Introduction à la complexité algorithmique

Lydem - SSMIL

Décembre 2018

Sommaire

But du cours : pouvoir quantifier les performances en temps et en espace d'un algorithme et en comparer plusieurs, connaître les notations et leur signification mathématique.

Pré-requis : algorithmique de base (variables, boucles, fonctions), représentation graphique de fonctions mathématiques, syntaxe C++ de base.

1 Introduction

Lorsqu'on réalise un algorithme, il y a plusieurs choses que l'on souhaite ; La première, et pas des moindres, est sa validité. C'est-à-dire que quel que soit les valeurs en entrée, le résultat renvoyé est celui attendu.

```
bool estImpair(int x)
{
  return x % 2;
}
```

Algorithme 1: "x est-il impair ?"

% représente l'opération modulo, c'est-à-dire le reste de la division de x par 2. Si x est impair, le résultat de la division $x/2$ n'est pas un entier, donc x % 2 donne 1. La validité est confirmée !

La deuxième est l'efficacité de l'algorithme, que ce soit en temps ou en espace.

```
bool estImpair(int x)
{
  bool impair(0);
  for(int i(1); i <= x: ++i)
    impair = !impair;
  return impair;
}
```

Algorithme 2: "x est-il impair ?" (idiot)

Cet algorithme [2] réalise la même chose que l'algorithme [1] mais de manière plus atypique.

Avant d'évaluer les deux exemples plus haut, donnons les outils pour le faire.

2 Complexité en temps et Grand-O

La performance d'un algorithme ne peut être basé sur le temps de calcul, tout simplement car ce dernier dépend fortement de la machine sur laquelle il est exécuté, de la manière dont il est compilé ou interprété, etc. On va donc quantifier les performances par le nombre d'opérations qu'effectue la procédure.
On va représenter ce nombre d'opérations par un polynôme, qui sera en fonction de ou des entrées.

Par exemple, si on prend l'algorithme [2], on peut voir qu'on effectue deux déclarations (dont une dans la boucle Pour), x comparaisons (vérification de la borne sur i dans la boucle Pour), $2x$ affectations (incrémentation de i et affectation de impair une fois par tour) et un retour.
On considère la déclaration, l'affectation, l'incrémentation et la comparaison comme des opérations élémentaires, leur fonction de complexité sera une constante.
Le retour ne compte pas.

Au final, l'algorithme [2] effectue $3x + 2$ opérations, donc on peut définir sa fonction de complexité comme étant $f(x) = 3x + 2$.

Les notations de Landau permettent de classifier les fonctions de complexité. Prenons Grand-O (Grand Omicron), la notation de Landau la plus usitée. Elle consiste à majorer "asymptotiquement" la fonction de complexité.
En d'autres termes, il s'agit de choisir une fonction qui, à partir d'une valeur, sera constamment "au dessus" (courbe plus haute) de la fonction initiale, et ce avec au moins un facteur strictement positif.

De manière mathématique, cela signifie que $f(x) \in O(g(x))$ si et seulement si $\exists k > 0 \ tq \ \forall x > x0, |f(x)| \leq k|g(x)|$, $g(x)$ étant la fonction majorante.

Il est important d'essayer de se rapprocher au maximum de la fonction de complexité. En effet, même s'il est vrai qu'une fonction exponentielle majore une fonction linéaire, cela n'apporte aucune information viable quant à l'efficacité de l'algorithme.

Par exemple, si on prend notre $f(x) = 3x + 2$, on remarque de $g(x) = kx$ majore bien $f(x)$, et ce avec n'importe quel $k > 3$ (car à partir d'une valeur de x, $g(x)$ sera supérieur à $f(x)$). Notre algorithme a donc une complexité en $O(x)$.

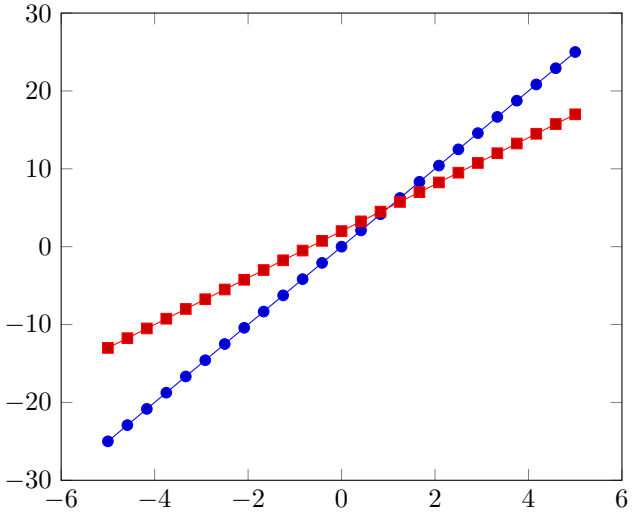

Figure 1: $\forall x \geq 1$, $g(x) = kx$ (courbe bleue ; Avec $k = 5$) dépasse $f(x) = 3x + 2$ (courbe rouge)

Pour l'algorithme [1], c'est différent.
En effet, lorsqu'on évalue un algorithme, on va souvent considèrer les additions, multiplications et divisions comme étant des opérations élémentaires, donc en temps constant, c'est-à-dire en $O(1)$.

Cependant, cela n'est qu'une simplification qui est effectuée en fonction de l'utilité des programmes, ainsi que de la taille des nombres à manipuler.
En effet, les opérations élémentaires (dont le modulo) sont bien soumises à un paramètre : la taille de l'entier donné ; Plus l'entier sera "grand" (plus il sera composé de chiffres), plus l'opération sera longue.

Theorem 3.3. *Let a and b be arbitrary integers.*

(i) We can compute $a \pm b$ in time $O(\text{len}(a) + \text{len}(b))$.

(ii) We can compute $a \cdot b$ in time $O(\text{len}(a)\,\text{len}(b))$.

(iii) If $b \neq 0$, we can compute the quotient $q := \lfloor a/b \rfloor$ and the remainder $r := a \bmod b$ in time $O(\text{len}(b)\,\text{len}(q))$.

Figure 2: A Computational Introduction to Number Theory and Algebra, Victor Shoup

Si on se base sur le théorème énoncé par Victor Shoup, on peut dire que la fonction de complexité de l'opération $x \ \% \ p$ est
$f(x, p) = taille(x/p) * taille(p)$, où (x/p) est la partie entière de la division.

Dans notre cas, avec x % 2,
$f(x) = taille(x/2) * taille(2) = taille(x/2)$ (car $taille(2) = 1$).

Si on définit la taille en base décimale telle que $taille(x) = log10(x)+1$, notre fonction de complexité est donc $f(x) = log10(x/2) + 1 = log10(x) + log10(5)$ (en simplifiant).
On peut voir que la fonction $g(x) = k * log10(x)$ majore $f(x)$, et ce avec n'importe quel $k > 1$. La complexité de l'algorithme est donc en $O(log10(x))$.

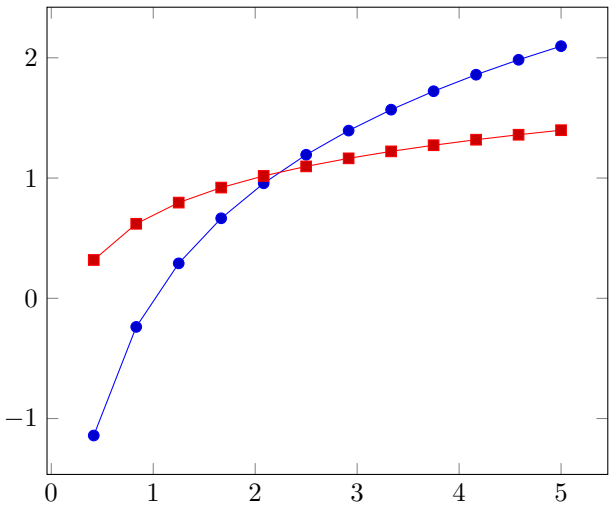

Figure 3: $\forall x \geq 2.4$, $g(x) = k * log10(x)$ (courbe bleue ; Avec $k = 3$) dépasse $f(x) = log10(x) + log10(5)$ (courbe rouge)

Mais du coup, comment comparer les deux algorithmes ? Il suffit de comparer leur fonction majorante respective.

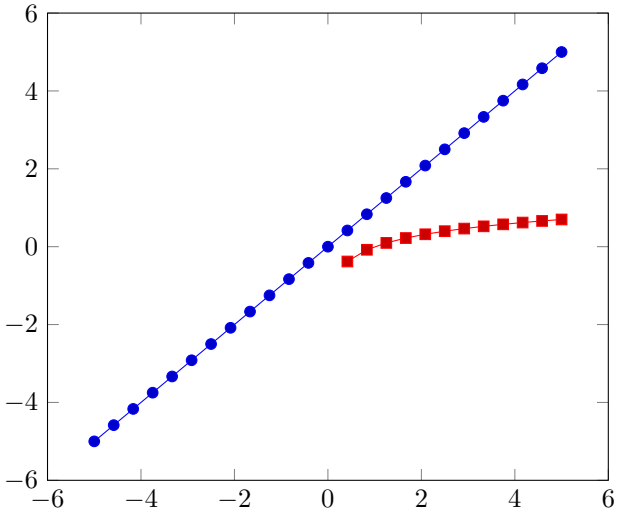

Figure 4: La fonction identité $f(x) = x$ (courbe bleue) est bien au dessus de $g(x) = log10(x)$ (courbe rouge), l'algorithme [1] est donc le plus optimal

Pour se faire une idée : avec $x = 18000$, il faut $log10(18000) \simeq 4$ opérations pour l'algorithme [1] et 18000 opérations pour l'algorithme [2].

On a donc montré qu'au moyen des fonctions de complexité, il y a moyen de savoir si un algorithme est plus ou moins efficace qu'un autre.
Toutefois, on peut aussi de manière indépendante savoir si un algorithme est performant selon le polynôme qui le majore.

$O(1)$	constante
$O(logx(n))$	logarithmique en base x
$O(n)$	linéaire
$O(n * logx(n))$	linéarithmique en base x
$O(n^k)$ (avec un $k > 1$ fixé)	polynomiale
$O(2^n)$	exponentielle

Table 1: Complexités les plus rencontrées, de la plus à la moins performante

3 Les notations de Landau

Nous avons introduit la notation Grand-O qui est la plus utilisée mais il existe d'autres notations de Landau :

→ Ω (Grand Oméga) qui représente une fonction non pas majorante (comme Grand-O) mais minorante, c'est-à-dire qu'il faut une fonction telle que, à partir d'une valeur, celle-ci soit constamment "en dessous" de la fonction de complexité, et ce avec au moins un facteur strictement positif.

De manière mathématique, cela signifie que $f(x) \in \Omega(g(x))$ si et seulement si $\exists k > 0 \ tq \ \forall x > x0, k|f(x)| \geq |g(x)|$, $g(x)$ étant la fonction minorante.

Reprenons les algorithmes [1] et [2].

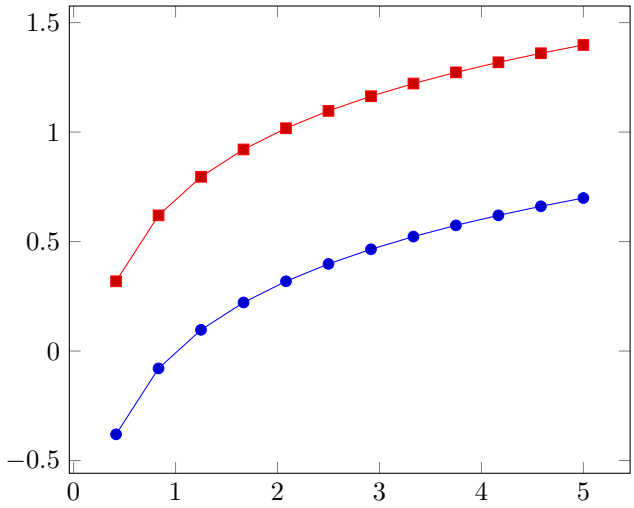

Figure 5: $\forall x \geq 0, \ f(x) = k(log10(x) + log10(5))$ (courbe rouge ; Avec $k = 1$) dépasse $g(x) = log10(x)$ (courbe bleue)

Donc l'algorithme [1] est en $\Omega(log10(x))$.

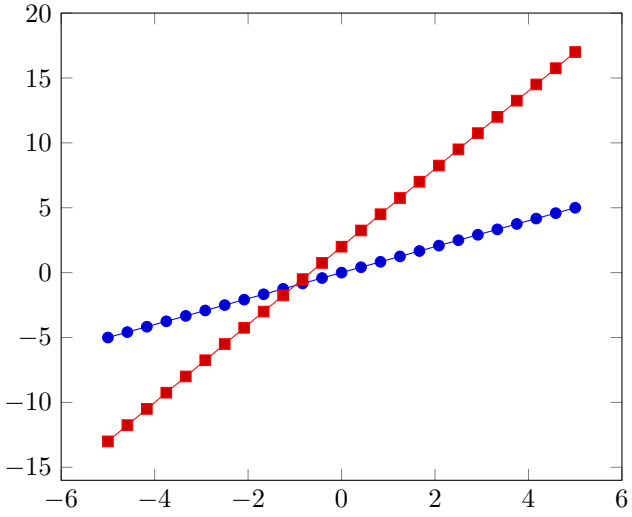

Figure 6: $\forall x \geq 0$, $f(x) = k(3x+2)$ (courbe rouge ; Avec $k = 1$) dépasse $g(x) = x$ (courbe bleue)

Donc l'algorithme [2] est en $\Omega(x)$.

→ Θ (Grand Thêta) qui représente une mesure plus précise de la complexité que la notation Grand-O.
En effet, comme dit précédemment, Grand-O peut être plus ou moins précis selon ce qu'on choisit comme fonction majorante (2^x majore 1, c'est vrai mais inutile).
Grand Thêta conserve donc cette notion de majorante mais rajoute la notion de minorante, et ce par la même fonction $g(x)$.
Il faut donc une fonction telle que, à partir d'une valeur, celle-ci soit constamment "au dessus" avec un facteur strictement positif et "en dessous" avec un autre facteur strictement positif de la fonction de complexité.

De manière mathématique, cela signifie que $f(x) \in \Theta(g(x))$ si et seulement si $\exists k0, k1 > 0 \ tq \ \forall x > x0, k0|g(x)| \leq |f(x)| \leq k1|g(x)|$, $g(x)$ étant la fonction majorante et minorante.

Les observateurs auront remarqué que la notation Grand Thêta est une union entre la notation Grand-O et la notation Grand Oméga.
En effet, un algorithme a une complexité en $\Theta(g(x))$ si et seulement s'il a une complexité en $O(g(x))$ et en $\Omega(g(x))$.

Reprenons les algorithmes [1] et [2].

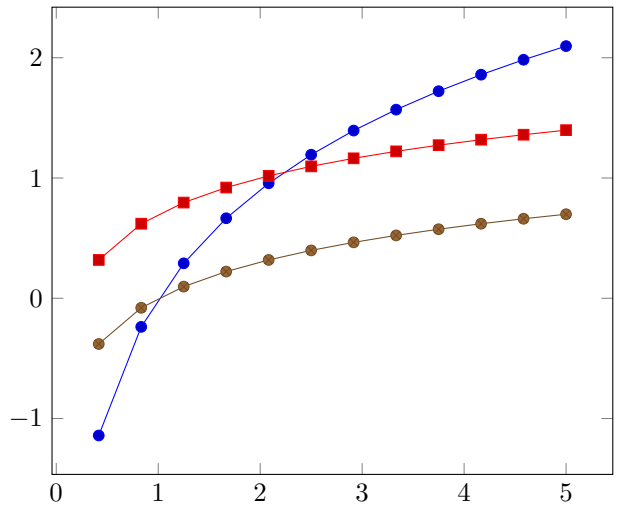

Figure 7: $\forall x \geq 2.4$, $g(x) = k * log10(x)$ (courbe bleue ; Avec $k = 3$) dépasse $f(x) = log10(x) + log10(5)$ (courbe rouge) et, $\forall x \geq 0$, $f(x)$ dépasse $h(x) = k * log10(x)$ (courbe marron ; Avec $k = 1$)

Donc l'algorithme [1] est en $\Theta(log10(x))$.
Ce résultat est logique car cet algorithme est en $O(log10(x))$ et en $\Omega(log10(x))$.

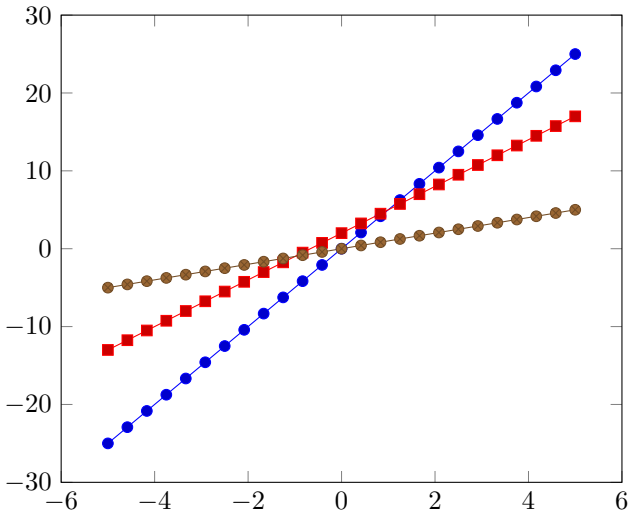

Figure 8: $\forall x \geq 1$, $g(x) = kx$ (courbe bleue ; Avec $k = 5$) dépasse $f(x) = 3x + 2$ (courbe rouge) et $\forall x \geq 0$, $f(x)$ dépasse $h(x) = kx$ (courbe marron ; Avec $k = 1$)

Donc l'algorithme [2] est en $\Theta(x)$.

Ce résultat est logique car cet algorithme est en $O(x)$ et en $\Omega(x)$.

→ o (Petit-o) (resp. ω (Petit Oméga)) qui est assez similaire à son grand frère, mis à part qu'il exprime la domination (resp. la soumission) de la fonction de complexité à une autre fonction, c'est-à-dire qu'il faut une fonction telle que, à partir d'une valeur, celle-ci soit constamment "au dessus" (resp. "en dessous") de la fonction de complexité, et ce quelque soit le facteur.

De manière mathématique, cela signifie que $f(x) \in o(g(x))$ (resp. $\omega(g(x))$) si et seulement si $\forall k > 0, \forall x > x0, |f(x)| \leq k|g(x)|$ (resp. $k|f(x)| \geq |g(x)|$).
Par définition, un algorithme en $\Theta(g(x))$ ne peut être ni en $o(g(x))$ ni en $\omega(g(x))$.

Reprenons les algorithmes [1] et [2].

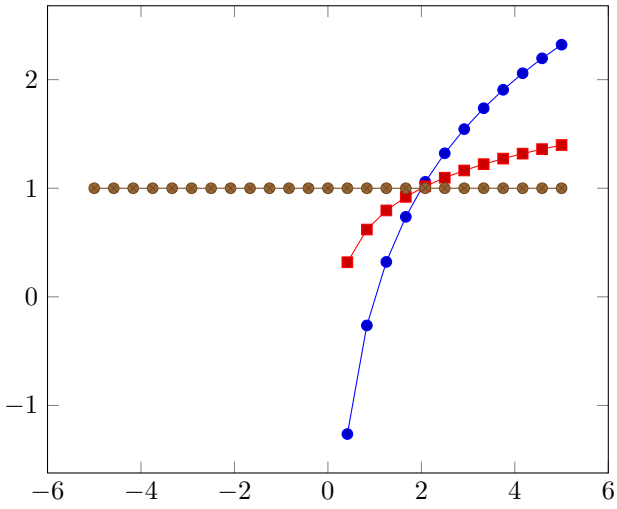

Figure 9: $\forall k > 0$, $g(x) = k * log2(x)$ (courbe bleue) dépassera $f(x) = log10(x) + log10(5)$ (courbe rouge) à partir d'un certain x, et $f(x)$ dépassera $h(x) = k$ (courbe marron) à partir d'un certain x

Donc l'algorithme [1] est en $o(log2(x))$ et en $\omega(1)$.

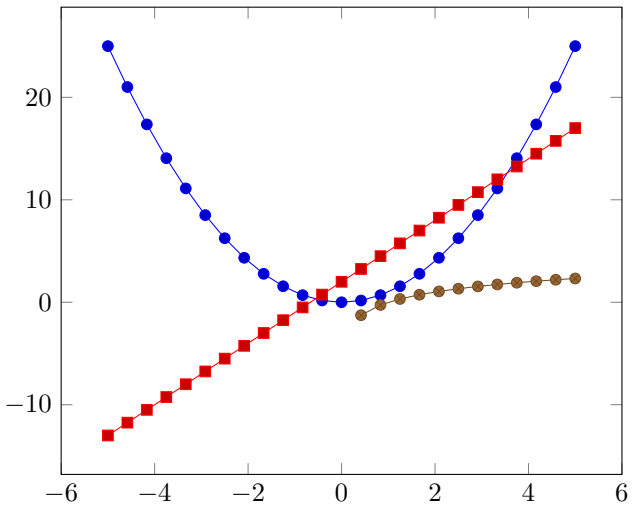

Figure 10: $\forall k > 0$, $g(x) = kx^2$ (courbe bleue) dépassera $f(x) = 3x + 2$ (courbe rouge) à partir d'un certain x, et $f(x)$ dépassera $h(x) = k * log2(x)$ (courbe marron) à partir d'un certain x

Donc l'algorithme [2] est en $o(x^2)$ et en $\omega(log2(x))$.

→ $\sim$ (tilde) est une notation signifiant l'équivalence, c'est-à-dire que "$f(x) \sim g(x)$" est synonyme de "$f(x)$ est similaire à $g(x)$".

De manière mathématique, $f(x) \sim g(x)$ si et seulement si $\forall k > 0, \forall x > x0, |f(x) - g(x)| \leq k|g(x)|$.

→ $\widetilde{O}$ (soft-O) qui représente une variante de la notation Grand-O qui ignore les facteurs logarithmiques, c'est-à-dire que les complexités linéarithmiques deviennent linéaires (cf. table [1]).
Par exemple, un algorithme de tri efficace avec une complexité en $O(n * log2(n))$ a une complexité en $\widetilde{O}(n)$.

De manière mathématique, cela signifie que $f(x) \in \widetilde{O}(g(x))$ si et seulement si $\exists k > 0\ tq\ \forall x > x0, |f(x)| \leq |g(x)|log^k(|g(x)|)$.

Le but n'est pas de connaître toutes ces notations sur le bout des doigts mais de comprendre comment elle fonctionne et pouvoir s'y retrouver si on les retrouve dans des ouvrages ou de la documentation.
On retrouve majoritairement la notation Grand-O, Grand Oméga et Grand Thêta en informatique, et Petit-o en mathématique.

4 Complexité dans le meilleur/pire des cas, et moyenne

Jusqu'à maintenant, les seuls algorithmes présentés avaient une complexité constante, c'est-à-dire que quelque soit l'entrée la fonction de complexité était identique, et donc soumise à une même fonction.
Toutefois, ce n'est pas toujours le cas.

```
bool existe(std::vector<int> & tab, int x)
{
  int i(0);
  bool trouve(0);
  while(i < tab.size() && !trouve)
  {
     if(tab[i] == x)
       trouve = 1;
     i++;
  }
  return trouve;
}
```

Algorithme 3: "x est-il présent dans le tableau tab ?" (naïf)

Ce morceau de code est un algorithme naïf qui vérifie si une valeur est dans un tableau donné en paramètre.
Pour la culture, il faut savoir qu'il est possible de savoir si une valeur est dans un tableau en $O(log2(n))$ (dans le pire des cas), en triant préalablement le tableau avec un algorithme en complexité linéarithmique.

→ Si x est en première position, l'algorithme va faire qu'une seule itération. Sa complexité sera donc en $O(1)$.
Comme il est impossible que cet algorithme fasse moins d'itérations (avec un tableau non-vide), on appelle cela la complexité dans le meilleur des cas.

→ Si x n'est pas dans le tableau, l'algorithme va parcourir tout le tableau. Sa complexité est donc en $O(n)$, où n est la taille du tableau.
Comme il est impossible que cet algorithme fasse plus d'itérations, on appelle cela la complexité dans le pire des cas.

→ De manière probabiliste, on peut savoir, avec une distribution donnée, la complexité dite moyenne de l'algorithme.
Nous n'allons pas le faire ici, il faut juste savoir que ça existe et que c'est souvent utilisé lorsque un algorithme est performant mais possède une complexité dans le pire des cas très mauvaise.
C'est le cas du tri rapide par exemple.

De manière générale, on estime un algorithme selon sa complexité dans le pire des cas et sa complexité moyenne si trop divergente de cette dernière.

5 Complexité en espace

La logique et les notations sont les mêmes que celles de la complexité en temps, sauf qu'au lieu d'évaluer les performances d'un algorithme en nombre d'itérations on va compter le nombre de cases mémoires qu'on va utiliser pour une entrée donnée.

```
std::vector<int> liste(int x)
{
  std::vector<int> tab;
  tableau.reserve(x + 1);
  for(int i(0); i <= x; ++i)
     tab.push_back(i);
  return tab;
}
```

Algorithme 4: "Renvoie un tableau contenant les valeurs de 0 à x"

Ici la complexité en temps est en $O(x)$, mais qu'en est-il de la complexité en espace ?
Cet algorithme [4] contient une variable entière i et un tableau de $x + 1$ cases, donc notre fonction de complexité spatiale $f(x) = x + 2$.
La complexité en espace est donc aussi en $O(x)$.

→ Il faut savoir que la structure vector de la STL prend un peu plus de place qu'un tableau style C. Ici, pour simplifier, on va considérer que ça prend le même nombre de cases mémoires.

→ L'algorithme [1], [2] et [3] ont une complexité spatiale en $O(1)$.

Cette mesure de la complexité, bien que moins utilisée, est très utile pour pondérer les performances d'un algorithme.
En effet, certains algorithmes ont une très bonne complexité en temps mais prennent énormément en mémoire, là où il peut exister un algorithme un peu moins véloce mais moins "volumineux".
Il est donc important d'analyser le tout et de pouvoir jauger ce qu'on souhaite faire, et surtout comment le faire.