



Ecole Supérieure d'Economie Numérique

## Cours Complexité algorithmique (MBDS)

### cours 1: Introduction à la complexité des algorithmes

Dr. Dhouha Maatar Razgallah  
2017/2018

### *Outline*

- ❑ *Introduction*
- ❑ *Théorie de la complexité*
- ❑ *Complexité algorithmique*
- ❑ *Application de calcul de complexité: produit de matrices*

| Introduction   | Qu'est ce que l'algorithmique? |
|--|--------------------------------|
| <ul style="list-style-type: none"> <li>❑ <i>L'algorithmique est l'étude des algorithmes.</i></li> <li>❑ <i>Un algorithme est une suite d'instructions qui décrit comment résoudre un problème particulier en un temps fini.</i></li> </ul> |                                |
| <p><i>Avez-vous déjà ouvert un livre de recettes de cuisine ?<br/>ou déjà indiqué un chemin à un touriste?</i></p> <p><i>Si oui, sans le savoir, vous avez déjà exécuté des algorithmes.</i></p>   |                                |
| <p>Si l'algorithme est juste, le résultat est le résultat voulu, et le touriste se retrouve là où il voulait aller. Si l'algorithme est faux, le résultat est aléatoire, et le touriste est perdu.</p>                                     |                                |
| <p>➤ Pour fonctionner, <i>un algorithme doit contenir uniquement des instructions compréhensibles par celui qui devra l'exécuter.</i></p>  |                                |
| 3  |                                |

| Introduction   | Algorithme et programme |
|--|-------------------------|
| <p>❑ Un <b>programme (code)</b> : la réalisation (l'implémentation) d'un <b>algorithme</b> au moyen d'un langage donné (sur une architecture donnée). Il s'agit de la mise en œuvre du principe.</p>   |                         |
| <pre> graph LR     A([Informations du problème]) --&gt; B([machine])     B --&gt; C([Résultats mis en forme])     subgraph ShadedBox [ ]         D([Données structurées])         E([Obtention de résultats])         F[Traitement]     end     D -.-&gt; B     B -.-&gt; E     F -.-&gt; E           </pre> |                         |
| 4  |                         |

**Exemple de tâche:** Décider si un tableau L est trié en ordre croissant.

- **Raisonnement:** Un tableau L est trié si tous ses éléments sont dans l'ordre croissant.  $L \text{ trié} \iff \forall i, 0 < i < |L| \quad L[i] < L[i+1]$
- **Algorithme:** Une fonction vérifiant cette propriété, supposera donc le tableau L, de taille n, trié au départ et cherchera une contradiction.

```

Fonction trie (L: tab, n: entier) : boolean
Variables   i, n: entier; Ok: boolean
Début
Ok ← vrai
pour i de 1 à n faire
    si (L[i] > L[i+1]) alors
        Ok ← Faux
    Finsi
Finpour
Retourner OK
Fin trie

boolean trie (tab L, int n)
{
    Ok = true;
    For (i = 0; i < n ; i++)
    {
        if (L[i] > L[i+1])
            Ok = Faux;
    }
    return Ok;
}

```

➤ **Code:**

5

- ❑ La question la plus fréquente que se pose à chaque programmeur est la suivante:

**Comment choisir parmi les différentes approches pour résoudre un problème?**

Exemples: Trier un tableau: algorithme de tri par insertion ou de tri rapide? ..., etc

- ❑ On attend d'un algorithme qu'il résolve correctement et de manière efficace le problème à résoudre, quelles que soient les données à traiter.

### 1. La correction: résout-il bien le problème donné?

Trouver une méthode de résolution (exacte ou approchée) du problème.

### 2. L'efficacité: en combien de temps et avec quelles ressources?

Il est souhaitable que nos solutions ne soient pas lentes, ne prennent pas de l'espace mémoire considérable.

➡ Savoir résoudre un problème est une chose, le résoudre efficacement en est une autre!

6

| Introduction  | Efficacité |
|---|------------|
| <p>L'efficacité d'un algorithme peut être évaluée par:</p> <ul style="list-style-type: none"> <li>❑ <b>Rapidité</b> (en terme de temps d'exécution)</li> <li>❑ <b>Consommation de ressources</b> (espace de stockage, mémoire utilisée)</li> </ul> <p>➤ <i>La théorie de la complexité</i> étudie l'efficacité des algorithmes.</p> <p>✓ On s'intéresse dans ce cours, essentiellement, à l'efficacité en terme de temps d'exécution.</p> |            |
| 7   |            |

| Théorie de la complexité  | Définition et Objectifs |
|---|-------------------------|
| <ul style="list-style-type: none"> <li>❑ <i>La théorie de la complexité</i> est une branche de l'informatique théorique, elle cherche à calculer, formellement, <b>la complexité algorithmique</b> nécessaire pour résoudre un problème <math>P</math> au moyen de l'exécution d'un algorithme <math>A</math>.</li> <li>❑ <i>La théorie de la complexité</i> vise à répondre aux besoins d'efficacité des algorithmes (programmes):</li> </ul> <p>Elle permet:</p> <ul style="list-style-type: none"> <li>• Classer les problèmes selon leur difficulté.</li> <li>• Classer les algorithmes selon leur efficacité.</li> <li>• Comparer les algorithmes résolvant un problème donné afin de faire un choix sans devoir les implémenter.</li> </ul> |                         |
| 8   |                         |

- ❑ *On ne mesure pas la durée en heure, minute, seconde... cela impliquerait d'implémenter les algorithmes qu'on veut comparer.*
- ❑ *Ces mesures ne seraient pas pertinentes car le même algorithme sera plus rapide sur **une machine** plus **puissante**.*
- ❑ *Au lieu de ça, on utilise des unités de temps abstraite proportionnelles au nombre d'opérations effectuées.*
- ❑ *Au besoin, on pourra alors adapter ces quantités en fonction de la machine sur laquelle l'algorithme s'exécute.*

**Règles:**

- Chaque instruction basique consomme une unité de temps (affectation d'une variable, lecture, écriture, comparaison,...).
- Chaque itération d'**une boucle** rajoute le nombre d'unités de temps consommés dans le corps de cette boucle.
- Chaque appel de **fonction** rajoute le nombre d'unités de temps consommées dans cette fonction.
- Pour avoir le nombre d'opération effectuées par l'algorithme, on additionne le tout.

*Exemple: Temps d'exécution de la fonction factorielle*

L'algorithme suivant calcule :  $n! = n*(n-1)*(n-2)*\dots*1$  avec  $0! = 1$

```
int factorielle(n)
fact = 1;
i = 2;
while (i <= n)
    fact = fact * i;
    i = i + 1;
return fact
```

11

*Exemple: Temps d'exécution de la fonction factorielle*

L'algorithme suivant calcule :  $n! = n*(n-1)*(n-2)*\dots*1$  avec  $0! = 1$

```
int factorielle(n)
fact = 1;          initialisation(1)
i = 2;             affectation (1)
while (i <= n)      comparaison (1)
    fact = fact * i;  aff. + multip.(2)
    i = i + 1;       affectation + addition (2)
return fact         retour (1)
```

**Temps de calcul** =  $1+1+(2+2+1)*(n-1)+1 = 5n-2$  opérations

12

| Théorie de la complexité   | Calcul du temps d'exécution |
|--|-----------------------------|
| <p>❖ <b>Problèmes</b></p> <p><b>Unités de temps abstraites:</b></p> <ul style="list-style-type: none"> <li>▪ Dépend des données.</li> <li>▪ Dépend de la nature des données: on ne sait pas toujours combien de fois exactement on va exécuter une boucle.</li> <li>▪ De même lors d'un branchement conditionnel, le nombre de comparaisons effectués n'est pas toujours le même.</li> </ul> <p><b>Temps exacte:</b></p> <ul style="list-style-type: none"> <li>▪ Dépend de la puissance de la machine.</li> <li>▪ Dépend de la nature des données (variables): si on change les données, le temps change.</li> </ul> <p>❖ <b>Solution</b></p> <p><b>Calcul de la complexité algorithmique</b></p> |                             |
| 13   |                             |

| Complexité algorithmique  | Définition |
|---|------------|
| <p>❑ La complexité d'un algorithme est la mesure du nombre d'opérations fondamentales qu'il effectue sur un jeu de données. Elle est exprimée comme une fonction de la taille du jeu de données.</p> <p>❑ Il existe trois types de complexité:</p> <ul style="list-style-type: none"> <li>▪ <b>Complexité au meilleur</b><br/>C'est le plus petit nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille n.</li> <li>▪ <b>Complexité au pire</b><br/>C'est le plus grand nombre d'opérations qu'aura à exécuter l'algorithme sur un jeu de données de taille n.</li> <li>▪ <b>Complexité en moyenne</b><br/>C'est la moyenne des complexités de l'algorithme sur des jeux de données de taille n.</li> </ul> |            |
| 14  |            |

| Complexité algorithmique  | Définition générale |
|---|---------------------|
| <p>❑ De façon générale: La complexité d'un algorithme est une mesure de sa performance <b>asymptotique</b> dans le <b>pire des cas</b>.</p> <ul style="list-style-type: none"> <li>▪ <b>asymptotique</b> ?<br/>✓ on s'intéresse à des données très grandes parce que les petites valeurs ne sont pas assez informatives.</li> <li>▪ <b>Pire des cas</b> ?<br/>✓ on s'intéresse à la performance de l'algorithme dans les situations où le problème prend le plus de temps à résoudre parce qu'on veut être sûr que l'algorithme ne prendra jamais plus de temps que ce qu'on a estimé.</li> </ul> <p>❑ Un algorithme est dit « <b>optimal</b> » si sa complexité est la complexité minimale parmi les algorithmes de sa classe.</p> |                     |

15

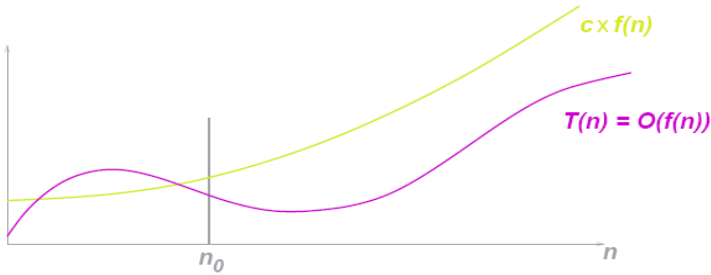
| Complexité algorithmique   | O-notation (Notation de Landau) |
|--|---------------------------------|
| <p>❖ Quand on calcule la complexité d'un algorithme, on ne calcule généralement pas sa complexité exacte, mais son ordre de grandeur.</p> <p>❖ C'est une approximation du temps de calcul de l'algorithme.</p> <div style="border: 1px solid black; padding: 10px; margin: 10px 0;"> <p>Par exemple, si on fait <math>n^2 + 2n - 5</math> opérations, on retiendra juste que l'ordre de grandeur est <math>n^2</math>. On utilisera donc la notation classique sur les ordres de grandeur.</p> </div> <p>❖ Pour ce faire, on a recours à la <b>notation asymptotique <math>O(.)</math></b> ou <b>Notation de Landau</b>.</p> |                                 |

16

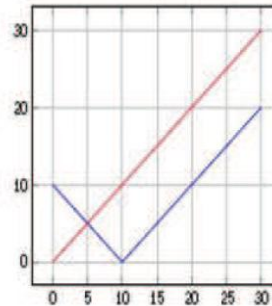
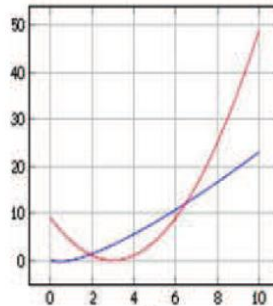
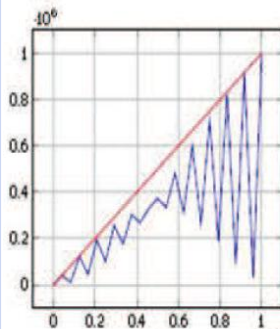


| Complexité algorithmique  | O-notation (Notation de Landau) |
|---|---------------------------------|
| <p>Cette notation exprime <b>la limite supérieure</b> d'une fonction dans un facteur constant.</p> <p>✓ Soit <math>n</math> la taille des données à traiter; on dit qu'une fonction <math>f(n)</math> est en <math>O(g(n))</math> si :</p> <div style="border: 1px solid red; padding: 5px; margin: 10px 0;"> <math display="block">\exists n_0 \in \mathbb{N}, \quad \exists c \in \mathbb{R}, \quad \forall n \geq n_0 :  f(n)  \leq c g(n) </math> </div> <p>✓ <math>f(n)</math> est en <math>O(g(n))</math> s'il existe un seuil à partir duquel la fonction <math>f(\cdot)</math> est toujours dominée par <math>g(\cdot)</math>, à une constante multiplicative fixée près.</p> <div style="border: 1px dashed purple; padding: 5px; margin: 10px 0;"> <p><b>Exemple:</b> <math>T(n) = O(n^2)</math> veut dire qu'il existe une constante <math>c &gt; 0</math> et une constante <math>n_0 &gt; 0</math> tel que pour tout <math>n &gt; n_0</math> <math>T(n) \leq c n^2</math></p> </div> <p><b>Utilité:</b> Le temps d'exécution est borné</p> <p><b>Signification:</b> Pour toutes les grandes entrées (i.e., <math>N \geq n_0</math>), on est assuré que l'algorithme ne prend pas plus de <math>c \cdot g(n)</math> étapes.</p> <p>➡ Borne supérieure.</p> |                                 |

17

| Complexité algorithmique   | O-notation (Notation de Landau) |
|--|---------------------------------|
| <div style="text-align: center;">  </div> <p>Caractérise le comportement asymptotique (i.e. quand <math>n \rightarrow \infty</math>).</p> <p><math>T(n) = O(f(n))</math> si <math>\exists c \exists n_0</math> tels que <math>\forall n &gt; n_0, T(n) \leq c \times f(n)</math></p> |                                 |

Exemples: quelques cas où  $f(n) = O(g(n))$



19

*Exemple 1: Prouver que  $f_1(n) = 5n+37$  est en  $O(n)$*

✓ **But:** trouver une constante  $c \in \mathbb{R}$  et un seuil  $n_0 \in \mathbb{N}$  à partir duquel  $|f_1(n_0)| \leq c|n_0|$

✓ On remarque que  $5n+37 \leq 6n$  si  $n \geq 37$

✓ On déduit donc que  $c=6$  fonctionne à partir de  $n_0=37$

**Remarque:** on ne demande pas d'optimiser (le plus petit  $c$  ou  $n_0$  qui fonctionnent) juste il faut donner des valeurs qui fonctionnent.

20

| Complexité algorithmique  | Exemples de calcul de $O(\cdot)$ |
|---|----------------------------------|
| <p><i>Exemple 2: Prouver que <math>f_2(n) = 6n^2 + 2n - 8</math> est en <math>O(n^2)</math></i></p> <p>✓ <b>But:</b> trouver une constante <math>c \in \mathbb{R}</math> et un seuil <math>n_0 \in \mathbb{N}</math> à partir duquel</p> $ 6n^2 + 2n - 8  \leq c n^2 $ <p>✓ Chercher d'abord <math>c</math>, <math>c=6</math> ne peut pas marcher, donc <math>c=7</math></p> <p>✓ On doit donc trouver un seuil <math>n_0</math> à partir duquel :</p> $ 6n^2 + 2n - 8  \leq 7 n^2  \quad \forall n \geq n_0$ <p>✓ <math>c=7</math> et <math>n_0=1</math></p> |                                  |

21

| Complexité algorithmique   | Exemples de calcul de $O(\cdot)$ |
|--|----------------------------------|
| <p><i>Exemple 3: Calculer la complexité de <math>T(n) = c_1n^2 + c_2n</math></i></p> <p>✓ On remarque que <math>c_1n^2 + c_2n \leq c_1n^2 + c_2n^2 = (c_1 + c_2)n^2</math><br/>pour tout <math>n \geq 1</math></p> <p>✓ <math>T(n) \leq cn^2</math> où <math>c = c_1 + c_2</math> et <math>n_0 = 1</math></p> <p>✓ Donc <math>T(n)</math> est en <math>O(n^2)</math></p> |                                  |

22

| Complexité algorithmique   | Exemples de calcul de $O(\cdot)$ |
|--|----------------------------------|
| <p><i>Exemple 4: Calculer la complexité de l'initialisation d'un tableau:</i><br/> <code>for (int i=0; i&lt;n; i++) Tab[i]=0;</code></p>   |                                  |
| <p>✓ On remarque qu'il y a <math>n</math> itérations; chaque itération nécessite un temps d'exécution <math>\leq c</math> où <math>c</math> est une constante (accès au tableau + une affectation)</p> <p>✓ le temps est donc <math>T(n) \leq c n</math></p> <p>✓ Donc <math>T(n)</math> est en <b><math>O(n)</math></b></p> |                                  |

23

| Complexité algorithmique  | Règles de calcul |
|---|------------------|
| <p>On calcule le temps d'exécution, mais on effectue les simplifications suivantes:</p> <ul style="list-style-type: none"> <li>➤ On oublie les constantes multiplicatives (elles valent 1)</li> <li>➤ On annule les constantes additives</li> <li>➤ On ne retient que les termes dominants</li> </ul>   |                  |
| <p><i>Exemple (simplifications) soit <math>g(n) = 4n^3 - 5n^2 + 2n + 3</math> :</i></p> <ul style="list-style-type: none"> <li>• On remplace les constantes multiplicatives par 1: <math>1n^3 - 1n^2 + 1n + 3</math></li> <li>• On annule les constantes additives : <math>n^3 - n^2 + n + 0</math></li> <li>• On garde le terme de plus haut degré: <math>n^3 + 0</math></li> <li>• On a donc la complexité de <math>g(n) = O(n^3)</math></li> </ul> |                  |

24

De façon générale, les règles de la notation en O sont les suivantes:

➤ Les termes constants :

$$O(c) = O(1)$$

➤ Les constantes multiplicatives sont omises:

$$O(cT) = c O(T) = O(T)$$

➤ L'addition est réalisée en prenant le maximum:

$$O(T1) + O(T2) = O(T1 + T2) = \max(O(T1), O(T2))$$

➤ La multiplication reste inchangée:

$$O(T1) O(T2) = O(T1 * T2)$$

25

❑ **Exemple:**

Supposons que le temps d'exécution d'un algorithme est décrit par la fonction:

$T(n) = 3n^2 + 10n + 10$ , calculer  $O(T(n))$ ?

$$\begin{aligned} O(T(n)) &= O(3n^2 + 10n + 10) \\ &= O(\max(3n^2, 10n, 10)) \\ &= O(3n^2) \\ &= O(n^2) \end{aligned}$$

❑ **Remarque:**

Pour  $n=10$ , nous avons:

Temps d'exécution de  $3n^2$  :  $3(10)^2 / (3(10)^2 + 10(10) + 10) = 73,2\%$

Temps d'exécution de  $10n$  :  $10(10) / (3(10)^2 + 10(10) + 10) = 24,4\%$

Temps d'exécution de  $10$  :  $10 / (3(10)^2 + 10(10) + 10) = 2,4\%$

Le poids de  $3n^2$  devient encore plus grand pour  $n=100$ , soit 96,7%, on peut donc négliger les quantités  $10n$  et  $10$ .

✓ Ceci explique les règles de notation O.

26

| Complexité algorithmique  | Calcul de complexité |   |  |   |  |             |          |  |   |   |                               |
|---|----------------------|---|--|---|--|-------------|----------|--|---|---|-------------------------------|
| <p>➤ <b>Cas d'une instruction simple</b> : Les instructions de base (lecture, écriture, affectation ...) prennent un temps constant, noté <math>O(1)</math>.</p> <p>➤ <b>Cas d'une suite d'instructions</b>: le temps d'exécution d'une séquence est déterminé par la règle de la somme.</p> <div style="margin-left: 40px;"> <table style="display: inline-table; vertical-align: middle;"> <tr> <td>Traitement1</td> <td><math>T_1(n)</math></td> <td rowspan="2" style="font-size: 3em; vertical-align: middle;">}</td> <td rowspan="2" style="vertical-align: middle;"> <math>T(n) = T_1(n) + T_2(n)</math><br/> <math>O(T) = O(T_1 + T_2) = \max(O(T_1); O(T_2))</math> </td> </tr> <tr> <td>Traitement2</td> <td><math>T_2(n)</math></td> </tr> </table> </div> <p>➤ <b>Cas d'une branche conditionnelle</b>: le temps d'exécution est déterminé aussi par la règle de la somme.</p> <div style="margin-left: 40px;"> <table style="display: inline-table; vertical-align: middle;"> <tr> <td style="vertical-align: top;"> <pre> if &lt;condition&gt;:     # instructions (1) else:     # instructions (2) </pre> </td> <td style="font-size: 3em; vertical-align: middle;">}</td> <td style="vertical-align: middle;"> <math>O(g(n))</math><br/> <math>O(f_1(n))</math><br/> <math>O(f_2(n))</math> </td> <td style="vertical-align: middle;"> <math>= O(g(n) + f_1(n) + f_2(n))</math> </td> </tr> </table> </div> |                      | Traitement1                             | $T_1(n)$   | } | $T(n) = T_1(n) + T_2(n)$<br>$O(T) = O(T_1 + T_2) = \max(O(T_1); O(T_2))$ | Traitement2 | $T_2(n)$ | <pre> if &lt;condition&gt;:     # instructions (1) else:     # instructions (2) </pre> | } | $O(g(n))$<br>$O(f_1(n))$<br>$O(f_2(n))$ | $= O(g(n) + f_1(n) + f_2(n))$ |
| Traitement1   | $T_1(n)$             | }                                       | $T(n) = T_1(n) + T_2(n)$<br>$O(T) = O(T_1 + T_2) = \max(O(T_1); O(T_2))$ |   |  |             |          |  |   |   |                               |
| Traitement2   | $T_2(n)$             |   |  |   |  |             |          |  |   |   |                               |
| <pre> if &lt;condition&gt;:     # instructions (1) else:     # instructions (2) </pre>  | }                    | $O(g(n))$<br>$O(f_1(n))$<br>$O(f_2(n))$ | $= O(g(n) + f_1(n) + f_2(n))$  |   |  |             |          |  |   |   |                               |

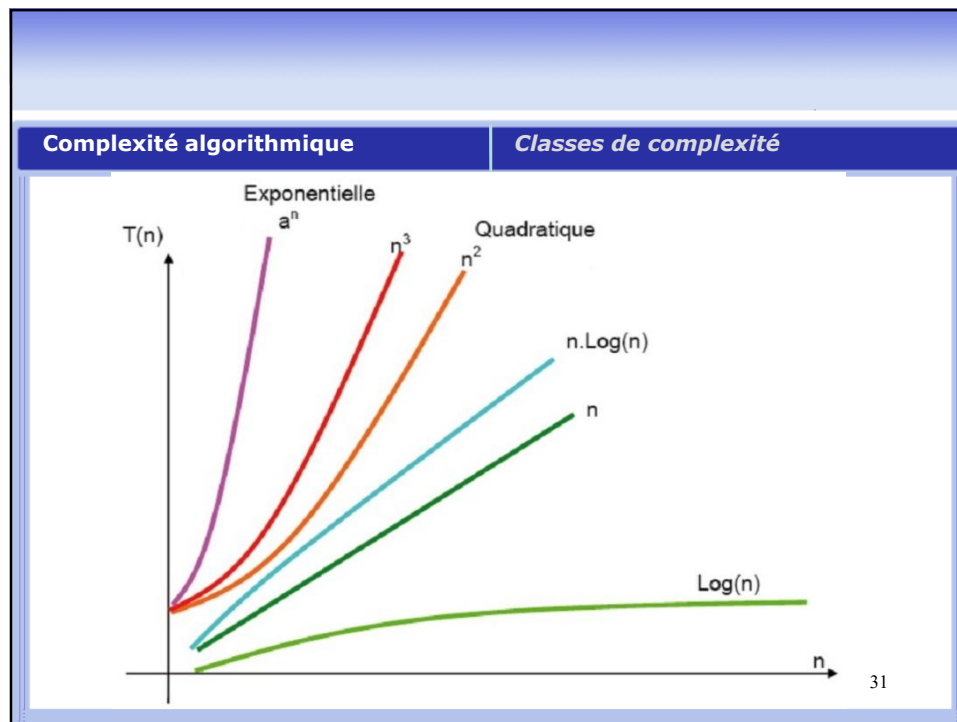
27

| Complexité algorithmique  | Calcul de complexité |                          |   |                          |
|---|----------------------|--------------------------|---|--------------------------|
| <p>➤ <b>Cas d'une boucle</b>: On multiplie la complexité du corps de la boucle par le nombre d'itérations.</p> <p>Exemple pour la boucle while (tant que), la complexité se calcule comme suit:</p> <div style="margin-left: 40px;"> <pre> # en supposant qu'on a m itérations while &lt;condition&gt;:     # instructions </pre> <table style="display: inline-table; vertical-align: middle;"> <tr> <td style="vertical-align: top;"> <math>O(g(n))</math><br/> <math>O(f(n))</math> </td> <td style="font-size: 3em; vertical-align: middle;">}</td> <td style="vertical-align: middle;"> <math>= O(m * (g(n) + f(n)))</math> </td> </tr> </table> </div> <p>❑ <b>Pour calculer la complexité d'un algorithme:</b></p> <ul style="list-style-type: none"> <li>• On calcule la complexité de chaque partie de l'algorithme.</li> <li>• On combine ces complexités conformément aux règles déjà vues.</li> <li>• On effectue sur le résultat les simplifications possibles déjà vues.</li> </ul> |                      | $O(g(n))$<br>$O(f(n))$   | } | $= O(m * (g(n) + f(n)))$ |
| $O(g(n))$<br>$O(f(n))$  | }                    | $= O(m * (g(n) + f(n)))$ |   |                          |

28

| Complexité algorithmique  | Calcul de complexité |
|---|----------------------|
| <p><b>Exemple: calcul de la complexité de la fonction factorielle</b></p> <div style="border: 1px dashed black; padding: 10px; margin: 10px 0;"> <pre> int factorielle(n) fact = 1;           O(1) i = 2;             O(1) while (i &lt;= n)      O(1)     fact = fact * i; O(1)     i = i + 1;      O(1) return fact         O(1) </pre> </div> <p style="margin-left: 40px;"> <math>O(1) + O(1) + O[(n-1) * (1+1+1)] + O(1)</math><br/> <math>= O(1) + O(1) + O(n) + O(1)</math><br/> <math>= \mathbf{O(n)}</math> </p> |                      |
| 29  |                      |

| Complexité algorithmique   | Classes de complexité |        |        |          |             |               |        |          |               |                  |          |             |          |         |          |             |         |             |    |
|--|-----------------------|--------|--------|----------|-------------|---------------|--------|----------|---------------|------------------|----------|-------------|----------|---------|----------|-------------|---------|-------------|----|
| <p>➤ On peut ranger les fonctions équivalentes dans la même classe.</p> <p>➤ Deux algorithmes de la même classe sont considérés de même complexité.</p> <p>➤ Les classes de complexité les plus fréquentes (par ordre croissant selon <math>O(.)</math>)</p>   |                       |        |        |          |             |               |        |          |               |                  |          |             |          |         |          |             |         |             |    |
| <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="padding: 5px;">Complexité</th> <th style="padding: 5px;">Classe</th> </tr> </thead> <tbody> <tr><td style="padding: 5px;"><math>O(1)</math></td><td style="padding: 5px;">constant</td></tr> <tr><td style="padding: 5px;"><math>O(\log n)</math></td><td style="padding: 5px;">logarithmique</td></tr> <tr><td style="padding: 5px;"><math>O(n)</math></td><td style="padding: 5px;">linéaire</td></tr> <tr><td style="padding: 5px;"><math>O(n \log n)</math></td><td style="padding: 5px;">sous-quadratique</td></tr> <tr><td style="padding: 5px;"><math>O(n^2)</math></td><td style="padding: 5px;">quadratique</td></tr> <tr><td style="padding: 5px;"><math>O(n^3)</math></td><td style="padding: 5px;">cubique</td></tr> <tr><td style="padding: 5px;"><math>O(2^n)</math></td><td style="padding: 5px;">exponentiel</td></tr> <tr><td style="padding: 5px;"><math>O(n!)</math></td><td style="padding: 5px;">factorielle</td></tr> </tbody> </table> | Complexité            | Classe | $O(1)$ | constant | $O(\log n)$ | logarithmique | $O(n)$ | linéaire | $O(n \log n)$ | sous-quadratique | $O(n^2)$ | quadratique | $O(n^3)$ | cubique | $O(2^n)$ | exponentiel | $O(n!)$ | factorielle | 30 |
| Complexité   | Classe                |        |        |          |             |               |        |          |               |                  |          |             |          |         |          |             |         |             |    |
| $O(1)$   | constant              |        |        |          |             |               |        |          |               |                  |          |             |          |         |          |             |         |             |    |
| $O(\log n)$  | logarithmique         |        |        |          |             |               |        |          |               |                  |          |             |          |         |          |             |         |             |    |
| $O(n)$   | linéaire              |        |        |          |             |               |        |          |               |                  |          |             |          |         |          |             |         |             |    |
| $O(n \log n)$  | sous-quadratique      |        |        |          |             |               |        |          |               |                  |          |             |          |         |          |             |         |             |    |
| $O(n^2)$   | quadratique           |        |        |          |             |               |        |          |               |                  |          |             |          |         |          |             |         |             |    |
| $O(n^3)$   | cubique               |        |        |          |             |               |        |          |               |                  |          |             |          |         |          |             |         |             |    |
| $O(2^n)$   | exponentiel           |        |        |          |             |               |        |          |               |                  |          |             |          |         |          |             |         |             |    |
| $O(n!)$  | factorielle           |        |        |          |             |               |        |          |               |                  |          |             |          |         |          |             |         |             |    |



Complexité algorithmique

Classes de complexité

Exemples de temps d'exécution en fonction de la taille de la donnée et de la complexité de l'algorithme, si on suppose qu'une instruction est de l'ordre de la  $\mu s$ ;

| T.\C.  | $\log n$  | $n$         | $n \log n$ | $n^2$      | $2^n$             |
|--------|-----------|-------------|------------|------------|-------------------|
| 10     | $3\mu s$  | $10\mu s$   | $30\mu s$  | $100\mu s$ | $1000\mu s$       |
| 100    | $7\mu s$  | $100\mu s$  | $700\mu s$ | $1/100s$   | $10^{14}$ siècles |
| 1000   | $10\mu s$ | $1000\mu s$ | $1/100s$   | $1s$       | astronomique      |
| 10000  | $13\mu s$ | $1/100s$    | $1/7s$     | $1,7mn$    | astronomique      |
| 100000 | $17\mu s$ | $1/10s$     | $2s$       | $2,8h$     | astronomique      |

32



Ecrire un algorithme qui permet de calculer le produit C de deux matrices carré A et B de dimension (n×n) ensuite de déterminer sa complexité?

**Principe:**

➤ L'équation  $C=A * B$  s'écrit:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & g \\ f & h \end{pmatrix}$$

**C                      A                      B**

➤ En développant cette équation, nous obtenons :

$$r = ae + bf, \quad s = ag + bh, \quad t = ce + df \quad \text{et} \quad u = cg + dh.$$

33

Ecrire un algorithme qui permet de calculer le produit C de deux matrices carré A et B de dimension (n×n) ensuite de déterminer sa complexité?

**Algorithme:**

**Procédure MULTI-MAT**(var C:mat; A, B: mat, n:entier)

**Var** i, j, k: entier

**Début**

Pour i de 1 à n faire

    Pour j de 1 à n faire

        C[i,j] ← 0

        Pour k de 1 à n faire

            C[i,j] ← C[i,j] + A[i,k] \* B[k,j]

    Finpour finpour finpour

**Fin**

34

Ecrire un algorithme qui permet de calculer le produit C de deux matrices carré A et B de dimension (n×n) ensuite de déterminer sa complexité?

**Complexité:**

```

Procédure MULTI-MAT(var C:mat; A, B: mat, n:entier)
Var i, j, k: entier
Début
  Pour i de 1 à n faire
    Pour j de 1 à n faire
      C[i,j] ← 0
      Pour k de 1 à n faire
        C[i,j] ← C[i,j] + A[i,k] * B[k,j]
      Finpour
    Finpour
  Finpour
Fin
  
```

Nbre d'iter: n  
 $\Rightarrow n * O(n) = O(n^2)$

Nbre d'iter: n  
 $\Rightarrow n * O(n^2) = O(n^3)$

Nbre d'iter: n  
 $\Rightarrow O(n)$

$$O(T) = O(n^3)$$