Les Tableaux et leurs opérations de base

Yann Strozecki
yann.strozecki@uvsq.fr

Septembre 2016

Rappels sur la complexité

La complexité d'un algorithme mesure le nombre d'opérations élémentaires effectuée en fonction de la taille de l'entrée.

C'est une mesure indépendante de la machine et du compilateur utilisée.

Elle doit permettre de comparer le temps pris par des algorithmes quand les entrées deviennent grande : complexité asymptotique.

Complexité asymptotique

- Caractérisation du comportement d'un algorithme $\mathcal{A}$ sur l'ensemble des données D_n de taille n.
- $Cout_{\mathcal{A}}(d)$: coût de l'algorithme $\mathcal{A}$ sur la donnée d.

Complexité dans le pire cas

$$Max_{\mathcal{A}}(n) = \max\{Cout_{\mathcal{A}}(d), d \in D_n\}$$

Problème de la taille

- Pour donner un sens aux fonctions de complexité, il faut savoir déterminer la taille d'une entrée.
- **Règle générale :** expliciter ce qui est considéré comme la taille de l'entrée.
- **Règle simple :** c'est le nombre d'objets de base dans l'entrée.
- Un tableau de n entiers est de taille n.

Problème de la taille

- Pour donner un sens aux fonctions de complexité, il faut savoir déterminer la taille d'une entrée.
- **Règle générale :** expliciter ce qui est considéré comme la taille de l'entrée.
- **Règle simple :** c'est le nombre d'objets de base dans l'entrée.
- Un tableau de n entiers est de taille n.
- Ambiguïté : il y a plusieurs paramètres, complexité d'un algo qui calcule a^b ?
- Ambiguïté : pour un entier, on considère sa valeur ou sa taille (ou rien) ?

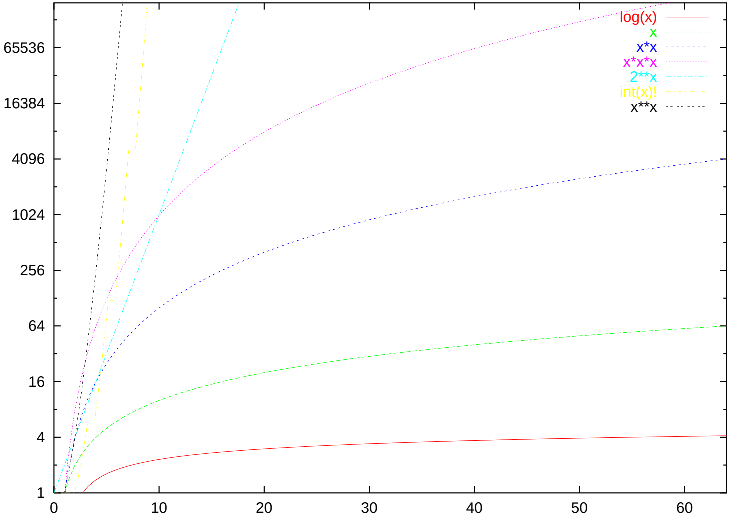
Problème de la taille

- Pour donner un sens aux fonctions de complexité, il faut savoir déterminer la taille d'une entrée.
- **Règle générale :** expliciter ce qui est considéré comme la taille de l'entrée.
- **Règle simple :** c'est le nombre d'objets de base dans l'entrée.
- Un tableau de n entiers est de taille n.
- Ambiguïté : il y a plusieurs paramètres, complexité d'un algo qui calcule a^b ?
- Ambiguïté : pour un entier, on considère sa valeur ou sa taille (ou rien) ?

Ordres de grandeurs

- Une approximation de la fonction de complexité est suffisante.
- On s'intéresse au nombre d'opérations à un facteur multiplicatif et additif près. On dit qu'une complexité est un $O(f(n))$ pour dire qu'elle est de l'ordre de $f(n)$.
- Utilisation d'une échelle de comparaison avec les fonctions $n!, 2^n, n^3, n^2, n \log n, n, \log n$
- Chacune de ces fonctions croît infiniment plus vite que celles qui sont à sa gauche.

Ordres de grandeurs



Définition formelle de la notations O

O « Borne Supérieure »

Soient f et $g : \mathbb{N} \rightarrow \mathbb{R}_+$: $f = O(g)$ ssi $\exists c \in \mathbb{R}_+, \exists n_0 \in \mathbb{N}$ tels que :

$$\forall n > n_0, f(n) \leq c \times g(n)$$

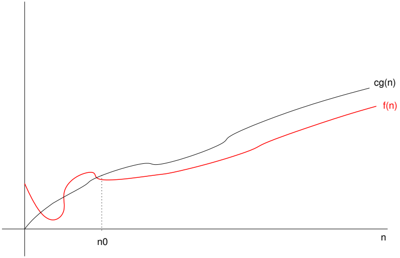


Figure – $f(n) = O(g(n))$

Exemples

$$2n = O(n^2)$$

$$2n = O(n)$$

$$3n^3 + n^2 \log(n)^4 + 5 = O(n^3)$$

$$2n \neq O(\log(n))$$

Un algorithme ...

- de complexité $O(1)$ effectue un nombre constant d'opérations
- de complexité $O(n)$ est un algorithme linéaire
- de complexité $O(n^k)$ est un algorithme polynomial

Quelques chiffres ...

		Complexité						
		1	$\log n$	n	$n \log n$	n^2	n^3	2^n
Taille des données	$n = 10^2$	$1\mu s$	$6.6\mu s$	$0.1ms$	$0.6ms$	$10ms$	$1s$	$4.10^{16}a$
	$n = 10^3$	$1\mu s$	$9.9\mu s$	$1ms$	$9.9ms$	$1s$	$16.6min$	∞
	$n = 10^4$	$1\mu s$	$13.2\mu s$	$10ms$	$0.1s$	$100s$	$11.5j$	∞
	$n = 10^5$	$1\mu s$	$16.6\mu s$	$0.1s$	$1.66s$	2.7h	$31.7a$	∞
	$n = 10^6$	$1\mu s$	$19.9\mu s$	$1s$	$19.9s$	$11.5j$	$31.7 * 10^3a$	∞

Temps d'exécution en fonction de la complexité d'un algorithme et de la taille des données.

$\infty =$ « $> 10^{25}$ années »

Nombre d'opérations par seconde $= 10^6$

Quelques chiffres ...

		Complexité						
		1	$\log n$	n	$n \log n$	n^2	n^3	2^n
Taille des données	$n = 10^2$	$1ps$	$6.64ps$	$0.1ns$	$0.66ns$	$0.01\mu s$	$1\mu s$	$4\ 10^{10}a$
	$n = 10^3$	$1ps$	$9.96ps$	$1ns$	$9.96ns$	$1\mu s$	$0.001s$	∞
	$n = 10^4$	$1ps$	$13.28ps$	$10ns$	$132.87ns$	10 ms	$1s$	∞
	$n = 10^5$	$1ps$	$16.6ps$	$0.1\mu s$	$1.6\mu s$	0.01 s	> 16 min	∞
	$n = 10^6$	$1ps$	$19.93ps$	$1\mu s$	$19.93\mu s$	$1s$	> 11 jours	∞

Temps d'exécution en fonction de la complexité d'un algorithme et de la taille des données.

$\infty =$ « $> 10^{25}$ années »

Nombre d'opérations par seconde $= 10^{12}$

La majorité des gains de vitesse ces dernières années vient de l'amélioration des algorithmes, pas de la loi de Moore.

Quelques chiffres ...

		Complexité						
		1	$\log n$	n	$n \log n$	n^2	n^3	2^n
Taille des données	$n = 10^2$	$1ps$	$6.64ps$	$0.1ns$	$0.66ns$	$0.01\mu s$	$1\mu s$	$4\ 10^{10}a$
	$n = 10^3$	$1ps$	$9.96ps$	$1ns$	$9.96ns$	$1\mu s$	$0.001s$	∞
	$n = 10^4$	$1ps$	$13.28ps$	$10ns$	$132.87ns$	10 ms	$1s$	∞
	$n = 10^5$	$1ps$	$16.6ps$	$0.1\mu s$	$1.6\mu s$	0.01 s	>16 min	∞
	$n = 10^6$	$1ps$	$19.93ps$	$1\mu s$	$19.93\mu s$	$1s$	>11 jours	∞

Temps d'exécution en fonction de la complexité d'un algorithme et de la taille des données.

$\infty =$ « $> 10^{25}$ années »

Nombre d'opérations par seconde $= 10^{12}$

La majorité des gains de vitesse ces dernières années vient de l'amélioration des algorithmes, pas de la loi de Moore.

Les structures de données

On va utiliser dans nos algorithmes de nombreuses structures de données présentée ici par ordre de complexité.

- Structures de base : entier, flottant, caractère
- Tableaux : structure linéaire de taille fixée en adressage direct

Les structures de données

On va utiliser dans nos algorithmes de nombreuses structures de données présentée ici par ordre de complexité.

- Structures de base : entier, flottant, caractère
- Tableaux : structure linéaire de taille fixée en adressage direct
- Liste, file, pile ... : structures linéaires de taille variable

Les structures de données

On va utiliser dans nos algorithmes de nombreuses structures de données présentée ici par ordre de complexité.

- Structures de base : entier, flottant, caractère
- Tableaux : structure linéaire de taille fixée en adressage direct
- Liste, file, pile . . . : structures linéaires de taille variable
- Structure hybride : table de hachage

Les structures de données

On va utiliser dans nos algorithmes de nombreuses structures de données présentée ici par ordre de complexité.

- Structures de base : entier, flottant, caractère
- Tableaux : structure linéaire de taille fixée en adressage direct
- Liste, file, pile ... : structures linéaires de taille variable
- Structure hybride : table de hachage
- Structures arborescentes : tas, arbre binaire de recherche

Les structures de données

On va utiliser dans nos algorithmes de nombreuses structures de données présentée ici par ordre de complexité.

- Structures de base : entier, flottant, caractère
- Tableaux : structure linéaire de taille fixée en adressage direct
- Liste, file, pile . . . : structures linéaires de taille variable
- Structure hybride : table de hachage
- Structures arborescentes : tas, arbre binaire de recherche
- Graphe

Les structures de données

On va utiliser dans nos algorithmes de nombreuses structures de données présentée ici par ordre de complexité.

- Structures de base : entier, flottant, caractère
- Tableaux : structure linéaire de taille fixée en adressage direct
- Liste, file, pile ... : structures linéaires de taille variable
- Structure hybride : table de hachage
- Structures arborescentes : tas, arbre binaire de recherche
- Graphe
- Base de données

Les structures de données

On va utiliser dans nos algorithmes de nombreuses structures de données présentée ici par ordre de complexité.

- Structures de base : entier, flottant, caractère
- Tableaux : structure linéaire de taille fixée en adressage direct
- Liste, file, pile . . . : structures linéaires de taille variable
- Structure hybride : table de hachage
- Structures arborescentes : tas, arbre binaire de recherche
- Graphe
- Base de données

Les structures de données

On va utiliser dans nos algorithmes de nombreuses structures de données présentée ici par ordre de complexité.

- Structures de base : entier, flottant, caractère
- Tableaux : structure linéaire de taille fixée en adressage direct
- Liste, file, pile . . . : structures linéaires de taille variable
- Structure hybride : table de hachage
- Structures arborescentes : tas, arbre binaire de recherche
- Graphe
- Base de données

Structures Linéaires

Eléments d'un même type stockés à la suite :

- **un tableau**
- **une liste**

Deux cas possibles :

- Eléments triés (l'ordre sur les éléments doit être maintenu)
- L'ordre des éléments n'a aucune importance.

Opérations sur les structures linéaires

- Insérer un nouvel élément
- Supprimer un élément
- Rechercher un élément
- Afficher l'ensemble des éléments
- Concaténer deux ensembles d'éléments
- ...

Définition des structures

Un tableau

```
Enregistrement Tab {
    T[NMAX] : entier;
    Fin     : entier;
}
```

Un tableau est une suite d'éléments contigus accessibles en temps constant. On doit connaître sa taille pour le manipuler.

Tableaux à plusieurs dimensions

On connaît les tableaux à deux dimensions : les matrices.

En général on peut utiliser n'importe quel tableau avec un nombre fixe de dimension. On utilise la syntaxe $T[n_1][n_2]\dots[n_k]$.

Comment faire si k est défini dans l'entrée du programme ?

Tableaux à plusieurs dimensions

On connaît les tableaux à deux dimensions : les matrices.

En général on peut utiliser n'importe quel tableau avec un nombre fixe de dimension. On utilise la syntaxe $T[n_1][n_2]\dots[n_k]$.

Comment faire si k est défini dans l'entrée du programme ?

Deux possibilités :

- Des tableaux de tableaux de tableaux ...
- Un tableau à une dimension tel que quand on considère $T'[i]$, on a accès à l'élément $T[i_1]\dots[i_k]$ c'est à dire que i code $i_1, \dots, i_k$.

Tableaux à plusieurs dimensions

On connaît les tableaux à deux dimensions : les matrices.

En général on peut utiliser n'importe quel tableau avec un nombre fixe de dimension. On utilise la syntaxe $T[n_1][n_2]\dots[n_k]$.

Comment faire si k est défini dans l'entrée du programme ?

Deux possibilités :

- Des tableaux de tableaux de tableaux ...
- Un tableau à une dimension tel que quand on considère $T'[i]$, on a accès à l'élément $T[i_1]\dots[i_k]$ c'est à dire que i code $i_1,\dots,i_k$.

Différentes opérations pour les sructures linéaires

1. Structures non triées
   - 1.1 Insertion un tableau
   - 1.2 Recherche dans un tableau
   - 1.3 Concaténation de deux tableaux
2. Structures triées
   - 2.1 Insertion dans un tableau
   - 2.2 Recherche dans un tableau
   - 2.3 Concaténation de deux tableaux

Tableau non trié : Insertion

Algorithme 1 Insertion dans un tableau non trié

Insertion(S : Tab, x : entier) : booléen

▷ ***Entrées : S (un tableau), x (élément à insérer)***

▷ ***Sortie : le tableau S dans lequel x a été inséré.***

Début

S.Fin $\leftarrow$ S.Fin + 1 ;

S.T[S.Fin] $\leftarrow$ x ;

Fin

Tableau non trié : Insertion

- **Opération fondamentale :** affectation
- **Nombre d'opérations fondamentales** : 2 affectations.
- **Complexité :** $\mathbf{O(1)}$ (Temps constant).

Ici on suppose qu'on peut redimensionner le tableau simplement. Faux en pratique.

Tableau non trié : Recherche

Algorithme 2 Recherche dans un tableau non trié

```
Recherche(S : Tab, x : entier) : booléen
▷ Entrées : S (un tableau), x (élément recherché)
▷ Sortie : vrai si l'élément x a été trouvé dans le tableau S, faux
sinon.
 Début
▷ Variable Locale
     i : entier ;
 pour i de 1 à S.Fin  faire
      si (S.T[i] = x)
           retourner  vrai ;
      fin si
 fin pour
 retourner  faux ;
 Fin
```

Tableau non trié : Recherche

- **Opération fondamentale :** comparaison
- **A chaque itération :**
  - 1 comparaison (`Si ... Fin Si`)
  - 1 comparaison (`Pour ... Fin Pour`)
- **Nombre d'itérations maximum :** nombre d'éléments du tableau
- **Complexité :** Si n est le nombre d'éléments du tableau $\mathbf{O(n)}$.

Concaténation de deux tableaux non triés

Algorithme 3 Concaténation de deux tableaux

```
Concaténation(S : Tab, T : Tab) : Tab
▷ Entrées : S et T deux tableaux
▷ Sortie : Un tableau U qui contient les deux tableaux S et T à la suite
 Début
▷ Variable Locale
     i : entier ;
U.Fin = S.Fin + T.Fin ;
 pour i de 1 à S.Fin  faire
     U[i] = S[i] ;
 fin pour
 pour i de 1 à T.Fin  faire
     U[S.fin + i] = T[i] ;
 fin pour
 retourner U ;
 Fin
```

Complexité en $O(S.Fin + T.Fin)$ i.e. linéaire en la taille de l'entrée.

Tableau Trié : Insertion

Algorithme 4 Insertion dans un tableau trié

```
Insertion(S : Tab, x : entier) : booléen
▷ Entrées : S (un tableau), x (élément recherché)
▷ Sortie : le tableau S dans lequel x a été inséré.
▷ Pré-condition : le tableau S trié par ordre croissant.
▷ Variable Locale
    i,k :entiers ;

Début
    si (S.Fin == -1)
        S.Fin ← 0 ;
        S.T[S.Fin] ← x ;
    sinon
        i ← 0 ;
        tant que (i < S.Fin et S.T[i] < x)
            i ← i + 1 ;
        fin tant que
        si (i = S.Fin et S.T[i] < x)
            k ← S.Fin + 1 ;
        sinon
            k ← i ;
        fin si
        pour i de S.Fin + 1 à k en décroissant faire
            S.T[i] ← S.T[i-1] ;
        fin pour
        S.T[k] ← x ;
        S.Fin ← S.Fin + 1 ;
    fin si
Fin
```

Tableau trié : insertion

- **Opération fondamentale :** affectation
- **Recherche de la bonne position :** k affectations
- **Décaler à droite :** $n - k$ affectations
- **Insérer élément :** 1 affectation
- **Total :** $n + 2$ affectations
- **Complexité :** $\mathbf{O(n)}$ si n est le nombre d'éléments du tableau.

Tableau trié : recherche

1. Première idée :
   - On compare l'élément recherché à tous les éléments du tableau comme on l'a fait pour un tableau non trié.
   - Problème : on ne tient pas compte de l'ordre des éléments.

2. Deuxième idée :
   - Optimisation : on s'arrête dès qu'on a trouvé un élément plus grand.
   - Complexité ?

Tableau trié : recherche

1. Première idée :
   - On compare l'élément recherché à tous les éléments du tableau comme on l'a fait pour un tableau non trié.
   - Problème : on ne tient pas compte de l'ordre des éléments.
2. Deuxième idée :
   - Optimisation : on s'arrête dès qu'on a trouvé un élément plus grand.
   - Complexité ?
3. Troisième idée :
   - Recherche dichotomique en utilisant diviser pour régner
   - Utilisation du fait que les éléments sont triés.

Tableau trié : recherche

1. Première idée :
   - On compare l'élément recherché à tous les éléments du tableau comme on l'a fait pour un tableau non trié.
   - Problème : on ne tient pas compte de l'ordre des éléments.
2. Deuxième idée :
   - Optimisation : on s'arrête dès qu'on a trouvé un élément plus grand.
   - Complexité ?
3. Troisième idée :
   - Recherche dichotomique en utilisant diviser pour régner
   - Utilisation du fait que les éléments sont triés.

Tableau trié : recherche dichotomique

Idée de l'algorithme :

- Soit M l'élément du milieu du tableau.
  - Si élément = M on a trouvé.
  - Si élément < M, l'élément est dans la première moitié du tableau.
  - Si élément > M, l'élément est dans la seconde moitié du tableau.

- On coupe le tableau en morceaux de plus en plus petits. On ne créé pas de nouvelles structures mais on utilise des indices de début et de fin.

Tableau trié : recherche dichotomique

Idée de l'algorithme :

- Soit M l'élément du milieu du tableau.
  - Si élément = M on a trouvé.
  - Si élément < M, l'élément est dans la première moitié du tableau.
  - Si élément > M, l'élément est dans la seconde moitié du tableau.

- On coupe le tableau en morceaux de plus en plus petits. On ne créé pas de nouvelles structures mais on utilise des indices de début et de fin.

Tableau trié : recherche

Algorithme 5 Recherche dichotomique

Recherche(x : entier, S : tableau) : booléen

▷ ***Entrées : x (élément recherché), S (espace de recherche)***

▷ ***Sortie : vrai si l'élément x a été trouvé dans le tableau S.***

▷ ***Variables Locales***

```
    m : entier ; g = 0 ; d = taille(S)-1

 Début
 tant que vrai faire
     m ← ⌊(g+d)/2⌋ ;
      si (x = S.T[m])
          retourner vrai ;
      sinon
          si (g = d) retourner faux
          si (x < S.T[m])
              d = m -1 ;
          sinon
              g = m+1 ;
          fin si
      fin si
 fin tant que
 Fin
```

Tableau trié : recherche

- **Opération fondamentale :** comparaison
- ***A chaque passage dans la boucle tant que, on diminue l'espace de recherche par 2*** et on fait au pire 3 comparaisons
- **Complexité :** Au pire on fera donc $\mathbf{O(\log_2 n)}$ tours de boucle et la complexité est donc en $\mathbf{O(\log_2 n)}$.
- Preuve de l'algorithme et de la complexité : au tableau.

Diviser pour régner

On veut résoudre un problème sur une instance x de taille n. Il suffit de trouver un algorithme A qui sait résoudre ce problème quand on lui fournit la solution à x_1 et x_2 avec x_1 et x_2 strictement plus petit que n.

- On coupe l'instance de départ x en de plus petites instances x_1,..., x_k.
- On résoud le problème sur les instances x_1,..., x_k ce qui nous donne les solutions $s_1, \ldots, s_k$.
- On combine les solutions $s_1, \ldots, s_k$ en une solution s à l'instance x.
- Il faut savoir résoudre le problème pour la taille 1.
- Pour que ça soit efficace il faut que k soit constant et que $taille(x_i) < cn$ avec c une constante inférieure à 1.

Diviser pour régner

On veut résoudre un problème sur une instance x de taille n. Il suffit de trouver un algorithme A qui sait résoudre ce problème quand on lui fournit la solution à x_1 et x_2 avec x_1 et x_2 strictement plus petit que n.

- On coupe l'instance de départ x en de plus petites instances x_1,. . ., x_k.
- On résoud le problème sur les instances x_1,. . ., x_k ce qui nous donne les solutions $s_1, \ldots, s_k$.
- On combine les solutions $s_1, \ldots, s_k$ en une solution s à l'instance x.
- Il faut savoir résoudre le problème pour la taille 1.
- Pour que ça soit efficace il faut que k soit constant et que $taille(x_i) < cn$ avec c une constante inférieure à 1.

Exponentiation rapide

Problème : on veut calculer a^n.

Y-a-t-il un algorithme plus rapide que l'algorithme naïf ?

Proposer un algorithme diviser pour régner.

Exponentiation rapide

Problème : on veut calculer a^n.

Y-a-t-il un algorithme plus rapide que l'algorithme naïf ?

Proposer un algorithme diviser pour régner.

Analyser sa complexité.

Exponentiation rapide

Problème : on veut calculer a^n.

Y-a-t-il un algorithme plus rapide que l'algorithme naïf ?

Proposer un algorithme diviser pour régner.

Analyser sa complexité.

Concaténation de deux tableaux triés

On pourrait les concaténer normalement puis les trier. Mais on n'a pas encore vu le tri et la complexité est plus élevée.

Solution en TD.

Concaténation de deux tableaux triés

On pourrait les concaténer normalement puis les trier. Mais on n'a pas encore vu le tri et la complexité est plus élevée.

Solution en TD.