Matière: Algorithmique Avancée et langage C

Complexité temporelle

Notions de base

Enseignant(e)s: **Feyrouz HAMDAOUI & Sofiane Ben Ahmed**

Niveau: **3ème Génie Informatique**

2024/2025 SU2

Plan

- Motivation et objectif
- Variantes
- Définition
- Propriétés
- Classes

Motivation et objectif

La théorie de la complexité algorithmique vise à répondre à ces besoins :

- classer les ***problèmes*** selon leur difficulté ;
- classer les ***algorithmes*** selon leur efficacité ;
- comparer les **algorithmes** résolvant un problème donné afin de faire un choix éclairé sans devoir les implémenter ;

Motivation et objectif

- Dans la méthode expérimentale de la complexité, si on faisait tourner plusieurs algorithmes sur des machines différentes, il fallait évaluer la puissance des machines.
- **Benchmark** (point de référence) : batterie de tests consistant à faire tourner certains programmes sur une machine pour évaluer sa puissance (un benchmark est orienté vers certains types de calculs).
- *Exemples de benchmark :*
  - Sandra : benchmark généraliste sous Windows (www.sisoftware.co.uk)
  - Sciencemark : benchmark orienté calculs scientifiques sous Windows
  - Benchmarks sous Linux : lbs.sourceforge.net

Motivation et objectif

- En méthode théorique, l'efficacité d'un algorithme peut être évalue en temps et en espace :
  - *complexité en temps :* évaluation du temps d'exécution de l'algorithme
  - *complexité en espace :* évaluation de l'espace mémoire occupe par l'exécution de l'algorithme
- Une règle non officielle de l'espace-temps informatique stipule que pour gagner du temps de calcul, on doit utiliser davantage d'espace mémoire. Néanmoins, on s'intéresse essentiellement à la complexité en temps (ce qui n'était pas forcément le cas quand les mémoires coutaient cher).

Plan

- Motivation et objectif
- Variantes
- Définition
- Propriétés
- Classes

Variantes de complexité temporelle

- La complexité temporelle dépend de l'instance considérée, elle n'est pas la même selon les déroulements du traitement:

❖ **La complexité au pire :** temps d'exécution maximum, dans le cas le plus défavorable.

$$T_{pire}(n) = \max_n \{ T(n) \}$$

❖ **La complexité au mieux :** temps d'exécution minimum, dans le cas le plus favorable.

$$T_{meil}(n) = \min_n \{ T(n) \}$$

Variantes de complexité temporelle

- **La complexité moyenne :** temps d'exécution dans un cas médian, ou moyenne des temps d'exécution.

$$T_{moy}(n) = \sum_n p(n) \,.\, T(n)$$

avec p(n) une loi de probabilité sur les entrées. Fréquemment, on utilise la loi de probabilité uniforme.

→ Le plus souvent, on utilise la complexité au pire, car on veut borner le temps d'exécution.

Exemples (extrait du tri par sélection)

```
1.  min ← i
2.  pour j de i+1 à n faire
3.      si A[j] < A[min] alors
4.          min ← j
```

→ 1 affectation
→ 1 aff.+ 1 comp. + (1 aff.+ 1 comp.).$\#_{boucles}$
→ 1 comparaison . $\#_{boucles}$
→ (si test vrai : 1 affectation) . $\#_{boucles}$

- Dans le pire des cas, quand la table est triée par ordre inverse.

4 (n-i) + 3

- Supposons que, sur les (n-i) tests, la (petite) moitié est évaluée à vrai. En moyenne,

4 [(n-i)/2] + 3 [(n-i)/2] + 3

- Dans le meilleur des cas, quand le tableau est trié, on n'exécute jamais l'instruction min ← j.

3 (n-i+1)

Plan

- Motivation et objectif
- Variantes
- Définition
- Propriétés
- Classes

Définition

La complexité d'un algorithme est une mesure de sa performance asymptotique dans le pire cas

- Que signifie asymptotique ?
  - on s'intéresse à des données très grandes ;

  pourquoi ?
  - les petites valeurs ne sont pas assez informatives ;
- Que signifie « dans le pire cas » ?
  - on s'intéresse à la performance de l'algorithme dans les situations où le problème prend le plus de temps à résoudre ;

  pourquoi ?
  - on veut être sûr que l'algorithme ne prendra jamais plus de temps que ce qu'on a estime ;

Plan

- Motivation et objectif
- Variantes
- Définition
- Propriétés
- Classes

Paramètre de la complexité

Le paramètre de la complexité est ce qui va faire varier le temps d'exécution de l'algorithme.

- Pour un algorithme qui opère sur **une structure de données** (tableau, ...), la complexité est généralement exprimée en fonction d'une dimension de la structure
- dans le cas où l'algorithme prend en entrée une structure linéaire, à une seule dimension, il n'y a pas d'ambiguité
- dans le cas où l'algorithme prend en entrée une structure à plusieurs dimensions (tableau multidimensionnel, arbre, graphe, ...), il faut préciser en fonction de quelle dimension on calcule la complexité
- dans le cas où l'algorithme prend en entrée une structure à plusieurs dimensions, l'algorithme peut avoir des complexités différentes selon la dimension considérée

- Pour un algorithme qui opère sur un **nombre**, la complexité est généralement exprimée en fonction de la valeur du nombre

Paramètre de la complexité

Exemple : la calcul de la factorielle

```
fonction avec retour entier factorielle1(entier n)
        entier i, resultat;
début
        resultat <- 1;
        pour (i allant de 2 à n pas 1) faire
            resultat <- resultat*i;
        finpour
        retourne resultat;
fin
```

Le paramètre de complexité est la valeur de n

Paramètre de la complexité

Exemple : multiplier tous les éléments d'un tableau d'entiers par un entier donné

```
fonction sans retour multiplie(entier[] tab, int x)
        entier i;
début
        pour (i allant de 0 à tab.longueur-1 pas de 1) faire
            tab[i] <- tab[i] * x;
        finpour
fin
```

Le paramètre de complexité est la longueur du tableau tab.

Paramètre de la complexité

Exemple : faire la somme des premiers éléments de chaque ligne d'un tableau à deux dimensions

```
fonction avec retour entier sommeTeteLigne(entier[][] tab)
        entier i,s;
début
        s <- 0;
        pour (i allant de 0 à tab[0].longueur-1 pas de 1) faire
            s <- s + tab[0][i];
        finpour
        retourne s;
fin
```

Le seul paramètre de complexité est la longueur de tab[0].

Calcul de la complexité d'un algorithme

- Pour calculer la complexité d'un algorithme :

➢ on calcule la complexité de chaque « partie » de l'algorithme ;

➢ on combine ces complexités conformément aux règles de calcul ;

➢ on simplifie le résultat grâce aux règles de simplifications

Règles de simplifications

- On calcule le temps d'exécution on effectuant les simplifications suivantes :

1. on oublie les constantes multiplicatives (elles valent 1) ;
2. on annule les constantes additives ;
3. on ne retient que les termes dominants ;

Exemple (simplifications)

Soit un algorithme effectuant $g(n) = 4n^3 - 5n^2 + 2n + 3$ opérations ;

1. on remplace les constantes multiplicatives par 1 : $1n^3 - 1n^2 + 1n + 3$
2. on annule les constantes additives : $n^3 - n^2 + n + 0$
3. on garde le terme de plus haut degré : $n^3 + 0$

et on a donc $g(n) = O(n^3)$.

Justification des simplifications

- Les processeurs actuels effectuent plusieurs milliards d'opérations à la seconde ;

1. qu'une affectation requière 2 ou 4 unités de temps ne change donc pas grand-chose ;

   → d'ou le remplacement des constantes par des 1 pour les multiplications

2. un nombre constant d'instructions est donc aussi négligeable par rapport à la croissance de la taille des données ;

   → d'ou l'annulation des constantes additives

3. pour de grandes valeurs de n, le terme de plus haut degré l'emportera ;

   → d'ou l'annulation des termes inferieurs

- **On préfère donc avoir une idée du temps d'exécution de l'algorithme plutôt qu'une expression plus précise mais inutilement compliquée**

Règles de calculs: combinaison des complexités

- Les instructions de base prennent un temps constant, noté O(1) ;
- On additionne les complexités d'opérations en séquence : $O(f_1(n)) + O(f_2(n)) = O(f_1(n) + f_2(n))$
- le temps d'un branchement conditionnel est égal, au pire des cas, au: t.e. du test + le max des deux t.e. correspondant aux deux alternatives ;

$$T(\textbf{si } C \textbf{ alors } I \textbf{ sinon } J) = T_C(n) + \max [T_I(n) , T_J(n)]$$

Règles de calculs: combinaison des complexités

- Dans les boucles, on multiplie la complexité du corps de la boucle par le nombre d'itérations ;

➢ Le temps d'une itération d'une boucle est égal à:

coût d'incrémentation + coût du corps de la boucle + coût du test de sortie de boucle.

Ce temps doit être *multiplié par le nombre d'itérations* pour calculer le temps d'une boucle « pour ».

T(pour i de e1 à e2 faire B) = (e2 – e1 + 1). [Te1(n) + TB(n) + Te2(n)]+T_{init}+$T_{cdArrêt}$

$$T(\textbf{tant que } C \textbf{ faire } B) = Nb_{boucles} \cdot [T_B(n) + T_C(n)] + T_C(n)$$
$$T(\textbf{répéter } B \textbf{ jusqu'à } C) = Nb_{boucles} \cdot [T_B(n) + T_C(n)]$$

Exemple (1)

- Considérons la boucle suivante (la boucle interne du tri par sélection) :

```
1) petit ← i
2) pour j=i+1 à n faire
3)     si (A[j] < A[i]) alors
4)         petit ← j
       fin si
   fin pour
```

➢ ***Temps d'exécution :***

- Analysons le temps d'exécution de cette boucle. Pour cela, nous devons définir une unité de mesure. Nous allons dire que chaque opération élémentaire va couter une unité de temps *ut*.

Exemple (2)

- La ligne 2) - 1*ut* pour incrémenter j
  - 1*ut* pour le test si j a atteint n ou non
  - → ce qui fait 2 *ut*
- La ligne 3) 1*ut* pour le test
- La ligne 4) c'est une affectation qui ne s'exécute que si la condition est vérifiée. Ici, on regardera surtout la complexité dans le pire des cas puisque c'est elle qui nous assure que l'algorithme se terminera toujours avec le temps au plus annoncé. On raisonne au pire et on considère qu'elle est exécutée à chaque itération donc on a 1*ut*.

Exemple (3)

- La boucle est exécutée n – (i+1) + 1 = n – i fois,

  → Ce qui nous donne 4(n – i) ut pour la boucle au pire des cas.

  A cela s'ajoute :

  - 1*ut* pour l'affectation de la ligne 1)

  - 1*ut* pour l'initialisation de j dans la ligne 2) au début de l'exécution de la boucle.

  - 1*ut* pour le test si j est supérieure à n dans la ligne 2) au début de l'exécution de la boucle.

→ Ce qui donne à la fin un temps d'exécution total de 4(n – i) + 3.

Exemple (4)

- ***Taille de l'instance :***

- Remarquons que ce code est exécuté sur une portion du tableau qui contient (n – i +1) entiers, c'est-à-dire une instance de taille n – i + 1.

- ***Analyse de complexité :***

- Donc essayons d'exprimer le temps d'exécution obtenu en fonction de la taille de l'instance. Soit m = n – i + 1, nous avons alors : 4 (n – i) + 3 = 4 (n – i + 1) – 1 = 4m – 1, d'où pour une instance de taille m, le temps d'exécution est donné par la fonction T(m) = 4m – 1.

Plan

- Motivation et objectif
- Variantes
- Définition
- Propriétés
- Classes

Classes

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

- **$O(n^2)$: complexité quadratique,** quand le paramètre double, le temps d'exécution est multiplié par 4. Cette complexité est acceptable uniquement pour des données de petite taille. *Exemple : algorithmes avec deux boucles imbriquées.*
- **$O(n^i)$: complexité polynomiale,** quand le paramètre double, le temps d'exécution est multiplié par 2^i. Un algorithme utilisant i boucles imbriquées est polynomial. *Exemple : algorithme utilisant i boucles imbriquées.*
- **$O(i^n)$: complexité exponentielle,** quand le paramètre double, le temps d'exécution est élevé à la puissance 2.
- **O(n!) : complexité factorielle,** asymptotiquement équivalente à n^n. Un algorithme de complexité factorielle ne sert à rien.

Relations asymptotiques entre les complexités

$$O(1) < O(\log) < O(n) < O(n*\log(n)) < O(n^2) < O(n^3) = O(n^i) = O(2^n) = O(n!)$$

Hiérarchie de classes

- Pour faire un choix éclairé entre plusieurs algorithmes, il faut être capable de situer leur complexité ;
- On fait une première distinction entre les deux classes suivantes :
  - les algorithmes dits polynomiaux, dont la complexité est en $O(n^k)$ pour un certain k ;
  - les algorithmes dits exponentiels, dont la complexité ne peut pas être majorée par une fonction polynomiale.

Bibliographie

- Sylvie Hamel. «Analyse et complexité des algorithmes ». Université de Montréal; IFT2810, A2009
- Frédéric Fürst. « complexité ». Cours «Algorithmique et programmation », Licence Informatique, Université de Picardie. 2015/2016
- Paola Flocchini. « Structures de données et algorithmes».Université d'Ottawa, 2010.
- Anthony Labarre « Complexité algorithmique ». Structures de données et algorithmes fondamentaux. Université Paris-Est Marne la Vallée.
- Sandrine JULIA. « Complexité des algorithmes ». Cours «Informatique Théorique »