Université du Maine

IUT de LAVAL

IUT Info 1 - Module M1103

Algorithmes et structures de données

La complexité algorithmique

aous.karoui@univ-lemans.fr

Aous Karoui (PhD Student)

Summary

Objectifs du cours

IUT de LAVAL

- Les progrès des matériels informatiques et la complexité des problèmes posés demandent une nouvelle analyse. Il faut donc :
  - Fournir les outils mathématiques nécessaires à l'analyse des performances d'un algorithme
  - Améliorer les performances des problèmes faciles
  - Savoir comment aborder les problèmes difficiles
  - Donner un sens à la notion d'efficacité des algorithmes
  - Etudier des structures de données avancées pour mieux exploiter les ressources mémoires des machines

Complexité : Introduction

IUT de LAVAL

- Elle permet de mesurer les performances d'un algorithme et de le comparer avec d'autres algorithmes réalisant les mêmes fonctionnalités.
- Un concept fondamental pour tout informaticien, elle permet de déterminer :
  - si un algorithme A est meilleur qu'un algorithme B
  - s'il est optimal ou
  - s'il ne doit pas être utilisé

Complexité : Introduction

- Deux possibilités pour mesurer la complexité
  - En terme de temps d'exécution
  - En terme d'occupation mémoire
- Objectif
  - Trouver un moyen de comparer des algorithmes : trouver une mesure absolue indépendante du matériel utilisé

Complexité : Introduction

- Le temps d'exécution d'un programme dépend :
  1. du nombre de données en entrée
  2. de l'efficacité de l'algorithme
  3. de la taille du code, qualité de la programmation
  4. du type d'ordinateur utilisé (processeur, mémoire)
  5. La qualité du code généré par le compilateur

Complexité : Introduction

- Estimer la taille des ressources ?
  - Si l'on prend en compte tous les paramètres : fréquence d'horloge, nombre de processeurs, temps d'accès disque. . . l'estimation de ces ressources peut :
    - ★ être assez compliquée, voire irréalisable
    - ★ devenir irréaliste dès que l'on change d'architecture
  - Pour cela on se contente souvent d'estimer l'influence de la taille des données sur la taille des ressources nécessaires

Complexité en temps

- Soit n la taille des données du problème et T(n) le temps d'exécution de l'algorithme. On distingue :
  - Le temps du pire cas $T_{max}(n)$ qui correspond au temps maximum pris par l'algorithme pour un problème de taille n
  - Le temps moyen $T_{moy}(n)$: temps moyen d'exécution sur des données de taille n (suppositions sur la distribution des données)
  - Le temps du meilleur cas $T_{min}(n)$ qui correspond au temps minimum pris par l'algorithme pour un problème de taille n
- Pour cela on se contente souvent d'estimer l'influence de la taille des données sur la taille des ressources nécessaires

Exemple: recherche d'un élément dans un tableau

Complexité en temps

- Règles générales :
  1. Le temps d'exécution d'une affectation ou d'un test est considéré comme constant c
  2. Le temps d'une séquence d'instructions est la somme des temps d'exécution des instructions qui la composent
  3. Le temps d'un branchement conditionnel est égal au t.e. du test + le max des deux t.e. correspondant aux deux alternatives (dans le cas d'un temps max)
  4. Le temps d'une boucle est égal à la somme du coût du test + du corps de la boucle x nombre d'itérations

Complexité en temps

IUT de LAVAL

- Evaluation de la complexité
  - Soit T(n) =
    - ★ Nombre d'unités de temps prises pour l'exécution
    - ★ Nombre d'instructions élémentaires exécutées
  - Instructions élémentaires (hors entrées/sorties) :
    - ★ Additions ou soustractions
    - ★ Multiplications ou divisions
    - ★ Operations arithmétiques simples (sans appel de fonctions)
    - ★ Tests d'égalités, d'inégalités
    - ★ Accès à la valeur d'un tableau
    - ★ Affectations
    - ★ Opérations de lectures et écritures simples
    - ★ Les operations booléennes (et,ou,non)

Complexité en temps

- Exemple: Calcul de la somme des n premiers entiers
  - Approche naïve

Complexité en temps

- Exemple: Calcul de la somme des n premiers entiers
  - Approche naïve (n additions)

Complexité en temps

- Exemple: Calcul de la somme des n premiers entiers
  - Approche naïve (n additions)

```
DEBUT
Lire (n);
somme <- 0;
POUR i = 1 à n
somme = somme + i;
FIN POUR
Ecrire (somme);
FIN
```

Complexité en temps

- Exemple: Calcul de la somme des n premiers entiers
  - Approche par récurrence

Complexité en temps

- Exemple: Calcul de la somme des n premiers entiers
  - Approche par récurrence 1 addition, 2 multiplications/divisions

Complexité en temps

- Exemple: Calcul de la somme des n premiers entiers
  - Approche par récurrence 1 addition, 2 multiplications/divisions

```
DEBUT
Lire (n) ;
somme = n*(n+1)/2 ;
FIN POUR
Ecrire (somme) ;
FIN
```

Complexité en temps

- La complexité dépend de la taille des données: « L'algorithme A est meilleur que l'algorithme B pour des données de telle taille »
  - Tri d'un tableau

Complexité en temps

- La complexité dépend de la taille des données: « L'algorithme A est meilleur que l'algorithme B pour des données de telle taille »
  - Tri d'un tableau
    - ★ taille du tableau (nombre d'éléments)

Complexité en temps

- La complexité dépend de la taille des données: « L'algorithme A est meilleur que l'algorithme B pour des données de telle taille »
  - Tri d'un tableau
    - ★ taille du tableau (nombre d'éléments)
  - Multiplications d'entiers

Complexité en temps

- La complexité dépend de la taille des données: « L'algorithme A est meilleur que l'algorithme B pour des données de telle taille »
  - Tri d'un tableau
    - ★ taille du tableau (nombre d'éléments)
  - Multiplications d'entiers
    - ★ nombre de chiffres

Complexité en temps

- La complexité dépend de la taille des données: « L'algorithme A est meilleur que l'algorithme B pour des données de telle taille »
  - Tri d'un tableau
    - ★ taille du tableau (nombre d'éléments)
  - Multiplications d'entiers
    - ★ nombre de chiffres
  - Multiplication de matrices

Complexité en temps

- La complexité dépend de la taille des données: « L'algorithme A est meilleur que l'algorithme B pour des données de telle taille »
  - Tri d'un tableau
    - ★ taille du tableau (nombre d'éléments)
  - Multiplications d'entiers
    - ★ nombre de chiffres
  - Multiplication de matrices
    - ★ dimensions des matrices

Complexité en temps

- Comparaison d'algorithmes
  - Entre deux algorithmes résolvant le même problème, le meilleur est celui qui possède le meilleur comportement assymptotique
  - Temps nécessaire pour traiter le pire des cas :
    - ★ Quelque soit l'entrée, l'algorithme ne dépassera pas le temps de calcul prévu
    - ★ Le pire des cas est souvent fréquent !

Complexité en temps

- Exemple : Tri naïf d'un tableau
  - Algorithme :

Complexité en temps

- Exemple : Tri naïf d'un tableau
  - Algorithme : pour chaque rang i(>= 1) dans le tableau
    - ★ Rechercher le maximum des éléments non triés (de i à n)
    - ★ Echanger ce maximum avec le premier élément non trié (en i)

Complexité en temps

- Exemple : Tri naïf d'un tableau
  - Algorithme : pour chaque rang i(>= 1) dans le tableau
    - ★ Rechercher le maximum des éléments non triés (de i à n)
    - ★ Echanger ce maximum avec le premier élément non trié (en i)
  - Complexité :

Complexité en temps

- Exemple : Tri naïf d'un tableau
  - Algorithme : pour chaque rang i(>= 1) dans le tableau
    - ★ Rechercher le maximum des éléments non triés (de i à n)
    - ★ Echanger ce maximum avec le premier élément non trié (en i)
  - Complexité :
    - ★ Pire des cas : tableau trié à l'envers
    - ★ Pour le rang i, la recherche coûte au plus $n - i + 1$ tests

Complexité en temps

- Exemple : Tri naïf d'un tableau
  - Algorithme : pour chaque rang i(>= 1) dans le tableau
    - ★ Rechercher le maximum des éléments non triés (de i à n)
    - ★ Echanger ce maximum avec le premier élément non trié (en i)
  - Complexité :
    - ★ Pire des cas : tableau trié à l'envers
    - ★ Pour le rang i, la recherche coûte au plus $n - i + 1$ tests
  - Au total: $n + (n-1) + (n-2) + ... + 1 = n(n+1)/2$
  - Complexité quadratique : $O(n2)$

Complexité en temps: *Notation «O»*

IUT de LAVAL

- Définition
  - Donne une majoration de l'ordre de grandeur du nombre d'opérations -> comportement assymptotique (taille n des données croît et tend vers l'infini)

$$T(n) = O(f(n)) \Leftrightarrow$$
$$\exists n_0 \in \aleph, \exists k \in \aleph^+ \textit{telsque} :$$
$$\forall n \geq n_0, T(n) \leq k \times f(n)$$

Complexité en temps: *Notation «O»*

IUT de LAVAL

- Définition
  - Donne une majoration de l'ordre de grandeur du nombre d'opérations -> comportement assymptotique (taille n des données croît et tend vers l'infini)

$$T(n) = O(f(n)) \Leftrightarrow$$
$$\exists n_0 \in \aleph, \exists k \in \aleph^+ \textit{telsque} :$$
$$\forall n \geq n_0, T(n) \leq k \times f(n)$$

  - A partir d'un certain rang, la fonction f majore la fonction T
  - $T(n)$ est au plus égale à une constante multipliée par $f(n)$

Complexité en temps: *Notation «O»*

- Exemple
  - Soient 2 programmes A1 et A2 ayant les temps d'exécution
    - ★ $T1(n) = 25n$
    - ★ $T2(n) = 3n^2$

Complexité en temps: *Notation «O»*

- Exemple
  - Soient 2 programmes A1 et A2 ayant les temps d'exécution
    - ★ $T1(n) = 25n$
    - ★ $T2(n) = 3n^2$
  - A1 est meilleur que A2 pour tout $n > 8$
  - Les constantes (25 et 3) ne sont donc pas importantes

Complexité en temps: *Notation «O»*

- Exemples

  $T(n) = n^3 + 3n^2$
  $T(n) = n^3 + 3n^2 \leq n^3 + n^3 = 2n^3 \forall n \geq 3$
  En posant $n_0 = 3$ et $k = 2$, $T(n) \in O(n^3)$ La complexité est cubique

  $T(n) = (n+1)^2$
  $T(n) = (n+1)^2 = n^2 + 2n + 1 \leq n^2 + 2n^2 + n^2 = 4n^2 \forall n \geq 1$
  En posant $n_0 = 1$ et $k = 4$, $T(n) \in O(n^2)$
  La complexité est quadratique

Complexité en temps: *Notation «O»*

IUT de LAVAL

- Propriétés
  - Les constantes ne sont pas importantes

$$T(n) \in O(f(n)) => T(n) \in O(c \times f(n))$$

Complexité en temps: *Notation «O»*

IUT de LAVAL

- Propriétés
  - Les constantes ne sont pas importantes

$$T(n) \in O(f(n)) => T(n) \in O(c \times f(n))$$

  - Les termes d'ordre inférieur sont négligeables

$$T(n) = a_k n^k + ... + a_1 n + a_0 => T(n) \in O(n^k)$$

Complexité en temps: *Notation «O»*

IUT de LAVAL

- Propriétés
  - Les constantes ne sont pas importantes
$$T(n) \in O(f(n)) => T(n) \in O(c \times f(n))$$
  - Les termes d'ordre inférieur sont négligeables
$$T(n) = a_k n^k + ... + a_1 n + a_0 => T(n) \in O(n^k)$$
  - Somme
$$\begin{gathered} T_1(n) \in O(f_1(n)) \\ T_2(n) \in O(f_2(n)) \\ => (T_1(n) + T_2(n)) \in O(f_1(n) + f_2(n)) \end{gathered}$$

Complexité en temps: *Notation «O»*

IUT de LAVAL

- Propriétés
  - Les constantes ne sont pas importantes
$$T(n) \in O(f(n)) => T(n) \in O(c \times f(n))$$
  - Les termes d'ordre inférieur sont négligeables
$$T(n) = a_k n^k + ... + a_1 n + a_0 => T(n) \in O(n^k)$$
  - Somme
$$\begin{gathered} T_1(n) \in O(f_1(n)) \\ T_2(n) \in O(f_2(n)) \\ => (T_1(n) + T_2(n)) \in O(f_1(n) + f_2(n)) \end{gathered}$$
  - Multiplication
$$\begin{gathered} T_1(n) \in O(f_1(n)) \\ T_2(n) \in O(f_2(n)) \\ => (T_1(n) \times T_2(n)) \in O(f_1(n) \times f_2(n)) \end{gathered}$$

Complexité en temps: *Notation «O»*

IUT de LAVAL

- Remarque
  - En notation assymptotique, le temps de calcul ne dépend pas de l'ordinateur utilisé

Complexité en temps: *Notation «O»*

IUT de LAVAL

- Remarque
  - En notation assymptotique, le temps de calcul ne dépend pas de l'ordinateur utilisé
- Ordres de grandeurs courantes
  - $O(1)$ constante
  - $O(n)$ linéaire
  - $O(log(n))$ logarithmique
  - $O(n^2)$ quadratique
  - $O(n^3)$ cubique
  - $O(2^n)$ exponentielle
  - $O(nlog(n))$ n log n

Complexité en temps: *Notation «O»*

IUT de LAVAL

- Etude de cas
  - Recherche d'un élément dans un tableau trié
  - Cas 1 : algorithme naïf

Complexité en temps: *Notation «O»*

IUT de LAVAL

- Etude de cas
  - Recherche d'un élément dans un tableau trié
  - Cas 1 : algorithme naïf

```
Booleen Fonction Appartient1(Reel x, TABLEAU de Réel E[n])
Début
Pour i de 0 à n
    Si (x==E[i]) alors Renvoyer (true);
    FinSi
Renvoyer(false);
Fin Pour
Fin
```

Complexité en temps: *Notation «O»*

- Complexité pour le cas 1 :

?

A vous de jouer

Complexité en temps: *Notation «O»*

- Complexité pour le cas 1

Complexité en temps: *Notation «O»*

- Complexité pour le cas 1

1. Affectation de la valeur 0 à i -> 1
2. Calcul de la taille de E 0 à i -> 1
3. Vérification de la condition 0 à i -> 1
4. Accès au ième élément de E et comparaison de cet élément avec x 0 à i -> 2
5. Incrémentation de i et retour en 2 -> 1

Complexité en temps: *Notation «O»*

- Complexité pour le cas 1

1. Affectation de la valeur 0 à i -> 1
2. Calcul de la taille de E 0 à i -> 1
3. Vérification de la condition 0 à i -> 1
4. Accès au ième élément de E et comparaison de cet élément avec x 0 à i -> 2
5. Incrémentation de i et retour en 2 -> 1

Les étapes 3, 4 et 5 sont faites autant de fois qu'il y a d'éléments dans E.
Soit n = taille(E):

$$\textbf{Complexité} = O(2 + 4n) = O(n)$$

Complexité en temps: *Notation «O»*

- Cas 2 : approche par dichotomie

```
Appartient2(x,E) {Dichotomie(x, E, 0, taille(E)); }
Dichotomie(x,E,i,j) {
   if (i>j) return false;
   else {
   k = (i+j) /2; u=E[k];
     if (x==u) return true;
     else {
        if (x<u) return (Dichotomie(x,E,i,k-1);
        else return (Dichotomie(x,E,k+1,j));
     }}}
```

Complexité en temps: *Notation «O»*

- Complexité pour le cas 2

Complexité en temps: *Notation «O»*

- Complexité pour le cas 2
  - S'il existe des choix : prendre le pire des cas
  - Pire des cas : celui qui correspond à la sous-liste la plus grande

=> 2ème appel de dichotomie

Complexité en temps: *Notation «O»*

- Complexité pour le cas 2
  - S'il existe des choix : prendre le pire des cas
  - Pire des cas : celui qui correspond à la sous-liste la plus grande

=> 2ème appel de dichotomie

1. Calcul de la taille de E et copie de x, E, 0 et taille(E) -> 5
2. Comparaison (==) de 2 valeurs i et j -> 1
3. Calcul de k = (i+j) /2 ; accès au kème élément de E et affectation de sa valeur à u -> 5
4. Comparaison (==) de x avec u -> 1
5. Calcul de k+1, copie de x,E,k+1 et j et retour en 2 -> 5

Complexité en temps: *Notation «O»*

- Les étapes de 2 à 5 sont réalisées autant de fois que n la taille(E) est divisé par 2
  - Partie entière de $log_2(n)$

$$\text{Complexité} = O(5 + log_2(n)(1 + 3 + 1 + 5)) = O(10log_2(n) + 5) = O(log_2(n))$$

Complexité en temps: *Notation «O»*

IUT de LAVAL

- Les étapes de 2 à 5 sont réalisées autant de fois que n la taille(E) est divisé par 2
  - Partie entière de $log_2(n)$

$$\text{Complexité} = O(5 + log_2(n)(1 + 3 + 1 + 5)) = O(10log_2(n) + 5) = O(log_2(n))$$

  - L'algorithme 2 est donc plus efficace que le premier
  - Dans le cas général, tous les algorithmes type « Diviser pour régner » sont en $O(log_2(n))$

Complexité en temps: *Notation «O»*

- Cas des boucles imbriquées

```
Tri(E)
{
   for (i=0;i<(taille(E)-1);i++)
   { MinI = i;
     for (j=i+1;j<taille(E);j++)
        { if(E[j] < E[i])MinI = j; }
     MinVal = E[MinI];
     E[MinI] = E[i];
     E[i] = MinVal;
     }}
```

Complexité en temps: *Résumé*

- Déroulement pire des cas
  - Chaque boucle *for* est parcourue complètement

Complexité en temps: *Résumé*

- Déroulement pire des cas
  - Chaque boucle *for* est parcourue complètement
- Nombre d'instructions élémentaires
  - Boucle intérieure

Complexité en temps: *Résumé*

- Déroulement pire des cas
  - Chaque boucle *for* est parcourue complètement
- Nombre d'instructions élémentaires
  - Boucle intérieure
    - ★ 7 par itération. Nombre d'itérations $= 1 + (n - 1) - (i + 1)$
    - ★ Total $= 7 \times (n - 1 - i)$

Complexité en temps: *Résumé*

- Déroulement pire des cas
  - Chaque boucle *for* est parcourue complètement
- Nombre d'instructions élémentaires
  - Boucle intérieure
    - ★ 7 par itération. Nombre d'itérations $= 1 + (n - 1) - (i + 1)$
    - ★ Total $= 7 \times (n - 1 - i)$
- Boucles extérieure

Complexité en temps: *Résumé*

- Déroulement pire des cas
  - Chaque boucle *for* est parcourue complètement
- Nombre d'instructions élémentaires
  - Boucle intérieure
    - 7 par itération. Nombre d'itérations $= 1 + (n-1) - (i+1)$
    - Total $= 7 \times (n - 1 - i)$
- Boucles extérieure
  - 8 + les $7 \times (n - 1 - i)$. Nombre d'itérations $= n - 1$ (de 0 à $n - 2$)

Complexité en temps: *Résumé*

- Déroulement pire des cas
  - Chaque boucle *for* est parcourue complètement
- Nombre d'instructions élémentaires
  - Boucle intérieure
    - 7 par itération. Nombre d'itérations $= 1 + (n - 1) - (i + 1)$
    - Total $= 7 \times (n - 1 - i)$
- Boucles extérieure
  - 8 + les $7 \times (n - 1 - i)$. Nombre d'itérations $= n - 1$ (de 0 à $n - 2$)
- Nombre d'instructions total

Complexité en temps: *Résumé*

- Déroulement pire des cas
  - Chaque boucle *for* est parcourue complètement
- Nombre d'instructions élémentaires
  - Boucle intérieure
    - 7 par itération. Nombre d'itérations $= 1 + (n-1) - (i+1)$
    - Total $= 7 \times (n - 1 - i)$
- Boucles extérieure
  - 8 + les $7 \times (n - 1 - i)$. Nombre d'itérations $= n - 1$ (de 0 à $n - 2$)
- Nombre d'instructions total
  - $8+7(n-1)+8+7(n-2)+8+7(n-3)+\ldots = 8(n-1)+7n(n-1)/2$

Complexité en temps: *Résumé*

IUT de LAVAL

- Déroulement pire des cas
  - Chaque boucle *for* est parcourue complètement
- Nombre d'instructions élémentaires
  - Boucle intérieure
    - ★ 7 par itération. Nombre d'itérations $= 1 + (n-1) - (i+1)$
    - ★ Total $= 7 \times (n - 1 - i)$
- Boucles extérieure
  - 8 + les $7 \times (n - 1 - i)$. Nombre d'itérations $= n - 1$ (de 0 à $n - 2$)
- Nombre d'instructions total
  - $8+7(n-1)+8+7(n-2)+8+7(n-3)+\ldots = 8(n-1)+7n(n-1)/2$

$$\text{Complexité} = O(8(n-1) + 7n(n-1)/2) = O(n^2)$$

Complexité en temps: *Résumé*

- La complexité d'un algorithme se mesure essentiellement en calculant le nombre d'opérations élémentaires pour traiter une (ou des) donnée(s) de taille n.
- Les opérations élémentaires considérées sont :
  - Le nombre de comparaisons.
  - Le nombre d'affectations.
  - Le nombre d'opérations (+,*) réalisées par l'algorithme.

Complexité en temps: *Conseils*

IUT de LAVAL

- Qualités d'un algorithme :
  1. Maintenable (facile à comprendre, coder, déboguer)
  2. Rapide
- Conseils :
  1. Privilégier le point 2 sur le point 1 uniquement si on gagne en complexité
  2. « ce que fait » l'algorithme doit se lire lors d'une lecture rapide : Une idée par ligne
- La rapidité d'un algorithme est un élément d'un tout, définissant les qualités de celui-ci

Les tris

- Fonction souvent utilisée et sur de nombreuses données
- Caractéristiques:
  - Complexité
  - Stabilité (ordre préservé en cas d'égalité)
  - Progressivité
- Exemples:
  - Tri bulle
  - Tri par fusion
  - Tri rapide
  - Tri par sélection

Tri par insertion

- Principe
  - Une sous séquence non triée
  - Une sous séquence triée
  - On prend un élément quelconque non trié
  - On l'insère à sa place dans la séquence triée N fois
- Remarque: recherche par dichotomie possible

Tri rapide

IUT de LAVAL

- On choisit un élément «pivot»
- On échange (partition) pour que:
  - partie avant pivot $<$ pivot
  - partie après pivot $>$ pivot
- Détail:
  - On part de la gauche pour trouver le 1er élément $>$ pivot
  - On part de la droite pour trouver le 1er élément $<$ pivot
  - On les permute
- On partitionne récursivement

Les tris

- Complexité minimale en $O(n \times log(n))$
- Choisir:
  - Peu de données (< 100): tri par sélection (moins d'échanges) ou insertion (si presque trié)
  - Sinon: tri rapide (quicksort)

IUT de LAVAL

Fin