
# Cours2
# Complexité d'un algorithme

Année 2016-2017 J-F. Kamp - IUT de Vannes

# PLAN

- Java :
  - JavaDoc
  - style
  - break
  - transtypage ou cast
- Notion de complexité
  - Problème versus algorithme
  - Comparaison d’algorithmes
  - Complexité théorique des algorithmes

# Java encore un peu

# Java : JavaDoc

Le commentaire dans un code Java est de deux types :

- Le commentaire qui explique un détail d'implémentation

```
// ces lignes sont mises en commentaire
// et ne seront pas compilées
```

```
/* ces lignes sont mises en commentaire
   et ne seront pas compilées non plus
   etc… etc… etc… etc… etc… etc…
   etc… etc… etc… etc… etc… etc… */
```

- Les commentaires de documentation embarquée JavaDoc. C'est ce commentaire là qui sera extrait du source Java pour être transformé en page HTML.

```
/** ces lignes sont destinées à compléter
    le code d'une documentation qui
    explique à quoi sert la classe, la
    méthode etc… */
```

# Java : JavaDoc

Le commentaire JavaDoc peut documenter une classe selon 3 niveaux :

- pour commenter le rôle général de la classe

```
/**
  * Cette classe effectue des opérations élémentaires..
  * etc.
  * etc.
  */

class SimplesTableau {
        …
}
```

- pour commenter le rôle d'un attribut (variable globale) de la classe

```
class SimplesTableau {

        /** La taille par défaut d'un tableau */
        final int TAILLE = 50;

        …
}
```

# Java : JavaDoc

- pour commenter le rôle d'une méthode de la classe

```
class SimplesTableau {
        ...

        /**
         * Affiche le contenu des nbElem cases d'un
         * tableau une par une.
         */
        void afficherTab ( int[] leTab, int nbElem ) {
                ...
        }
}
```

A l'intérieur d'un commentaire JavaDoc, toutes les balises HTML standards (<P>, <STRONG>, <B>, <I>, …) de mise en forme sont acceptées.

# Java : JavaDoc

Pour documenter le rôle d'une classe, javaDoc reconnaît un certain nombre de balises (tags) qui lui sont propres :

@author + « l'auteur de la classe »
@version + « numéro de version de la classe »
@since + « version du JDK »
@see + « See Also / référence à quelque chose »

En reconnaissant une balise, JavaDoc choisira le format approprié pour afficher les informations.

```
/**
  * Cette classe effectue des opérations élémentaires..
  *
  * @author J-F. Kamp – octobre 2016
  * @version 1.1.0
  */

class SimplesTableau {

        …
}
```

# Java : JavaDoc

Pour commenter une méthode, des balises JavaDoc spécifiques sont définies :

@param + « nomParamètre » + « description du paramètre de la méthode » (une balise/paramètre)

@return « description du type retourné » (au maximum un seul type retourné)

```
/**
 * Affiche le contenu des nbElem cases d'un tableau
 * une par une.
 *
 * @param leTab le tableau à afficher
 *
 * @param nbElem le nombre d'entiers que contient le
 * tableau
 *
 */

void afficherTab ( int[] leTab, int nbElem ) { ... }
```

# Java : style

Règle 1 : seule la classe et la constante commencent obligatoirement par une majuscule. Le reste (variables, méthodes) commencent toujours par une minuscule.

Règle 2 : tous les mots-clé du langage doivent être écrits en minuscules (class, int, for, while, new, …)

Règle 3 : le nom que l'on donne à un attribut, classe, constante ou méthode doit être clair, non ambigu. Il doit être parlant pour le programmeur.

```
z = x.meth10 ( y ); // NON !!

monCapital.ajoute ( interet ); // OUI !!
```

# Java : style

Règle 4 : Si plusieurs noms composent le nom principal, chaque nom est séparé par une majuscule.

```
String mot_courant; // NON !!
String motcourant; // NON !!

String motCourant; // OUI !!
```

Règle 5 : Choisir un verbe pour le nom des méthodes.

```
void ajouterInteret ( Monnaie somme ) {…}
boolean estVisible ( ) {…}
```

Règle 6 : Le nom d'une constante se compose uniquement de majuscules.

```
final int MIN_HAUTEUR = 4;
```

# Java : style

- L’unité d’indentation vaut 4 espaces.
- Chaque code qui commence à l’intérieur d’un bloc délimité par {…} doit être indenté d’une unité.
- L’accolade « { » doit suivre l’expression sur la même ligne, l’accolade « } » doit se trouver au même niveau que l’expression.

```
for ( i = 1; i < maxLoops; i++ ) {
        ...
        instructions;
        …
}
```

# Java : break

L'instruction break; permet la sortie forcée d'une boucle SANS tester la condition de sortie de cette boucle.

Exemple (trivial)

```
// sans break
boolean var = true;
int i = 0;

while ( var ) {
   i++;
   // boucle infinie !!
}

// avec break
…
i = 0;
while ( var ) {
   i++;
   if ( i == 10 ) break;
}
```

# Java : break

```
// avec break
…
i = 0;
while ( var ) {
        i++;
        if ( i == 10 ) break;
}
```

L'instruction break; s'écrit dans 99% des cas à l'intérieur d'une alternative. Cette alternative précise dans quelle(s) condition(s) exactement on force la sortie de la boucle englobante.

Intérêt : réduire la longueur des expressions booléennes de sortie de boucle.

```
// sans break mais 1 condition de sortie en +
…
i = 0;
while ( var && (i != 10) ) {
        i++;
}
```

# Java : break

Ne pas abuser du break :

- D’abord écrire la boucle avec l’expression booléenne habituelle (cf. TestAlgo). Si expression illisible ou si certains termes ne sont pas « évaluables » (i.e. débordement de tableau) alors envisager le break.

- Ne pas confondre break; et return; (forcer la sortie d’une méthode) : « return; » est interdit.

- Bien savoir de quelle boucle on force la sortie.

```
for ( i = 1; i <= 20; i++ ) {
    …
    j = 0;
    while ( var ) {
        j++;
        if ( j == 10 ) break;
    }
}
```

# Java : transtypage (cast)

Les types primitifs entiers et réels peuvent toujours s'affecter entre eux s'ils respectent le principe simple :

un nombre codé sur X bits acceptera toujours une valeur codée sur Y bits ssi Y <= X.

byte ⇨ short ⇨ int ⇨ float ⇨ double

```
int x = 153;
double y = 22.69;

y = x;
// OK car x codé sur 32 bits et y codé sur 64 bits
```

# Java : transtypage (cast)

Le transtypage pour les types primitifs entiers et réels, c'est FORCER l'affectation d'une variable dans une autre alors que la place n'est pas disponible (Y > X).

Exemple : la division entière

```
float var1 = 7 ;
int var2 = 2 ;
int res ;

res = ( var1 / var2 ) ; // 7/2 = 3.5 (pas un entier)
// Erreur de compilation

res = (int) ( var1 / var2 ) ; // compile
// Transtypage => conservation de la partie entière
// 7/2 = 3.5
// (int) 3.5 = 3
```

# Notion de complexité

# Problème et algorithmes

Un problème peut généralement être résolu de +sieurs façons.

Donc il existe +sieurs algorithmes différents pour résoudre un même problème.

Exemple : calcul du factoriel

$n! = 1 \times 2 \times 3 \times \ldots \times (n - 1) \times n$

En math : f(n) telle que

$f(0) = 1$

$f(n) = 1 \times 2 \times 3 \times \ldots \times (n - 1) \times n$

# Problème et algorithmes

Premier algorithme possible : la récursivité (pour la fin du module !).

```
int factorielle ( int n ) {

    int ret;

    if ( n == 0 ) {
        ret = 1;
    }

    else {

        ret = n * factorielle ( n – 1 );
    }

    return ret;
}
```

En fait c'est 1 boucle exécutée n fois…

# Problème et algorithmes

Deuxième algorithme possible : l'algorithme itératif (cf. TestAlgo).

```
int factorielle ( int n ) {

    int ret, i:

    ret = 1;

    for ( i = 1; i <= n; i++ ) {

        ret = ret * i;
    }

    return ret;
}
```

C'est aussi 1 boucle exécutée n fois…

Quel est l'algorithme le + performant : récursif ou itératif ?

# Comparaison des algos

Sur quels critères comparer les algorithmes ?

- L’espace mémoire occupé par le fichier contenant le code ?
- L’espace mémoire occupé par le fichier compilé ?
- La quantité de mémoire vive occupée lors de l’exécution ?
- Le nombre d’octets transférés sur le réseau ?
- **Le temps de calcul.**

# Critère de comparaison
# Le temps de calcul

Première suggestion : chronométrer le temps de calcul.

Mais ça va dépendre :

- De la machine (brouette ou Ferrari)
- Du nombre de données traitées
- De la fréquence d’horloge
- Temps d’accès aux données
- …

# Critère de comparaison
# Le temps de calcul

Le temps de calcul va dépendre de la complexité (taille) du travail à accomplir.

Exemple : pour un algorithme de tri, la complexité va dépendre du nombre N de valeurs à trier.

Si N est la taille du problème, on voudrait exprimer le temps de calcul en fonction de N.

# Le temps de calcul empirique

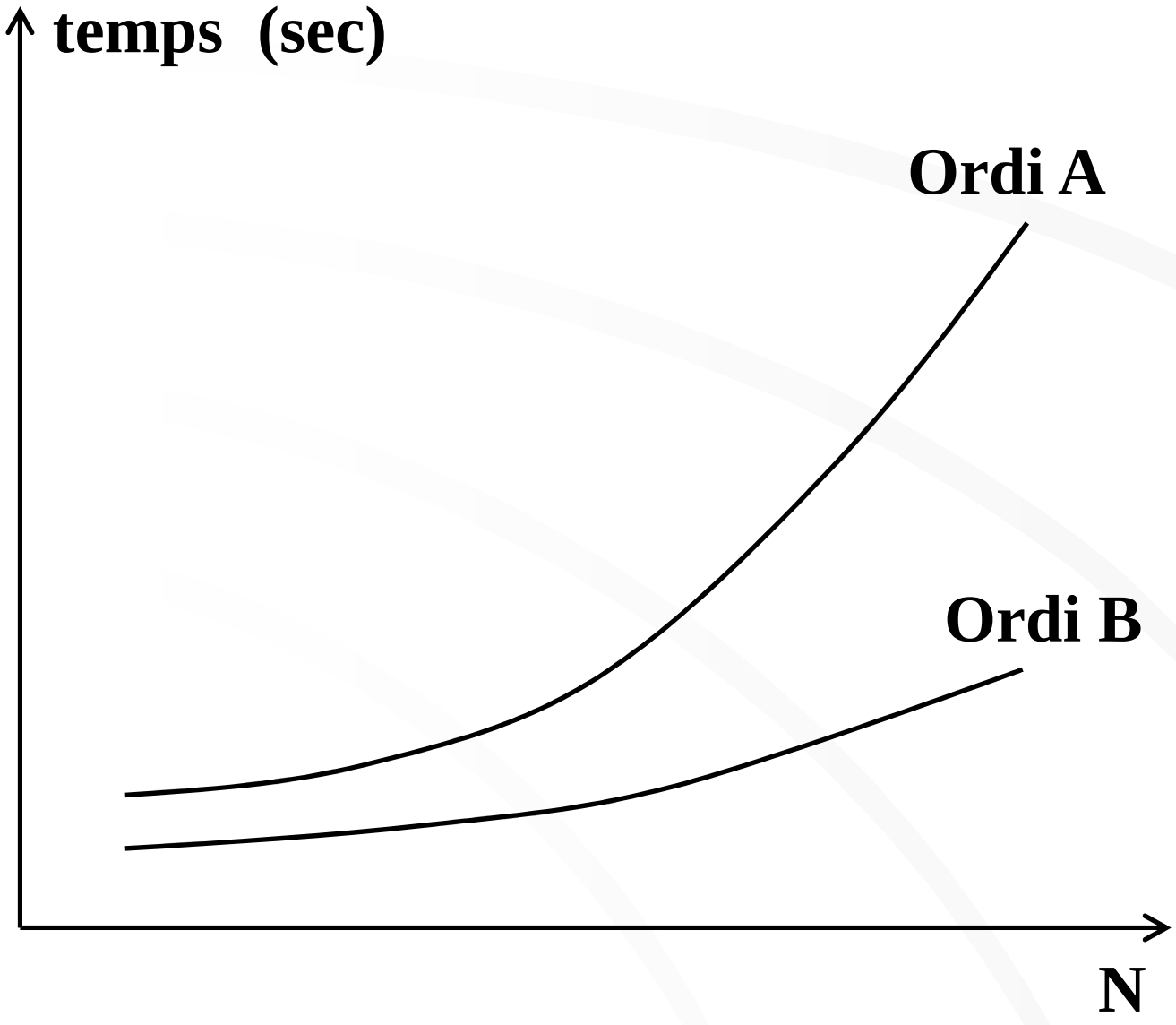
Exemple

Mesures du temps de calcul (en secondes) pour un algo. de tri sur 2 machines A et B.

| N | Ordi. A | Ordi. B |
|---|---|---|
| 125 | 12,5 | 2,8 |
| 250 | 49,3 | 11 |
| 500 | 195,8 | 43,4 |
| 1000 | 780,3 | 172,9 |
| 2000 | 3114,9 | 690,5 |

# Le temps de calcul empirique

Ajustement de courbes sur les données.



# Le temps de calcul empirique

- Mise en équation

Ordi A

$t(N) = 0{,}00078\ N^2 + 0{,}003\ N + 0{,}001$

Ordi B

$t(N) = 0{,}00017\ N^2 + 0{,}0004\ N + 0{,}01$

- Observations

Les courbes sont en $N^2$ pour les 2 ordis.

Si $N > 500$, $t(N)$ ne dépend que du terme en $N^2$.

# Le temps de calcul empirique

La variation du temps de calcul en fonction de la taille (ici N = nbre de valeurs à trier) est appelée complexité de l'algorithme.

Dans cet exemple, il s'agit d'une complexité empirique car fondée sur des mesures (ici du temps).

# Complexité théorique

Il est possible de déterminer les équations précédentes de manière théorique pour une bonne majorité des problèmes.

Idée : compter les opérations élémentaires exécutées par l'algorithme en fonction de N (taille du problème).

Hypothèse (forte) : **une opération élémentaire s'exécute en un temps élémentaire** qui est 1 coup d'horloge (T = 1/F, F = fréquence en Hz).

# Complexité théorique

Exemple de calcul théorique pour la factorielle.

Hypothèses (fixons les règles) :

- Une déclaration = 1 opération élémentaire
- Une affectation = 1 opération élém.
- Une addition ou multiplication = 1 opération élém.
- Une incrémentation (i++) ou décrémentation (i--) = 2 opérations élém.
- Un test de comparaison (<=, <, >=, >, !=, ==) = 1 opération élém.

# Complexité théorique

```
int factorielle ( int n ) {

    int ret, i:                     // 2 op

    ret = 1;                        // 1 op
    i = 1;                          // 1 op

    while ( i <= n ) {              // 1 op

        ret = ret * i;              // 2 op
        i++;                        // 2 op
    }

    return ret;
}
```

- Avant la boucle : 2 + 1 + 1 = 4 op
- Boucle effectuée (n – 1 + 1) = n fois
- Corps de la boucle : 2 + 2 = 4 op
- Comparaison : 1 op
- Comparaison effectuée (n + 1) fois

# Complexité théorique

- Avant la boucle : 2 + 1 + 1 = 4 op
- Boucle effectuée (n – 1 + 1) = n fois
- Corps de la boucle : 2 + 2 = 4 op
- Comparaison : 1 op
- Comparaison effectuée (n + 1) fois

Nbre opérations élémentaires = f ( n )

= 4 + ( n x 4 ) + ( n + 1 ) x 1

= 5 + 5n

Temps de calcul : t ( n ) = f ( n ) x T

où T = 1/F (F = fréquence de l'horloge Hz)

Conclusion : le temps de calcul théorique de la factorielle est linéaire en « n ».

# Une meilleure intuition

Supposons une horloge à 1MHz

=> $T = 10^{-6}$ secondes

- Premier algo. de tri (tri1)

  $t(n) = n^2 \times T$

- Second algo. de tri (tri2)

  $t(n) = n \log_2 (n) \times T$

Soit $n = 10^5$ (100.000)

- tri1 : $t(n) = 10^{10} \times 10^{-6} = 10^4$ secondes = 2,7 heures
- tri2 : $t(n) = 16,6 \times 10^5 \times 10^{-6} = 1,66$ secondes