - Le temps d'exécution
  - L'espace mémoire utilisé
- Le critère le plus important est le temps d'exécution qu'on l'appelle aussi parfois « Le coût d'un algorithme »

Complexité : Définition

- La complexité d'un algorithme c'est l'évaluation de nombres d'opérations élémentaires effectuées en fonction de la nature et la taille de données dans le meilleur , moyen et pire de cas;

Complexité : Définition

- Nature de données : variable simple ou composée (ex : tableau)
- Taille de données :
  - variable simple :
    - Taille = l'entier lui-même ou le nombre de chiffres qui les compose
  - Variable composée :
    - Taille = Taille de tableau (nombres de cases)

Complexité : Définition

- **La complexité au pire :** temps d'exécution maximum, dans le cas le plus défavorable.
- **La complexité au mieux :** temps d'exécution minimum, dans le cas le plus favorable (en pratique, cette complexité n'est pas très utile).
- **La complexité moyenne :** temps d'exécution dans un cas médian, ou moyenne des temps d'exécution.

➔Le plus souvent, on utilise la complexité au pire, car on veut borner le temps d'exécution et on s'assure que l'algorithme ne prendra jamais plus de temps

Modèle de calcul de complexité

Coût d'algorithme = ∑ coûts d'opérations élémentaires + ∑ coûts d'opérations complexes

- **Coût des opérations élémentaires:**
  - opération élémentaire: affectation, lecture/écriture, opérations arithmétiques, test de comparaison…, ont un faible temps d'exécution. Elles représentent l'unité de mesure de coût d'u algorithme
  - coût d'exécution =1
  - **Exemple Algorithme A:**

```
Somme ← n + 1
Somme ← somme * n
Somme ← somme/2
```

=> coût(A) = 3 ou 6 si calcul détaillé

Modèle de calcul de complexité

- **Coût d'opérations complexes:**
  - Opération complexe: instruction conditionnelle ou boucle ou appel de module.
    - coût (inst_cond) = coût(test)+max(coût(T1),coût(T2))
    - coût(boucle Pour) = n * ∑coût(inst(i))
    - coût(boucle TantQue) = n*∑coût(inst(i))+coût(comparaison)
    - coût(module) = ∑coûts (opérations élémentaires internes)
  - **Exemple Algorithme B:**

```
TantQue i <= n faire
    Somme ← Somme + i
    i ← i + 1
FinTantQue
```

=> coût (B) = 3n ou 5n si calcul détaillé

Complexité : notations

- Exemple :

```
Répéter
        ecrire (" donner n« )
        Lire (n)
jusqu'à (n>0)
Pour i  de 1 à n faire
        Pour j  de 1 à n faire
                ecrire (i*j)
        fin pour
Fin pour
```

➔Le temps d'exécution proportionnel à $2n^2 + 3n$

On note quoi comme complexité de cet algorithme ?

Complexité : notations

Le temps d'exécution est proportionnel à $n^2 + 3n$

Donc, si n ↗ ➔ $3n <<< n^2$

(3n ↗ beaucoup moins vite que n^2: négligeable)

Pour décrire l'efficacité d'un algorithme, on s'intéresse seulement au terme qui ↗ le plus vite par rapport aux autres terme lorsque la taille du problème ↗ , donc à : n^2

Complexité : notations

On dit que $n^2 + 3n$ est « un grand O de n^2 »

Noté $O(n^2)$

Complexité : notations

- n : Taille de données
- T(n) : nombres d'opérations élémentaires
- $T_i(n)$: coût de $i^{ème}$ itération
- $T(n) = T_1(n) + T_2(n) + \ldots + T_k(n)$
- T(n) s'écrit : O(f(n))
- O(f(n)) : Notation de landau
- O(f(n)) est la complexité de l'algorithme
- O est lue « en grandeur de »

Complexité temporelle vs spatiale

Exemple : échange de deux valeurs entières

```
// (1) échange des valeurs
de deux variables entier x,
y, z;
... // initialisation de x
et y

z <- x;
x <- y;
y <- z;
```

```
// (2) échange des
valeurs de deux variables
entier x, y;
... // initialisation de
x et y

x <- y-x;
y <- y-x;
x <- y+x;
```

- - la première méthode utilise une variable supplémentaire et réalise 3 affectations
- - la deuxième méthode n'utilise que les deux variables dont on veut échanger les valeurs, mais réalise 3 affectations et 3 opérations

Classes de complexité

- **O(1) :** complexité constante, pas d'augmentation du temps d'exécution quand le paramètre croit

Exemple :

```
Procédure p (n : entier)
Var i: entier
Début
    Pour i de 1 à 11 faire
        Écrire (i * n)
    Fin pour
Fin
```

La boucle toujours exécutée 11 fois et ne comporte qu'une multiplication, donc le temps d 'exécution ne dépend pas de l'entrée n
➔ la complexité est O(1)

Classes de complexité

- **O(log(n)) :** complexité logarithmique, augmentation très faible du temps d'exécution quand le paramètre croit.
  - **Exemple :** algorithmes qui décomposent un problème en un ensemble de problèmes plus petits (dichotomie).

Classes de complexité

- **O(n)** : complexité linéaire, augmentation linéaire du temps d'exécution quand le paramètre croit (si le paramètre double, le temps double).
  - **Exemple :** algorithmes qui parcourent séquentiellement des structures linéaires.
  - A et b deux nombres entiers à n chiffres : Faire la somme de a+b ? Revient à lire tous les chiffres de a, lire tous les chiffres de b, faire la somme, Écrire tous les chiffres de résultat ➔ Le temps d'exécution est proportionnel à n

Classes de complexité

Exemple :

```
Procédure p (n : entier)
Var i: entier
Début
    Pour i de 1 à n faire
        Écrire (i * n)
    Fin pour
Fin
```

La boucle exécutée n fois, donc le temps d ’exécution dépend de l’entrée n ➔ la complexité est O(n)

Classes de complexité

- **O(n log(n))** : complexité quasi-linéaire, augmentation un peu supérieure à O(n).
  - **Exemple :** algorithmes qui décomposent un problème en d'autres plus simples, traités indépendamment et qui combinent les solutions partielles pour calculer la solution générale.

Classes de complexité

- **$O(n^2)$:** complexité quadratique, quand le paramètre double, le temps d'exécution est multiplié par **4**.

  –**Exemple :** algorithmes avec deux boucles imbriquées, chacune effectuant n répétitions de son corps.

```
Pour i de 1 à n faire
    Pour j de 1 à n faire
```

Exercice: Evaluer la complexité

```
Algorithme Cal_Pol
    Variable A, x, P : réel
             i, n : entier
Début
  Ecrire(« Donner x : »)
  Lire(x)
  Répéter
    Ecrire(« Donner n : »)
    Lire(n)
  jusqu'à n>1
  Ecrire(« Donner A : »)
  Lire(A)
  P←A
  Pour i de n-1 pas=-1 à 0
faire
      Ecrire(« Donner A »,
i)
      Lire(A)
      P←P*X+A
  FinPour
  Ecrire(« P(», X, « ) =», P)
Fin
```

Exercice: Evaluer la complexité

```
Algorithme Cal_Duree
      variable duree, etape : entier
Début
    Ecrire(« Saisir le n° de l'étape= »)
    Lire(etape)
    Si (etape=1) ou (etape=4) alors
        duree←15
     sinon
        si etape=2 alors
           duree←10
        sinon
           si etape=3 alors
              duree←25
           sinon
              duree←0
           Finsi
        Finsi
   Finsi
   Si duree=0 alors
      Ecrire(« Arrêt du processus »)
   sinon
      Ecrire(« La durée de l'étape n° », etape, « = », duree, « secondes »)
   Finsi
Fin
```

Exercice: Evaluer la complexité

```
1     i ← 0
2     j ← 0
3     Tant que(i < n) faire
4           Si ( i mod 2 = 0) alors
5                 j ← j + 1
6           sinon
7                 j ← j / 2
8           Fin si
9           i ← i + 1
10    Fin tant que
```

le temps d'exécution du corps de la boucle est donc un Θ(1). Comme la boucle s'exécute n fois, le temps d'exécution du programme est alors en Θ(n).

Exercice

```
1       Pour i de 1 à n faire
2               Pour j de 0 à 2 faire
3                       i ← i * j + 1
4               Fin pour
10      Fin pour
```

L'analyse commence comme toujours par la boucle la plus interne la boucle s'exécute exactement trois fois pour j prenant les valeurs 0, 1 et 2. la boucle interne réalise un nombre constant d'opérations Donc le nombre d'opérations réalisées par la boucle complète est un $\Theta(1)$. boucle externe s'exécute pour les valeurs de i allant de 1 à n ; On a donc n exécutions du corps de la boucle, soit un nombre total d'opérations en $n \times \Theta(1)$, c'est-à-dire $\Theta(n)$.