Introduction à l'algorithmique

Hélène Collavizza

PeiP2

Année 2021-2022

Objectifs : se poser les bonnes questions

1) Caractéristiques d'un « bon algorithme »
2) Présentation de structures de données et algorithmes classiques

Plan

Critères pour caractériser un « bon algorithme »

- *Validité : terminaison et correction*
- *Notions élémentaires sur la complexité algorithmique*

Parcours élémentaires de tableaux

Tris

Algorithmes de base sur les listes

Rappels de récursivité

Algorithmes de base sur les arbres

Algorithmes de base sur les graphes

Validité : correction et terminaison

Exemple 1: Recherche d'un élément dans une séquence

Données : une séquence de n entiers distincts e_0, e_1, …., e_{n-1}
un entier x

Résultat : -1 si x n'est pas dans la séquence e_0,…., e_{n-1}
j si x = e_j

Recherche de 6 dans {3, 10, 4, 1, 6, 3} → 4
Recherche de 31 dans {3, 10, 4, 1, 6, 3} → -1

Principe de l'algorithme : parcourir la séquence en comparant x à e_0 puis à e_1, puis à e_2, ….

Si l'on trouve un i tel que e_i=x, le résultat est i,

Si l'on parcourt les e_i jusqu'à i=n, le résultat est -1

Question 1 : l'algorithme est-il correct ?

- Précondition PRE : conditions d'application de l'algorithme
- Postcondition POST : conditions vérifiées après l'algorithme
- Propriétés Pi : sur les variables internes qui assurent que si PRE est vrai alors POST le sera après exécution

Question 2 : l'algorithme termine-t-il ?

- Propriétés Pi : sur les variables internes qui assurent que si PRE est vrai alors la condition de sortie de boucle est atteinte ou l 'arrêt de la récursion est atteint

Question 3 : quelle est la place utilisée en mémoire ?

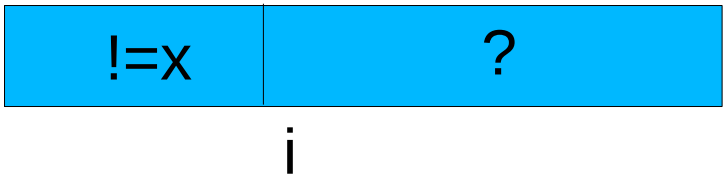
- Variables supplémentaires

Question 4 : quel est le temps d'exécution ?

- En ordre de grandeur

Question 1 : l'algorithme est-il correct ?

```
/** méthode pour rechercher l'indice d'un élément
* PRE : tab est un tableau d'entiers distincts,
*       x est un entier
* POST : renvoie i si tab[i] == x,
*        -1 si x n'est pas dans le tableau
*/
public int recherche(int[] tab, int x) {
    int i=0;
    // P1 : ∀ 0 ≤ j < i, tab[j]!=x
    while(i<tab.length){
        if (tab[i]==x) {
            // P2 : tab[i]==x
            return i ;
        }
        i++;
    }
    // P3 : ∀ 0 ≤ j < tab.length, tab[j]!=x
    return -1;
}
```



Question 2 : l'algorithme termine-t-il ?

au pire i croît de 0 à tab.length qui est une valeur finie

Question 3 : quelle est la place utilisée en mémoire ?

La place nécessaire pour stocker x et tab

Si tab contient n éléments on dit :

Complexité en espace = $\theta(n)$

Question 4 : quel est le temps d'exécution ?

Temps CPU : dépend de la machine

« Temps de l'algorithme » : on fait au plus tab.length passages dans la boucle for

Si le tableau contient n éléments :

Complexité en temps dans le pire des cas = n

Complexité en temps dans le meilleur des cas = 1

Complexité en temps en moyenne = p (n+1)/2 + n (1-p)

(où p est la probabilité pour que x soit dans le tableau)

Exemple 2 : Recherche d'un élément dans une séquence ordonnée

Données : une séquence de n entiers distincts e_0, e_1, …., e_{n-1} ordonnés par ordre croissant

un entier x

Résultat : -1 si x n'est pas dans la séquence e_0, e_1,…, e_{n-1}

i si $x = e_i$

Recherche de 6 dans {3, 4, 6, 10, 34} → 2

Recherche de 31 dans {3, 4, 6, 10, 34} → -1

Solution 1 : le même algorithme que dans le cas où les éléments ne sont pas ordonnés.

Solution 2 : utiliser le fait que les éléments sont ordonnés pour appliquer le paradigme « diviser pour régner »

Principe : comparer x à l'élément m qui est au milieu de la partie du tableau considérée.

Si x = m renvoyer l'indice de m

Si x < m chercher x dans la partie du tableau à gauche de m

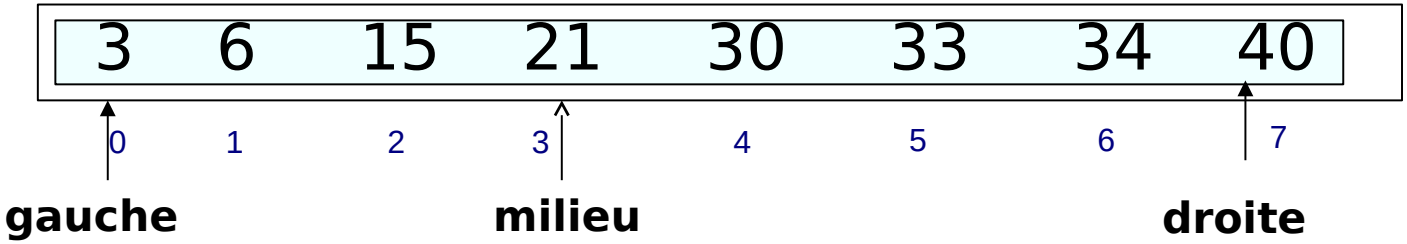
Si x > m chercher x dans la partie du tableau à droite de m

Si la partie considérée est vide, renvoyer –1

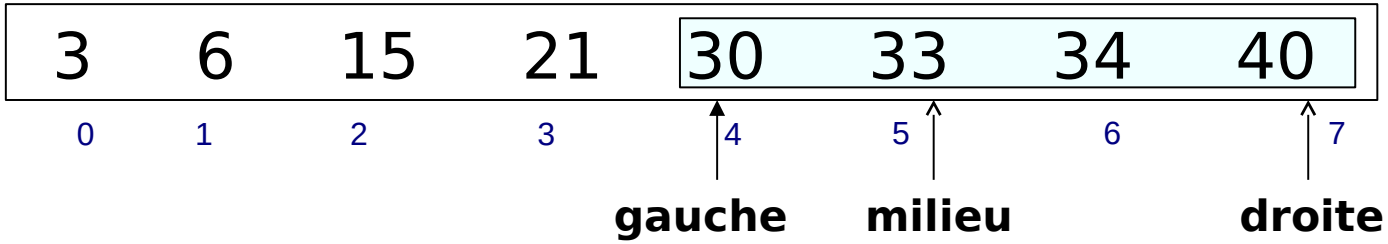
Avantage : meilleure complexité en temps (i.e en $O(\log_2 n)$)

Zone de recherche

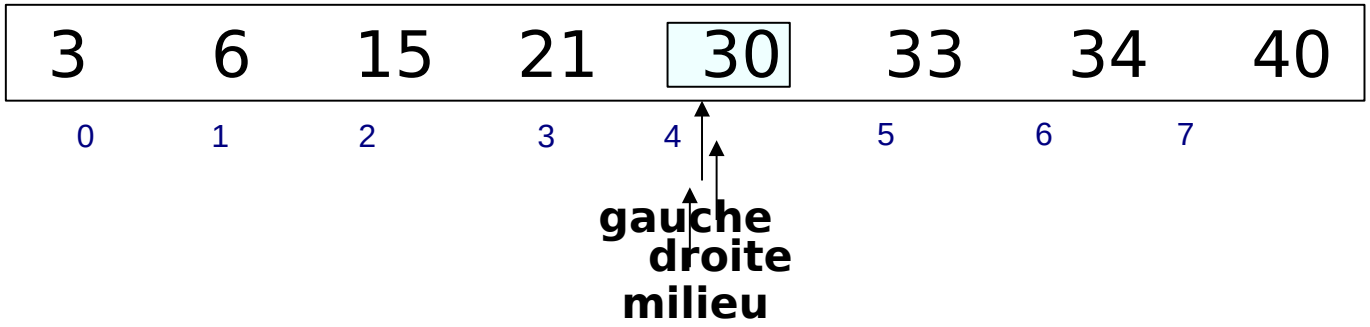
Recherche de 30



30>21



30<33



30 est trouvé; il est à l'indice 4

```
/** pour rechercher l'indice d'un élément
* antécédent : tab est un tableau d’entiers
*              distincts ordonnés par ordre croissant,
*              x est un entier
* conséquent : renvoie i si tab[i] == x,
*              -1 si x n'est pas dans le tableau
* complexité : O(log n)
*/
public int rechercheVite(int[] tab, int x) {
   int gauche = 0;
   int droite = tab.length - 1;
   int milieu;
```

```
   while (gauche <= droite) {
       // P1 :  ∀ j < gauche tab[j]!=x
       //       ∀ j > droite tab[j]!=x
       milieu  = (gauche + droite) / 2 ;
       if (x==tab[milieu])
           // P2 : x est à l'indice milieu
           return milieu;
       if (x<tab[milieu]) droite = milieu - 1;
          else gauche = milieu + 1;
       }
   // P3 : x n'est pas dans le tableau
    return -1;
}
```

A VOUS

Que se passe-t-il si les éléments du tableau ne sont pas distincts dans le cas de la recherche simple ?

21	6	2	21	21	33	21	40
0	1	2	3	4	5	6	7

Que se passe-t-il si les éléments du tableau ne sont pas distincts dans le cas de la recherche dichotomique ?

3	6	21	21	21	33	34	40
0	1	2	3	4	5	6	7

Comment peut-on prendre en compte le fait que les éléments sont triés dans la recherche simple ?

Notations asymptotiques et complexité

Donner un ordre de grandeur de la durée d'exécution du programme, indépendamment de la machine sur lequel il est exécuté

- Quelques rappels mathématiques indispensables en informatique
- Notations asymptotiques ou comment négliger les constantes
- Recettes pour évaluer la complexité de programmes itératifs

Rappels mathématiques

Sommations

$$\sum_{i=0}^{n} f(i) = f(0) + f(1) + \ldots + f(n)$$

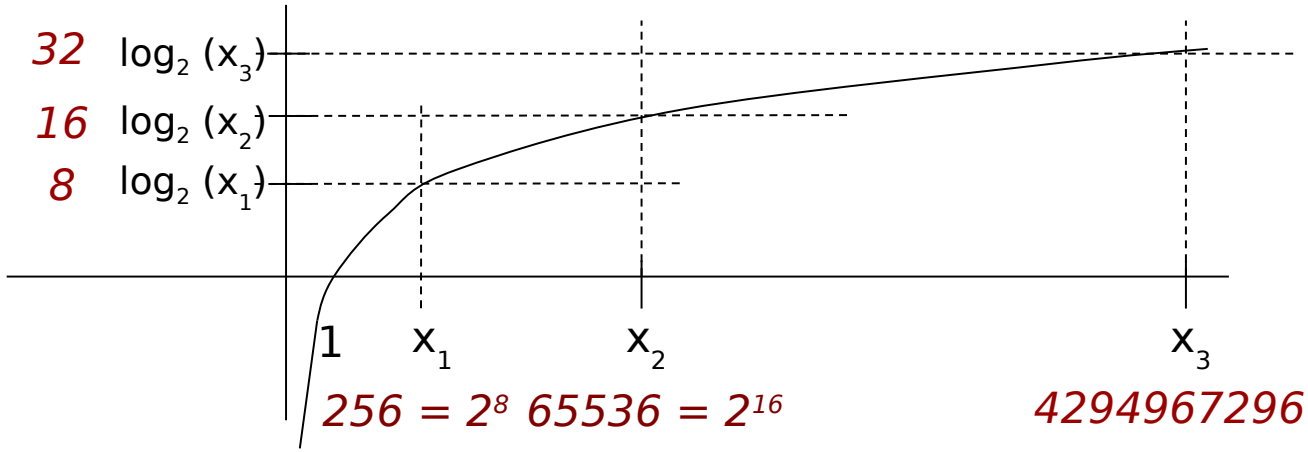
$$\sum_{i=0}^{n} i = n(n+1)/2$$

$$\sum_{i=0}^{n} a\, r^{i} = a\,(r^{n+1} - 1)/(r - 1)$$

Logarithmes

Pour b>1, x>0, y est le logarithme en base b de x si et seulement si $b^y = x$ (noté $\log_b x = y$)

Cas particulier connu le logarithme népérien : $e^y = x$ ssi $\ln(x)=y$



croissance très lente !

Propriétés :

Pour b et c >1

$\log_b b^a = a$

$\log_b (x\ y) = \log_b (x) + \log_b (y)$

$\log_b (x^a) = a \log_b (x)$

$\log_c (x) = \log_b (x) / \log_b (c)$

$\forall$ n entier strictement positif, $\exists$ k tel que $2^k \leq n < 2^{k+1}$

Notations asymptotiques ou comment négliger les constantes

Quelles sont les opérations élémentaires significatives ?

```
int j = 8;
for (int i=0;i<tab.length-1;i++) {
    int k = tab[i];
    if (tab[i] < tab[i-1]) tab[i] = tab[i] + tab[i+1];
}
```

Que choisir entre un algorithme qui fait $n^3/2$ opérations et un algorithme qui fait $5n^2$ opérations ?

si n=3, $n^3/2$ vaut 13,5 et $5n^2$ vaut 45

si n=100, $n^3/2$ vaut 500 000 et $5n^2$ vaut 5000

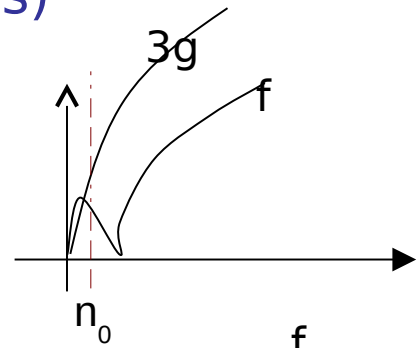
Besoin d'ignorer les facteurs constants et les petites valeurs des entrées → notations asymptotiques

Ordres de grandeur des fonctions (positives)

O(g)

f est un « grand O de g » ssi

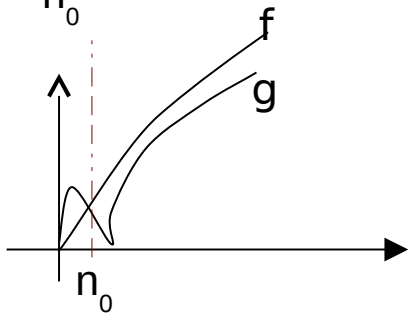
$\exists\, c > 0, \exists\, n_0 \geq 0\ \forall\, n \geq n_0, f(n) \leq c\, g(n)$



Ω(g)

f est un « grand oméga de g » ssi

$\exists\, c > 0, \exists\, n_0 \leq 0\ \forall\, n \geq n_0, f(n) \geq c\, g(n)$

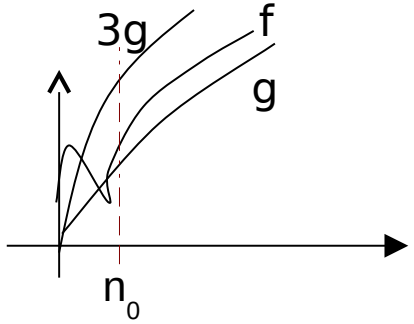


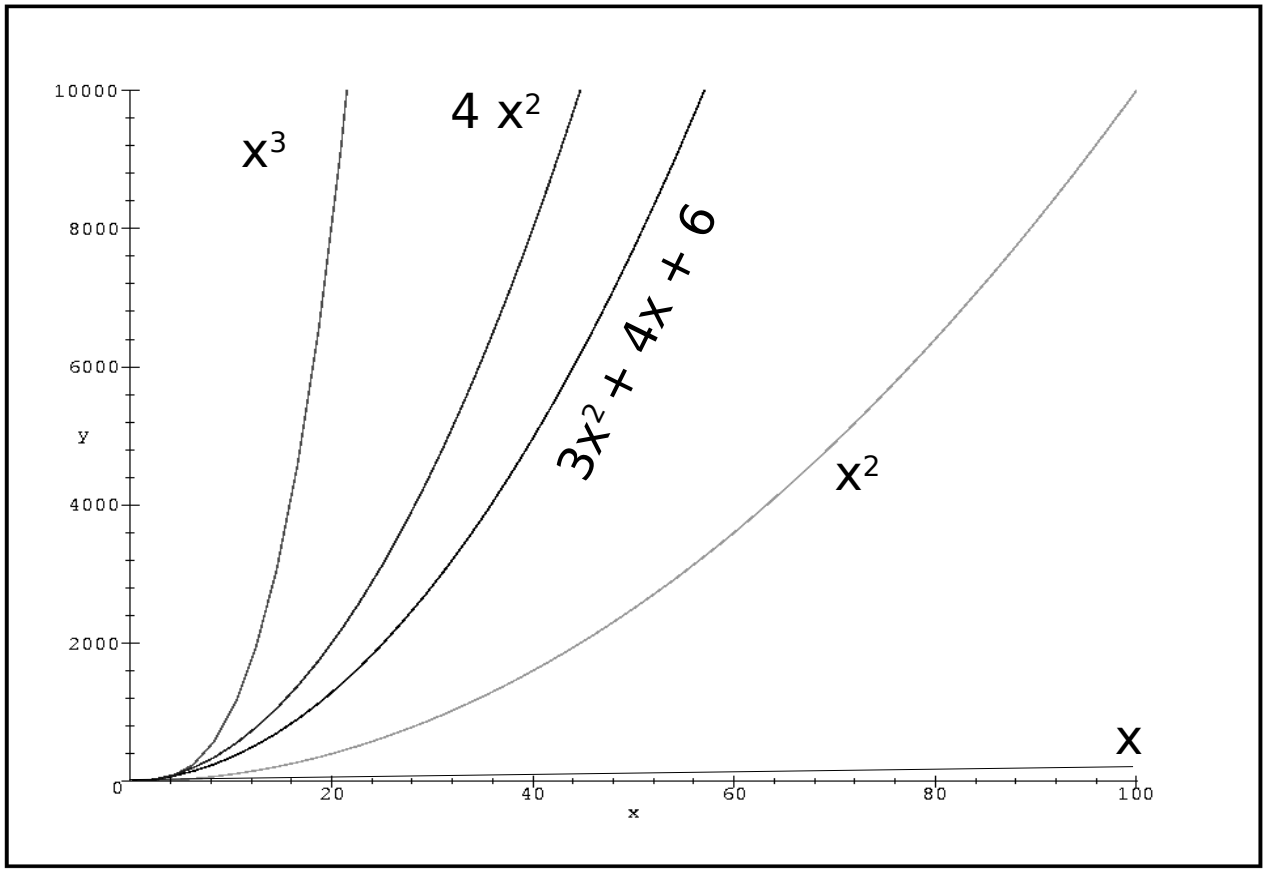
Θ(g)

f est « du même ordre que g » ssi

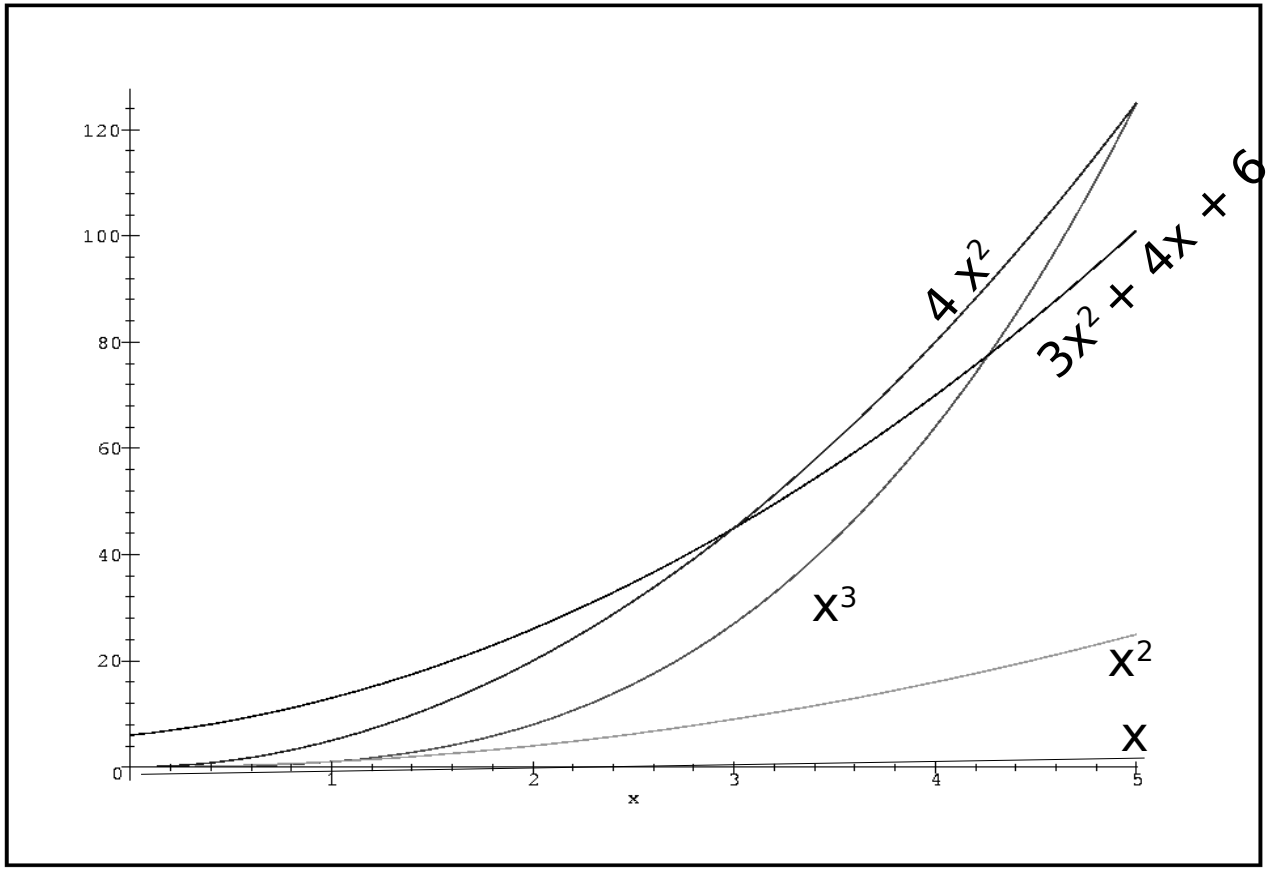
f est un « grand O de g » et

f est un « grand oméga de g »





$$3x^2 + 4x + 6 = \Theta(x^2)$$
$$= O(x^3)$$
$$= \Omega(x)$$


120
100
80
60
40
20
0
1
2
3
4
5
x
4 x²
3x² + 4x + 6
x³
x²
x

Propriétés

$f = \Theta(g) \Leftrightarrow g = \Theta(f)$

$O(c\,f) = O(f)$ si c constante — *peu importe les constantes*

vrai aussi pour Θ et Ω

$O(f + g) = O(\max(f,g))$

vrai aussi pour Θ et Ω

$$\sum_{i=0}^{n} a_{d-i}\, n^{d-i} = \Theta(n^d)$$ — *un polynôme de degré d est en $\Theta(n^d)$*

$$\sum_{i=0}^{n} a_{d-i}\, n^{d-i} = O(n^{d+1})$$ — *un polynôme de degré d est en $O(n^{d+1})$*

$\log_a n = \Theta(\log_2 n)$ — *peu importe la base du logarithme*

Complexités les plus courantes

- $\Theta(n)$ algorithme linéaire
- $\Theta(n^2)$ algorithme quadratique
- $\Theta(n^3)$ algorithme cubique
- $\Theta(\log_2 n)$ algorithme logarithmique
- $\Theta(n \log_2 n)$

- $\Theta(2^n)$ algorithme exponentiel

Un algorithme $\mathcal{A}$ est un :

O(g) : si $\mathcal{A}$ effectue *au plus* de l'ordre de g opérations

"borne supérieure"

Ω(g) : si $\mathcal{A}$ effectue *au moins* de l'ordre de g opérations

"borne inférieure"

Θ(g) : si $\mathcal{A}$ effectue *au moins* et *au plus* de l'ordre de g opérations

Exemple numérique arbitraire : on suppose qu'un algorithme prend 10^{-9}s pour traiter un tableau ayant 1 élément. Que se passe-t-il si on change la complexité de cet algorithme ?

n : nombre d'éléments	**$\Theta(n)$**	**$\Theta(\log_2 n)$**	**$\Theta(n \log_2 n)$**	**$\Theta(n^2)$**	**$\Theta(2^n)$**
5	$5\ 10^{-9}$ s	$3\ 10^{-9}$ s	$12\ 10^{-9}$ s	$25\ 10^{-9}$ s	$32\ 10^{-9}$ s
10	10^{-8} s	$4\ 10^{-9}$ s	$3\ 10^{-8}$ s	10^{-7} s	10^{-6} s
1000	10^{-6} s	10^{-8} s	10^{-5} s	10^{-3} s	$3\ 10^{282}$ siècles
10^6	10^{-3} s	$2\ 10^{-8}$ s	$2\ 10^{-2}$ s	10^3 s 20mn	siècles

Nombre d'éléments que l'on peut traiter en 24h selon la complexité

n	**$\log_2 n$**	**$n \log_2 n$**	**n^2**	**2^n**
$9\ 10^{13}$	10^{3e13}	$2\ 10^{12}$	10^7	32

Complexité

Complexité en fonction de quoi ?

de la taille des données auxquelles s'appliquent l'algorithme

Sur un entier : l'entier (ou la taille nécessaire pour le coder en binaire)

Sur une chaîne de caractères : longueur de la chaîne

Sur un objet java : taille des attributs

Sur un tableau : nombre d'éléments du tableau

NB: si on applique sur chaque élément une méthode dont la complexité dépend de la taille des éléments, il faut en tenir compte.

Sur un arbre : nombre d'éléments de l'arbre

Sur un graphe : nombre de sommets et de liaisons

Complexité en espace

Espace mémoire nécessaire pour stocker les données (ici, on se préoccupe des constantes).

Exemple : pour trier un tableau contenant n éléments, la complexité en espace optimale est n; certains algorithmes utilisent 2 x n.

Complexité en temps

Ordre de grandeur du temps d'exécution de l'algorithme (indépendamment de la machine) en fonction de la taille des données

Cette complexité peut dépendre de la configuration des données

Complexité dans le meilleur des cas

Temps d'exécution le plus faible; utile pour vérifier que l'algorithme ne perd pas de temps inutilement

Exemple : algorithme de tri appliqué à un tableau déjà trié !

Complexité dans le pire des cas

La plus importante. Donne une borne supérieure du temps d'exécution.

> *Commenter chaque méthode java avec sa complexité dans le pire des cas*

Complexité en moyenne

Intéressante quand on sait la calculer; donne le temps d'exécution moyen, quand on traite successivement des données n'ayant aucune propriété particulière.

Soit D_n l'ensemble des données de taille n. Soit I un sous ensemble de D_n et soit t(I) le nombre d'opérations élémentaires pour exécuter I.

Complexité dans le meilleur des cas

meilleur(n) = min $\{t(I), I \in D_n\}$

Complexité dans le pire des cas

pire(n) = max $\{t(I), I \in D_n\}$

Complexité en moyenne

moyenne(n) = $\sum_{I \subseteq D_n}$ Pr (I) t(I) où Pr(I) est la probabilité de I

Complexité temporelle des algorithmes itératifs

Les instructions élémentaires (affectation, comparaison) sont en temps constant

Séquence de blocs

I_1 ;

I_2 ;

est de complexité $\Theta(max(f_1(n),f_2(n)))$

où $\Theta(f_1(n))$ est la complexité de I_1 et $\Theta(f_2(n))$ la complexité de i_2

If then else

if (C) I_1 ; else I_2 ;

est de complexité $O(max(f(n),f_1(n),f_2(n))$

où $\Theta(f(n))$ est la complexité de C, $\Theta(f_1(n))$ la complexité de I_1 et $\Theta(f_2(n))$ la complexité de i_2

Itération for

for (int i=0 ; i< n ; i++)

I ;

est de complexité Θ (n (f(n)))

si I n'a aucun effet sur les variables i et n et que Θ(f(n)) est la complexité de I.

Si la complexité de I dépend de i, la complexité est en

$$\sum_{i=0}^{n-1} \Theta(f(i))$$

Itération while

While (C)

I;

*est de complexité Θ (g(n)*max(f_1 (n),f_2 (n)))*

si C est en $\Theta(f_1(n))$, I en $\Theta(f_2(n))$ et que la boucle while est exécutée Θ(g(n)) fois

Exemple : Recherche du maximum d'un tableau

```
public int chercheMax(int[] tab) {
   int maxCourant = tab[0];
   for (int i=1;i<tab.length;i++)
       if (maxCourant < tab[i])
              maxCourant = tab[i];
   return maxCourant;
}
```

taille des données : n = tab.length

opérations comptées : comparaison ou affectation ?

pire(n) = meilleur(n) = moyenne(n) = $\Theta(n)$

Exemple : Recherche d'un élément dans un tableau

```
public int recherche(int[] tab, int x) {
  for (int i=0;i<tab.length;i++)
      if (x==tab[i]) return i;
  return -1;
}
```

Taille des données : n = tab.length

Opération effectuée dans la boucle : une comparaison d'entiers en $\Theta(1)$

Complexité dans le meilleur des cas : $\Theta(1)$

x est l'élément d'indice 0; une comparaison et sortie de la boucle

Complexité dans le pire des cas : $\Theta(n)$

sortie de la boucle quand i = tab.length

correspond à quelles données ?

Complexité en moyenne

On suppose que *les éléments du tableau sont distincts* et que si x est dans le tableau, il peut être placé n'importe où, avec la même probabilité.

Complexité en moyenne quand x *est* dans le tableau :

- Les données qui contiennent x sont les tableaux qui contiennent x à l'indice 0, les tableaux qui contiennent x à l'indice 1, …, les tableaux qui contiennent x à l'indice i, … , les tableaux qui contiennent x à l'indice n-1.
- Quand x est dans le tableau, la probabilité pour que x soit à la place i est 1/n (1 chance sur n possibilités)
- Quand x est à la place i on fait i+1 comparaisons

$$C\_moyenne_{trouvé}(n) = \sum_{i=0}^{n-1} (1/n) * (i+1) = (n+1)/2$$

Complexité en moyenne quand x *n'est pas* dans le tableau

Quelle que soit la donnée, il y a n comparaisons.

$C\_moyenne_{pasTrouvé}(n) = n$

La complexité en moyenne est

$$moyenne(n) = p\ moyenne_{trouvé}(n) + (1-p)\ moyenne_{pasTrouvé}(n)$$

où p est la probabilité pour que x soit dans le tableau.

$C\_moyenne(n) = p * (n + 1)/2 + (1-p) * n$

Si x est dans le tableau ou x n'est pas dans le tableau, on retrouve les complexités précédentes

Si x a 50% de chance d'être dans le tableau, on fait en moyenne (n + 1)/4 + n / 2 comparaisons c'est à dire environ 3 comparaisons sur 4.

La recherche dichotomique est en $\log_2(n)$!

Bibliographie

T. Cormen, C. Leiserson, R. Rivest : « Introduction à l'algorithmique », Dunod.

S. Baase, A. V. Gelder : « Computers algorithms : Introduction to Design & Analysis », Addison Wesley

U. Mander : « Introduction to algorithms, A Creative Approach », Addison Wesley