LA COMPLEXITE C'EST SIMPLE COMME LA DICHOTOMIE (LYCEE MATHS/ISN)

Guillaume CONNAN (*)
Irem de Nantes

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 de Heron d'Alexandrie. 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.

Il ne s'agit pas d'une activité « clé en main ». L'orientation est plutôt vers la formation continue car nous avons besoin d'en savoir un peu plus que nos élèves avant de leur introduire une notion. Cependant, comme cela a déjà été évoqué (Aldon et coll. [2012]) dans cette revue, l'étude de la complexité peut être abordée au lycée et les auteurs de cet article avaient proposé de nombreux exemples.

Nous proposons ici des approches théoriques et expérimentales de ce problème. On met ainsi en évidence qu'une notion mathématique importante (les suites numériques) peut être illustrée par un problème informatique théorique et pratique et inversement, en cours d'ISN par exemple, on fait prendre conscience aux élèves que l'informatique, ce n'est pas taper sur des touches, c'est surtout noircir du papier avec des raisonnements très proches des mathématiques.

Nous avons du mal en français à trouver des termes pour désigner les domaines gravitant autour de l'informatique. Ce dernier mot est souvent remplacé par Tice (avec un T comme technologie... cela restreint le domaine et permet de comprendre pourquoi pendant si longtemps informatique rimait avec bureautique) ou bien par numérique et la dernière mode est de parler de code. Les anglo-saxons parlent de Computer Science, désignation qui a l'avantage de comporter le mot science. Computer Science est en relation avec les domaines Scientific Computing et Computer Engineering mais n'est pas du tout réduit à ces deux composantes proches.

Le département de Computer Science de l'Université de Boston s'adresse ainsi à ses futur(e)s étudiant(e)s (University [2015]) :

> *La science de l'ordinateur concerne la résolution de problèmes. Ainsi, les qualités requises dans ce domaine sont entre autres d'être passionné par la recherche de solutions élégantes, d'être capable d'utiliser analyse mathématique et rigueur logique afin d'évaluer ces solutions, de faire preuve de créativité pour modéliser des problèmes complexes en utilisant l'abstraction, être attentif aux détails et aux affirmatins cachées, être capable de reconnaître des variantes d'un même problème dans différents contextes, et d'adapter des solutions efficaces connues à de nouveaux contextes. Si vous aimez résoudre des énigmes, alors la science de l'ordinateur est faite pour vous !*

Cela ressemble beaucoup à ce que l'on attend d'élèves étudiant les mathématiques !...

(*) guillaume.connan@univ-nantes.fr

1. — Le lancer d'œuf ou comment faire de l'informatique sans ordinateur...

1.1. *L'expérience*

Un groupe d'élèves a fabriqué des œufs synthétiques dans le cadre des TPE (très transversal : biologie, physique, mathématique et informatique) et voudrait étudier sa solidité et en particulier sa résistance en cas de chute.

Il mettent au point un protocole d'expérimentation. Ils acheminent k œufs en bas du bâtiment principal de leur lycée qui a $N = 2^n$ étages, le rez-de-chaussée portant le numéro 0. Le but de l'expérimentation est de déterminer l'étage à partir duquel lancer un œuf par la fenêtre lui (l'œuf..) 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 l'œuf survit au lancer, on peut le réutiliser. Pour économiser le temps et le souffle des expérimentateurs, on va chercher un algorithme qui minimise le nombre d'essais effectués (et non pas le nombre d'œufs détruits).

Une première idée serait de commencer au premier étage et de lancer un œuf de chaque étage jusqu'à atteindre l'étage fatal.

Si jamais l'étage fatal est le dernier, cela nécessitera d'effectuer $N - 2$ 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 $(N - 2)/2$ essais en moyenne. Mais on peut faire mieux...

1.2. *Diviser pour régner*

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.

Sun Zi
(544 496
av. J.-C.)

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 de Sun Zi
(VIème siècle avant JC)

```
inf ← Étage inférieur
sup ← Étage supérieur
milieu ← (inf + sup)/2
Si estFatal(milieu) Alors
  | ChercherEntre(inf,milieu)
Sinon
  | ChercherEntre(milieu, 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 directement au-dessus de l'étage inférieur. La fonction devient celle de l'encadré 1. On peut donner une version non récursive équivalente mais moins naturelle : se dire « je recommence l'expérience dans la moitié supérieure » fait clairement penser à un processus récursif (encadré 2).

```
Fonction ChercherEntre( inf, sup : Entiers ) : Entier
{ pré-condition: il existe au moins un étage fatal entre inf et sup, inf n'est pas fatal et inf < sup }
{ invariant: le plus petit étage fatal est entre inf (non compris) et sup }
{ post-condition: la valeur retournée est le plus petit étage fatal }
Si sup == inf + 1 Alors
    Retourner sup
Sinon
    milieu ← (inf + sup)/2
    Si estFatal(milieu) Alors
        ChercherEntre(inf,milieu)
    Sinon
        ChercherEntre(milieu, sup)
    FinSi
FinSi
```

encadré 1

encadré 2

```
Fonction ChercherEntre( N : Entier ) : Entier
{ pré-condition: il existe au moins un étage fatal entre inf et sup, inf n'est pas fatal et inf < sup }
{ invariant: le plus petit étage fatal est entre inf (non compris) et sup }
{ post-condition: la valeur retournée est le plus petit étage fatal }
inf ← 0
sup ← N
TantQue sup > inf + 1 Faire
    { le plus petit étage fatal est entre inf (non compris) et sup et sup > inf + 1 }
    milieu ← (inf + sup)/2
    Si estFatal(milieu) Alors
        sup = milieu
    Sinon
        inf = milieu
    FinSi
FinTantQue
{ le plus petit étage fatal est entre inf (non compris) et sup et sup = inf + 1 }
Retourner sup
{  la valeur retournée est le plus petit étage fatal }
```

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] ?

1 — Pour répondre à la première question, il suffit de remarquer qu'au départ, l'intervalle d'étude est l'ensemble $\{0, 1, ..., N-1\}$ 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

$$l_i = \frac{N}{2^i} = \frac{2^n}{2^i} = 2^{n-i}.$$

Ainsi $l_n = 1$ 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 $\{inf, inf+1\}$ et *inf* n'est pas fatal donc $inf+1$ l'est et est la valeur cherchée.

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 les bâtiments d'un lycée. Il faut compter le temps de chute, le temps de remontée des œufs encore en état, le temps de manipulation pour passer les œufs par la fenêtre, l'examen de l'état de l'œuf ayant chu. Selon l'efficacité des muscles et l'état de santé des expérimentateurs, 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 de l'œuf par un expert [2].

... Mais c'est un peu compliqué... On va supposer que l'ascenseur est ultra-performant et que les œufs sont très légers ce qui entraîne que tous ces temps sont à peu près constants, quelque soit l'étage [3].

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 les jambes des expérimentateurs.

1.3. *Le bug de Java*

Pendant neuf ans, jusqu'en 2006, les fonctions de recherche dichotomique en Java cachaient un bug... Voici (encadré 3) ce que l'on trouvait dans la bibliothèque java. util .Arrays :

1 Un algorithme ne s'exécute pas. On va voir qu'il n'existe pas un temps par algorithme.

2 C'est ce qui correspondra par exemple en informatique au parcours d'une liste chaînée.

3 C'est ce qui correspondra par exemple en informatique au parcours d'un tableau statique.

```
1   public static int binarySearch(int[] a, int key) {
2           int low  = 0;
3           int high = a.length - 1;
4
5           while (low <= high) {
6               int mid    = (low + high) / 2;
7               int midVal = a[mid];
8
9               if (midVal < key)
10                  low = mid + 1
11              else if (midVal > key)
12                  high = mid - 1;
13              else
14                  return mid; // key found
15          }
16          return -(low + 1);  // key not found.
17      }
```

Regardez la ligne 6 :

int mid = (low + high) / 2;

Tout va bien... mais en fait il faut se souvenir que les entiers de type int sont codés sur 32 bits en complément à 2^{32} et que le successeur de $2^{32} – 1$ n'est pas 2^{32} mais son opposé car on représente les nombres relatifs sur machine d'une manière spéciale[4]. Étudier la manière de représenter les relatifs sur machine est un problème intéressant.

Le bug a été corrigé[5] en remplaçant cette ligne par :

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 $n = \log_2(N)$ œufs pour tester ?

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

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

Ainsi, s'il nous reste assez d'œufs (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

$$k - 1 + \frac{N}{2^{k-1}} - 1 \text{ tests.}$$

2. Que se passe-t-il si nous n'avons que 2 œufs 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...

4 Voir par exemple le bug de l'an 2038 : https://fr.wikipedia.org/wiki/Bug_de_l%27an_2038

5 http://bugs.java.com/bugdatabase/view_bug.do?bug_id=5045582

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 œuf (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 l'immeuble. Or, pour tout entier naturel N, il existe un entier n tel que:

$$2^n \leq N < 2^{n+1} .$$

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

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

donc $T(N) = n = \lfloor \log_2 (N) \rfloor$ [6].

4. Et si nous avions fait de la trichotomie ?

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 trois)

$$2\log_3(N) = 2\,\frac{ln(N)}{ln(2)} \times \frac{ln(2)}{ln(3)} \approx 1{,}3\log_2(N)$$

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 :

$$\begin{aligned} xy &= (10^m x_1 + x_2)\,(10^m y_1 + y_2) \\ &= 10^{2m} x_1 y_1 + 10^m (x_2 y_1 + x_1 y_2) + x_2 y_2 . \end{aligned}$$

6 On note $\lfloor x \rfloor$ la partie entière inférieure de x

```
Fonction MUL( x : entier , y : entier ) : entier
Si n==1 Alors
    Retourner x.y
Sinon
    m ← ⌊n/2⌋
    x1 ← ⌊x/10^m⌋
    x2 ← x mod 10^m
    y1 ← ⌊y/10^m⌋
    y2 ← y mod 10^m
    a ← MUL(x1, y1, m)
    b ← MUL(x2, y1, m)
    c ← MUL(x1, y2, m)
    d ← MUL(x2, y2, 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 n donc le temps d'exécution est défini par :

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

Peut-on exprimer $T(n)$ 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...

Le grand Андрей Николаевич Колмогоров (Andreï Nikolaïevitch Kolmogorov) finit même par conjecturer, dans les années 1950, que l'on ne pourra jamais trouver un algorithme de multiplication d'entiers en $o(n^2)$. Lors d'un séminaire sur ce sujet en 1960, un jeune étudiant soviétique, Анатолий Алексеевич Карацуба (Anatoly Alexievitch Karatsouba) propose cependant au Maître une solution plus simple et navrante de simplicité...

Il fait remarquer à Колмогоров que :

$$bc + ad = ac + bd - (a - b)(c - d)$$

En quoi cela simplifie-t-il le problème ? Quelle est alors la nouvelle complexité ?

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

On voit alors que l'on n'a plus que trois produits différents à calculer au lieu de quatre ce qui va améliorer notre complexité. La relation de récurrence devient :

$$x_k = 3x_{k-1} + \lambda\, 2^k \qquad x_0 = 1$$

On obtient :

$$x_k = 3(3x_{k-2} + \lambda' 2^{k-1}) + \lambda\, 2^k =$$
$$3^k x_0 + \sum_{i=1}^{k} \Lambda_k 2^k = 3^k + k\Lambda_k 2^k =$$
$$n^{\log_2(3)} + \Lambda_k n \log n \sim n^{\log_2(3)} .$$

ce qui est mieux, car $\log_2(3) \approx 1{,}6$.

Bref, avec un peu de réflexion, on peut améliorer parfois certains algorithmes. D'autres algorithmes ont permis depuis d'améliorer encore la complexité asymptotique de la multiplication (cf Knuth [1998]).

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 **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	$1+2^{-10}$	$1+2\times2^{-10}$	$1+3\times2^{-10}$	...	$1+2^{10}\times2^{-10}$

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. Nous en présentons (ci-dessous) une version en pseudo-code et une autre en Python... Bon, le pseudo-code a longtemps été en fait du pseudo-Pascal mais sacrifions à la mode et écrivons du pseudo-Python à côté du code Python. Petit jeu : qui est qui ? Nous obtenons :

```
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
```

```
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. Pour plus de précisions, nhésitez pas à parcourir :

http://download.tuxfamily.org/tehessinmath/les%20pdf/JA2014.pdf

ou

http://download.tuxfamily.org/tehessinmath/les%20pdf/PolyAnalyse14.pdf

```
Fonction racineDicho(prec : Réel ) : Réel
cpt ← 0
inf ← 1
sup ← 2
TantQue sup - inf > prec Faire
    m ← inf + (sup − inf)/2
    cpt ← cpt + 1
    Si m × m ⩽ 2 Alors
        inf ← m
    Sinon
        sup ← m
    FinSi
FinTantQue
Retourner sup,cpt
```

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. *Le problème*

Le problème de la complexité peut être abordé de manière plus habituelle en étudiant un algorithme utilisant des boucles, plus familières des élèves que les fonctions récursives et les logarithmes de base 2.

On dispose d'une liste de N nombres. Déterminez le nombre de triplets dont la somme est nulle.

Ici, diviser va difficilement nous permettre de régner. Intuitivement, on peut facilement diviser notre liste en 2 et rechercher les triplets annulateurs dans chaque moitié mais il faudra recommencer sur toute la liste pour chercher des triplets d'éléments appartenant à des moitiés différentes donc on a peu de chance de gagner du temps.

La complexité théorique est elle-même facile à calculer : nous en parlerons ensuite. Nous allons plutôt commencer par une approche expérimentale pour savoir si elle se rapproche des résultats théoriques et si elle dépend du langage utilisé. On essaiera ensuite d'améliorer un peu l'algorithme et voir si cela a une influence sur la complexité au sens où nous allons la calculer.

4.2. *Méthode expérimentale*

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
```

```
Fonction trois_sommes( xs : Liste
d'entiers ) : Entier
N ← longueur de la liste xs
cpt ← 0
Pour i de 0 à N-1 Faire
    Pour j de i+1 à N-1 Faire
        Pour k de j+1 à N-1 Faire
            Si xs[i] + xs[j] + xs[k] = 0 Alors
                cpt ← cpt + 1
            FinSi
        FinPour
    FinPour
FinPour
Retourner cpt
```

Comparons les temps pour différentes tailles. On teste sur des listes d'entiers aléatoirement choisis selon une loi uniforme dans l'intervalle [[–10 000; 10 000]] :

```
In [1]: %timeit trois_sommes([randint(-10000,10000) for _ in range(100)])
100 loops, best of 3: 21.3 ms per loop

In [2]: %timeit trois_sommes([randint(-10000,10000) for _ in range(200)])
10 loops, best of 3: 168 ms per loop

In [3]: %timeit trois_sommes([randint(-10000,10000) for _ in range(400)])
1 loops, best of 3: 1.38 s per loop
```

On aurait pu être plus concis en utilisant des listes par compréhension :

```
def trois_sommes_comp(xs):
    n = len(xs)
    return len([(i,j,k) 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 ])
```

Mais les temps sont les mêmes :

```
In [24]: %timeit trois_sommes_comp([randint(-10000,10000) for _ in range(100)])
10 loops, best of 3: 21.2 ms per loop

In [25]: %timeit trois_sommes_comp([randint(-10000,10000) for _ in range(200)])
10 loops, best of 3: 168 ms per loop

In [26]: %timeit trois_sommes_comp([randint(-10000,10000) for _ in range(400)])
1 loops, best of 3: 1.39 s per loop
```

Bon, il semble que quand la taille double, le temps est multiplié par $8 = 2^3$. Il ne semble donc pas aberrant de considérer que le temps de calcul est de aN^3 mais que vaut a ?

$$1{,}39 = a \times 400^3 \text{ donc } a \approx 2{,}17 \times 10^{-8}$$

Donc pour $N = 1000$, on devrait avoir un temps de $2{,}17 \times 10^{-8} \times 10^9 = 21{,}7$ s .

```
In [7]: %timeit trois_sommes([randint(-10000,10000) for _ in range(1000)])
1 loops, best of 3: 22.2 s per loop
```

Ça marche ! Voici la même chose en C, sans vraiment chercher à optimiser le code :

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

typedef int Liste[100000] ;

int trois_sommes( int N )
{
  int cpt = 0 ;
  Liste liste ;
  // on forme une liste d'entiers
  for ( int k = 0; k < N; k++ )
    {
      liste[k] = ( rand() % N ) - (N / 2) ;
    }

  // on utilise le même algorithme
  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 ;
}

// on compte le temps nécessaire à chaque recherche
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) ;

}

// on lance la recherche sur différents temps, en allant plus loin car C est plus rapide...
int main(void)
{
  for ( int i = 1; i < 8; i++ )
    {
      chrono( 100 << i ) ; // on multiplie le temps par 2 à chaque fois
    }
  return 1;
}
```

Et on obtient en compilant ce fichier en donnant quelques options au compilateur gcc :

```
$ gcc  -std=c99 -Wall -Wextra  -Werror -pedantic -O1 -o somm3 Trois_Sommes.c
$ ./somm3
Temps en sec pour 200 : 0.004842
Temps en sec pour 400 : 0.018059
Temps en sec pour 800 : 0.084797
Temps en sec pour 1600 : 0.622067
Temps en sec pour 3200 : 4.918256
Temps en sec pour 6400 : 39.424683
Temps en sec pour 12800 : 315.236694
```

On a la même évolution en N^3 avec un rapport de 8 entre chaque doublement de taille mais la constante est bien meilleure :

$$39{,}42 = a \times 6400^3 \qquad \text{d'où } a \approx 1{,}5 \times 10^{-10}$$

$$315{,}24 = a \times 12800^3 \qquad \text{d'où } a \approx 1{,}5 \times 10^{-10}$$

En Haskell :

```haskell
import Criterion.Main

-- On utilise le même algorithme qui filtre les éléments d'une liste mais écrit à la mode fonctionnelle : on forme
       ↪  la liste des triplets dont la somme est nulle. La fonction drop j permet d'éliminer les j premiers
       ↪  éléments d'une liste.
troisSommes :: [Int] -> [(Int,Int,Int)]
troisSommes xs =
    let inds = [0 .. (length xs -1)]
    in [(xs!!i , xs!!j , xs!!k) | i <- inds, j <- drop i inds, k <- drop j inds, xs!!i + xs!!j + xs!!k == 0]

-- Le "benchmarking" avec le module Criterion pour comparer les différents temps
main :: IO()
main = defaultMain [
        bgroup "TroisSommes" [ bench "3som 100"  $ whnf troisSommes [-50..49]
                             , bench "3som 200"  $ whnf troisSommes [-100..99]
                             , bench "3som 400"  $ whnf troisSommes [-200..199]
                             , bench "3som 800"  $ whnf troisSommes [-400..399]
                             , bench "3som 1600"  $ whnf troisSommes [-800..799]
                             , bench "3som 3200"  $ whnf troisSommes [-1600..1599]
                        ]
       ]
```

et on observe :

```
$ ./TroisSom --output troisSom.html
benchmarking TroisSommes/3som 100
time                 640.9 μs   (636.3 μs .. 645.0 μs)

benchmarking TroisSommes/3som 200
time                 4.322 ms   (4.305 ms .. 4.344 ms)

benchmarking TroisSommes/3som 400
time                 31.72 ms   (31.43 ms .. 32.08 ms)

benchmarking TroisSommes/3som 800
time                 261.7 ms   (259.7 ms .. 263.6 ms)

benchmarking TroisSommes/3som 1600
time                 2.318 s    (2.289 s .. 2.339 s)

benchmarking TroisSommes/3som 3200
time                 16.47 s    (15.00 s .. 17.51 s)
```

Encore un rapport de 8 pour chaque doublement de taille mais Haskell est 60 fois plus rapide que Python.

Conclusion : Python, Haskell 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 quand on parlera de complexité) est donc un ORDRE DE CROISSANCE. Cependant, les temps sont bien distincts : ici, C va 200 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 un très grand nombre à faible coût.

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 établies expérimentalement.

4.3. *Notations*

Loi de Brooks [*prov.*] :

« Ajouter des personnes à un projet informatique en retard accroît son retard. » Cela vient du fait que partager le travail entre N programmeurs permet d'espérer un gain en O(N) mais le coût associé au temps de communication dû à la coordination et à la fusion de leurs travaux est en $O(N^2)$

in « Le nouveau dictionnaire Hacker »
http://outpost9.com/reference/jargon/jargon_17.html#SEC24

Les notations de Landau(1877-1938) ont en fait été créées par Paul Bachmann(1837-1920) en 1894. Rappelons une définition pour plus de clarté :

Définition 1 (« Grand *O* ») *Soit f et g deux fonctions de* $\mathbf{N}$ *dans* $\mathbf{R}$. *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* $n \in \mathbf{N}$.

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, $\frac{75}{n^2} \leq \frac{75}{1} \times 1$ donc $\frac{75}{n^2} = O(1)$ mais on peut dire mieux : $\frac{75}{n^2} \leq \frac{75}{1} \times \frac{1}{n}$ et ainsi on prouve que $\frac{75}{n^2} = O(\frac{1}{n})$. 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 dé nition :

Définition 2 (« Grand Oméga ») *Soit f et g deux fonctions de* $\mathbf{R}$ *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* $|f(n)| \geq C|g(n)|$ *, pour tout* $n \in \mathbf{N}^*$.

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 »)

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

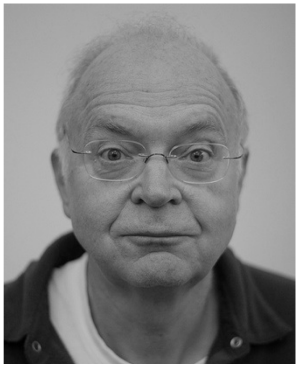
Voici maintenant une petite table pour illustrer les di érentes classes de complexité rencontrées habituellement (Voir tableau en haut de la page suivante)…

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

4.4. *Analyse mathématique*

Une bonne lecture de chevet est le troisième volume de The Art of Computer Programming (Knuth [1998]).

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	$\approx 10^{30}$	$> 10^{300}$	$> 10^{10^5}$	$> 10^{10^8}$



D. E. Knuth, Né en 1938

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.

Dans un langage comme C, on peut avoir le code assembleur associé et donc estimer le coût en cycles du processeur. Dans des langages de plus haut niveau comme Python, c'est beaucoup moins immédiat mais dans le cas de l'accès à un élément d'une liste on peut en gros le savoir car une liste de Python est construite en... C. Voici par exemple (voir ci-dessous) le code permettant d'accéder à un élément dans une liste Python.

On peut donc considérer ce temps d'accès comme constant, comme en C, mais ce coût existe. Dans notre algorithme des « 3 sommes » en Python, on peut donc gagner du temps en ne recalculant pas ce qui l'a déjà été car cela économise des temps d'accès (ci-contre, en haut).

C'est mieux mais le rapport est toujours de 8 entre chaque doublement…

```
1  static PyObject *
2  list_item(PyListObject *a, int i)
3  {
4          if (i < 0 || i >= a->ob_size) {
5                  if (indexerr == NULL)
6                          indexerr = PyString_FromString(
7                                  "list index out of range");
8                  PyErr_SetObject(PyExc_IndexError, indexerr);
9                  return NULL;
10         }
11         Py_INCREF(a->ob_item[i]);
12         return a->ob_item[i];
13 }
```

```
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
```

```
In [28]:  xs = list(range(-50,50))
In [29]: %timeit trois_sommes(xs)
100 loops, best of 3: 11.4 ms per loop

In [30]:  xs = list(range(-100,100))
In [31]: %timeit trois_sommes(xs)
10 loops, best of 3: 84.3 ms per loop

In [32]:  xs = list(range(-200,200))
In [33]: %timeit trois_sommes(xs)
1 loops, best of 3: 728 ms per loop

In [34]:  xs = list(range(-400,400))
In [35]: %timeit trois_sommes(xs)
1 loops, best of 3: 6.02 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 changer la taille des nombres utilisés. Cela ralentira le traitement mais le rapport de 8 est toujours mis en évidence (voir encadré 3 page suivante). On peut aussi visualiser en échelle log-log (encadré 4)…

Le « temps de calcul des trois sommes selon la taille de la liste en échelle log-log » est assez rectiligne (encadré 5).

Regardons de nouveau le code des trois sommes et comptons le nombre d'opérations élémentaires : ces opérations élémentaires représentent notre unité de temps théorique. Dans certains cas, il est difficile de les distinguer. Ici, notre algorithme est suffisamment basique pour pouvoir distinguer des opérations que nous considèrerons à coût constant : une déclaration de variable (d unités de temps), une affectation (a unités de temps), l'accès à un élément d'une liste (x unités de temps), une somme d'entiers ou un incrément (s unités de temps), une comparaison entre deux entiers (c unités de temps).

Chacune de ces actions va compter pour un nombre constant d'unités de temps qu'on ne connait pas a priori, qui dépendra du système sur lequel on travaille, mais on verra que leur valeur réelle ne nous intéresse pas. Il ne reste plus qu'à déterminer leurs fréquences dans l'algorithme :

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

```
In [46]: xs = list(range(-2**1000, -2**1000 + 100)) + list(range(2**1000 - 100, 2**1000))
In [47]: %timeit trois_sommes(xs)
10 loops, best of 3: 147 ms per loop

In [48]: xs = list(range(-2**1000, -2**1000 + 200)) + list(range(2**1000 - 200, 2**1000))
In [49]: %timeit trois_sommes(xs)
1 loops, best of 3: 1.21 s per loop

In [50]: xs = list(range(-2**1000, -2**1000 + 400)) + list(range(2**1000 - 400, 2**1000))
In [51]: %timeit trois_sommes(xs)
1 loops, best of 3: 9.84 s per loop
```

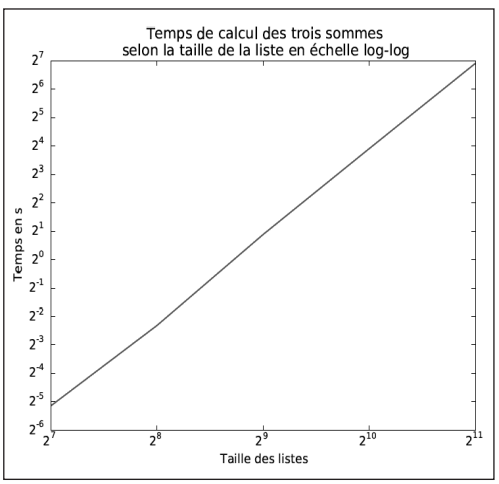
Encadré 3

Encadré 4

```
from time import perf_counter

def temps(xs):
    debut = perf_counter()        # on déclenche le chrono
    trois_sommes(xs)              # on lance le calcul
    return perf_counter() - debut # on arrête le chrono quand c'est fini
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 echelle log-log' )
plt.xlabel( 'Taille des listes' )
plt.ylabel( 'Temps en s' )
```



Encadré 5

Que valent S1 et S2 ? Pour S1 il s'agit du nombre d'itérations de la boucle incrémentée par j et pour S2 ce sera k. Cela implique un petit travail classique sur les sommes, voire les doubles sommes, d'entiers.

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

(avec $i' = N-(i+1)$)

$$S_2 = \sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} [N-(j+1)]$$

$$= \sum_{i=0}^{N-1} \sum_{j'=0}^{N-(i+2)} j' \qquad (\text{avec } j' = N-(j+1))$$

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

$$S_2 = \sum_{i'=1}^{N-2} \frac{i'(i'+1)}{2} \qquad \text{(avec } i' = N-(i+2)\text{)}$$

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

$$= \frac{1}{2}\left(\frac{(N-2)(2N-3)(N-1)}{6} + \frac{(N-2)(N-1)}{2} \right)$$

$$= \frac{N(N-1)(N-2)}{6}$$

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 :

$$\begin{aligned}\tau(N) \leq\ & (2d + 3d + 3a + a) \\ & + (d + a + x + d + s + a)N \\ & + (d + a + x + s + d + s + a)S_1 \\ & + (x + s + c + s)S_2 \ .\end{aligned}$$

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 !

ANNEXE

Plus fort que la dichotomie…
l'algorithme de Héron et sa complexité

La dichotomie est peu efficace[7] pour trouver la solution d'une équation numérique par rapport à l'algorithme de Héron. Pour démontrer cette supériorité du point de vue de la complexité, il nous faut introduire aux élèves quelques outils mathématiques.

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 $\frac{1}{2}(x_n + \frac{a}{x_n})$ et constitue une meilleure approximation que les deux précédentes. On peut montrer que c'est une approximation par excès (en développant $(x_n - \sqrt{a})^2$ 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
```

```
Fonction heron_rec( a,fo : Réels  n : Entier ) : Réel
Si n = 0 Alors
  Retourner 0
Sinon
  Retourner heron_rec(a,(fo + a / fo) / 2, n
- 1)
FinSi
Fonction heron_it( a,fo : Réels  n : Entier ) : Réel
app ← 0
Pour k de 0 à n - 1 Faire
  app ← (app + a / app) / 2
FinPour
Retourner 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
```

7 Pour visualiser un algorithme dichotomique peu effcace, on peut regarder le tri rapide dansé : https://www.youtube.com/watch?v=ywWBy6J5gz8...

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 abordable au lycée.

Théorème 2 (Un théorème « light » du point fixe) Soit I un intervalle fermé de **R**, soit f une fonction continue 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 $r_{n+1} = f(r_n) = \dfrac{r_n + \dfrac{2}{r_n}}{2}$ et $r_0 = 1$. On peut alors démontrer par récurrence que :

1. pour tout entier naturel non nul n, on a $r_n \geq \sqrt{2}$;
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 $r_n \geq 1$, donc :

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

Le nombre de décimales d'un nombre x positif plus petit que 1 est $- \log_{10} x$.

En posant $d_n = - \log_{10} \left| r_n - \sqrt{2} \right|$, on obtient donc que $d_{n+1} \geq 2d_n$: 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*, ce qui est bien plus efficace que la dichotomie.

Cette méthode a d'ailleurs permis à Henry Briggs de construire ses tables de logarithmes en 1617. Voir l'excellente reconstruction de ces tables par Denis Roegel (Roegel [2010]).

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.

G. Aldon, J. Germoni et J.-M. Mény, *« Complexité d'un algorithme : une question cruciale et abordable »*, Repères IREM, , no 86, p. 27 50, 2012.

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.citilab.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 arti cielle, InterEditions, ISBN 9782729605810, 1995.

D. Goldberg, *« What every computer scientist should know about oating point arithmetic »*, ACM Computing Surveys, vol. 23, no 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.

D. Roegel, *« A reconstruction of the tables of Briggs' Arithmetica logarithmica (1624) »*, Research report, adresse : https://hal.inria.fr/inria00543939, 2010.

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.

C. S. D. B. University, What is computer science ?, adresse : http://www.cs.bu.edu/AboutCS/, 2015.