Cours : Complexité

La capacité de calcul des ordinateurs a connu une croissance très rapide depuis les premiers ordinateurs jusqu'à aujourd'hui. Un indicateur souvent utilisé est le nombre d'opérations élémentaires qu'un ordinateur peut effectuer : l'unité de mesure communément admise est le nombre d'additions et de multiplications qu'il peut effectuer sur les nombres flottants en une seconde. En 1961, l'ordinateur le plus rapide du monde (L'IBM 7030 Stretch) était capable de 500 000 Flops/s (floating-point operations per second). Aujourd'hui, un ordinateur portable est capable d'effectuer 200 GFlops/s (Giga Flops/s), soit 200 milliards d'opérations par seconde. Un téléphone portable n'est pas en reste puisque les modèles de 2019 peuvent avoir une capacité de calcul de 50 GFlops/s. Toujours en 2019, l'ordinateur Summit, l'un des plus puissant actuellement, est capable d'exécuter 200 PFlops (Peta Flops/s) soit 200 millions de milliards de Flops/s. Cependant, il consomme 10 MW, c'est-à-dire la puissance nécessaire au fonctionnement d'une ville de 40 000 habitants.

Bien que ces capacités de calcul soient très grandes, elles ne sont pas illimitées. Prenons par exemple le problème du voyageur de commerce qui consiste à trouver dans quel ordre visiter n villes pour minimiser la distance parcourue, les trajets commençant et finissant tous dans la même ville. Ce problème est extrêmement difficile à résoudre et l'algorithme naïf consiste à tester tous les ordres différents. Il nécessite donc d'effectuer $(n-1)!$ tests. Pour chaque test, le calcul de la distance parcourue nécessite d'effectuer n additions. L'algorithme naïf nécessite donc d'effectuer $n!$ additions. Voici les plus grandes valeurs de n que les différents ordinateurs cités ci-dessus peuvent traiter en un jour :

- IBM 7030 : $n = 13$
- Téléphone portable de 2019 : $n = 17$
- Ordinateur portable de 2019 : $n = 18$
- Summit : $n = 22$

On voit bien que malgré leur puissance, les ordinateurs actuels ne peuvent pas grand chose pour aider le voyageur de commerce. Il existe des algorithmes du voyageur de commerce trouvant une solution en $n^2 2^n$ opérations. Les plus grandes valeurs de n que ces mêmes ordinateurs peuvent traiter deviennent alors :

- IBM 7030 : $n = 25$
- Téléphone portable de 2019 : $n = 41$
- Ordinateur portable de 2019 : $n = 43$
- Summit : $n = 61$

La première remarque est que même avec un ordinateur très puissant et un très bon algorithme, certains problèmes sont hors d'atteinte. La résolution du problème du voyageur de commerce pour $n = 100$ avec l'ordinateur le plus puissant actuel et le meilleur des deux algorithmes dont nous venons de parler nécessiterait 2 milliards d'années, ce qui n'est pas très loin de l'âge de l'univers, estimé à 14 milliards d'années. La seconde remarque est que les ordinateurs auxquels vous êtes confrontés ont tous une puissance de calcul similaire, que ce soit un téléphone portable, un ordinateur portable ou un ordinateur de bureau. La dernière remarque est que c'est avant tout l'algorithme utilisé et non la machine qui va déterminer la faisabilité d'utiliser un programme pour résoudre un problème donné. Remarquez par exemple que l'IBM 7030 datant de 1961 va faire tourner le second algorithme plus rapidement que ce qui est nécessaire à l'ordinateur le plus puissant actuellement pour faire tourner l'algorithme naif du problème du voyageur de commerce.

1 Introduction

La complexité temporelle est un outil pour avoir une estimation du temps nécessaire pour qu'un calcul s'exécute. Malheureusement, les processeurs sont aujourd'hui tellement complexes qu'il est très difficile d'obtenir quelque chose à la fois simple à calculer et précis. D'une part, certaines opérations sont plus rapides que d'autres, comme la division par deux entiers qui est 20 fois plus lente que la multiplication de deux nombres flottants, et ce, sur la plupart des processeurs actuels. L'accès à la mémoire est encore plus lent et peut être d'une lenteur exceptionnelle. Par exemple, l'accès aléatoire à un élément d'une très grande liste peut être 1000 fois plus lent que la multiplication de deux de ses éléments. Pire, le temps d'accès mémoire à un élément dépend de l'historique des calculs, ce qui rend une estimation théorique précise impossible en pratique.

Cependant, on peut considérer que les temps d'exécution des opérations élémentaires ne diffèrent pas plus d'un facteur 1000. Il est de plus intéressant de savoir comment évolue le temps d'exécution d'un algorithme en fonction de la taille des données sur lequel il travaille. Par exemple, pour chercher un élément dans une liste de longueur n, l'algorithme suivant

```
>>> def est_present(t, x):
        n = len(t)
        ans = False
        for k in range(n):
            if t[k] == x:
                est_present = True
        return est_present
```

nécessite n accès mémoire au tableau (accès à $t[k]$), n comparaisons entre $t[k]$ et x, et quelques affectations. Si pour les opérations élémentaires, on compte le nombre de comparaisons, on obtient n opérations élémentaires. Si pour opérations élémentaires, on compte maintenant le nombre d'accès mémoire et le nombre de comparaisons, on obtient $2n$. Bref, quelles que soient les opérations élémentaires choisies, le nombre d'opérations c_n est de la forme $an + b$ où a et b sont des valeurs de taille raisonnable. Plus précisément, ces nombres sont très petits devant le facteur 1000 qui peut exister encore le temps d'un accès mémoire et le temps nécessaire à une comparaison. On dit que cet algorithme est de complexité linéaire.

Si on souhaite trouver la plus grande valeur de $t[j] - t[i]$ pour $0 \leqslant i \leqslant j < n$, l'algorithme suivant

```
>>> def gain_maximal(t, x):
        n = len(t)
```

```
        ans = 0.0
        for j in range(n):
            for i in range(0, j + 1):
                gain = t[j] - t[i]
                if gain > ans:
                    ans = gain
        return ans
```

fonctionne. Pour les opérations élémentaires, nous allons compter le nombre de soustractions et le nombre de comparaisons. Pour chaque i et j fixés, il y a une soustraction et une comparaison, ce qui fait 2 opérations élémentaires. Pour chaque valeur de j, i va prendre les valeurs $0, 1, \dots, j$, soit $j+1$ valeurs. Il y a donc au total

$$c_n = \sum_{j=0}^{n-1} (j+1) = \frac{n(n+1)}{2}$$

opérations. Dans ce cas, il existe a, b et c de taille raisonnable tels que $c_n = an^2 + bn + c$. On dit que l'algorithme est de complexité quadratique.

Le Python étant un langage réputé lent, et ces calculs n'exploitant pas le parallélisme des ordinateurs actuels, on peut supposer en première approximation que les ordinateurs exécutant du code Python effectuent de l'ordre de un million d'opérations par seconde. Pour n assez grand, la complexité c_n de ces algorithmes est de l'ordre de an pour le premier exemple et an^2 pour le second exemple. Dans la mesure où a est un nombre de taille raisonnable, on peut supposer que $a \approx 1$. Si tel est le cas, c'est pour $n = 1\,000\,000$ que le programme `est_present` tournera en une seconde. Le programme `gain_maximal` tournera en une seconde pour une valeur beaucoup plus petite de n : 1 000. Il est impensable de faire tourner ce programme dans un temps raisonnable pour $n = 1\,000\,000$, même avec un ordinateur assez puissant. S'il existait un algorithme de complexité linéaire qui résout ce problème, il permettrait d'attaquer des problèmes de taille beaucoup plus grande. Il se trouve qu'un tel algorithme existe, mais il nécessite un peu de réflexion pour le trouver.

2 Relations de comparaison

Soit (c_n) une suite réelle positive et (α_n) une suite réelle strictement positive à partir d'un certain rang. On dit que (c_n) est un grand « O » de (α_n) lorsque n tend vers $+\infty$ et on note

$$c_n = \underset{n\to+\infty}{\mathrm{O}}(\alpha_n)$$

lorsqu'il existe $K \in \mathbb{R}_+$ tel que

$$\frac{c_n}{\alpha_n} \leqslant K$$

à partir du rang où α_n est strictement positif. Si tel est le cas, on a donc $0 \leqslant c_n \leqslant K\alpha_n$ à partir de ce rang. Par exemple, si il existe a, b, $c \in \mathbb{R}$ tels que $c_n = an^2 + bn + c$, alors

$$c_n = \underset{n\to+\infty}{\mathrm{O}}\left(n^2\right).$$

En effet

$$\forall n \in \mathbb{N}^* \quad \frac{c_n}{n^2} = a + \frac{b}{n} + \frac{c}{n^2} \leqslant |a| + \frac{|b|}{n} + \frac{|c|}{n^2} \leqslant |a| + |b| + |c|$$

donc $K = |a| + |b| + |c|$ convient. Plus généralement, si $c_n = a_p n^p + \dots + a_1 p + a_0$, on montre de même que $c_n = \underset{n\to+\infty}{\mathrm{O}}(n^p)$.

Prenons l'exemple de l'algorithme du calcul du diamètre d'une famille de n points du plan. On se donne une liste t de n couples de réels (x_k, y_k) qui sont les coordonnées des points $M_0, \dots, M_{n-1}$ du plan ramené à un repère orthonormé. On cherche à calculer

$$\max_{0\leqslant i,j<n} \mathrm{d}(M_i, M_j) = \max_{0\leqslant i,j<n} \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

Si on compte pour opérations élémentaires, les soustractions, les additions et les carrés, l'algorithme suivant

```
>>> import numpy as np
>>> def diametre(t):
        n = len(t)
        ans = 0.0
        for i in range(n):
            for j in range(n):
                distance_2 = (t[i][0] - t[j][0])**2 + \
                             (t[i][1] - t[j][1])**2
                if distance_2 > ans:
                    ans = distance_2
        return np.sqrt(ans)
```

effectue 5 opérations pour chaque valeur de i et j. Comme i peut prendre n valeurs et j peut prendre n valeurs, on en déduit que l'algorithme a une complexité $c_n = 5n^2$. On a donc

$$c_n = \underset{n\to+\infty}{\mathrm{O}}\left(n^2\right).$$

En remarquant que la distance de M_i à M_j est égale à la distance de M_j à M_i, et que la distance entre M_i et M_i est nulle, on peut se restreindre au calcul des distances entre M_i à M_j pour $0 \leqslant j < i < n$. On obtient alors l'algorithme suivant :

```
>>> import numpy as np
>>> def diametre_bis(t):
        n = len(t)
        ans = 0.0
        for i in range(n):
            for j in range(i):
                distance_2 = (t[i][0] - t[j][0])**2 + \
                             (t[i][1] - t[j][1])**2
                if distance_2 > ans:
                    ans = distance_2
        return np.sqrt(ans)
```

Dans ce cas, le nombre d'opérations élémentaire est

$$c_n = 5 \times (0 + 1 + \cdots + (n-1)) = \frac{5}{2} n(n-1)$$

et on a toujours

$$c_n = \underset{n\to+\infty}{\mathrm{O}}\left(n^2\right)$$

ce qui fait que cet nouvel algorithme est toujours quadratique. Cet algorithme est en pratique à peu près deux fois plus rapide que la version précédente, mais ce n'est pas le but des calculs de complexité temporelle que de déterminer cela.
On dira qu'un algorithme a une complexité *linéaire* lorsque

$$c_n = \underset{n\to+\infty}{\mathrm{O}}\left(n\right)$$

et *quadratique* lorsque

$$c_n = \underset{n\to+\infty}{\mathrm{O}}\left(n^2\right).$$

En pratique, on retiendra qu'un algorithme possédant une boucle est le plus souvent de complexité linéaire. Lorsqu'il possède deux boucles imbriquées, il est le plus souvent de complexité quadratique.

Si on applique cette définition à la lettre, un algorithme de complexité linéaire serait aussi un algorithme de complexité quadratique. En effet, si il existe $K \in \mathbb{R}_+^*$ tel que $c_n \leqslant Kn$, alors $c_n \leqslant Kn^2$. En pratique, il est sous-entendu lorsqu'on dit qu'un algorithme est de complexité quadratique, qu'il n'existe pas de $\alpha < 2$ tel que

$$c_n = \underset{n\to+\infty}{\mathrm{O}}\left(n^\alpha\right).$$

Il arrive que le nombre d'opérations élémentaires ne dépende pas seulement de la taille des données, mais aussi des données elles-mêmes. Par exemple, dans l'algorithme de recherche dans une liste quelconque donné ci-dessous

```
>>> def est_present(t, x):
        n = len(t)
        for k in range(n):
            if t[k] == x:
                return True
        return False
```

le nombre de tests d'égalité est inférieur où égal à la taille de la liste n. Cependant, si t est une liste comportant la valeur x en première position, le nombre de tests est égal à 1. La complexité c_n est alors mal définie. Une solution est donc de définir c_n comme étant *la complexité dans le pire des cas* : c_n est le nombre d'opérations élémentaires pour la liste t et le tableau x qui engendrent le plus grand nombre d'opérations élémentaires. Ici, dans le cas où le tableau ne contient pas la valeur x, il y a n tests d'égalité, donc la complexité dans le pire des cas est n. On a donc

$$c_n = \underset{n\to+\infty}{\mathrm{O}}\left(n\right)$$

et on dit que la complexité de cet algorithme est linéaire dans le pire des cas. Si on note $c_n(t,x)$ le nombre de tests d'égalité lors de l'appel de la fonction avec une liste quelconque t et une valeur x, on a donc $0 \leqslant c_n(t,x) \leqslant c_n \leqslant Kn$.

3 Complexité de la recherche dichotomique

Si t est une liste triée dans l'ordre croissant et x est une valeur, l'algorithme de recherche dichotomique permet de déterminer si x est un élément de t. Pour le calcul de complexité, nous allons compter le nombre d'accès mémoire au tableau t.

```
>>> def est_present_dichotomie(t, x):
        g = 0
        d = len(t) - 1
        while g <= d:
            m = (g + d) // 2
            if x < t[m]:
                d = m - 1
            elif x > t[m]:
                g = m + 1
            else:
                return True
        return False
```

Remarquons tout d'abord que le nombre d'accès mémoire dépend non seulement du nombre d'éléments n dans la liste mais aussi de ces éléments. En effet, supposons que l'élément recherché soit exactement au milieu de la liste, alors on sort de la fonction après deux accès mémoire. Si l'élément recherché n'est pas présent dans la liste, le nombre d'accès mémoire est beaucoup plus grand. On définit donc c_n comme la complexité dans le pire des cas.

Afin de majorer c_n, nous devons majorer le nombre de fois où l'algorithme exécute le corps de la boucle `while`. Pour cela, on note $n_k = d_k - g_k$ la largeur de la partie de la liste sur lequel s'effectue la recherche après k exécutions du corps de la boucle. Avant d'entrer dans la boucle, elle est égale à $n-1$, donc $d_0 = n-1 \leqslant n$. Nous allons montrer que quel que soit l'entier k, on a $n_{k+1} < n_k/2$.

— Supposons que $x < t[m]$. Dans ce cas

$$\begin{aligned} n_{k+1} &= \left(\left\lfloor \frac{d_k+g_k}{2} \right\rfloor - 1\right) - g_k \\ &\leqslant \left(\frac{d_k+g_k}{2} - 1\right) - g_k \\ &\leqslant \frac{d_k-g_k}{2} - 1 \\ &< \frac{d_k-g_k}{2} = \frac{n_k}{2} \end{aligned}$$

— Supposons maintenant que $x > t[m]$. Alors, en utilisant le fait que $x < \lfloor x \rfloor + 1$, on obtient

$$\begin{aligned} n_{k+1} &= d_k - \left(\left\lfloor \frac{d_k+g_k}{2} \right\rfloor + 1\right) \\ &< d_k - \frac{d_k+g_k}{2} \\ &< \frac{d_k-g_k}{2} = \frac{n_k}{2} \end{aligned}$$

Par récurrence finie, on en déduit donc que pour tout entier k, on a $n_k \leqslant n/2^k$. Or :

$$\begin{aligned}\frac{n}{2^k} < 1 &\iff 2^k > n\\ &\iff k\ln(2) > \ln(n)\\ &\iff k > \frac{\ln(n)}{\ln(2)}\end{aligned}$$

Donc, au bout d'au plus

$$k = \left\lfloor \frac{\ln(n)}{\ln(2)} \right\rfloor + 1$$

itérations, on a $n_k \leqslant 0$ et donc $n_{k+1} < 0$, ce qui implique que l'on sort de la boucle. On a donc prouvé que le nombre d'itérations est inférieur à

$$\left\lfloor \frac{\ln(n)}{\ln(2)} \right\rfloor + 2.$$

Ce nombre est par ailleurs inférieur à

$$\frac{\ln(n)}{\ln(2)} + 2.$$

On en déduit, le nombre d'accès mémoire par itération étant inférieur à 2, que pour $n \geqslant 2$

$$\begin{aligned}\frac{c_n}{\ln(n)} &\leqslant \frac{2}{\ln(2)} + \frac{4}{\ln(n)}\\ &\leqslant \frac{2}{\ln(2)} + \frac{4}{\ln(2)}\\ &\leqslant \frac{6}{\ln(2)}\end{aligned}$$

ce qui prouve que

$$c_n = \underset{n\to+\infty}{\mathrm{O}}\left(\ln(n)\right).$$

On dit que la complexité de la recherche dichotomique est logarithmique. Cela rend cet algorithme extrêmement efficace. Par exemple, si $n = 1\,000\,000\,000$, $\ln n \approx 20$ ce qui montre que le nombre d'accès mémoire au tableau de compte en dizaines, ce qui est très faible devant la taille du tableau.

4 Évaluation d'un polynôme, algorithme de Horner

Étant donné un polynôme

$$P(X) = \sum_{k=0}^{n} a_k X^k,$$

représenté par la liste de longueur $n+1$ notée $p = [a_0, \ldots, a_n]$, on souhaite calculer la complexité de différents algorithmes d'évaluation de $P(x)$. Dans cette partie, nous nous interdisons l'utilisation de l'opérateur `**` qui n'est pas traduit en une unique opération pour le processeur, notamment lorsqu'on travaille avec des entiers. Commençons par l'algorithme le plus naif :

```
>>> def evaluation_polynome(p, x):
        n = length(p) - 1
        ans = 0
        for k in range(n + 1):
            x_puissance_k = 1
            for i in range(k):
                x_puissance_k = x_puissance_k * x
            ans = ans + p[k] * x_puissance_k
        return ans
```

Pour le calcul de la complexité, nous allons compter le nombre d'opérations élémentaires : additions et multiplications. Pour chaque valeur de k, k multiplications sont nécessaires pour calculer x^k, et une addition et une multiplication sont nécessaires pour ajouter $a_k x^k$ à la somme partielle. Il y a donc $k+2$ opérations élémentaires. Il y a donc au total

$$c_n = \sum_{k=0}^{n} k + 2 = \frac{n(n+1)}{2} + 2(n+1)$$

opérations élémentaires. On en déduit que

$$c_n = \underset{n\to+\infty}{\mathrm{O}}\left(n^2\right)$$

et donc que l'évaluation naïve d'un polynôme se fait en complexité quadratique.

On peut rendre plus efficace l'évaluation de ce polynôme en gardant en mémoire x^k avant de calculer x^{k+1}. On obtient ainsi l'algorithme suivant

```
>>> def evaluation_polynome(p, x):
        n = length(p) - 1
        ans = 0
        x_puissance_k = 1
        for k in range(n + 1):
            ans = ans + p[k] * x_puissance_k
            x_puissance_k = x_puissance_k * x
        return ans
```

qui nécessite

$$c_n = \sum_{k=0}^{n} 3 = 3(n+1)$$

opérations élémentaires. On en déduit que

$$c_n = \underset{n\to+\infty}{\mathrm{O}}\left(n\right)$$

ce qui fait de ce nouveau programme un algorithme en complexité linéaire. Ce changement d'algorithme permet de faire changer de classe de complexité. Il est donc considéré comme un changement algorithmique majeur.

Il est cependant possible de faire baisser la constante 3 dans le $3(n+1)$ en utilisant l'algorithme de *Horner*. Pour cela, on remarque que :

$$P(x) = (\cdots((a_n x + a_{n-1})\,x + a_{n-2})\,x + \cdots a_1)\,x + a_0$$

Ce nouvel algorithme s'implémente de la façon suivante :

```
>>> def horner(p, x):
        n = length(p) - 1
        ans = 0
        for k in range(n + 1):
            ans = ans * x + p[n - k]
        return ans
```

Il nécessite $c_n = 2n$ opérations élémentaires. L'algorithme est donc toujours de complexité linéaire, mais il nécessite une multiplication de moins à chaque étape. Cela fait de l'algorithme de Horner l'algorithme le plus efficace pour évaluer un polynôme.

5 Tri par insertion

Nous avons vu que le fait d'avoir une liste triée permet d'utiliser ensuite des algorithmes beaucoup plus rapide. Par exemple, la recherche dichotomique dans une liste a une complexité en $O(\ln(n))$ alors que la recherche dans une liste quelconque a une complexité en $O(n)$. Il est donc fondamental de savoir trier efficacement une liste. Nous présenterons différents algorithmes plus ou moins efficaces pour trier une liste cette année. Commençons par l'un des plus simples d'entre eux qui est celui que vous utilisez pour trier vos cartes lorsque vous jouez à la belote : le tri par insertion. L'idée est d'insérer, carte après carte, la i-ième carte dans les groupe des cartes de gauche qui est déjà triée.

- La carte de gauche forme le groupe initial qui est déjà trié.
- On va insérer la seconde carte dans ce groupe. Pour cela, il suffit de savoir si elle est plus forte ou plus faible que la première carte. On échange alors ou non ces deux cartes. On a ainsi formé un groupe de 2 cartes sur la gauche de notre main qui est déjà trié.
- Supposons que l'on souhaite insérer la carte au rang i dans notre groupe déjà trié. On compare notre carte successivement aux cartes du groupe, en commençant par la plus forte. On descend dans la force des cartes, jusqu'à trouver une carte plus faible que celle que l'on souhaite insérer ou éventuellement réaliser que notre carte est la plus faible.

L'algorithme de tri par insertion n'est rien d'autre que l'implémentation de cet algorithme. Il marche par effet de bord.

```
>>> def tri_insertion(t):
        n = len(t)
        for i in range(1, n):
            x = t[i]
            j = i
            while j > 0 and t[j - 1] > x:
                t[j] = t[j - 1]
                j = j - 1
            t[j] = x
```

Pour calculer la complexité de cet algorithme, nous allons compter le nombre de comparaisons. Bien entendu, le nombre d'opérations élémentaires effectué ne dépend pas seulement de la taille de la liste mais aussi de l'ordre des éléments dans la liste. Cela est dû au `while` dont le test d'arrêt dépend de l'ordre des éléments dans la liste. On travaillera donc avec c_n la complexité dans le pire des cas. Pour chaque valeur de i, le nombre de comparaisons entre un élément du tableau et x est inférieur ou égal à i. On en déduit que

$$c_n \leqslant \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

puis que

$$c_n = \underset{n\to+\infty}{\mathrm{O}}\left(n^2\right)$$

ce qui montre que la complexité du tri par insertion est quadratique.

6 Les différentes classes de complexité

Cette année, nous rencontrerons différentes classes de complexité. Nous dirons qu'un algorithme a une complexité

- *en temps constant* lorsque $c_n = \underset{n\to+\infty}{\mathrm{O}}(1)$
- *logarithmique* lorsque $c_n = \underset{n\to+\infty}{\mathrm{O}}(\ln n)$
- *linéaire* lorsque $c_n = \underset{n\to+\infty}{\mathrm{O}}(n)$
- *quasi-linéaire* lorsque $c_n = \underset{n\to+\infty}{\mathrm{O}}(n\ln n)$
- *quadratique* lorsque $c_n = \underset{n\to+\infty}{\mathrm{O}}\left(n^2\right)$.
- *cubique* lorsque $c_n = \underset{n\to+\infty}{\mathrm{O}}\left(n^3\right)$.
- *polynomiale* lorsqu'il existe $p \in \mathbb{N}$ tel que $c_n = \underset{n\to+\infty}{\mathrm{O}}(n^p)$
- *exponentielle* lorsqu'il existe $a > 1$ tel que $c_n = \underset{n\to+\infty}{\mathrm{O}}(a^n)$
- *factorielle* lorsque $c_n = \underset{n\to+\infty}{\mathrm{O}}(n!)$

On peut dire qu'un algorithme s'exécute en un temps raisonnable lorsqu'il nécessite de l'ordre de 1 000 000 000 opérations. Lorsque l'algorithme s'exécute en temps constant ou même en complexité logarithmique, la taille des données que l'on peut traiter dans un temps raisonnable est virtuellement illimitée. En pratique, ce sera plutôt la mémoire utilisée par les données qui sera un facteur limitant. Pour un algorithme linéaire, on pourra pousser jusqu'à des tailles de 1 milliard. Pour un algorithme quasi-linéaire, on pourra aller jusqu'à 100 millions, ce qui est guère différent du cas linéaire. Pour un algorithme de complexité quadratique, on pourra aller jusqu'à des valeurs de n de l'ordre de 10 000 voire 100 000. Pour un algorithme de complexité cubique, on dépassera difficilement 1 000. Pour un algorithme de complexité exponentielle, on ne pourra pas dépasser quelques dizaines. Enfin, pour un algorithme de complexité factorielle, il va être difficile de dépasser beaucoup plus de 10.

Tous ces nombres sont des ordres de grandeur et rappelons que certaines opérations sont parfois 1000 fois plus rapides que d'autres sur les processeurs actuels. Cela explique par exemple que le facteur $\ln(n)$ n'a souvent pas une grande importance, ce qui explique le nom d'algorithme quasi-linéaire. Il arrive même qu'il y ait des algorithmes cubiques presque aussi

rapides que des algorithmes quadratiques, pour des valeurs de n de l'ordre de 1000.

Enfin, il est essentiel de se rappeler que les calculs de complexité ne nous apprennent pas grand chose sur ce qui se passe pour les « petites » valeurs de n. Par exemple, certains des algorithmes les plus efficaces pour trier une liste utilisent la méthode du tri par insertion lorsque $n \leqslant 20$, méthode qui est de complexité quadratique, alors que nous verrons bientôt des méthodes de complexité quasi-linéaire. Bien entendu, lorsque n devient grand, ces algorithmes utilisent le plus souvent le tri rapide, algorithme quasi-linéaire en moyenne (nous verrons plus tard ce que ce « en moyenne » signifie).

Bref, tout cela nous rappelle que l'informatique est avant tout une science expérimentale, surtout lorsqu'on essaie de faire le lien entre la complexité temporelle avec le temps de calcul.