

Introduction

Slide 1

La programmation

Dans le domaine de **l'informatique**, la programmation est l'ensemble des activités qui permettent **l'écriture des programmes informatiques**.

C'est une étape importante du **développement** de logiciels.

Qu'est-ce qu'un programme ?

Un programme informatique est **un ensemble d'opérations** destinées à être exécutées par un ordinateur :

- Un programme source est un **code** écrit par un informaticien dans un **langage** de programmation.

Il peut être **compilé** vers une forme binaire, ou directement **interprété**.

Qu'est-ce qu'un langage ?

En informatique, **un langage de programmation** est une notation conventionnelle destinée à formuler des **algorithmes**.

Un langage de programmation est composé :

- d'un **alphabet**,
- d'un **vocabulaire**,
- de **règles de grammaire**
- et de **significations**.

Qu'est-ce qu'un langage ?

L'apprentissage **d'un premier langage** de programmation est à l'image de l'apprentissage **d'une langue** parfaitement inconnue.

Cela ne se fait pas du jour au lendemain, et cela requiert **beaucoup** de pratique, d'exercices et surtout de projets.

Les différents paradigmes

- **Programmation séquentielle** : l'ordinateur déroule une liste d'instructions et les exécute pas à pas.
- **Programmation procédurale** : des procédures, contenant du code, peuvent elles-mêmes être appelées par du code, suivant des enchaînements contrôlés.
- **Programmation orientée objet** : des objets définissent des fonctions et des contenus, et peuvent s'appeler les uns les autres => on abstrait des méthodes, mais aussi des contenus.

Quel langage pour quelle application ?

Chaque langage a ses spécificités.

- Pour le web (html, css, php, javascript, ...).
- Pour des logiciels (C, C++, C#, Java, ...).
- Pour le mobile (Objectif C, Swift, Java, ...).
- Pour des scripts (Python, Ruby, Perl, ...).
- Pour les bases de données (SQL, ...).

Les compilateurs

Un compilateur est, en informatique, le terme utilisé pour désigner **un programme qui transforme** :

- **un code source** écrit dans un langage de programmation (le langage source)
- **en un langage machine** (appelé langage cible).

Les exécutables

Un exécutable est **le fichier** qui résulte de la compilation.

Il ne peut être exécuté que par une machine en mesure de le faire.

Un .exe (programme windows) ne peut pas être exécuté sur une machine Linux.

Qu'est-ce qu'un algorithme ?

Un **algorithme** est une procédure de calcul bien définie permettant de répondre à un problème.

Un algorithme prend en **entrée** un ensemble de valeurs et qui délivre en **sortie** un ensemble de valeurs.

On réalise une stratégie pour répondre à ce problème.
Il existe souvent différents algorithmes permettant de répondre au même problème mais pas de la même manière.

Exemple

Problème : trier une suite de nombres entiers dans l'ordre croissant.

Entrée : suite de n nombres entiers $(a_1, a_2, \dots, a_n)$

Sortie : une permutation de la suite donnée en entrée $(a'_1, a'_2, \dots, a'_n)$ telle que $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

À partir de la suite $(6, 9, 2, 4)$, un algorithme de tri fournira le résultat $(2, 4, 6, 9)$.

Un algorithme

Un algorithme est **correct**, si pour toute instance du problème il se termine et produit une sortie correcte.

L'efficacité d'un algorithme est mesurée par son coût, **sa complexité**, en temps et en mémoire.

Pourquoi parler de complexité ?

Deux algorithmes peuvent résoudre le même problème...

Rechercher un élément dans une liste :

- **Algorithme A** : trouve l'élément après quelques opérations.
- **Algorithme B** : doit parcourir toute la liste.

Pourquoi parler de complexité ?

Sur une liste de 10 éléments, la différence peut sembler insignifiante.

Mais que se passe-t-il avec :

- 1 000 éléments ?
- 1 000 000 d'éléments ?
- 1 milliard d'éléments ?

Pourquoi parler de complexité ?

L'objectif de la complexité algorithmique est de mesurer comment le coût d'un algorithme évolue lorsque la quantité de données augmente.

Un bon algorithme n'est pas seulement un algorithme qui fonctionne.

C'est aussi un algorithme qui **reste efficace lorsque les données deviennent importantes.**

Qu'est-ce que la complexité algorithmique ?

La **complexité algorithmique** permet d'étudier les ressources nécessaires à l'exécution d'un algorithme.

On s'intéresse principalement à deux ressources :

- **Complexité temporelle** : elle mesure comment évolue le nombre d'opérations nécessaires à l'algorithme.
- **Complexité spatiale** : elle mesure la quantité de mémoire supplémentaire utilisée par l'algorithme.

Qu'est-ce que la complexité algorithmique ?

“Dire qu’un algorithme prend 0,02 seconde.”

Qu'est-ce que la complexité algorithmique ?

“Dire qu’un algorithme prend 0,02 seconde.”

Hum... ce résultat n'est pas très intéressant.

Qu'est-ce que la complexité algorithmique ?

Le temps dépend :

- du processeur,
- de la mémoire,
- du langage utilisé,
- de l'implémentation,
- du compilateur...

Nous préférons donc compter approximativement **le nombre d'opérations effectuées.**

Qu'est-ce que la complexité algorithmique ?

Par exemple :

- 10 éléments $\rightarrow$ environ 10 opérations
- 100 éléments $\rightarrow$ environ 100 opérations
- 1 000 éléments $\rightarrow$ environ 1 000 opérations

On cherche surtout à connaître **la façon dont le nombre d'opérations évolue avec la taille des données.**

La taille des données : n

Pour simplifier, on utilise généralement la lettre n pour représenter la taille du problème.

Si notre tableau contient $[12, 5, 8, 21, 4]$ alors $n = 5$

La question devient alors :

Pour le même algorithme, comment évolue le nombre d'opérations lorsque n augmente ?

La notation Big O

Pour décrire cette évolution, on utilise **la notation Big O**.

Quelques exemples :

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(2^n)$

Big O permet de décrire l'ordre de grandeur de la croissance du coût d'un algorithme.

Ce que Big O ignore

Big O ne cherche pas à connaître le nombre exact d'opérations.

Par exemple **$3n + 5$** et **$100n + 2\,000$** sont tous les deux en $O(n)$.

Pourquoi ?

Parce que lorsque n devient très grand, c'est le terme qui dépend de n qui devient prépondérant.

De la même manière $5n^2 + 3n + 10$ est en $O(n^2)$

Big O $\neq$ temps d'exécution

Deux algorithmes peuvent tous les deux être en $O(n)$ sans avoir le même temps d'exécution.

Par exemple :

- Algorithme A : $2n$ opérations
- Algorithme B : $100n$ opérations

Les deux sont en $O(n)$, mais B peut être beaucoup plus lent en pratique.

Big O $\neq$ temps d'exécution

Big O ne donne donc pas une durée d'exécution exacte.

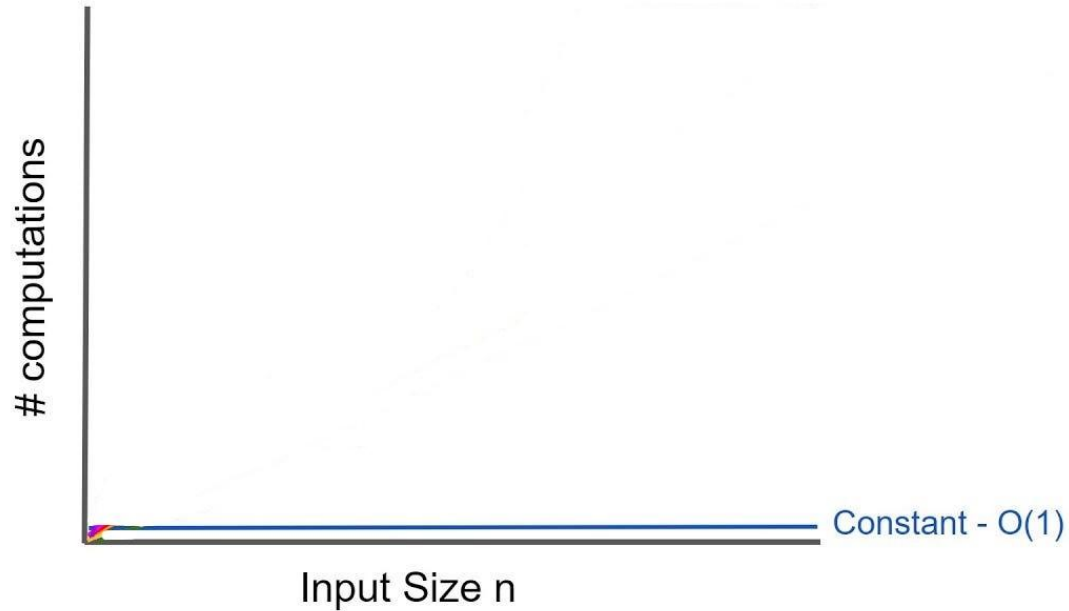
Il permet surtout de comparer la manière dont les algorithmes évoluent lorsque n augmente.

$O(1)$: temps constant

Un algorithme est en **$O(1)$** lorsque son nombre d'opérations **ne dépend pas de la taille des données**.

Que le tableau contienne 10 éléments ou 10 000 000 éléments l'opération prend toujours approximativement le même nombre d'étapes.

$O(1)$: temps constant



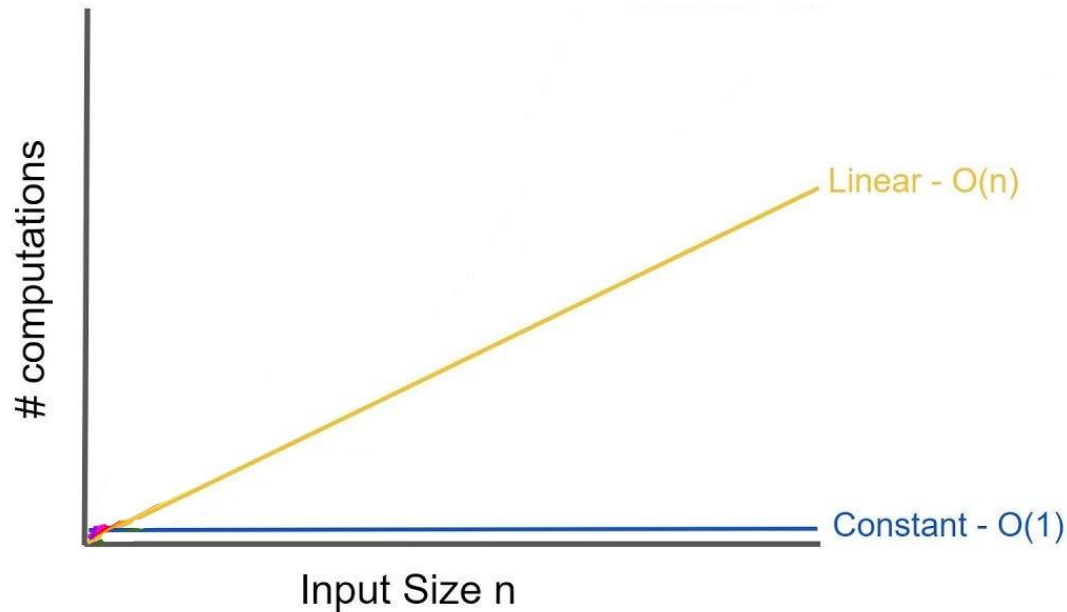
$O(n)$: temps linéaire

Un algorithme est en $O(n)$ lorsque le nombre d'opérations **augmente proportionnellement à la taille des données**.

Exemple : trouver le max d'un tableau :

- 10 éléments $\rightarrow$ 9 étapes (comparaisons)
- 1 000 éléments $\rightarrow$ 999 étapes (comparaisons)

$O(n)$: temps linéaire



$O(n^2)$: temps quadratique

Un algorithme est en $O(n^2)$ lorsqu'il doit effectuer **approximativement $n \times n$ opérations**.

C'est très souvent le résultat de deux boucles imbriquées.

Exemple :

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        // opération  
    }  
}
```

$O(n^2)$: temps quadratique

Pour $n = 10$, on obtient environ :

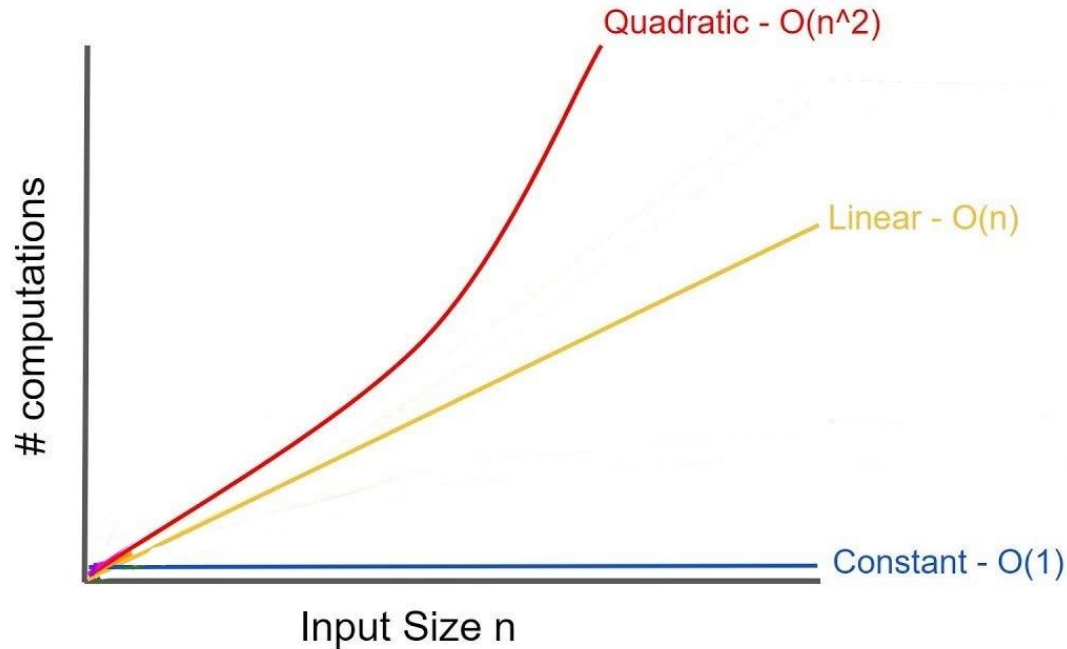
- $10 \times 10 = 100$ opérations

Pour $n = 1\,000$, on obtient :

- $1\,000 \times 1\,000 = 1\,000\,000$ opérations

$O(n^2)$ augmente beaucoup plus rapidement que $O(n)$.

$O(n^2)$: temps quadratique



$O(\log n)$: temps logarithmique

Cette complexité est un peu moins intuitive.

Un algorithme en **$O(\log n)$** réduit généralement fortement la taille du problème à chaque étape.

Un algorithme en $O(\log n)$ est très efficace.

$O(\log n)$: temps logarithmique

Exemple avec une recherche dichotomique :

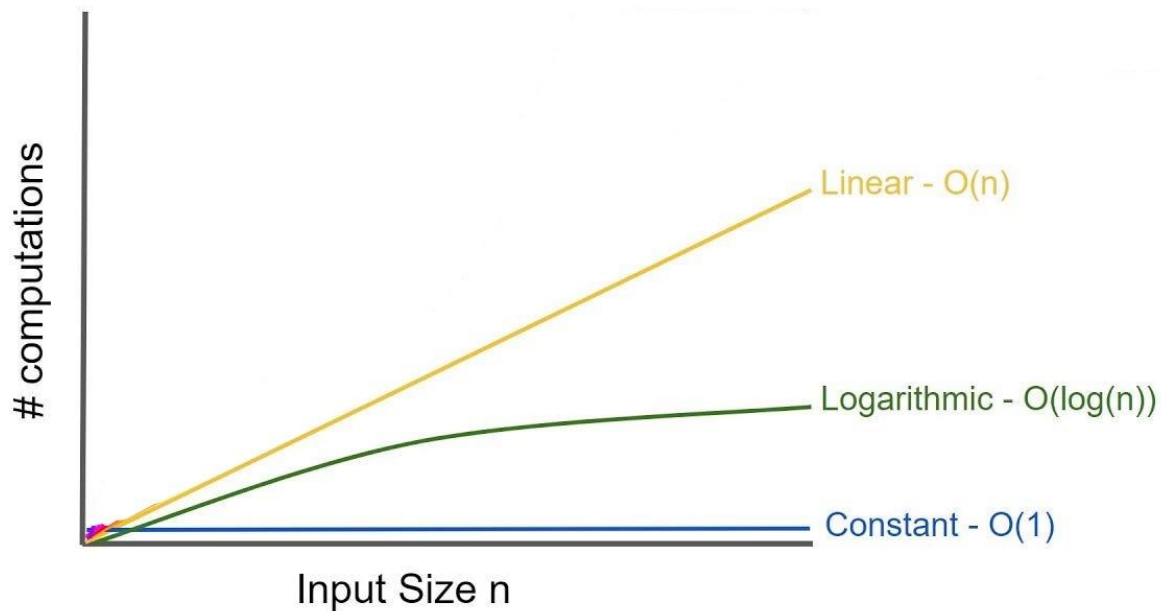
- 1 000 éléments $\rightarrow$ environ 10 étapes
- 1 000 000 éléments $\rightarrow$ environ 20 étapes
- 1 000 000 000 éléments $\rightarrow$ environ 30 étapes

Pourquoi ?

- $2^{10} \approx 1\,000$
- $2^{20} \approx 1\,000\,000$
- $2^{30} \approx 1\,000\,000\,000$

$O(\log n)$ signifie généralement qu'à **chaque étape**, on élimine **une grande partie du problème**.

$O(\log n)$: temps logarithmique



$O(n \log n)$

Nous avons maintenant deux notions **$O(n)$** et **$O(\log n)$** .

Lorsque les deux sont combinées, on obtient **$O(n \log n)$**

Dans l'idée, le nombre de parcours dépend de n mais la quantité d'opération diminue très fortement à chaque étapes.

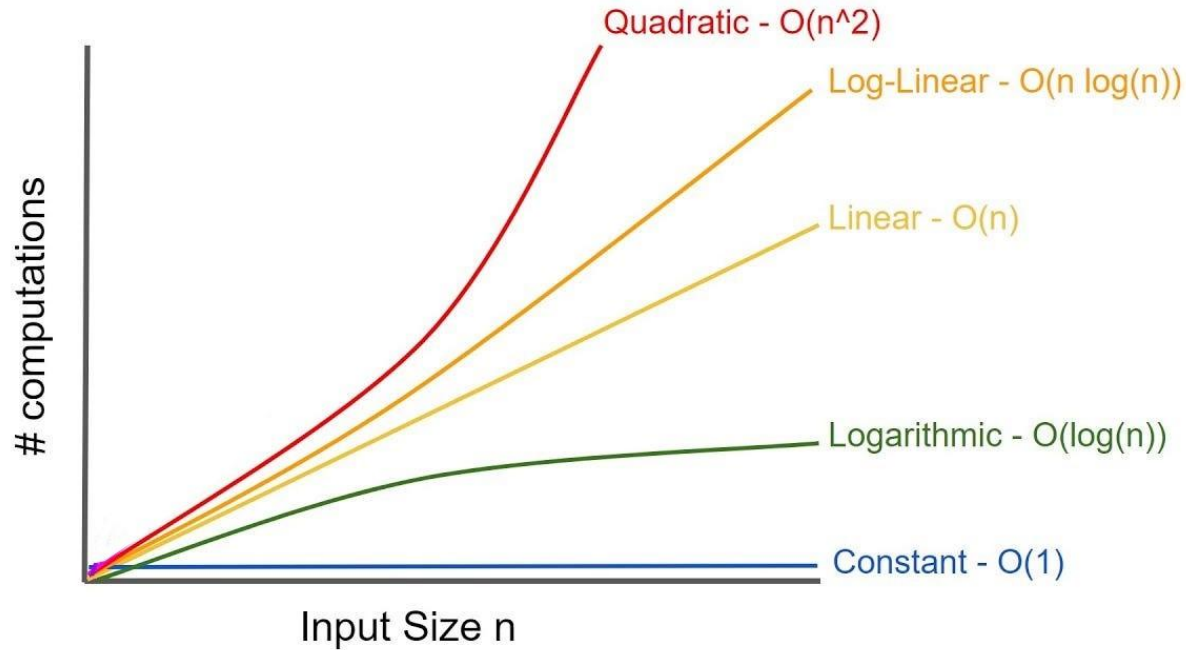
$O(n \log n)$

Cette complexité apparaît notamment dans certains algorithmes de tri efficaces.

Exemples :

- Merge Sort
- Heap Sort
- Quick Sort dans son cas moyen

$O(n \log n)$



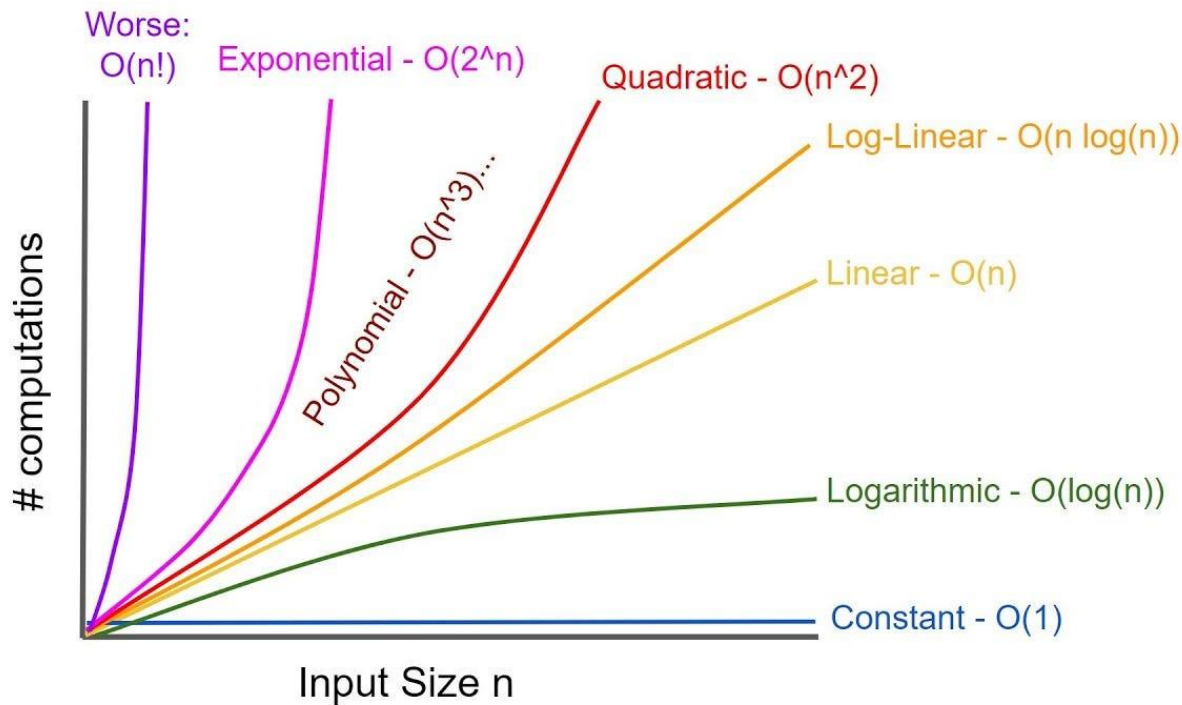
$O(2^n)$: temps exponentiel

Complexité qui augmente **extrêmement rapidement** $O(2^n)$
On parle de complexité exponentielle.

Le nombre d'opérations double à chaque fois que n augmente :

- $n = 1 \rightarrow 2$ opérations
- $n = 2 \rightarrow 4$ opérations
- $n = 3 \rightarrow 8$ opérations
- $n = 4 \rightarrow 16$ opérations
- $n = 10 \rightarrow 1\,024$ opérations
- $n = 20 \rightarrow 1\,048\,576$ opérations
- $n = 30 \rightarrow 1\,073\,741\,824$ opérations

$O(2^n)$: temps exponentiel



Additionner les complexités

Exemple :

```
for (int i = 0; i < n; i++) {  
    // Opération  
}  
for (int i = 0; i < n; i++) {  
    // Opération  
}
```

On a : **$O(n) + O(n)$** soit **$O(2n)$**

En Big O : **$O(2n) = O(n)$**

Additionner les complexités

Exemple :

```
for (int i = 0; i < n; i++) {  
    // Opération  
}  
for (int i = 0; i < n; i++) {  
    // Opération  
}
```

On a : $O(n) + O(n)$ soit $O(2n)$

En Big O : $O(2n) = O(n) \rightarrow$ La complexité reste linéaire

Multiplier les complexités

Exemple :

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        // Opération  
    }  
}
```

On obtient $n \times n$ donc **$O(n^2)$**

Le cas moyen, le meilleur et le pire cas

La complexité **peut dépendre du scénario.**

Prenons une recherche dans un tableau [12, 5, 18, 7, 15]

- Si nous cherchons 12 $\rightarrow$ 1 seule comparaison
- Si nous cherchons 15 $\rightarrow$ plusieurs comparaisons
- Si nous cherchons 42 $\rightarrow$ il faut parcourir tout le tableau

Le cas moyen, le meilleur et le pire cas

On distingue donc :

- **Meilleur cas** : Le scénario le plus favorable.
- **Cas moyen** : Le comportement moyen de l'algorithme.
- **Pire cas** : Le scénario nécessitant le plus de travail.

En algorithmique, on utilise très souvent le pire cas pour garantir une limite maximale du coût.

Conclusion

Un algorithme efficace n'est pas nécessairement celui qui est le plus rapide pour 10 éléments.

C'est celui dont le coût reste maîtrisable lorsque la taille du problème augmente.