Chapitre n°4

Algorithmique

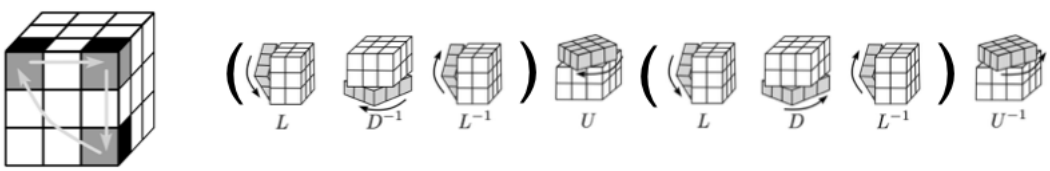
Table des matières

1. Notion d'algorithme

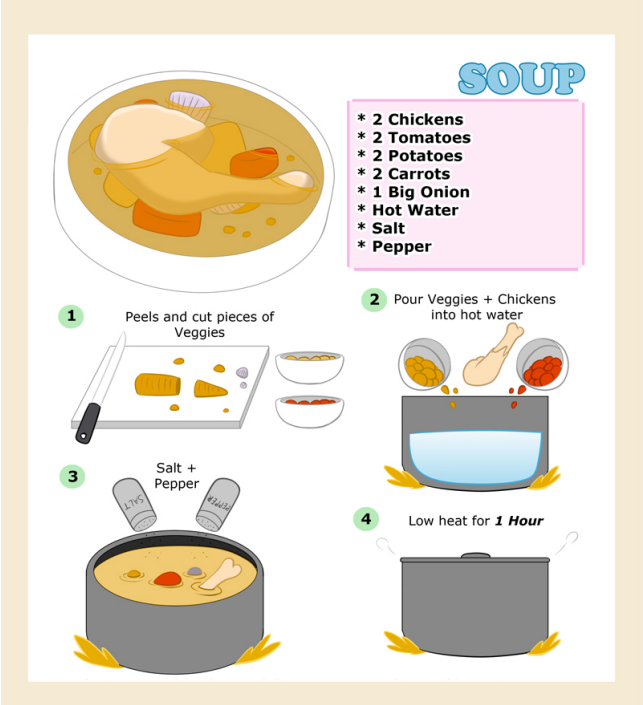
Un **algorithme** est, de manière générale, une suite d'instructions simples permettant de résoudre un problème donné. Le problème à résoudre peut dépendre d'un jeu de données qu'on représente par des **entrées**. Lors de son exécution, l'algorithme retourne une **sortie** constituant la réponse au problème posé.

Le concept d'algorithme dépasse largement le cadre de l'informatique et s'applique à de nombreuses situations de la vie de tous les jours – voici quelques exemples d'algorithmes.

- la résolution du Rubik's cube : le problème à résoudre est l'obtention d'un cube où chaque face affiche une unique couleur, l'entrée est la configuration de départ du cube, et la sortie est le cube correctement reconstitué.



- une recette de cuisine : le problème à résoudre est la réalisation d'un plat donné, les entrées sont les ingrédients utilisés dans la recette, et la sortie est le plat préparé à l'issue de la recette.



En programmation, le **code** n'est que la traduction de l'algorithme dans le **langage de programmation** choisi (Python, dans notre cas). Le code s'appuie alors sur une syntaxe et des règles d'écriture propres au langage choisi : pour Python, il faut respecter l'indentation, on distingue entiers et flottants, on peut utiliser des fonctions, des boucles for et while, des blocs if, etc.

L'algorithmique consiste à étudier l'algorithme même, sans se préoccuper du code (i.e. de son implémentation dans un langage de programmation quelconque). Pour étudier un algorithme en dehors de tout langage informatique, on peut utiliser une écriture en **pseudo-code**, qui est une traduction des opérations à effectuer sous une forme *quasi*-naturelle.

Ex : voici, par exemple, une manière d'écrire le pseudo-code permettant de déterminer si un entier est divisible ou non par 9 :

ALGORITHME : critère de divisibilité par 9

Entrée : n
Sortie : b

```
Tant que (nombre de chiffres de n) ⩾ 2 faire :
    n ⟵ somme des chiffres de n
Si n = 9 alors :
    b ⟵ VRAI
Sinon :
    b ⟵ FAUX
Retourner b
```

2. Terminaison

2.1. Problème de la terminaison d'un algorithme

La première question à se poser face à un algorithme donné est celui de la **terminaison** : « Est-ce que, pour une entrée quelconque, l'exécution de cet algorithme conduit à un arrêt après un nombre fini d'étapes ? ».

Il existe des algorithmes dont on ne sait pas dire s'il termine ou non, comme la célèbre suite de Syracuse.

Ex :

L'algorithme ci-contre termine lorsqu'on arrive à $n = 1$ au bout d'un nombre fini d'étapes.

On a prouvé par la force brute que pour toute entrée $n \leq 2^{68} \approx 10^{20}$, l'algorithme Syracuse termine. Mais on ne sait toujours pas le prouver pour tout entier n. De plus, il est possible que, même si c'était vrai, on ne puisse pas le prouver (on dit que le problème est indécidable).

Algorithme : Syracuse

```
Entrée : n
Sortie : ∅
1 Tant que n > 1 faire :
2     Si n est pair alors :
3         n ⟵ n/2
4     Sinon :
5         n ⟵ 3n+1
6 Retourner ∅
```

Dans le cadre du cours de CPGE, on se limitera aux cas simples d'algorithme dont on peut démontrer la terminaison.

On comprend assez facilement que les blocs « **si** » (et ses variantes « **sinon** », etc.) ainsi que les boucles « **pour** » ne sont pas des obstacles à la terminaison (sauf cas très pathologiques...). Les deux principales structures qui peuvent poser un problème de terminaison sont :

- Les boucles « **tant que** », car contrairement à « pour », on ne peut pas savoir à l'avance combien de fois cette boucle sera répétée. En cas de boucle infinie, l'algorithme ne termine pas. Il faut donc s'assurer que la condition de cette boucle sera fausse après un nombre fini d'étapes.
- Les **fonctions récursives**, qui sont des fonctions qui s'appellent elles-mêmes. Sans condition d'arrêt, elles peuvent s'appeler indéfiniement et l'algorithme ne termine pas. Cette notion sera vue plus loin dans ce chapitre.

Ex : l'algorithme `decompte` ci-dessous ne termine pas si la valeur n en entrée est négative :

Algorithme : decompte

```
Entrée : n
Sortie : ∅
1 Tant que n ≠ 0 faire :
2     n ⟵ n − 1
3 Retourner ∅
```

2.2. Preuve de la terminaison (algorithme itératif)

Démontrer qu'un algorithme itératif (donc sans fonction récursive) termine consiste à montrer que chacune de ses boucles « tant que » se termine quelle que soit les arguments donnés en entrée. Cette preuve repose sur la notion de **variant de boucle**.

Définition : variant de boucle

Un variant de boucle est une quantité construite à partir des variables utilisées dans la boucle, et qui est *entière*, *positive*, et *strictement décroissante* à chaque itération de la boucle.

Si un tel variant de boucle existe, alors la boucle « Tant que » associée se termine nécessairement. Ainsi, prouver la terminaison revient dans ce cas à exhiber un variant de boucle. Très souvent, le variant de boucle est construit à partir des variables qui apparaissent dans la condition de la boucle « tant que ».

Pour les preuves de terminaison, étant donné une variable `a`, il est très pratique et fréquent d'appeler a_k la valeur de `a` après k itérations de la boucle « tant que ». Auquel cas a_0 représente la valeur dans `a` juste avant l'entrée dans la boucle.

Application

On considère l'algorithme suivant, permettant de déterminer la position d'un élément dans une liste :

Algorithme : decomposition

Entrée : L,a
Sortie : i

```
1  n ⟵ nombre d'éléments de L
2  i ⟵ 0
3  trouve ⟵ FAUX
4  Tant que i < n et trouve = FAUX faire :
5  |  Si L(i) = a alors :
6  |  |  trouve ⟵ VRAI
7  |  i ⟵ i + 1
8  Si trouve = VRAI alors :
9  |  Retourner i
10 Sinon :
11 |  Retourner None
```

Déterminer un variant de boucle, noté v, très simple, construit à partir des variables `i` et `n` pour montrer la terminaison de la boucle « tant que » de cet algorithme.

S'il y a plusieurs boucles « tant que » imbriquées, il faut commencer par prouver la terminaison de la boucle la plus indentée (mais ce cas est assez rare en pratique).

2.3. Preuves de la terminaison (algorithme récursif)

Comme dit plus haut, une fonction récursive est une fonction qui s'appelle elle-même au cours de son exécution. L'exemple le plus classique étant le calcul d'une factorielle :

Ex : on considère l'algorithme suivant qui calcule la factorielle d'un entier naturel n :

Algorithme : `factorielle`

Entrée : `n` : entier naturel
Sortie : la factorielle de `n`

```
1 Si n ≤ 1 alors :
2 |  Retourner 1
3 Sinon :
4 |  Retourner n × factorielle(n-1)
```

Ainsi, lorsqu'on exécute `factorielle(3)`... :

- Comme `3 > 1`, `factorielle(3)` renvoie `3 * factorielle(2)`. Il faut encore calculer `factorielle(2)`.
- Comme `2 > 1`, `factorielle(2)` renvoie `2 * factorielle(1)`. Il faut encore calculer `factorielle(1)`.
- Comme `1` $\leq$ `1`, `factorielle(1)` renvoie `1`.

Finalement, `factorielle(3)` renvoie `3 * 2 * 1`, càd `6`.

La terminaison d'un algorithme récursif se prouve presque systématiquement par récurrence.

Application

Montrer que l'algorithme `factorielle` ci-dessus termine (i.e. que quel que soit l'entrée `n`, l'algorithme retourne un résultat après un nombre fini d'opérations).

3. Correction

3.1. Problème de la correction d'un algorithme

Obtenir un algorithme qui termine n'est absolument pas suffisant pour une utilisation de celui-ci : en effet, la terminaison d'un algorithme ne garantit en rien que le résultat obtenu soit cohérent avec la réponse au problème posé.

Ex : on considère l'algorithme suivant permettant de déterminer la première décimale d d'un réel x (avec $\lfloor \cdot \rfloor$ l'opérateur « partie entière ») :

Algorithme : premiere_decimale

Entrée : x
Sortie : d

```
1 d ⟵ ⌊ 10 × (x − ⌊x⌋) ⌋
2 Retourner d
```

Cet algorithme est constitué d'une séquence finie d'instructions : il termine donc de manière triviale. On peut vérifier que cet algorithme retourne bien la valeur attendue sur l'exemple $x = \pi$:

$$\begin{aligned}\texttt{premiere\_decimale}(\pi) &= \lfloor 10 \times (\pi - \lfloor \pi \rfloor) \rfloor \\ &= \lfloor 10 \times (3,1415... - \lfloor 3,1415... \rfloor) \rfloor \\ &= \lfloor 10 \times (3,1415... - 3) \rfloor \\ &= \lfloor 1,415... \rfloor \\ &= 1\end{aligned}$$

Cependant, cet algorithme ne retourne pas la valeur attendue pour une valeur de x négative :

$$\begin{aligned}\texttt{premiere\_decimale}(-4,2) &= \lfloor 10 \times ((-4,2) - \lfloor (-4,2) \rfloor) \rfloor \\ &= \lfloor 10 \times (-4,2+5) \rfloor \\ &= \lfloor 8 \rfloor \\ &= 8 \neq 2\end{aligned}$$

Si l'algorithme est purement séquentiel (ou s'il contient des blocs « si » et ses variantes), il est possible de vérifier « à la main » le contenu des différentes variables au fur et à mesure de l'exécution de celui-ci afin de confirmer qu'il produit (ou non) le résultat attendu. Cette démarche peut conduire à travailler par disjonction de cas, comme dans l'exemple ci-dessous.

Ex :
L'algorithme suivant permet, à partir d'une liste de trois réels a, b et c quelconques, d'obtenir la liste `L` triée par ordre croissant de ces trois réels :

Algorithme : `tri`

Entrée : a, b, c
Sortie : L

```
1 L ⟵ (a, b, c)
2 Si c < b alors :
3 |  inverser les éléments #2 et #3 de la liste L
4 Si c < a alors :
5 |  inverser les éléments #1 et #3 de la liste L
6 Si b < a alors :
7 |  inverser les éléments #1 et #2 de la liste L
8 Retourner L
```

On peut tester quelques entrées correspondant à des conditions différentes sur a, b et c :

- dans le cas $a \leqslant b \leqslant c$, seules les lignes 1 et 8 sont exécutées : l'algorithme retourne alors la liste $\texttt{tri}(a,b,c) = (a,b,c)$, ce qui correspond bien à la liste triée.
- dans le cas $b < a \leqslant c$, les lignes 1, 6, 7 et 8 sont exécutées (ce qui correspond à un échange des variables a et b) : l'algorithme retourne alors la liste $\texttt{tri}(a,b,c) = (b,a,c)$ ce qui correspond bien à la liste triée.
- etc.

On peut vérifier, par disjonction de cas, que toute entrée possible conduit à l'obtention d'une liste correctement triée.

Le seul problème est celui des boucles : en effet, il faudrait faire une vérification quel que soit le nombre d'itérations, qui peut être arbitrairement grand en fonction de l'entrée, cf plus loin.

3.2. Preuve de correction

La correction d'une boucle se démontre par récurrence en utilisant un **invariant de boucle**.

Définition : invariant de boucle

Un invariant de boucle est une *assertion* construite à partir des variables utilisées dans la boucle, et qui reste vraie du début à la fin de l'exécution de la boucle.

Méthodologie : démonstration de la correction d'une boucle

La démonstration de la correction d'une boucle repose sur la méthodologie suivante :

1. **construire un invariant de boucle à partir des variables utilisées dans la boucle ;**
2. **initialisation : démontrer que l'invariant de boucle est vrai avant d'entrer dans la boucle ;**
3. **hérédité : démontrer que, si l'invariant de boucle est vrai au début de la $k^{\text{ième}}$ itération de la boucle, alors l'invariant est toujours vrai à l'issue de la $k^{\text{ième}}$ itération de la boucle ;**
4. **vérification : démontrer que l'invariant de boucle répond bien au résultat attendu à la fin de la boucle.**

Application

Réaliser la division euclidienne de l'entier naturel $a \in \mathbb{N}$ par l'entier naturel (non-nul) $b \in \mathbb{N}^*$ revient à réécrire a (de manière unique) sous la forme :

$$a = b \times Q + R$$

où $Q \in \mathbb{N}$ désigne le quotient de la division euclidienne de a par b et $R \in \mathbb{N}$, avec $R \leqslant b$ désigne le reste dans la division euclidienne de a par b.

On considère l'algorithme de division euclidienne suivant :

Algorithme : division_euclidienne

Entrée : a, b
Sortie : Q, R

```
1 R ⟵ a
2 Q ⟵ 0
3 Tant que R ⩾ b faire :
4 |   R ⟵ R − b
5 |   Q ⟵ Q + 1
6 Retourner (Q, R)
```

On choisit l'invariant de boucle suivant :

$$\mathcal{P}_k \ : \qquad a = b \times \mathtt{Q}_k + \mathtt{R}_k$$

avec $\texttt{x}_k$ la valeur stockée dans la variable `x` à la fin de la $k^{\text{ième}}$ itération. Par convention, on note $\texttt{x}_0$ la valeur stockée dans la variable `x` avant d'entrer dans la boucle.

▶ Montrer que l'invariant de boucle est vrai avant d'entrer dans la boucle (initialisation).

▶ Montrer que, si l'invariant de boucle est vrai à la fin de la $k^{\text{ième}}$ itération, alors il est également vrai à la fin de la $(k+1)^{\text{ième}}$ itération (hérédité).

▶ Montrer que, à l'issue de la boucle, les variables `Q` et `R` contiennent bien les valeurs attendues.

Démontrer la correction (seule) d'un algorithme ne constitue qu'une preuve de **correction partielle** ; prouver la **correction totale** d'un algorithme demande de prouver à la fois sa terminaison et sa correction (partielle).

4. Complexité

4.1. Notion de complexité

La **complexité** d'un algorithme est une mesure de la quantité de ressources nécessaires pour mener à bien son exécution. La complexité d'un algorithme est une fonction de la **taille d'instance** (on rappelle qu'une instance correspond à la valeur d'une entrée particulière, par exemple 3 pour `factorielle`).

Définition : taille d'instance

La taille d'instance est un nombre caractéristique de l'instance utilisée. Elle est choisie de manière à croître lorsque l'objet manipulé devient plus « volumineux ». Ainsi, on peut choisir les tailles d'instances suivantes :

- **pour un entier ou un flottant : le nombre lui-même, ou le nombre de chiffres utilisés pour représenter ce nombre en binaire ;**
- **pour une séquence (liste, tuple, etc.) : le nombre d'éléments constituant la séquence ;**
- **pour un tableau ou une matrice : le nombre de coefficients, ou encore le nombre de lignes / de colonnes ;**
- **etc.**

Ex : selon ce qu'on souhaite tester, l'entrée `42` pourra être représentée par la taille d'instance 42 (nombre lui-même), 6 (nombre de chiffres dans la représentation de ce nombre en binaire[1], à savoir $\overline{101010}^{(2)}$, etc.

On peut considérer deux mesures différentes de la complexité :

- la **complexité temporelle** (complexité en temps) est une mesure de l'évolution du temps de calcul nécessaire à l'exécution d'un algorithme en fonction de la taille d'instance ;
- la **complexité spatiale** (complexité en espace) est une mesure de l'évolution du nombre de cases mémoire requises lors de l'exécution d'un algorithme en fonction de la taille d'instance.

Dans le cadre de ce cours, on s'intéressera principalement à la complexité en temps.

Ex : on considère la complexité en temps $\mathcal{C}$ d'un algorithme `decomposition` qui consiste à écrire un entier naturel non-nul n sous la forme :

$$n = 2^i \times m$$

avec i un entier naturel et m un entier naturel impair. Pour ce faire, on trace la graphe $\Delta t = f(n)$ obtenu en mesurant le temps d'exécution Δt de l'algorithme `decomposition`[2] en fonction de la valeur de l'entier n :

1. Voir le chapitre dédié (« Numération »).
2. En fait on mesure le temps d'exécution de 10^6 fois cet algorithme, afin que les durées mises en jeu soient mesurables et qu'on puisse négligeer les petites variations dues au processeur.

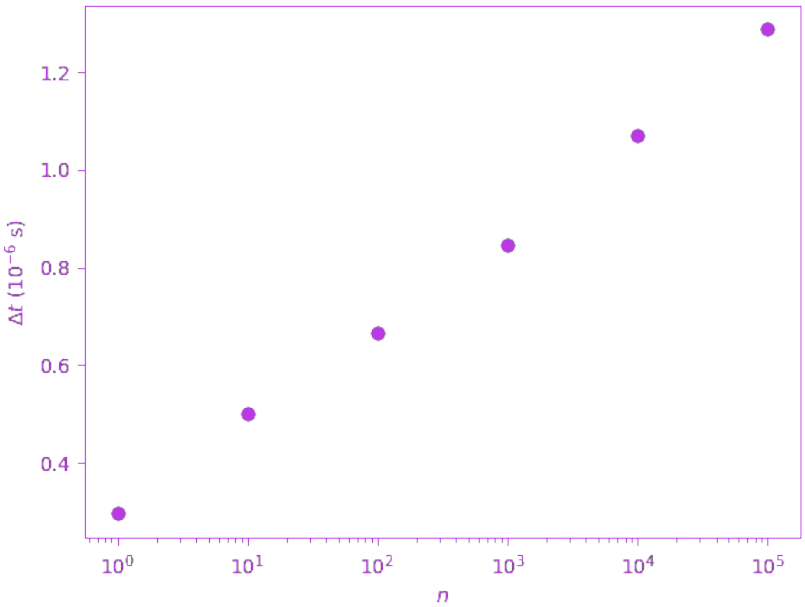


Fig. 1 – temps d'exécution de l'algorithme `decomposition` en fonction de la valeur de l'entier n (affichage en échelle semi-logarithmique)

Le temps requis pour exécuter un algorithme est intimement lié à deux paramètres :

- le nombre d'**opérations élémentaire** requises pour mener à bien l'algorithme ;
- le temps nécessaire à la machine utilisée pour réaliser *une* opération élémentaire.

Le concept d' « opération élémentaire » est relativement flou et dépend du contexte utilisé. Par exemple, on considère usuellement que l'addition de deux entiers est une opération élémentaire. Cependant, nous verrons par la suite que l'addition de deux "grands" entiers prend plus de temps que pour deux "petits" entiers[3]. Un processeur réalise une "grosse" addition par une succession de "petites" additions, dont chacune peut être considérée comme une opération élémentaire.

Méthodologie : opérations élémentaires

On peut *techniquement* considérer comme *une* opération élémentaire chacun des éléments suivants :

- **Les opérations arithmétiques élémentaires sur les entiers ou les flottants : « `+` », « `-` », « `*` », « `/` », division euclidienne (« `//` » pour le quotient et « `%` » pour le reste), exponentiation (« `**` »), etc.**
- **Les opérations de comparaison : égalité (« `==` »), différence (« `!=` »), supériorité (« `>` » et « `>=` », infériorité (« `<` » et « `<=` »), etc.**
- **Les opérations logiques : appartenance (« `in` »), coïncidence (« `is` »), conjonction (« `and` »), disjonction (« `or` »), négation (« `not` »), etc.**
- **L'affectation d'une variable (« `=` ») ;**
- **L'évaluation d'un élément d'une liste, i.e. `L[k]`.**
- **L'appel de fonctions natives de Python : `print`, `range`, `return`, etc. ;**
- **etc.**

3. Plus précisément, cela dépend du nombre de *bits* nécessaire pour représenter chacun de ces entiers en binaire.

Ces conventions sont (en partie) *arbitraires*, mais nous verrons par la suite que modifier ce qu'on considère ou non comme étant une opération élémentaire n'a, sauf exception, pas de conséquence sur la complexité d'un algorithme. Il faut rappeler (rapidement) les conventions choisies lorsqu'on réalise un calcul de complexité lors d'une épreuve (parfois cette convention peut vous être imposée).

Prendre les deux premiers items comme on l'a fait jusqu'à présent mène à une complexité correcte pour la grande majorité des algorithmes, mais pas tous :

Algorithme : `creer_liste_1...n`

Entrée : `n`
Sortie : `L`

```
L ⟵ liste vide
Pour k dans ⟦1, n⟧ faire :
    Ajouter l'élément k à L
Retourner L
```

Application

On considère l'algorithme suivant :

Algorithme : `algo_mystere`

Entrée : `L`
Sortie : `s`

```
n ⟵ nombre d'éléments de la liste L
s ⟵ 0
Pour k dans ⟦0, n − 1⟧ faire :
    s ⟵ s + L(k)
Retourner s
```

- Quel est le problème résolu par l'algorithme `algo_mystere` ?
- Proposer une taille d'instance pour cet algorithme.
- Déterminer la complexité (temporelle) $\mathcal{C}$ de cet algorithme.

Selon l'instance fournie à l'algorithme, la complexité peut – dans certains cas – grandement varier. On peut donc affiner la notion de complexité en proposant différentes mesures de celle-ci :

- La **complexité dans le pire cas** correspond à la complexité maximale pour une taille d'instance donnée.
- La **complexité dans le meilleur cas** correspond à la complexité minimale pour une taille d'instance donnée.
- La **complexité moyenne** correspond à la complexité moyenne obtenue en considérant toutes les instances possibles correspondant à une même taille d'instance donnée.

La notion la plus significative est la complexité dans le pire cas : c'est celle qu'on sous-entend quand on demande d'évaluer la complexité d'un algorithme.

Application

On considère l'algorithme suivant permettant de trouver, dans une liste `L` d'entiers naturels non-nuls, le rang du premier élément pair :

Algorithme : `rang_premier_entier_pair`

Entrée : `L`
Sortie : `rang`

```
1 n ⟵ nombre d'éléments de la liste L
2 rang ⟵ +∞
3 k ⟵ 0
4 Tant que k < n et rang = +∞ faire :
5     Si (reste dans la division euclidienne de L(k) par 2) = 0 alors :
6         rang ⟵ k
7     Sinon :
8         k ⟵ k + 1
9 Retourner rang
```

▶ Quel est le comportement de cet algorithme si la liste L fournie en entrée ne comporte aucun entier pair ?

▶ Quel est le « meilleur cas » pour l'exécution de cet algorithme en prenant une instance de taille n ? Déterminer la complexité de cet algorithme dans le meilleur cas.

▶ Quel est le « pire cas » pour l'exécution de cet algorithme en prenant une instance de taille n ? Déterminer la complexité de cet algorithme dans le pire cas.

4.2. Comportement asymptotique

Une question centrale qu'un programmeur est amené à se poser est la suivante : « Comment évolue la complexité d'un algorithme en fonction de la taille d'instance ? » – donc : « Comment évoluera la durée d'exécution du programme associé lorsque la taille d'instance augmente ? ». On peut illustrer l'importance de cette question en considérant l'évolution de quelques complexités typiques (exprimées en nombre d'opérations élémentaires) en fonction de la taille d'instance n :

n	10	20	50	100	1 000
$\mathcal{C}(n) = 1$	1	1	1	1	1
$\mathcal{C}(n) = \log(n)$	1	~ 1	~ 2	2	3
$\mathcal{C}(n) = n$	10	20	50	100	1 000
$\mathcal{C}(n) = n^2$	100	400	2 500	10^4	$\sim 10^6$
$\mathcal{C}(n) = 2^n$	1 024	$\sim 10^6$	$\sim 10^{15}$	$\sim 10^{30}$	$\sim 10^{300}$
$\mathcal{C}(n) = n!$	$\sim 10^6$	$\sim 10^{18}$	$\sim 10^{64}$	$\sim 10^{157}$	$\sim 10^{2\,567}$

Sachant qu'un processeur générique présente[4] une fréquence d'horloge typique de l'ordre du GHz – c'est-à-dire qu'un processeur est capable de traiter de l'ordre de 10^9 opérations élémentaires par seconde – l'exécution de programmes correspondant aux complexités présentées ci-dessus conduirait (en ordre de grandeur) aux durées d'exécution suivantes :

n	10	20	50	100	1 000
$\mathcal{C}(n) = 1$	1 ns	1 ns	1 ns	1 ns	1 ns
$\mathcal{C}(n) = \log(n)$	1 ns	$\sim$ 1 ns	$\sim$ 2 ns	2 ns	3 ns
$\mathcal{C}(n) = n$	10 ns	20 ns	50 ns	100 ns	1 μs
$\mathcal{C}(n) = n^2$	100 ns	400 ns	2,5 μs	10 μs	$\sim$ 1 ms
$\mathcal{C}(n) = 2^n$	$\sim$ 1 μs	$\sim$ 1 ms	$\sim$ 12 jours	$\sim 10^{13}$ ans	$\sim 10^{283}$ ans
$\mathcal{C}(n) = n!$	$\sim$ 1 ms	$\sim$ 32 ans	$\sim 10^{47}$ ans	$\sim 10^{140}$ ans	$\sim 10^{2\,550}$ ans

À titre de comparaison, on estime l'âge de l'Univers à un peu moins de $1{,}4 \cdot 10^{10}$ ans... Afin d'estimer la complexité asymptotique d'un algorithme, on utilise la **notation de Landau $\mathcal{O}$**.

Définition : notation de Landau $\mathcal{O}$

On dit qu'une fonction $f : \mathbb{R} \to \mathbb{R}$ est dominée par la fonction $g : \mathbb{R} \to \mathbb{R}$ en $+\infty$ si :

$$\exists C \in \mathbb{R}_+^* \quad \exists x_0 \in \mathbb{R} \quad \forall x > x_0 \qquad |f(x)| \leqslant C \times |g(x)|$$

On note alors :

$$f(x) = \mathcal{O}(g(x))$$

Ex : si on considère la fonction polynomiale P définie par :

$$P(x) = 4 \times x^2 + 2 \times x + 1$$

on peut écrire que : $P(x) = \mathcal{O}(x^2)$ car les termes $2 \times x$ et 1 deviennent négligeables devant le terme $4 \times x^2$ lorsque $x \longrightarrow +\infty$.

4. À ce jour de l'an 2021...

Le symbole $\mathcal{O}$ ne permet cependant pas de décrire correctement à quelle vitesse croit $P(x)$: en effet, on peut dire que $P(x)$ est aussi bien un $\mathcal{O}(x^2)$ qu'un $\mathcal{O}(x^3)$... On peut alors utiliser le **symbole Θ** pour préciser que la complexité obtenue est « en $\mathcal{O}(...)$, et pas mieux ».

Définition : notation de Landau Θ

On écrira $f(x) = \Theta(g(x))$ pour signifier que :

$$f(x) = \mathcal{O}(g(x)) \qquad \textbf{et} \qquad g(x) = \mathcal{O}(f(x))$$

soit :

$$\exists C_1, C_2 \in \mathbb{R}_+^* \quad \exists x_0 \in \mathbb{R} \quad \forall x > x_0 \qquad C_1 \times |g(x)| \leqslant |f(x)| \leqslant C_2 \times |g(x)|$$

On dit que les fonctions f et g sont du même ordre de grandeur.

Ex : si on considère de nouveau la fonction polynomiale $P(x) = 4 \times x^2 + 2 \times x + 1$, on peut écrire que :

$$\forall x > 3\,,\ 4 \times |x^2| \leqslant |P(x)| \leqslant 5 \times |x^2|$$

c'est-à-dire

$$P(x) = \Theta(x^2)$$

On peut alors identifier les différents algorithmes en fonction du comportement asymptotique de leur complexité (exprimée en notation de Landau) – classés ici des complexités les moins « gourmandes » aux complexités les plus contraignantes :

- un algorithme de **complexité constante** correspond à $\mathcal{C}(n) = \Theta(1)$;
- un algorithme de **complexité logarithmique** correspond à $\mathcal{C}(n) = \Theta(\log(n))$;
- un algorithme de **complexité linéaire** correspond à $\mathcal{C}(n) = \Theta(n)$;
- un algorithme de **complexité quasi-linéaire** correspond à $\mathcal{C}(n) = \Theta(n \times \log(n))$;
- un algorithme de **complexité quadratique** correspond à $\mathcal{C}(n) = \Theta(n^2)$;
- plus généralement, un algorithme de **complexité polynomiale** correspond à $\mathcal{C}(n) = \Theta(n^k)$ (avec $k \in \mathbb{N}, k \geqslant 2$) ;
- un algorithme de **complexité exponentielle** correspond à $\mathcal{C}(n) = \Theta(q^n)$ (avec $q > 1$) ;
- un algorithme de **complexité factorielle** correspond à $\mathcal{C}(n) = \Theta(n!)$.

D'un point de vue pratique, un algorithme de complexité exponentielle ou factorielle est impossible à mettre en œuvre en raison d'un temps d'exécution démesuré.