La complexité c'est simple comme la dichotomie

Guillaume Connan*- Irem de Nantes

22 mars 2015

Résumé

La dichotomie, c'est couper un problème en deux. On ne la présente souvent au lycée que dans le cadre restreint de la recherche dichotomique de la solution réelle d'une équation du type $f(x) = 0$ et on la dénigre car elle est bien plus lente que la fulgurante méthode des tangentes. Pourtant, elle est bien plus riche que son utilisation dans ce contexte étroit le laisserait penser. Nous en profiterons pour introduire la notion de complexité algorithmique.

1 Le lancer de piano ou comment faire de l'informatique sans ordinateur...

1.1 L'expérience

Une entreprise de déménagement syldave veut réduire son personnel et rationnaliser ses méthodes de transport de piano. Elle engage un spécialiste de chez Dicho & Co qui propose l'approche scientifique suivante. L'entreprise achemine k pianos en bas du bâtiment de l'Université de Klow qui a $N = 2^n$ étages, le rez-de-chaussée portant le numéro 0.

Figure 1 – Université de Klow

Le but de l'expérimentation est de déterminer l'étage à partir duquel lancer un piano par la fenêtre lui est fatal. On suppose qu'un tel étage existe et que le rez-de-chaussée n'est pas fatal.
On suppose également que tant que le piano survit au lancer, on peut le réutiliser. Pour économiser le temps et le dos des expérimentateurs, on va chercher un algorithme qui minimise le nombre d'essais effectués (et non pas le nombre de pianos détruits).
Une première idée serait de commencer au rez-de-chaussée et de lancer un piano de chaque étage jusqu'à atteindre l'étage fatal. Si jamais l'étage fatal est le dernier, cela nécessitera d'effectuer $N - 1$ essais : c'est le pire cas.
Si l'on considère que tous les étages sont potentiellement fatals avec la même probabilité, on effectuera avec cette méthode $(N - 1)/2$ essais en moyenne.
Mais on peut faire mieux....

*Guillaume.Connan@univ-nantes.fr

1.2 Diviser pour régner

Le commandement du grand nombre est le même pour le petit nombre, ce n'est qu'une question de division en groupes.

in « L'art de la guerre » 孙子兵法 de Sun Zi (VI[e] siècle avant JC)

FIGURE 2 – 孫子 (544–496 av. J.-C.)

Nous allons diviser l'immeuble en deux parties égales en considérant l'étage du milieu : si cet étage est fatal, il suffira de chercher dans la moitié inférieure, sinon, dans la moitié supérieure de l'immeuble. Nous allons donc effectuer une **recherche dichotomique** de l'étage fatal.

```
inf ← Étage inférieur
sup ← Étage supérieur
milieu ← (inf + sup)/2
IestFatal(milieu) ChercherEntre(inf,milieu)
Sinon
ChercherEntre(milieu + 1, sup)
FinSi
```

Quand va-t-on s'arrêter de découper l'immeuble ? Lorsqu'il n'y aura plus d'ambiguité, c'est-à-dire lorsque l'étage supérieur sera juste au-dessus de l'étage inférieur. La fonction devient :

```
Fonction ChercherEntre( inf, sup : Entiers ) : Entier
{ pré-condition: il existe au moins un étage fatal entre inf et sup }
{ invariant: le plus petit étage fatal est entre inf et sup }
{ post-condition: la valeur retrournée est le plus petit étage fatal }
Isup == inf + 1 Retourner sup
Sinon
milieu ← (inf + sup)/2
IestFatal(milieu) ChercherEntre(inf,milieu)
Sinon
ChercherEntre(milieu + 1, sup)
FinSi
FinSi
```

Qu'a-t-on gagné ? Mais d'abord, est-ce que notre fonction nous renvoie l'étage attendu ? Et d'abord, renvoie-t-elle quelque chose ?
Déterminer les réponses à ces questions constitue les trois étapes indispensables de l'étude d'un algorithme :

1. étude de la *terminaison* de l'algorithme : est-ce que la fonction renvoie effectivement une valeur ?
2. étude le la *correction* de l'algorithme : est-ce que la fonction renvoie la valeur attendue ?
3. étude de la *complexité* de l'algorithme : peut-on estimer la vitesse d'exécution de cet algorithme ?

1. Pour répondre à la première question, il suffit de remarquer qu'au départ, l'intervalle d'étude est l'ensemble $\{1, 2, \ldots, N\}$ de longueur N. Après chaque appel récursif, la longueur de l'intervalle est divisée par deux. Notons ℓ_i cette longueur après i appels récursifs. La suite ℓ est géométrique de raison $1/2$ et de premier terme N.
   On a donc $\ell_i = \frac{N}{2^i} = \frac{2^n}{2^i} = 2^{n-i}$. Ainsi $\ell_n = 1$ et l'algorithme s'arrête.

2. Vérifions d'abord que l'invariant est valide à chaque appel récursif : c'est vrai au départ car on précise qu'un étage fatal existe entre le rez-de-chaussée et le dernier étage.

   Par construction, on choisit inf et sup pour que le plus petit reste entre inf et sup et que la borne inférieure ne soit pas fatale.

   Quand la récursion s'arrête, l'intervalle est en fait $\{\text{inf}, \text{inf}+1\}$ et inf n'est pas fatal donc inf + 1 l'est et est la valeur cherchée, eh !

3. Combien de temps nous a-t-il fallu ? Mais d'abord comment mesure-t-on le temps ?

   Nous ne sommes pas sur machine mais dans la grande université de Klow. Il faut compter le temps de chute, le temps de remontée des pianos encore en état, le temps de manipulation pour passer les pianos par la fenêtre, l'examen de l'état du piano ayant chu. Selon l'efficacité des muscles et l'état de santé des déménageurs, nous pouvons poser que ces temps sont proportionnels au nombre d'étages, mis à part le temps de passage par la fenêtre qui est constant ainsi que l'examen du piano par un expert [a].

   ...Mais c'est un peu compliqué...On va supposer que l'ascenseur est ultra-performant et que les pianos sont très lourds ce qui entraîne que tous ces temps sont à peu près constants, quelque soit l'étage [b].

   Il suffit donc de compter combien de lancers ont été effectués. La réponse est dans l'étude faite pour prouver la terminaison : c'est à l'étape n que l'on atteint la condition de sortie de l'algorithme.

   Que vaut n ? On a posé au départ que le nombre d'étages était une puissance de 2 : $N = 2^n$. Ainsi, $n = \log_2 N$.

   Finalement, le nombre de tests effectués est égal au logarithme en base 2 du nombre d'étages. Et le temps ? Et bien, ce n'est pas ce qui nous intéressait puisqu'on nous demandait de mesurer le nombre de tests effectués mais cela nous a permis d'introduire la notion de *complexité* d'un algorithme dont la mesure va dépendre de l'unité choisie et sera plus ou moins difficile à déterminer.

   Si notre immeuble a 1024 étages, nous sommes donc passés de 1022 tests au maximum à 10 tests ce qui n'est pas négligeable, surtout pour le dos des déménageurs.

1.3 Le bug de Java

Pendant 9 ans, jusqu'en 2006, les fonctions de recherche dichotomique en Java cachaient un bug...
Voici ce que l'on trouvait dans la bibliothèque `java.util.Arrays` :

```java
public static int binarySearch(int[] a, int key) {
        int low  = 0;
        int high = a.length - 1;

        while (low <= high) {
            int mid    = (low + high) / 2;
            int midVal = a[mid];

            if (midVal < key)
                low = mid + 1
            else if (midVal > key)
                high = mid - 1;
            else
                return mid; // key found
        }
        return -(low + 1);  // key not found.
    }
```

Regardez la ligne 6 : `int mid    = (low + high) / 2;`.
Tout va bien...mais en fait il faut se souvenir que les entiers sont codés en général sur 64 bits et que le successeur de $2^{63} - 1$ n'est pas 2^{63} mais son opposé car on représente les nombres relatifs sur machine d'une manière spéciale [c].
Le bug a été corrigé [d] en remplaçant cette ligne par :

a. C'est ce qui correspondra par exemple en informatique au parcours d'une liste chaînée.
b. C'est ce qui correspondra par exemple en informatique au parcours d'un tableau statique.
c. Voir par exemple le bug de l'an 2038 : https://fr.wikipedia.org/wiki/Bug_de_l%27an_2038
d. http://bugs.java.com/bugdatabase/view_bug.do?bug_id=5045582

```
int mid = low + ((high - low) / 2);
```
Il faudra faire de même dans nos fonctions !

1.4 Questions subsidiaires

1. **Que se passe-t-il si nous avons moins de $n = \log_2(N)$ pianos pour tester ?**

   Aïe ! Notre dichotomie ne peut aller jusqu'au bout. L'idée est de l'appliquer $k-1$ fois (k étant le nombre de pianos à diposition). Il se peut que nous n'ayons pas abîmé de piano. Il se peut aussi que nous ayons cassé un piano à chaque fois si l'étage fatal est très petit. Il nous reste donc *dans le pire des cas* un piano et $N/2^{k-1}$ étages non testés.

   On va donc partir du plus bas et remonter jusqu'à casser notre dernier piano. *Dans le pire des cas*, ce sera le dernier testé.

   Ainsi, s'il nous reste assez de pianos (si l'étage fatal n'est pas trop bas), nous pourrons appliquer la dichotomie jusqu'au bout. Dans le pire des cas, il faudra effectuer $k - 1 + \frac{N}{2^{k-1}} - 1$ tests.

2. **Que se passe-t-il si nous n'avons que 2 pianos et que n est pair ?**

   Si l'on applique une fois la dichotomie, il nous restera $N/2$ tests à effectuer.

   On peut faire mieux...

   Il suffit de découper l'immeuble en $\sqrt{N} = 2^{n/2}$ paquets de $\sqrt{N}$ étages.

   On commence par tester l'étage $\sqrt{N}$ puis éventuellement $2\sqrt{N}$, etc. Bref, on effectue une recherche séquentielle du paquet fatal.

   Ensuite, il suffit d'utiliser le deuxième piano (le premier est cassé pour déterminer le paquet fatal) afin d'effecteur une recherche séquentielle de l'étage fatal cette fois. Dans le pire des cas, cela va nous demander $\sqrt{N} - 1$ tests.

   Bref, dans le pire des cas, nous effectuerons $2\sqrt{N} - 1$ tests.

3. **Que se passe-t-il quand N n'est pas une puissance de 2 ?**

   Ce n'est pas grave car le nombre de tests croît avec le nombre d'étages dans limmeuble. Or, pour tout entier naturel N, il existe un entier n tel que :

   $$2^n \leqslant N < 2^{n+1}$$

   Si on appelle T la fonction qui, au nombre d'étages, associe le nombre de tests par la méthode dichotomique, alors :

   $$T(2^n) = n \leqslant T(N) < n + 1$$

   donc $T(N) = n = \lfloor \log_2(N) \rfloor$.

4. **Et si nous avions fait de la trichotomie**

FIGURE 3 – Trichotomie manuelle

On pourrait penser que diviser notre immeuble en 3 pourrait être plus efficace car on divise la taille de l'intervalle suivant par 3 *mais* on effectue au pire deux tests à chaque fois.

Ainsi, dans le pire des cas on effectuera (en supposant que N est une puissance de 3) $2\log_3(N) = 2\frac{\ln(N)}{\ln(2)} \times \frac{\ln(2)}{\ln(3)} \approx 1,3\log_2(N)$ tests donc ce n'est pas mieux et ça ne fera qu'empirer si on augmente le nombre de divisions...

La division par 2 serait-elle magique ?

2 Diviser ne permet pas toujours de régner

Considérons un problème mathématique simple : la multiplication de deux entiers.
Vous savez additionner deux entiers de n chiffres : cela nécessite $\lambda_1 n$ additions de nombres de un chiffre compte-tenu des retenues avec λ_1 une constante positive.
Multiplier un entier de n chiffres par un entier de 1 chiffre prend $\lambda_2 n$ unités de temps selon le même principe.
En utilisant l'algorithme de l'école primaire pour multiplier deux nombres de n chiffres, on effectue n multiplications d'un nombre de n chiffres par un nombre de 1 chiffre puis une addition des n nombres obtenus. On obtient un temps de calcul en $\lambda_3 n^2$.
On peut espérer faire mieux en appliquant la méthode *diviser pour régner*.
On coupe par exemple chaque nombre en deux parties de $m = \lfloor n/2 \rfloor$ chiffres :

$$xy = (10^m x_1 + x_2)(10^m y_1 + y_2) = 10^{2m} x_1 y_1 + 10^m (x_2 y_1 + x_1 y_2) + x_2 y_2$$

Fonction `MUL(x:entier ,y: entier):` *entier*
`In==1` **Retourner** `x.y`
Sinon
$m \leftarrow \lfloor n/2 \rfloor$
$x_1 \leftarrow \lfloor x/10^m \rfloor$
$x_2 \leftarrow x \bmod 10^m$
$y_1 \leftarrow \lfloor y/10^m \rfloor$
$y_2 \leftarrow y \bmod 10^m$
$a \leftarrow$ `MUL(`x_1`,`y_1`,m)`
$b \leftarrow$ `MUL(`x_2`,`y_1`,m)`
$c \leftarrow$ `MUL(`x_1`,`y_2`,m)`
$d \leftarrow$ `MUL(`x_2`,`y_2`,m)`
Retourner $10^{2m} a + 10^m (b + c) + d$
FinSi

Les divisions et les multiplications par des puissances de 10 ne sont que des décalages effectués en temps constant.
L'addition finale est en λn donc le temps d'exécution est défini par :

$$T(n) = 4T(\lfloor n/2 \rfloor) + \lambda n \qquad T(1) = 1$$

Peut-on exprimez $T(n)$ explicitement en fonction de n sans récursion ?
Si nous avions une idée de la solution, nous pourrions la démontrer par récurrence.
Ça serait plus simple si n était une puissance de 2. Voyons, posons $n = 2^k$ et $T(n) = T(2^k) = x_k$.
Alors la relation de récurrence devient :

$$x_k = 4x_{k-1} + \lambda 2^k \qquad x_0 = 1$$

On obtient :

$$x_k = 4(4x_{k-2} + \lambda' 2^{k-1}) + \lambda 2^k = 4^k x_0 + \sum_{i=1}^{k} \Lambda_k 2^k = 4^k + k\Lambda_k 2^k = n^2 + \Lambda_k n \log n \sim n^2$$

Car nous verrons que l'on peut encadrer ce Λ_k entre deux constantes positives dans la dernière section.
On montre alors par récurrence forte que ce résultat est vrai pour tout entier naturel non nul n.
Bref, tout ça pour montrer que l'on n'a pas amélioré la situation...

3 Et la recherche dichotomique d'une solution d'une équation réelle ?

Quel rapport entre la recherche dichotomique dans un tableau (ou un immeuble) et la recherche des solutions d'une équation $f(x) = 0$ dans $\mathbb{R}$?

Pour mettre en œuvre une telle méthode, il faut donner une précision. On ne travaille que sur des approximations décimales ou plutôt à l'aide de nombres à virgule flottante en base 2.
Par exemple, si nous cherchons une approximation de $x^2-2=0$ par la méthode de dichotomie avec une précision de 2^{-10} entre 1 et 2, il va falloir chercher un nombre (un étage) dans un tableau (un immeuble) de 2^{10} nombres (étages) :

1	$1+2^{-10}$	$1+2\times 2^{-10}$	$1+3\times 2^{-10}$	...	$1+2^{10}\times 2^{-10}$

Notre fonction booléenne « estFatal » est alors le test $x \mapsto$ x * x <= 2 et l'on va chercher une cellule de ce tableau par dichotomie comme on cherchait un étage dans un immeuble.

```
def racineDicho(prec):
    cpt = 0
    inf = 1
    sup = 2
    while (sup - inf > prec):
        m = inf + (sup - inf) / 2
        cpt += 1
        if m*m <= 2:
            inf = m
        else:
            sup = m
    return sup,cpt
```

Nous obtenons :

```
In [1]: racineDicho(2**(-10))
Out[1]: (1.4150390625, 10)

In [2]: racineDicho(2**(-15))
Out[2]: (1.414215087890625, 15)

In [3]: racineDicho(2**(-20))
Out[3]: (1.4142141342163086, 20)

In [4]: racineDicho(2**(-30))
Out[4]: (1.4142135623842478, 30)

In [5]: racineDicho(2**(-50))
Out[5]: (1.4142135623730958, 50)
```

Attention ! Il faudra s'arrêter à la précison 2^{52} compte-tenu de la représentation des nombres sur machine. N'hésitez pas à parcourir
http://download.tuxfamily.org/tehessinmath/les%20pdf/JA2014.pdf
ou
http://download.tuxfamily.org/tehessinmath/les%20pdf/PolyAnalyse14.pdf
pour plus de précisions ;-)
On « voit » que le nombre de tests correspond à la longueur de l'intervalle exprimé en « logarithme de la précision » : ici, l'intervalle est de longueur 1.

4 La complexité sur machine : approche expérimentale et théorique

4.1 Méthode scientifique ;-)

Le problème : *on dispose d'une liste de N nombres. Déterminez les triplets dont la somme est nulle.*
Utilisons la force brute (pour le lecteur impatient : nous ferons mieux plus tard...) :

```
def trois_sommes(xs):
    N = len(xs)
    cpt = 0
    for i in range(N):
        for j in range(i + 1, N):
            for k in range(j + 1, N):
                if xs[i] + xs[j] + xs[k] == 0:
                    cpt += 1
    return cpt
```

Comparons les temps pour différentes tailles :

```
In [1]: %timeit trois_sommes([randint(-10000,10000) for k in range(100)])
10 loops, best of 3: 27.5 ms per loop

In [2]: %timeit trois_sommes([randint(-10000,10000) for k in range(200)])
1 loops, best of 3: 216 ms per loop

In [3]: %timeit trois_sommes([randint(-10000,10000) for k in range(400)])
1 loops, best of 3: 1.82 s per loop
```

On peut plutôt s'intéresser aux ratios :

```
from time import perf_counter
from math import log

def temps(xs):
    debut = perf_counter()
    trois_sommes(xs)
    return perf_counter() - debut

t = [temps(range(100 * 2**k)) for k in range(4)]

ratio = [t[k + 1] / t[k] for k in range(3)]

def logD(x):
    return log(x)/log(2)

logratio = map(logD,ratio)
```

et on obtient :

```
In [4]: ratio
Out[4]: [7.523860206447286, 9.118789882406599, 8.5098312160934]

In [5]: list(logratio)
Out[5]: [2.940628541715559, 3.133284732580891, 3.128693841844642]
```

Bon, il semble que quand la taille double, le temps est multiplié par 2^3.
Il ne semble donc pas aberrant de considérer que le temps de calcul est de aN^3 mais que vaut a ?

```
In [6]: temps(range(400))
Out[6]: 4.005484320001415
```

$4,00 = a \times 400^3$ donc $a \approx 6,25 \times 10^{-8}$
Donc pour $N = 1000$, on devrait avoir un temps de $6,25 \times 10^{-8} \times 10^9 = 62,5$

```
In [7]: temps(range(1000))
Out[7]: 68.54615448799996
```

Presque mais Python reste Python...

Voici la même chose en C, sans vraiment chercher à optimiser le code.

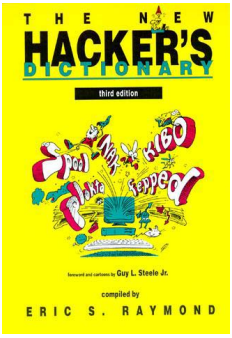
```c
#include <stdio.h>
#include <time.h>

typedef int Liste[13000];

int trois_sommes(int N)
{
  int cpt = 0;
  Liste liste;

  for ( int k = 0; k < N; k++)
    {
      liste[k] = k - (N / 2);
    }

  for (int i = 0; i < N; i++)
    {
      for (int j = i + 1; j < N; j++)
        {
          for (int k = j + 1; k < N; k++)
            {
              if (liste[i] + liste[j] + liste[k] == 0) {cpt++;}
            }
        }
     }
  return cpt;
}

void chrono(int N)
{
  clock_t temps_initial, temps_final;
  float temps_cpu;

  temps_initial = clock();
  trois_sommes(N);
  temps_final   = clock();
  temps_cpu = ((double) temps_final - temps_initial) / CLOCKS_PER_SEC;
  printf("Temps en sec pour %d : %f\n",N, temps_cpu);

}

int main(void)
{
  chrono(100);
  chrono(200);
  chrono(400);
  chrono(800);
  chrono(1600);
  chrono(3200);
  chrono(6400);
  chrono(12800);

  return 1;
}
```

Et on obtient :

```
$ gcc  -std=c99 -Wall -Wextra  -Werror -pedantic -O4 -o somm3 Trois_Sommes.c
$ ./somm3
Temps en sec pour 100 : 0.00000
Temps en sec pour 200 : 0.000000
Temps en sec pour 400 : 0.020000
Temps en sec pour 800 : 0.090000
Temps en sec pour 1600 : 0.720000
Temps en sec pour 3200 : 5.760000
Temps en sec pour 6400 : 45.619999
Temps en sec pour 12800 : 360.839996
```

On a la même évolution en N^3 avec un rapport de 8 entre chaque doublement de taille mais la constante est bien meilleure :
$45,61 = a \times 6400^3$ d'où $a \approx 1,74 \times 10^{-10}$
$360,84 = a \times 12800^3$ d'où $a \approx 1.72 \times 10^{-10}$
Conclusion : Python ou C, l'algo est le même et on constate la même progression selon le cube de la taille.
Ce qui nous intéresse en premier lieu (et ce que l'on mesurera aux concours) est donc un **ORDRE DE CROISSANCE**.
Cependant, les temps sont bien distincts : ici, C va 400 fois plus vite que Python ;-)
On a donc une approche expérimentale et les expériences sont plus facilement menées que dans les autres sciences et on peut en effectuer de très grands nombres.
La mauvaise nouvelle, c'est qu'il est difficile de mesurer certains facteurs comme l'usage du cache, le comportement du ramasse-miettes, etc.
En C, on peut regarder le code assembleur généré mais avec Python, c'est plus mystérieux.
La bonne nouvelle, c'est qu'on peut cependant effectuer une analyse mathématique pour confirmer nos hypothèses expérimentales.

4.2 Notations



> *Brooks's Law [prov.]*
> *« Adding manpower to a late software project makes it later » – a result of the fact that the expected advantage from splitting work among N programmers is $O(N)$, but the complexity and communications cost associated with coordinating and then merging their work is $O(N^2)$*
>
> in « The New Hacker's Dictionnary »
> http://outpost9.com/reference/jargon/jargon_17.html#SEC24

Les notations de LANDAU(1877-1938) ont en fait été crées par Paul BACHMANN(1837-1920) en 1894, mais bon, ce sont tous deux des mathématiciens allemands.
Par exemple, si l'on considère l'expression :

$$f(n) = n + 1 + \frac{1}{n} + \frac{75}{n^2} - \frac{765}{n^3} + \frac{\cos(12)}{n^{37}} - \frac{\sqrt{765\,481}}{n^{412}}$$

Quand n est « grand », disons 10 000, alors on obtient :

$$f(10\,000) = 10\,000 + 1 + 0,0001 + 0,00000000075 - 0,0000000000000765 + \text{ peanuts}$$

Tous les termes après n comptent pour du beurre quand n est « grand ». Donnons une définition pour plus de clarté :

Définition 1 : « Grand O »

Soit f et g deux fonctions de $\mathbb{N}$ dans $\mathbb{R}$. On dit que f est un « grand O » de g et on note $f = O(g)$ ou $f(n) = O(g(n))$ si, et seulement si, il existe une constante strictement positive C telle que $|f(n)| \leqslant C|g(n)|$ pour tout $n \in \mathbb{N}$.

Dans l'exemple précédent, $\frac{1}{n} \leqslant \frac{1}{1} \times 1$ pour tout entier n supérieur à 1 donc $\frac{1}{n} = O(1)$.
De même, $\frac{75}{n^2} \leqslant \frac{75}{1^2} \times 1$ donc $\frac{75}{n^2} = O(1)$ mais on peut dire mieux : $\frac{75}{n^2} \leqslant \frac{75}{1} \times \frac{1}{n}$ et ainsi on prouve que $\frac{75}{n^2} = O\left(\frac{1}{n}\right)$.
En fait, un grand O de g est une fonction qui est au maximum majorée par un multiple de g.
On peut cependant faire mieux si l'on a aussi une minoration.
C'est le moment d'introduire une nouvelle définition :

Définition 2 : « Grand Oméga »

Soit f et g deux fonctions de $\mathbb{R}$ dans lui-même. On dit que f est un « grand Oméga » de g et on note $f = \Omega(g)$ ou $f(n) = \Omega(g(n))$ si, et seulement si, il existe une constante strictement positive C telle que $|f(n)| \geqslant C|g(n)|$ pour tout $n \in \mathbb{N}^*$.

Remarque 1 :

Comme Ω est une lettre grecque, on peut, par esprit d'unification, parler de « grand omicron » au lieu de « grand O »...

Si l'on a à la fois une minoration et une majoration, c'est encore plus précis et nous incite à introduire une nouvelle définition :

Définition 3 : « Grand Théta »

$f = \Theta(g) \iff f = O(g) \quad \wedge \quad f = \Omega(g)$

Voici maintenant une petite table pour illustrer les différentes classes de complexité rencontrées habituellement :

coût \ n	100	1000	10^6	10^9
$\log_2(n)$	≈ 7	≈ 10	≈ 20	≈ 30
$n \log_2(n)$	≈ 665	$\approx 10\,000$	$\approx 2 \cdot 10^7$	$\approx 3 \cdot 10^{10}$
n^2	10^4	10^6	10^{12}	10^{18}
n^3	10^6	10^9	10^{18}	10^{27}
2^n	$\approx 10^{30}$	$> 10^{300}$	$> 10^{10^5}$	$> 10^{10^8}$

Gardez en tête que l'âge de l'Univers est environ de 10^{18} secondes...

4.3 Analyse mathématique

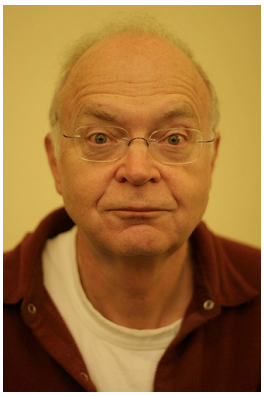


FIGURE 4 – D. E. Knuth
Né en 1938

Une bonne lecture de chevet est le troisième volume de « Zi Arte » (Knuth [1998]).
Le temps d'exécution est la somme des produits (coût × fréquence) pour chaque opération.
— le coût dépend de la machine, du compilateur, du langage ;
— la fréquence dépend de l'algorithme, de la donnée en entrée.
Par exemple, l'accès à un élément dans un itérateur en Python n'est pas gratuit.

```
In [12]: xs = range(-500,501)

In [13]: %timeit xs[900]
10000000 loops, best of 3: 129 ns per loop
In [14]: %timeit xs[900] + xs[800]
1000000 loops, best of 3: 292 ns per loop
In [15]: %timeit xs[900] + xs[800] + xs[700]
1000000 loops, best of 3: 424 ns per loop
```

Mais si on travaille sur une liste :

```
In [22]: xs = list(range(-500,501))

In [23]: %timeit xs[900]
10000000 loops, best of 3: 40.7 ns per loop
In [24]: %timeit xs[900] + xs[800]
10000000 loops, best of 3: 114 ns per loop
In [25]: %timeit xs[900] + xs[800] + xs[700]
10000000 loops, best of 3: 165 ns per loop
```

oui mais :

```
In [26]: %timeit range(-500,501)
1000000 loops, best of 3: 292 ns per loop
In [27]: %timeit list(range(-500,501))
100000 loops, best of 3: 13.9 µs per loop
```

Dans notre algorithme des 3 sommes en python, on peut donc gagner du temps :

```
def trois_sommes(xs):
    N = len(xs)
```

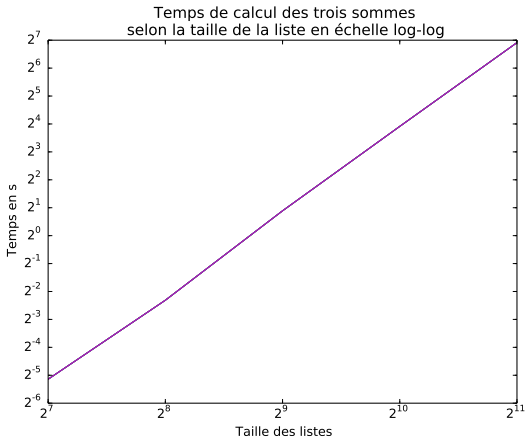
```
    cpt = 0
    for i in range(N):
        xi = xs[i]
        for j in range(i + 1, N):
            sij = xi + xs[j]
            for k in range(j + 1, N):
                if sij + xs[k] == 0:
                    cpt += 1
    return cpt
```

C'est mieux :

```
In [28]:  xs = list(range(-50,51))
In [29]: %timeit trois_sommes(xs)
100 loops, best of 3: 12.9 ms per loop

In [30]:  xs = list(range(-100,101))
In [31]: %timeit trois_sommes(xs)
10 loops, best of 3: 94.4 ms per loop

In [32]:  xs = list(range(-200,201))
In [33]: %timeit trois_sommes(xs)
1 loops, best of 3: 851 ms per loop

In [34]:  xs = list(range(-400,401))
In [35]: %timeit trois_sommes(xs)
1 loops, best of 3: 7.04 s per loop
```

En C, ça ne changera rien, car le compilateur est intelligent et a remarqué tout seul qu'il pouvait garder en mémoire certains résultats.
On peut visualiser en échelle log-log :

```
tailles = [2**k for k in range(7,11)]*
listes = [list(range(- N//2, N//2 + 1)) for N in tailles]
t = [temps(xs) for xs in listes]
for taille in tailles:
    plt.loglog(tailles, t, basex = 2, basey = 2)
plt.title('Temps de calcul selon la taille de la liste en échelle log-log')
plt.xlabel('Taille des listes')
plt.ylabel('Temps en s')
```



C'est assez rectiligne.
Regardons de nouveau le code des trois sommes et comptons le nombre d'opérations :

OPÉRATION	FRÉQUENCE
Déclaration de la fonction et du paramètre (l. 1)	2
Déclaration de N, cpt et i (l. 2, 3 et 4)	3
Affectation de N, cpt et i (l. 2, 3 et 4)	3
Déclaration de xi (l. 5)	N
Affectation de xi (l. 5)	N
Accès à xs[i] (l. 5)	N
Déclaration de j (l.6)	N
Calcul de l'incrément de i (l. 6)	N
Affectation de j (l.6)	N
Déclaration de sij (l. 7)	S_1
Affectation de sij (l. 7)	S_1
Accès à xs[j] (l.7)	S_1
Somme (l.7)	S_1
Déclaration de k (l.8)	S_1
Incrément de j (l. 8)	S_1
Affectation de k (l.8)	S_1
Accès à x[k] (l.9)	S_2
Calcul de la somme (l.9)	S_2
Comparaison à 0 (l.9)	S_2
Incrément de cpt (l.9)	entre 0 et S_2
Affectation de la valeur de retour (l.11)	1

Que valent S_1 et S_2 ?

$$S_1 = \sum_{i=0}^{N-1} N-(i+1) = \sum_{i'=0}^{N-1} i' = \frac{N(N-1)}{2}$$

$$\begin{aligned}
S_2 &= \sum_{i=0}^{N-1}\sum_{j=i+1}^{N-1} N-(j+1) \\
&= \sum_{i=0}^{N-1}\sum_{j'=0}^{N-(i+2)} j' \\
&= \sum_{i=0}^{N-2} \frac{(N-(i+2))(N-(i+1))}{2} \\
&= \sum_{i'=1}^{N-2} \frac{i'(i'+1)}{2} \\
&= \frac{1}{2}\left(\sum_{i=1}^{N-2} i^2 + \sum_{i=1}^{N-2} i\right) \\
&= \frac{1}{2}\left(\frac{(N-2)(2N-3)(N-1)}{6} + \frac{(N-2)(N-1)}{2}\right) \\
&= \frac{N(N-1)(N-2)}{6}
\end{aligned}$$

Notons a le temps constant d'affectation, d le temps constant de déclaration, x le temps constant d'accès à une cellule, s le temps constant d'une somme, c le temps constant d'une comparaison.
Le temps d'exécution vérifie :

$$\tau(N) \leqslant (2d+3d+3a+a) + (d+a+x+d+s+a)N + (d+a+x+s+d+s+a)S_1 + (x+s+c+s)S_2$$

Or $S_1 \sim N^2$ et $S_2 \sim N^3$ quand N est « grand ».
Finalement...

$$\tau(N) = O(N^3)$$

...et ce, que l'on utilise C, Python, BrainFuck (https://fr.wikipedia.org/wiki/Brainfuck), etc.
C'est aussi ce que l'on a observé expérimentalement

♪ À retenir (1) :

Moralité, on ne s'occupe que de ce qu'il y a dans la boucle la plus profonde et on oublie le reste....

5 Plus fort que la dichotomie : l'algorithme de Héron

Le mathématicien Héron d'Alexandrie n'avait pas attendu Newton et le calcul différentiel pour trouver une méthode permettant de déterminer une approximation de la racine carrée d'un nombre entier positif puisqu'il a vécu seize siècles avant Sir Isaac.
Si x_n est une approximation strictement positive par défaut de $\sqrt{a}$, alors a/x_n est une approximation par excès de $\sqrt{a}$ et vice-versa.
La moyenne arithmétique de ces deux approximations est $\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ et constitue une meilleure approximation que les deux précédentes.
On peut montrer c'est une approximation par excès (en développant $\left(x_n - \sqrt{a}\right)^2$ par exemple).
En voici deux versions, une récursive et une itérative.

```
1  def heron_rec(a,fo,n):
2          if n == 0:
3                  return fo
4          return heron_rec(a,(fo + a / fo) / 2, n - 1)
5
6  def heron_it(a,fo,n):
7          app = fo
8          for k in range(n):
9                  app = (app + a / app) / 2
10         return app
```

Ce qui donne par exemple :

```
1  In [12]: heron_rec(2,1,6)
2  Out[12]: 1.414213562373095
3
4  In [13]: heron_it(2,1,6)
5  Out[13]: 1.414213562373095
6
7  In [14]: from math import sqrt
8
9  In [15]: sqrt(2)
10 Out[15]: 1.4142135623730951
```

Est-ce que la suite des valeurs calculées par la boucle converge vers $\sqrt{2}$? Nous aurons besoin de deux théorèmes que nous admettrons et dont nous ne donnerons que des versions simplifiées.

Théorème 1 : Théorème de la limite monotone

Toute suite croissante majorée (ou décroissante minorée) converge.

Un théorème fondamental est celui du point fixe. Il faudrait plutôt parler des théorèmes du point fixe car il en existe de très nombreux avatars qui portent les noms de mathématiciens renommés : Banach, Borel, Brouwer,

Kakutani, Kleene,... Celui qui nous intéresserait le plus en informatique est celui de Knaster-Tarski. Ils donnent tous des conditions d'existence de points fixes (des solutions de $f(x) = x$ pour une certaine fonction). Nous nous contenterons de la version light vue au lycée.

Théorème 2 : Un théorème light du point fixe

Soit I un intervalle fermé de $\mathbb{R}$, soit f une fonction de I vers I et soit (r_n) une suite d'éléments de I telle que $r_{n+1} = f(r_n)$ pour tout entier naturel n. SI (r_n) est convergente ALORS sa limite est UN point fixe de f appartenant à I.

Cela va nous aider à étudier notre suite définie par $r_{n+1} = f(r_n) = \frac{r_n + \frac{2}{r_n}}{2}$ et $r_0 = 1$. On peut alors démontrer par récurrence que

1. pour tout entier naturel non nul n, on a $r_n \geqslant \sqrt{2}$;
2. la suite est décroissante ;
3. $\sqrt{2}$ est l'unique point fixe positif de f.

On peut alors conclure et être assuré que notre algorithme va nous permettre d'approcher $\sqrt{2}$.

Quelle est la vitesse de convergence ? Ici, cela se traduit par « combien de décimales obtient-t-on en plus à chaque itération ? ». Introduisons la notion d'ordre d'une suite :

Définition 4 : Ordre d'une suite - Constante asymptotique d'erreur

Soit (r_n) une suite convergeant vers ℓ. S'il existe un entier $k > 0$ tel que :

$$\lim_{n \to +\infty} \frac{|r_{n+1} - \ell|}{|r_n - \ell|^k} = C$$

avec $C \neq 0$ et $C \neq +\infty$ alors on dit que (r_n) est d'ordre k et que C est la constante asymptotique d'erreur.

Déterminons l'ordre et la constante asymptotique d'erreur de la suite de Babylone donnant une approximation de $\sqrt{2}$. On démontre que

$$\left|r_{n+1} - \sqrt{2}\right| = \left|\frac{(r_n - \sqrt{2})^2}{2r_n}\right|$$

Ainsi la suite est d'ordre 2 et sa constante asymptotique est $\frac{1}{2\sqrt{2}}$. Ici, on peut même avoir une idée de la vitesse sans étude asymptotique.
En effet, on a $r_n \geqslant 1$ donc

$$\left|r_{n+1} - \sqrt{2}\right| \leqslant \left|(r_n - \sqrt{2})^2\right|$$

Le nombre de décimales d'un nombre positif plus petit que 1 est $-\log_{10} x$. En posant $d_n = -\log_{10} |r_n - \sqrt{2}|$ on obtient donc que $d_{n+1} \geqslant 2d_n$: à chaque tour de boucle, on double au minimum le nombre de bonnes décimales. La précision de la machine étant de 16 décimales si les nombres sont codés en binaires sur 64 bits, **il faut au maximum 6 itérations pour obtenir la précision maximale**.

Références

H. Abelson, G. Sussman et J. Sussman,
Structure and interpretation of computer programs :,
vol. MIT electrical engineering and computer science series, MIT Press, ISBN 0-262-51087-1, 1996.

A. Benoit, Y. Robert et F. Vivien,
A Guide to Algorithm Design : Paradigms, Methods, and Complexity Analysis,
vol. Applied Algorithms and Data Structures series, Chapman & Hall/CRC, août 2013.

A. Brygoo, T. Durand, M. Pelletier, C. Queinnec et M. Soria,
Programmation récursive (en Scheme) - Cours et exercices corrigés,
vol. Informatique, Dunod, ISBN 9782100528165, 2004.

T. H. Cormen, C. E. Leiserson, R. L. Rivest et C. Stein,
Introduction to Algorithms, Third Edition,
The MIT Press, 3rd édition, ISBN 0262033844, 9780262033848, 2009.

E. W. Dijkstra,
« Algol 60 translation : An algol 60 translator for the x1 and making a translator for algol 60 »,
cwireport MR 34/61, Stichting Mathematisch Centrum, adresse : http://oai.cwi.nl/oai/asset/9251/9251A.pdf, 1961,
(ALGOL Bulletin, 1).

F. de Dinechin,
« Page personnelle », adresse : http://perso.citi-lab.fr/fdedinec/, 2014.

G. Dowek,
Les principes des langages de programmation,
Éditions de l'École Polytechnique, 2008.

J. Dufourd, D. Bechmann et Y. Bertrand,
Spécifications algébriques, algorithmique et programmation,
vol. I.I.A. Informatique intelligence artificielle, InterEditions, ISBN 9782729605810, 1995.

D. Goldberg,
« What every computer scientist should know about floating point arithmetic »,
ACM Computing Surveys, vol. 23, n° 1, p. 5–48, 1991.

F. Goualard,
« Page personnelle », adresse : http://goualard.frederic.free.fr/, 2014.

W. Kahan,
« Page personnelle », adresse : http://www.cs.berkeley.edu/~wkahan/, 2014.

D. E. Knuth,
The Art of Computer Programming, Volume 1 (3rd Ed.) : Fundamental Algorithms,
Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, ISBN 0-201-89683-4, 1997.

D. E. Knuth,
The Art of Computer Programming, Volume 3 : (2Nd Ed.) Sorting and Searching,
Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, ISBN 0-201-89685-0, 1998.

J.-M. Muller, N. Brisebarre, F. de Dinechin, C.-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé et S. Torres,
Handbook of Floating-Point Arithmetic,
Birkhäuser Boston, 2010,
ACM G.1.0 ; G.1.2 ; G.4 ; B.2.0 ; B.2.4 ; F.2.1., ISBN 978-0-8176-4704-9.

M. Pichat,
« Correction d'une somme en arithmétique à virgule flottante »,
Numer. Math., vol. 19, p. 400–406, 1972.

G. J. E. Rawlins,
Compared to What ? An Introduction to the Analysis of Algorithms,
Computer Science Press, 1992.

R. Sedgewick et P. Flajolet,
An Introduction to the Analysis of Algorithms,
Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, ISBN 0-201-40009-X, adresse : http://aofa.cs.princeton.edu/home/, 1996.

R. Sedgewick et K. Wayne,
Algorithms, 4th Edition.,
Addison-Wesley, ISBN 978-0-321-57351-3, adresse : http://algs4.cs.princeton.edu/home/, 2011.