R3.02 - Développement efficace

Introduction, objectifs, mémoire et structures

Imad Assayakh

imad.assayakh@univ-lorraine.fr

2025 – 2026

1. **Choisir et utiliser des structures de données adaptées.**

2. **Initier à l'analyse et à l'optimisation des performances** (temps, mémoire, énergie).

Message clé : ce cours aide à devenir de *meilleurs développeurs*, pas seulement à faire fonctionner un programme.

Allons détailler ces besoins

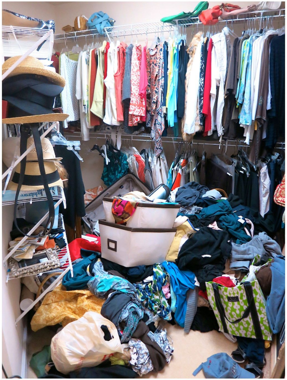
avec des exemples concrets.

Mise en contexte

Comment organiser un placard ?

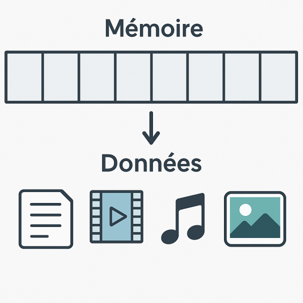
Quels sont les problèmes si on met tout en vrac dans un placard ?

Oups !



1. On a de la place mais on ne peut pas tout ranger efficacement.
2. On perd du temps à retrouver un objet.
3. Les petits objets se perdent parmi les grands.

Analogie



En informatique,

- placard $\leftrightarrow$ la mémoire
- objets $\leftrightarrow$ données (textes, vidéos, sons, etc.).

Qu'est-ce que la mémoire ?

Comment les variables sont-elles organisées en mémoire ?

Mémoire

1000	0 1 0 1 0 1 0 1
1001	1 1 0 0 1 1 0 0
1002	0 0 0 0 0 0 0 0
1003	1 1 1 1 0 0 0 0
1004	1 0 1 0 1 0 1 0
1005	0 1 1 0 0 0 1 0
1006	1 1 1 1 1 1 1 1
1007	0 0 0 1 1 1 1 0
1008	1 0 0 0 0 0 0 1
1009	0 0 1 1 1 0 1 1

1. Que signifie vraiment l'instruction : `int x = 5;` ?
2. Quelle est la taille en mémoire de chaque type de données : `int`, `float`, `char` ?
3. Pour stocker **1000 entiers**, vaut-il mieux utiliser : **1000 variables séparées** ou une **structure adaptée** (par ex. un **tableau**) ?

`int x = 5;`

Que signifie cette instruction ?

- On demande au compilateur de **réserver un espace en mémoire** capable de contenir un entier (en général **4 octets**).
- Cet espace est initialisé avec la valeur **5**.

- `int` : type de la variable (**entier**).
- `x` : nom de la variable.
- `= 5` : valeur affectée.

Taille en mémoire des types de base

Tailles usuelles (32/64 bits modernes)

- `int` : 4 octets (32 bits)
- `float` : 4 octets (32 bits, IEEE 754 simple précision)
- `char` : 1 octet (8 bits, code ASCII)

Exemple : int

`int x = 5;` ⇒ occupe 4 octets.
En binaire (32 bits) : 00000000 00000000 00000000 00000101

Décomposition : $5 = 2^2 + 2^0$

Exemple : char

`char c = 'A';` ⇒ 1 octet (ASCII 65).
Binaire (8 bits) : 01000001

Décomposition : $65 = 2^6 + 2^0$

Stockage de plusieurs entiers

Problème posé

Pour stocker **1000 entiers**, vaut-il mieux utiliser :

- 1000 variables séparées ?
- Une **structure adaptée** (par ex. un tableau) ?

Réponse : une **structure adaptée**.

- Manipulation plus simple (parcours avec des **boucles**).
- Données organisées en mémoire $\Rightarrow$ meilleure efficacité.
- ...

Retour sur l'analogie : placard $\leftrightarrow$ mémoire

Comparaison

- Placard organisé en **boîtes** $\Rightarrow$ **tableau**.
- Placard avec **étiquettes** $\Rightarrow$ **dictionnaire / table de hachage**.
- Placard avec **tiroirs hiérarchiques** $\Rightarrow$ **arbre**.

- **Mauvaise organisation** $\Rightarrow$ gaspillage d'espace et perte de temps.
- **Bonne organisation** $\Rightarrow$ efficacité et optimisation.

Initier à l'analyse et à l'optimisation des performances

Question :

Si vous devez monter au 10^{e} étage d'un immeuble, préférez-vous prendre les ***escaliers*** *ou l'****ascenseur*** *?*

- **Escaliers** : ça marche toujours, mais c'est **long** et **coûteux en énergie** (surtout si on le fait souvent).
- **Ascenseur** : **plus rapide**, **moins d'effort**.

Transition vers l'informatique : pour un même problème, il existe **plusieurs solutions** ; certaines sont **lentes** (temps/mémoire), d'autres **optimisées**.

Notion de complexité : notation $\mathcal{O}$

Idée clé

La notation $\mathcal{O}$ décrit **l'ordre de grandeur du coût d'un algorithme** en fonction de la taille de l'entrée n.

Par défaut, on considère la **complexité dans le pire cas**.

Exemples :

- $\mathcal{O}(1)$: temps constant (ex. accès à `tab[5]`).
- $\mathcal{O}(n)$: temps linéaire (ex. parcours d'un tableau de n éléments).
- $\mathcal{O}(n^2)$: temps quadratique (ex. deux boucles imbriquées sur n).

Remarque : plus n augmente, plus l'écart entre $\mathcal{O}(n)$ et $\mathcal{O}(n^2)$ devient considérable.

Complexité : Temps vs. Mémoire

Complexité en temps

Mesure le nombre d'opérations élémentaires en fonction de la taille de l'entrée n.

Indique **la durée d'exécution**.

Exemples :

- $\mathcal{O}(1)$: accès direct à `tab[5]`.
- $\mathcal{O}(n)$: parcours d'un tableau.
- $\mathcal{O}(n^2)$: deux boucles imbriquées.

Complexité en mémoire

Mesure la quantité d'espace nécessaire en fonction de la taille de l'entrée n.

Indique **la mémoire utilisée**.

Exemples :

- $\mathcal{O}(1)$: quelques variables.
- $\mathcal{O}(n)$: tableau de n éléments.
- $\mathcal{O}(n^2)$: matrice $n \times n$.

Exercice 1 : Sommes cumulatives dans un tableau

On dispose d'un tableau d'entiers `T`.
On définit la somme cumulative à l'indice i :

$$\text{sommeCumulative}[i] = \mathsf{T}[0] + \mathsf{T}[1] + \cdots + \mathsf{T}[i]$$

Écrire une fonction qui retourne un nouveau tableau des sommes cumulatives.

```java
public int[] sommeCumulative(int[] T)
```

Exemples :

- $[1, 2, 3, 4] \to [1, 3, 6, 10]$
- $[1, 1, 1, 1, 1] \to [1, 2, 3, 4, 5]$
- $[3, 1, 2, 10, 1] \to [3, 4, 6, 16, 17]$

Exercice 1 : Solution naïve : $\mathcal{O}(n^2)$ (comme les escaliers)

Pour chaque i, on recalcule la somme depuis 0.

```
public int[] sommeCumulativeNaive(int[] T) {
    int n = T.length;
    int[] result = new int[n];
    for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = 0; j <= i; j++) {
            sum += T[j];
        }
        result[i] = sum;
    }
    return result;
}
```

Complexité temps : $\mathcal{O}(n^2)$

Complexité espace : $\mathcal{O}(n)$

Exercice 1 : Solution optimisée : $\mathcal{O}(n)$ (comme l'ascenseur)

On réutilise la somme précédente (un seul passage).

```java
public int[] sommeCumulative(int[] T) {
    int n = T.length;
    int[] result = new int[n];
    result[0] = T[0];
    for (int i = 1; i < n; i++) {
        result[i] = result[i-1] + T[i];
    }
    return result;
}
```

Complexité temps : $\mathcal{O}(n)$
Complexité espace : $\mathcal{O}(n)$

// Si l'énoncé autorise de modifier T : O(n) temps, O(1) espace suppl.

```
public int[] sommeCumulative(int[] T) {
    int n = T.length;
    for (int i = 1; i < n; i++) {
        T[i] = T[i-1] + T[i];
    }
    return T;
}
```

Remarque

- Deux solutions, **même résultat** ⋯ mais **coûts très différents**.
- Naïve : $\mathcal{O}(n^2) \Rightarrow$ inefficace pour grands n.
- Optimisée : $\mathcal{O}(n) \Rightarrow$ un seul passage, bien plus rapide.

Exercice 2 : Nombre manquant

On dispose d'un tableau d'entiers `T` de taille n contenant tous les nombres de 0 à n, à l'exception d'un seul nombre manquant. L'objectif est de retrouver ce nombre.

```java
public int nombreManquant(int[] T)
```

Exemples :

- $T = [0, 1, 2, 4] \Rightarrow$ nombre manquant $= 3$
- $T = [1, 2, 3, 4, 5] \Rightarrow$ nombre manquant $= 0$

Exercice 2 : Solution naïve

```
public int nombreManquant(int[] T) {
    int n = T.length;
    for (int i = 0; i <= n; i++) {
        int j = 0;
        while (j < n && T[j] != i) {
            j++;
        }
        if (j == n) {
            return i;
        }
    }
    return -1;
}
```

Complexité :

- Temps : $\mathcal{O}(n^2)$
- Espace : $\mathcal{O}(1)$

Exercice 2 : Solution avec marquage

```java
public int nombreManquant(int[] T) {
    int n = T.length;
    boolean[] present = new boolean[n+1];

    for (int i = 0; i < n; i++) {
        present[T[i]] = true;
    }

    for (int i = 0; i <= n; i++) {
        if (!present[i]) {
            return i;
        }
    }
    return -1;
}
```

Complexité : Temps $= \mathcal{O}(n)$ | Espace $= \mathcal{O}(n)$

Exercice 2 : Quel algorithme choisir ?

Si l'on considère uniquement les deux solutions :

- **Peu de mémoire disponible** $\Rightarrow$ choisir la **solution naïve** (temps très élevé : $\mathcal{O}(n^2)$, mais espace minimal : $\mathcal{O}(1)$).
- **Temps d'exécution critique** $\Rightarrow$ choisir la **solution avec marquage** (rapide : $\mathcal{O}(n)$, mais nécessite plus de mémoire : $\mathcal{O}(n)$).

Remarque : le choix d'un algorithme dépend du contexte et des ressources disponibles.

Exercice 2 : Solution plus optimale (par somme)

```java
public int nombreManquant(int[] T) {
    int n = T.length;
    int sommeTheorique = n * (n + 1) / 2;
    int sommeReelle = 0;

    for (int i = 0; i < n; i++) {
        sommeReelle += T[i];
    }
    return sommeTheorique - sommeReelle;
}
```

Complexité :

- Temps : $\mathcal{O}(n)$
- Espace : $\mathcal{O}(1)$

Conclusion

Bilan des objectifs

1. Savoir **choisir et utiliser des structures de données adaptées** pour organiser efficacement la mémoire et manipuler l'information.
   - Le **choix de la structure de données** est crucial.
   - La mémoire peut être comparée à un **placard** : une mauvaise organisation entraîne de l'inefficacité.
2. Être capable **d'analyser et d'optimiser les performances** en tenant compte du temps et de l'espace mémoire.
   - Identifier la **complexité en temps** d'un algorithme.
   - Évaluer la **consommation mémoire** des structures utilisées.
   - Comparer différentes approches pour choisir la plus **efficace**.