

Grace Hopper (1906 – 1992)



Informaticienne américaine. Manipuler les nombreux interrupteurs de valeur 0 et 1 dans les premiers calculateurs électromécaniques, l'a conduite à proposer les premiers compilateurs, dont la fonction est de traduire des instructions en langage humain vers les positions de ces interrupteurs. La légende raconte que c'est en constatant une erreur dans un calcul réalisé sur une machine à cartes perforées qu'elle aurait introduit le terme de *bug* (insecte en anglais) : une mite se serait logée dans l'un des trous.

Bref, avec Grace Hopper, les compilateurs nous facilitent la programmation !

Source : [1]

UE09 – Algorithmique Complexité – La notation « Big-O »

Nicolas HENDRIKX & Simon LIÉNARDY

Sommaire du thème 1

1. Un premier exercice
2. Définitions
3. Exemples d'algorithme
4. Caractéristiques (domaine, exactitude, efficacité)
5. La notion de complexité
6. Comparaison d'algorithmes polynomiaux
7. Types de problèmes

Sommaire du jour

1. Exercice introductif
2. Constatations et tentatives d'explications
3. Définition
4. L'outil dont nous avons besoin : la notation grand-O
5. Calculer la complexité en pratique

La notion de complexité

```
private static String exemple1() {  
    StringJoiner sj = new StringJoiner("");  
    for (int i = 0; i < count; i++) {  
        String s = lignes[i % lignes.length];  
        sj.add(s.split(",")[1]);  
    }  
    return sj.toString();  
}
```

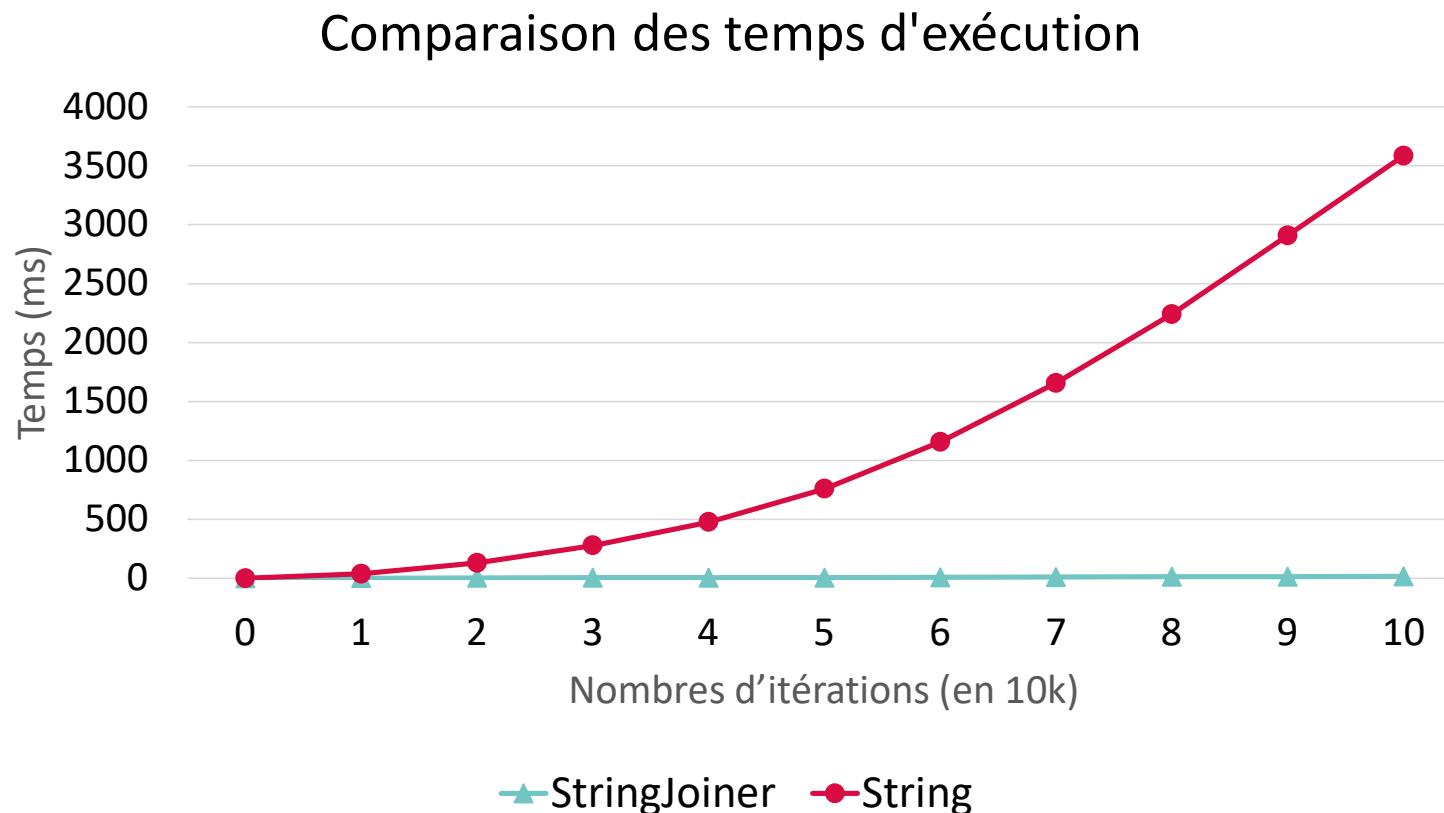
Faisons l'[exercice introductif](#)
(voir HELMo LEARN)

```
private static String exemple2() {  
    String result = "";  
    for (int i = 0; i < count; i++) {  
        String s = lignes[i % lignes.length];  
        result += s.split(",")[1];  
    }  
    return result;  
}
```

Sommaire

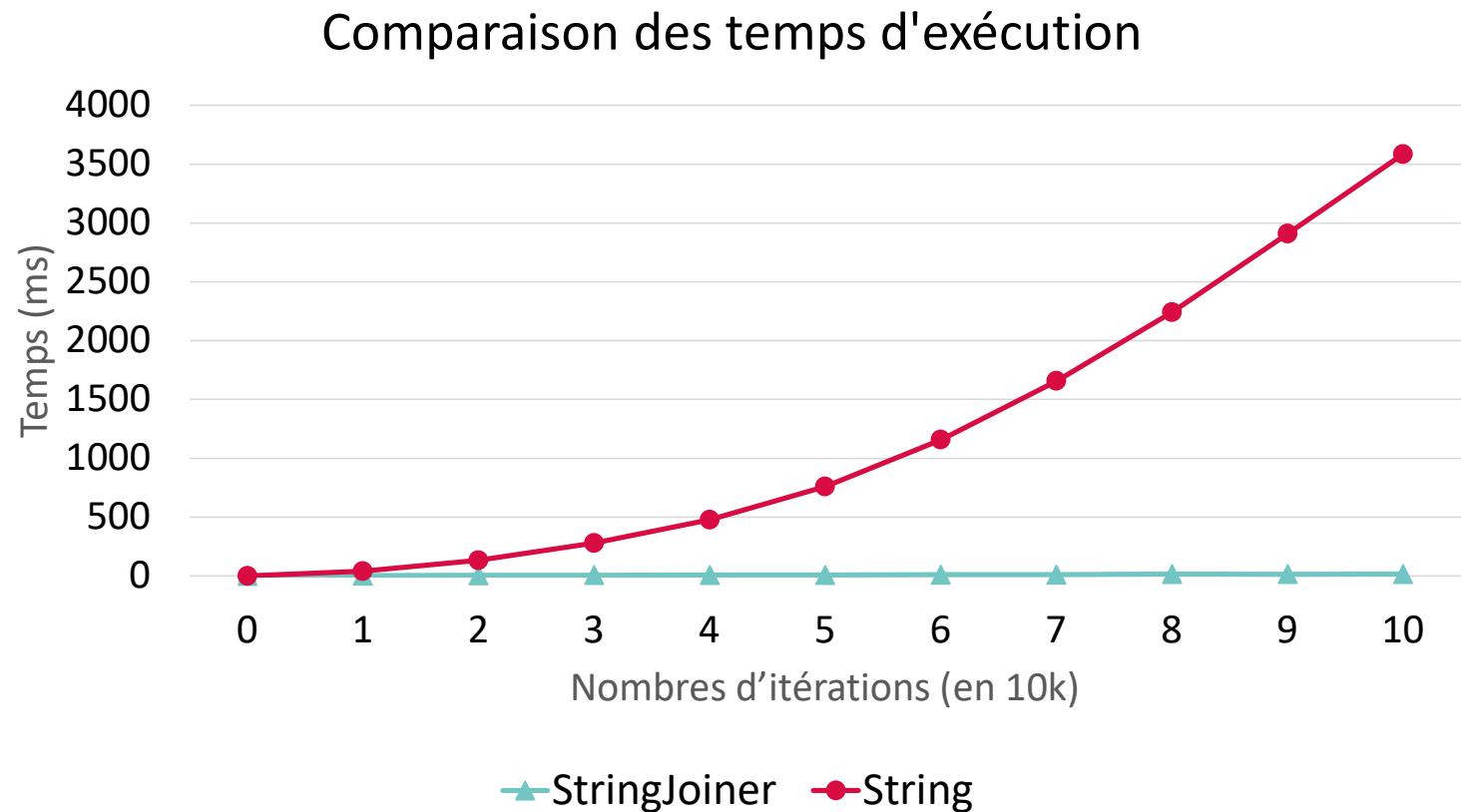
1. Exercice introductif
2. Constatations et tentatives d'explications
3. Définition
4. L'outil dont nous avons besoin : la notation grand-O
5. Calculer la complexité en pratique

Constatations



Constatations

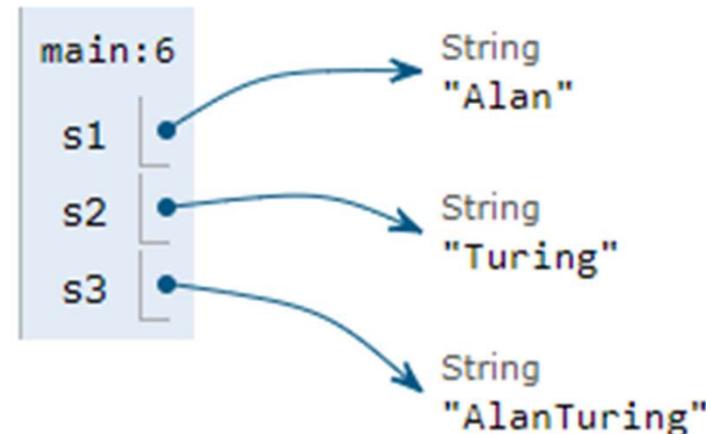
- Le temps d'exécutions **exact** dépend de nos ordinateurs particuliers.
- Mais nous avons tous observé ce genre de courbe, **quel que soit le matériel dont nous disposons !**
- *Pourquoi ?* 🤔



Concaténation de chaîne

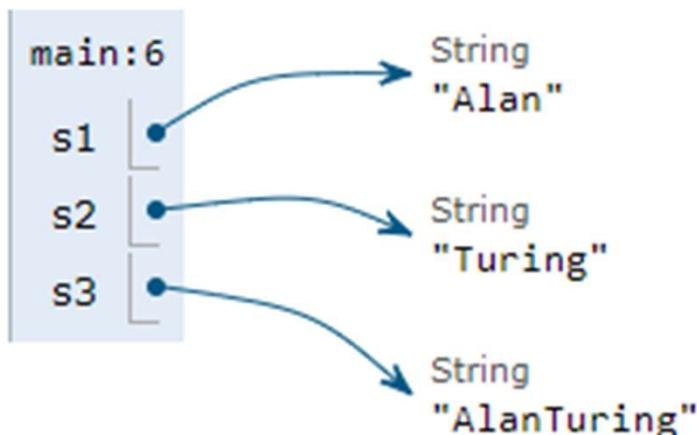
- Le temps d'exécutions **exact** dépend de nos ordinateurs particuliers.
- Mais nous avons tous observé ce genre de courbe, **quel que soit le matériel dont nous disposons !**
- *Pourquoi ?* 🤔

```
// À visualiser ICI
String s1 = "Alan";
String s2 = "Turing";
String s3 = s1 + s2;
```



Concaténation de chaîne

```
// À visualiser ICI
String s1 = "Alan";
String s2 = "Turing";
String s3 = s1 + s2;
```



Ce qui importe, c'est le nombre **d'opérations élémentaires** effectuées par le programme.

Ce nombre ne change pas d'un ordinateur à l'autre

Ici, concaténer la chaîne va demander de recopier toute la chaîne s1 et toute la chaîne s2 dans s3

D'où vient la courbe en forme de parabole ?

- Supposons qu'il y ait N itérations
- Pour simplifier, supposons que toutes les String ont une longueur p
- À la 1^{re} itération, on concatène la chaîne vide et une chaîne de longueur p, cela demande p opérations.
- À la 2^e itération, on concatène le résultat (p caract.) et une nouvelle chaîne (p) = 2p opérations.
- À la 3^e itérations, on concatène le résultat (2p caract.) et une nouvelle chaîne (p) = 3p opérations.
- Pour les n itérations, on a au total $p + 2p + 3p + 4p + \dots + Np$
 $= p (1 + 2 + 3 + \dots + N)$ // Mise en évidence

$$1 + 2 + 3 + \dots + (N - 1) + N$$

$$1 + 2 + 3 + \dots + (N - 1) + N$$

$$1 + 2 + 3 + \dots + (N - 1) + N$$

$$1 + 2 + 3 + \dots + (N - 1) + N$$

$$N + (N - 1) + \dots + 3 + 2 + 1$$

$$\sum_{i=1}^N i = \frac{N \times (N + 1)}{2}$$

$$(N + 1) + \dots + (N + 1) + (N + 1)$$

N fois

D'où vient la courbe en forme de parabole ?

- Pour les n itérations, on a au total $p + 2p + 3p + 4p + \dots + Np$
 $= p (1 + 2 + 3 + \dots + N)$ // Mise en évidence
 $= p (N (N-1)) / 2$ // Voir slide précédent

Conclusion → Le nombre d'opérations à réaliser pour les N itérations va être une fonction du carré de N, d'où la forme de la **parabole**.

Comment exprimer simplement ceci ?

- *Le temps pris par la version du code avec concaténation évolue avec le carré du nombre d'itérations*
 - *Le temps pris par la version avec StringJoiner évolue proportionnellement au nombre d'itérations*
 - *La version du StringJoiner est plus efficace*
 - *Exprimer comment l'espace mémoire utilisé par les programmes varie*
- BESOIN D'UNE NOTION : LA COMPLEXITÉ

Sommaire

1. Exercice introductif
2. Constatations et tentatives d'explications
3. Définition
4. L'outil dont nous avons besoin : la notation grand-O
5. Calculer la complexité en pratique

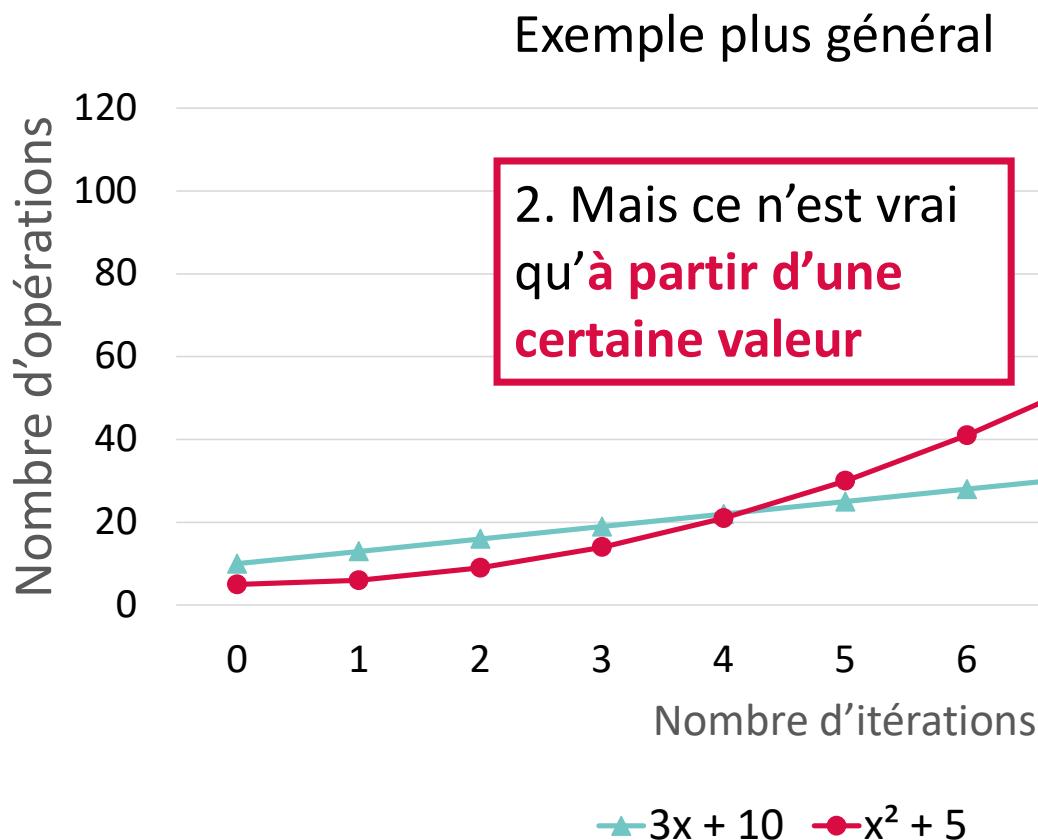
Définition

L'analyse de la complexité d'un algorithme est

- l'étude **formelle** [= nécessitant des calculs]
- de la quantité de **ressources** nécessaire [= dans ce cours : temps et espace]
- à l'exécution de cet **algorithme** [2].

= propriété de l'algorithme, indépendant du matériel
(détail important : cela doit rester un ordinateur avec un processeur et une RAM)

Que souhaitons-nous exprimer exactement ?



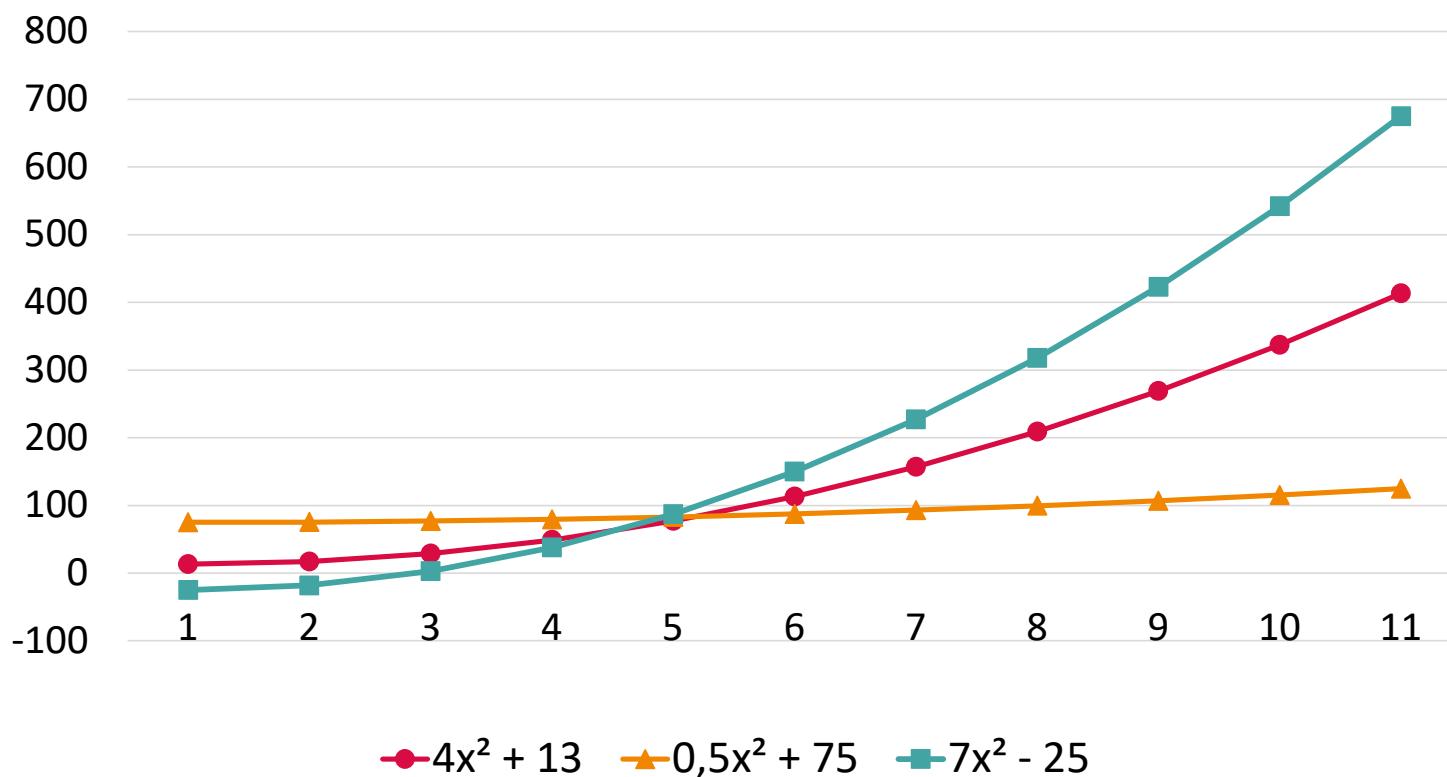
2. Mais ce n'est vrai
qu'à partir d'une
certaine valeur

1. La courbe rouge
est clairement **au
dessus** de la verte

3. Il est plus facile de
comparer une fonction à un
comportement bien connu !

Toutes les fonctions en x^2 se ressemblent !

Comparaison de plusieurs fonction en x^2



Dans la suite, pour avoir une idée de comment va varier la consommation de ressources, on va dire : « comme une droite » ou « comme une parabole », en se référant à des fonctions connues

Sommaire

1. Exercice introductif
2. Constatations et tentatives d'explications
3. Définition
4. L'outil dont nous avons besoin : la notation grand-O
5. Calculer la complexité en pratique

Récapitulons – Inventaires de nos besoins

- On veut pouvoir **comparer** des algorithmes entre eux
- On veut comparer des **comportements**
- Si possible à d'autres comportements **bien connus**
- Avec une notation **simple** à utiliser / intuitive

On veut pouvoir comparer des algorithmes

- En math, quand on veut comparer des choses, on utilise les opérateurs $<$, $>$, \leq , \geq
 - Exemple : $4 > 3$
- Pour deux algorithmes, écrire :

$$X < Y$$

N'a pas beaucoup de sens si X et Y sont fixes...

→ Si les données sont petites les ressources consommées seront petites, si elles sont grandes, on en aura besoin de plus.

Les ressources sont **fonctions** des données

- Si les données sont petites les ressources consommées seront petites, si elles sont grandes, on en aura besoin de plus.
- On va exprimer $X(\text{taille})$, la quantité de ressource utilisée par un algo en fonction de la taille du problème
- Taille du problème ?
 - Taille du tableau utilisé (ex. : tri)
 - Valeur des paramètres (ex. : calcul de la factorielle $f!$)

Certaines fonctions sont bien connues

- Si on compare des fonctions, autant les comparer à des fonctions bien connues :
 - Droite $y = x$
 - Parabole $y = x^2$
 - Logarithme $y = \log(x)$
- Dans la suite, on s'intéressera surtout à des fonctions **croissantes** (si votre algo consomme moins de ressources à mesure que la taille du problème augmente, les ressources consommées ne sont pas un sujet...)

Multiplier une fonction par une constante ne change pas sa forme

- Par une constante positive,
évidemment... sinon la fonction peut devenir décroissante !
- Exemple : toutes ces droites ont la même forme :

$$x, 2x, \frac{1}{2}x, 10x$$

- Voir dessins au tableau

Nous pouvons donc comparer l'allure de deux fonctions

Écrivons :

$$X(\text{taille}) \leq Y(\text{taille}) \times c$$

c, constante au choix, nous permet d'exprimer que X sera plus petit que l'allure de la fonction Y.

Pour autant que l'on puisse écrire cela avec au moins un c !

Il faut qu'un c qui vérifie cette inégalité existe : n'écrivez pas de fakenews!

Jusqu'ici, nous comparons les fonctions pour toutes les tailles possibles
Limitons-nous !

Les ressources ne sont critiques que pour les grands problèmes

- Petits problèmes → petites ressources : OSEF
- Problèmes assez grands : là, la consommation de l'algorithme commence à nous intéresser.
- On va dire qu'à partir d'une certaine taille (**seuil**), on compare nos fonctions (voir illustration précédemment)
- On parle de **comportement asymptotique**

Récapitulons :

- On compare à partir d'une **certaine taille**:

$$\exists \text{taille}_0 > 0$$

- On a besoin d'une constante qui nous permette de comparer des **allures** de fonction :

$$\exists c > 0$$

- On **compare** les allures de fonctions :

$$X(\text{taille}) \leq c \times Y(\text{taille})$$

- Pour toutes les tailles, après le **seuil** :

$$\forall \text{taille} > \text{taille}_0$$

Qui nous donne la notation suivante :

$$\exists \text{taille}_0 > 0, \exists c > 0 : X(\text{taille}) \leq c \times Y(\text{taille}), \forall \text{taille} > \text{taille}_0$$
 \Leftrightarrow $X \in O(Y)$

Ceci montre
l'appartenance de
 X à l'ensemble



Ceci est un ensemble
(toutes les fonctions qui
sont dominées par Y)

Lecture : X est grand-O de Y bien « Y domine X en $+\infty$ »

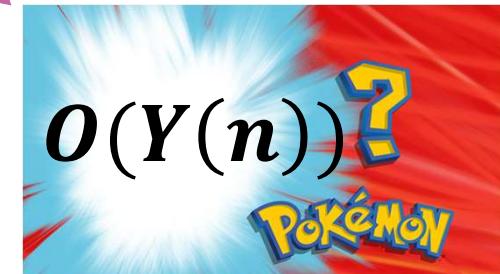
Comment faire tousser un mathématicien ?

- Écrivez ceci :

$$X(n) = O(Y(n))$$

Valeur de fonction ?

Égalité ?



Attention : souvent rencontré, même dans des bouquins de référence !

Certains additionnent même des grand-O à d'autres trucs !

Pour une fois, la taille (du problème) compte !

- Quand on exprime $X = \text{taille}^2$,

Écrire

$$X \in O(n^2)$$

Ne veut **rien dire** si n n'apparaît pas dans l'expression de X !

- Simplifier

$$n^2 + m \in O(n^2)$$

N'a **pas de sens** non plus :

→ si m ne dépend pas de n, gardez les deux variables !

Comment montrer que $f \in O(g)$?

- Si c'est vrai, on a dit qu'il existe une constante et un seuil.
 - Il suffit de les fournir
 - Exemple : $18t + 25$ est dans $O(t)$ car :

$$18t + 25 \leq 43t$$

$$t_0 = 1, c = 43$$

Exercice : montrez que dans la version de l'exercice introductif, le nombre d'opérations est $O(N^2)$, avec N le nombre d'itérations.

Comment montrer que $f \notin O(g)$?

- La plupart du temps partir du fait que c serait une constante puis **arriver à une absurdité**

→ Exemple : $b^2 \notin O(b)$

$$\begin{aligned} b^2 &\leq c \times b \\ b &\leq c \end{aligned}$$

Signifie que *pour n'importe quel* b (aussi grand qu'on veut), c constant est plus grand que lui) alors que b croît.

« Celui qui croit que la croissance peut être infinie dans un monde fini est soit un fou, [...] » [3].

→ Impossible donc c n'est pas constant !

Quelques fonctions connues

- Pour une taille du problème égale à 1000 ($n = 1000$) et un temps d'exécution de chaque étape (itération) de 1 microseconde(μs) :



Fonction	Taux de croissance	Exemple	Temps
$O(1)$	Constant	10	$10 \mu s$
$O(\log n)$	Logarithmique	$2 \log n + 3$	$9 \mu s$
$O(n)$	Linéaire	$2n + 50$	$2 \text{ ms } 50 \mu s$
$O(n^2)$	Quadratique	$2n^2 + 3n + 50$	$2\text{s } 3\text{ms } 50 \mu s$
$O(n^3)$	Cubique	$5n^3 + 2n^2 + 2n + 60$	$1\text{h } 23\text{min } + \dots$
$O(2^n)$	Exponentiel	2^n	10^{86} siècles
$O(n!)$	Factoriel	$n!$	$10^{2552} \text{ siècles}$

Dorothy Vaughan (1910 – 2008)



Lune !

Mathématicienne américaine, elle a compris rapidement la nécessité d'utiliser des ordinateurs pour réaliser des calculs complexes, jusque là effectués à la main. Dès les années 1940, elle a travaillé pour la NASA avec Katherine Johnson et Mary Jackson, dans un groupe de femmes réalisant des calculs. Elle a ensuite activement participé aux programmes spatiaux.

Bref, avec Dorothy Vaughan et ses collègues, Apollo 11 a trouvé sa trajectoire vers la

Source : [1]

UE09 – Algorithmique Complexité – La calculer en pratique

Nicolas HENDRIKX & Simon LIÉNARDY

Sommaire

1. Exercice introductif
2. Constatations et tentatives d'explications
3. Définition
4. L'outil dont nous avons besoin : la notation grand-O
5. **Calculer la complexité en pratique**

Complexité : 2 types

- Pour calculer la complexité **temporelle**, on va s'intéresser au nombre d'**opérations élémentaires** de notre algorithme
- Pour calculer la complexité **spatiale**, on va s'intéresser aux **variables** et structures de **données** requises pour l'exécution de l'algorithme.

Opérations élémentaires ?

Parmi les extraits de code suivants, quels sont les opérations élémentaires ? (⏳ 2 min)

```
Color helmo =  
    new Color(217, 11, 67)
```

```
i = x + 3
```

```
y % 2024
```

```
a = new int[17]
```

```
r = s1 + s2
```

```
s.matches("\s+")
```

De quoi sont composés nos codes ?

- Des instructions simples
- Des successions d'instructions
- Des alternatives : **if / switch**
- Des boucles : **for / while / do ... while**
- Des fonctions / méthodes : `call(args)`

Pour calculer la complexité temporelle, il « suffit » d'appliquer quelques règles de calculs dans chacune de ces situations.

Exemple - classe GrilleLotto

- Comme exemple, nous allons déterminer les complexités
 - Spatiales
 - Temporelles

de la méthode `GrilleLotto.tirageBoule` (Voir [Helmo LEARN](#))

Instructions simples

- Les déclarations de variables
- Les initialisations de constantes
- Les opérations sur des types primitifs

→ Tout cela compte pour 1 opération élémentaire

On peut ajouter aussi : les allocations de mémoire dynamique et les appels à certaines fonctions rapides comme Math.random

Successions d'instructions

instr1; // x opérations

instr2; // y opérations

Total : x + y opérations

Alternatives

```
if (condition) {  
    instr1; // x ops  
}  
  
else {  
    instr2; // y ops  
}
```

```
switch (calcul) {  
    case VAL1 -> instr1; // x1 ops  
    case VAL2 -> instr2; // x2 ops  
    ...  
    default -> instrD; // xD ops  
}
```

Total dans le pire des cas ?

$$\rightarrow \max(x,y) + a$$

$$\rightarrow \max(x_1, x_2, \dots, x_D) + b$$

 Il est aussi possible de réfléchir à un **cas moyen** en analysant plus finement à quelle fréquence les alternatives sont exécutées.

Boucles

```
INIT // xINIT ops
while(GARDIEN) { // xG ops
    CORPS // XCORPS ops
}
FIN // XFIN ops
```

Et pour les boucles **for**
et **do...while** ?

Total dans le pire des cas ?

$$Total = X_{Init} + X_{Fin} + \sum^{nb_{tours}} (X_{Corps} + X_G)$$

- On additionne toutes les opérations à tous les tours de boucles

Boucles – Simplification

$$Total = X_{Init} + X_{Fin} + \sum^{nb_tours} (X_{Corps} + X_G)$$

SI le corps de la boucle n'est pas dépendant de la variable d'itération :

$$Total = X_{Init} + X_{Fin} + \left(\sum^{nb_tours} 1 \right) \times (X_{Corps} + X_G)$$

$$Total = X_{Init} + X_{Fin} + nb_tours \times (X_{Corps} + X_G)$$

💡 Généralement, utiliser cette formule simplifiée pour toutes les boucles donne une approximation satisfaisante. Mais cela reste une approximation !

Fonctions

Pour calculer le nombre d'opérations élémentaires d'une fonction ou d'une méthode, il faut compter le nombre d'opérations que l'appel à cette fonction / méthode prend.

→ Bien souvent, ce nombre s'exprime en fonction des paramètres de la fonction / méthode !

Exemple : `Arrays.copyOf(int[] original, int newLength)`
prendra un nombre proportionnel à `newLength`

À vous !

Quand vous avez le nombre d'opérations, vous avez une expression qui utilise les paramètres du problèmes.

Utilisez la notation Grand-O pour exprimer comment évolue ce nombre d'opérations quand les valeurs des paramètres du problème augmentent.

Calculez la complexité temporelle de GrilleLotto.tirageBoule

Complexité spatiale

Pour la complexité spatiale, on compte les espaces mémoire nécessaires. Puis on utilise également la notation grand-O pour exprimer comment l'espace varie quand la taille du problème augmente.

```
int x; // 1 espace mémoire
int[] tableau = new int[X]; // X espaces mémoires
int[][] grille2D = new int[X][y]; // ??
```

Calculez la complexité spatiale de GrilleLotto.tirageBoule

Conclusion

- Certains algorithmes sont plus efficaces en temps mais consomment plus de mémoire.
- Ou inversement.
- Lequel choisir ? 

Sources

1. Maxime Amblard & Christine Leininger. (2019) *Famille « Algorithmes & programmation »*. <https://interstices.info/famille-algorithmes-programmation/> (Consulté le 27/01/24)
2. Wikipédien^{ne}s. *Analyse de la complexité des algorithmes*. https://fr.wikipedia.org/wiki/Analyse_de_la_complexit%C3%A9_des_algorithmes (Consulté le 10/02/24)
2. Ouest France. *Celui qui croit que la croissance....Citation du jour.* <https://citations.ouest-france.fr/citation-kenneth-ewart-boulding/celui-croit-croissance-peut-etre-123503.html> (Consulté le 10/02/24)