

La complexité c'est simple comme la dichotomie (Lycée Maths/ISN)

Guillaume CONNAN*- IREM de Nantes

19 avril 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 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 TICÉ (avec un T comme *technologie*...cela restreint le domaine et permet de comprendre pourquoi pendant si longtemps informatique rimait avec bureaucratie) 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]) :

« Computer Science is about problem solving. Thus, the qualities of a good computer scientist include a passion for finding elegant solutions, an ability to use mathematical analysis and logical rigor to evaluate such solutions, creativity in modeling complex problems through the use of abstractions, attention to details and hidden assumptions, an ability to recognize variants of the same problem in different settings, and being able to re-target known efficient solutions to problems in new settings. If you like to solve puzzles, then computer science is for you! »

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

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.

*Guillaume.Connand@univ-nantes.fr

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



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)

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
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'ambiguïté, c'est-à-dire lorsque l'étage supérieur sera directement 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, 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

```

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.

```

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 de 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 $\{0, 1, \dots, N-1\}$ 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 $\ell_i = \frac{N}{2^i} = \frac{2^n}{2^i} = 2^{n-i}$. 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.

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 ^a.

...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 ^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 les jambes des expérimentateurs.

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` :

```
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 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. Étudier la manière de représenter les relatifs sur machine est un problème intéressant : un exemple de recherche sur ce thème est donné en 4.4 page 17.

Le bug a été corrigé ^d en remplaçant cette ligne par :

```
int mid = low + ((high - low) / 2);
```

Il faudra faire de même dans nos fonctions!

- 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

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 de œufs à disposition). Il se peut que nous n'ayons pas abîmé de 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 de 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 $k - 1 + \frac{N}{2^{k-1}} - 1$ 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...

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'effectuer 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 $T(N) = n = \lfloor \log_2(N) \rfloor$.

4. Et si nous avons 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 3) $2 \log_3(N) = 2 \frac{\ln(N)}{\ln(3)} \times \frac{\ln(2)}{\ln(3)} \approx 1,3 \log_2(N)$ 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
```

```
Si n==1 Alors
```

```
  Retourner x.y
```

```
Sinon
```

```
  m ← ⌊n/2⌋
```

```
  x1 ← ⌊x/10m⌋
```

```
  x2 ← x mod 10m
```

```
  y1 ← ⌊y/10m⌋
```

```
  y2 ← y mod 10m
```

```
  a ← MUL(x1, y1, m)
```

```
  b ← MUL(x2, y1, m)
```

```
  c ← MUL(x1, y2, m)
```

```
  d ← MUL(x2, y2, m)
```

```
  Retourner 102ma + 10m(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 :

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

Peut-on exprimer $T(n)$ explicitement en fonction de n sans récursion ?

Si nous avons 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 :

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

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 $x \mapsto x * x \leq 2$ et l'on va chercher une cellule de ce tableau par dichotomie comme on cherchait un étage dans un immeuble.

```
1 def racineDicho(prec):
2   cpt = 0
3   inf = 1
4   sup = 2
5   while (sup - inf > prec):
```

```

6     m = inf + (sup - inf) / 2
7     cpt += 1
8     if m*m <= 2:
9         inf = m
10        else:
11            sup = m
12    return sup, cpt

```

Nous obtenons :

```

1 In [1]: racineDicho(2**(-10))
2 Out[1]: (1.4150390625, 10)
3
4 In [2]: racineDicho(2**(-15))
5 Out[2]: (1.414215087890625, 15)
6
7 In [3]: racineDicho(2**(-20))
8 Out[3]: (1.4142141342163086, 20)
9
10 In [4]: racineDicho(2**(-30))
11 Out[4]: (1.4142135623842478, 30)
12
13 In [5]: racineDicho(2**(-50))
14 Out[5]: (1.4142135623730958, 50)

```

Attention! Il faudra s'arrêter à la précision 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 Le problème

On dispose d'une liste de N nombres. Déterminez les 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.

4.2 Méthode expérimentale

Utilisons la force brute (pour le lecteur impatient : nous ferons mieux plus tard...) :

```

1 def trois_sommes(xs):
2     N = len(xs)
3     cpt = 0
4     for i in range(N):
5         for j in range(i + 1, N):
6             for k in range(j + 1, N):
7                 if xs[i] + xs[j] + xs[k] == 0:
8                     cpt += 1
9     return cpt

```

Comparons les temps pour différentes tailles :

```

1 In [1]: %timeit trois_sommes([randint(-10000,10000) for k in range(100)])
2 10 loops, best of 3: 27.5 ms per loop
3
4 In [2]: %timeit trois_sommes([randint(-10000,10000) for k in range(200)])
5 1 loops, best of 3: 216 ms per loop
6
7 In [3]: %timeit trois_sommes([randint(-10000,10000) for k in range(400)])
8 1 loops, best of 3: 1.82 s per loop

```

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

```

1 def trois_sommes_comp(xs):
2     n = len(xs)
3     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] ==
4                 → 0 ])

```

Mais les temps sont les mêmes :

```

1 In [24]: %timeit trois_sommes_comp([randint(-10000,10000) for k in range(100)])
2 10 loops, best of 3: 28.1 ms per loop
3
4 In [25]: %timeit trois_sommes_comp([randint(-10000,10000) for k in range(200)])
5 1 loops, best of 3: 222 ms per loop
6
7 In [26]: %timeit trois_sommes_comp([randint(-10000,10000) for k in range(400)])
8 1 loops, best of 3: 1.85 s per loop

```

On peut plutôt s'intéresser aux ratios :

```

1 from time import perf_counter
2 from math import log
3
4 def temps(xs):
5     debut = perf_counter()
6     trois_sommes(xs)
7     return perf_counter() - debut
8
9 t = [temps(range(100 * 2**k)) for k in range(4)]
10
11 ratio = [t[k + 1] / t[k] for k in range(3)]
12
13 def logD(x):
14     return log(x)/log(2)
15
16 logratio = map(logD,ratio)

```

et on obtient :

```

1 In [4]: ratio
2 Out[4]: [7.523860206447286, 9.118789882406599, 8.5098312160934]
3
4 In [5]: list(logratio)
5 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 ?

```
1 In [6]: temps(range(400))
2 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 $6,25 \times 10^{-8} \times 10^9 = 62,5$

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

Presque...

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

```
1 #include <stdio.h>
2 #include <time.h>
3
4 typedef int Liste[13000];
5
6 int trois_sommes(int N)
7 {
8     int cpt = 0;
9     Liste liste;
10
11     for ( int k = 0; k < N; k++)
12     {
13         liste[k] = k - (N / 2);
14     }
15
16     for (int i = 0; i < N; i++)
17     {
18         for (int j = i + 1; j < N; j++)
19         {
20             for (int k = j + 1; k < N; k++)
21             {
22                 if (liste[i] + liste[j] + liste[k] == 0) {cpt++;}
23             }
24         }
25     }
26     return cpt;
27 }
28
29 void chrono(int N)
30 {
31     clock_t temps_initial, temps_final;
32     float temps_cpu;
33
34     temps_initial = clock();
35     trois_sommes(N);
36     temps_final = clock();
37     temps_cpu = ((double) temps_final - temps_initial) / CLOCKS_PER_SEC;
38     printf("Temps en sec pour %d : %f\n",N, temps_cpu);
39 }
40
41
42 int main(void)
43 {
44     chrono(100);
45     chrono(200);
46     chrono(400);
47     chrono(800);
48     chrono(1600);
49     chrono(3200);
```

```

50     chrono(6400);
51     chrono(12800);
52
53     return 1;
54 }

```

Et on obtient :

```

1 $ gcc -std=c99 -Wall -Wextra -Werror -pedantic -O4 -o somm3 Trois_Sommes.c
2 $ ./somm3
3 Temps en sec pour 100 : 0.000000
4 Temps en sec pour 200 : 0.000000
5 Temps en sec pour 400 : 0.020000
6 Temps en sec pour 800 : 0.090000
7 Temps en sec pour 1600 : 0.720000
8 Temps en sec pour 3200 : 5.760000
9 Temps en sec pour 6400 : 45.619999
10 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 :

$$45,61 = a \times 6400^3 \text{ d'où } a \approx 1,74 \times 10^{-10}$$

$$360,84 = a \times 12800^3 \text{ d'où } a \approx 1.72 \times 10^{-10}$$

En Haskell :

```

1 import Criterion.Main
2
3 -- La fonction
4 troisSommes :: [Int] -> [(Int,Int,Int)]
5 troisSommes xs =
6     let inds = [0 .. (length xs -1)]
7         in [(xs!!i , xs!!j , xs!!k) | i <- inds, j <- drop i inds, k <- drop j inds, xs!!i + xs!!j + xs!!k == 0]
8
9 -- Le "benchmarking" avec le module Criterion
10 main :: IO ()
11 main = defaultMain [
12     bgroup "TroisSommes" [ bench "3som 100" $ whnf troisSommes [-50..49]
13                          , bench "3som 200" $ whnf troisSommes [-100..99]
14                          , bench "3som 400" $ whnf troisSommes [-200..199]
15                          , bench "3som 800" $ whnf troisSommes [-400..399]
16                          , bench "3som 1600" $ whnf troisSommes [-800..799]
17                          , bench "3som 3200" $ whnf troisSommes [-1600..1599]
18     ]
19 ]

```

et on observe :

```

1 $ ./TroisSom --output troisSom.html
2 benchmarking TroisSommes/3som 100
3 time                640.9 µs   (636.3 µs .. 645.0 µs)
4                    1.000 R²   (0.999 R² .. 1.000 R²)
5 mean                640.4 µs   (637.6 µs .. 644.4 µs)
6 std dev             11.88 µs   (9.047 µs .. 15.88 µs)
7
8 benchmarking TroisSommes/3som 200
9 time                4.322 ms   (4.305 ms .. 4.344 ms)
10                   1.000 R²   (0.999 R² .. 1.000 R²)
11 mean                4.358 ms   (4.342 ms .. 4.384 ms)
12 std dev             65.63 µs   (42.33 µs .. 119.0 µs)
13
14 benchmarking TroisSommes/3som 400

```

```

15 time                31.72 ms   (31.43 ms .. 32.08 ms)
16                   1.000 R2   (0.999 R2 .. 1.000 R2)
17 mean                31.58 ms   (31.50 ms .. 31.75 ms)
18 std dev             239.8 μs   (83.75 μs .. 368.0 μs)
19
20 benchmarking TroisSommes/3som 800
21 time                261.7 ms   (259.7 ms .. 263.6 ms)
22                   1.000 R2   (1.000 R2 .. 1.000 R2)
23 mean                262.8 ms   (262.0 ms .. 263.2 ms)
24 std dev             758.8 μs   (95.48 μs .. 967.9 μs)
25 variance introduced by outliers: 16% (moderately inflated)
26
27 benchmarking TroisSommes/3som 1600
28 time                2.318 s    (2.289 s .. 2.339 s)
29                   1.000 R2   (NaN R2 .. 1.000 R2)
30 mean                2.320 s    (2.314 s .. 2.323 s)
31 std dev             4.803 ms   (0.0 s .. 4.834 ms)
32 variance introduced by outliers: 19% (moderately inflated)
33
34 benchmarking TroisSommes/3som 3200
35 time                16.47 s    (15.00 s .. 17.51 s)
36                   0.999 R2   (0.998 R2 .. 1.000 R2)
37 mean                17.33 s    (16.82 s .. 17.64 s)
38 std dev             471.1 ms   (0.0 s .. 535.4 ms)
39 variance introduced by outliers: 19% (moderately inflated)

```

Encore un rapport de 8 pour chaque doublement de taille mais Haskell est 60 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 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.

4.3 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 Dictionary »

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, mais bon, ce sont tous deux des mathématiciens allemands.

Rappelons 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 $|f(n)| \leq C|g(n)|$ pour tout $n \in \mathbb{N}$.

Dans l'exemple précédent, $\frac{1}{n} \leq \frac{1}{1} \times 1$ pour tout entier n supérieur à 1 donc $\frac{1}{n} = O(1)$.

De même, $\frac{75}{n^2} \leq \frac{75}{1^2} \times 1$ 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 $\frac{75}{n^2} = O\left(\frac{1}{n}\right)$.

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 $|f(n)| \geq C|g(n)|$ 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 Theta ») $f = \Theta(g) \iff f = O(g) \wedge f = \Omega(g)$

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...

4.4 Analyse mathématique



Une bonne lecture de chevet est le troisième volume de « The Art » ([Knuth \[1998\]](#)).
 Le temps d'exécution est la somme des produits (coût \times 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.

D. E. Knuth
Né en 1938

```

1 In [12]: xs = range(-500,501)
2
3
4 In [13]: %timeit xs[900]
5 10000000 loops, best of 3: 129 ns per loop
6 In [14]: %timeit xs[900] + xs[800]
7 1000000 loops, best of 3: 292 ns per loop
8 In [15]: %timeit xs[900] + xs[800] + xs[700]
9 1000000 loops, best of 3: 424 ns per loop

```

Mais si on travaille sur une liste :

```

1 In [22]: xs = list(range(-500,501))
2
3 In [23]: %timeit xs[900]
4 10000000 loops, best of 3: 40.7 ns per loop
5 In [24]: %timeit xs[900] + xs[800]
6 10000000 loops, best of 3: 114 ns per loop
7 In [25]: %timeit xs[900] + xs[800] + xs[700]
8 10000000 loops, best of 3: 165 ns per loop

```

oui mais :

```
1 In [26]: %timeit range(-500,501)
2 1000000 loops, best of 3: 292 ns per loop
3 In [27]: %timeit list(range(-500,501))
4 100000 loops, best of 3: 13.9  $\mu$ s per loop
```

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

```
1 def trois_sommes(xs):
2     N = len(xs)
3     cpt = 0
4     for i in range(N):
5         xi = xs[i]
6         for j in range(i + 1, N):
7             sij = xi + xs[j]
8             for k in range(j + 1, N):
9                 if sij + xs[k] == 0:
10                    cpt += 1
11     return cpt
```

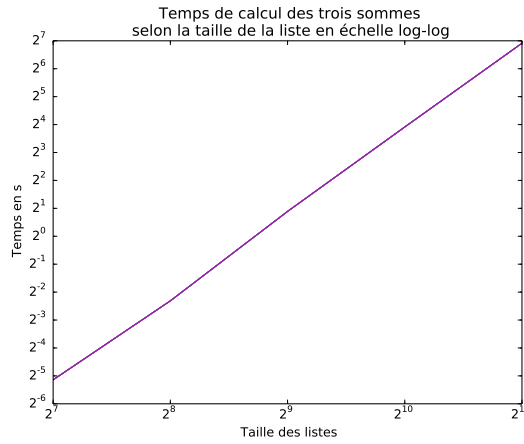
C'est mieux :

```
1 In [28]: xs = list(range(-50,51))
2 In [29]: %timeit trois_sommes(xs)
3 100 loops, best of 3: 12.9 ms per loop
4
5 In [30]: xs = list(range(-100,101))
6 In [31]: %timeit trois_sommes(xs)
7 10 loops, best of 3: 94.4 ms per loop
8
9 In [32]: xs = list(range(-200,201))
10 In [33]: %timeit trois_sommes(xs)
11 1 loops, best of 3: 851 ms per loop
12
13 In [34]: xs = list(range(-400,401))
14 In [35]: %timeit trois_sommes(xs)
15 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 :

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



C'est assez rectiligne.

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 considérerons 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 connaît 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

Que valent S_1 et S_2 ? Pour S_1 il s'agit du nombre d'itérations de la boucle incrémentée par j et pour S_2 ce sera k . Cela implique un petit travail classique sur les sommes voire les doubles sommes d'entiers.

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

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

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 :

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

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 !

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

Annexe 1 : Plus fort que la dichotomie...l'algorithme de Héron et sa complexité

La dichotomie est peu efficace^e 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 $\frac{1}{2} \left(x_n + \frac{a}{x_n} \right)$ et constitue une meilleure approximation que les deux précédentes.

On peut montrer c'est une approximation par excès (en développant $(x_n - \sqrt{a})^2$ 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

```

^e. Pour visualiser un algorithme dichotomique très efficace, on peut regarder le tri rapide dansé : <https://www.youtube.com/watch?v=ywWBy6J5gz8...>

```

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

```

```

4 In [13]: heron_it(2,1,6)
5 Out[13]: 1.414213562373095

```

```

7 In [14]: from math import sqrt

```

```

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 abordable au lycée.

Théorème 2 (Un théorème light du point fixe) *Soit I un intervalle fermé de \mathbb{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 $r_{n+1} = f(r_n) = \frac{r_n + \frac{2}{r_n}}{2}$ et $r_0 = 1$. On peut alors démontrer par récurrence que

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

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

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

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

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 $d_{n+1} \geq 2d_n$: 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.

Annexe 2 : un exemple de TP sur les relatifs et le complément à 2 puissance n

Le monde se sépare en 10 catégories : ceux qui comprennent cette phrase et les autres.

Anonyme

Nous allons, pour favoriser la compréhension et simplifier les écritures, travailler sur 8 bits. Cela signifie que nos nombres entiers peuvent être codés avec 8 chiffres égaux à 0 ou 1 en base 2.

1. Écrivez les nombres de 0 à 17 en base 2. Quel est le plus grand nombre que l'on peut écrire avec 8 bits ?
2. En fait, on aimerait avoir des entiers signés, c'est-à-dire pouvoir représenter des entiers tant positifs que négatifs avec cette contrainte.

On va donc réserver un bit (le plus à gauche) pour indiquer le signe du nombre : on mettra un 0 s'il est positif et un 1 s'il est négatif.

Écrivez en compréhension l'ensemble des nombres que l'on peut ainsi représenter et qui se note sur Haskell Int8.

3. Bon, pas de problème...enfin si quand même un peu. Que pensez-vous de ces deux nombres :

00000000 10000000

4. On voudrait additionner facilement ces nombres. Une idée serait d'utiliser l'addition posée comme à l'école primaire. Par exemple pour calculer $5 + 9$

```

00000101
+ 00001001
-----
00001110

```

On trouve bien 14.

Et maintenant si on veut calculer $5 + (-9)$, qu'est-ce que cela donne avec ce système ?

5. Considérez maintenant le nombre $m = 11111111$ et ajoutez-lui 1 (vous n'avez que 8 bits à disposition). Que vaut $m + 1$? Que cela vous donne-t-il comme idée ?

Considérez $n=11111110$ et calculez $n + 2$.

Considérez $p=11111100$ et calculez $n + 4$.

Considérez $p=11111000$ et calculez $n + 8$.

Considérez $p=11111010$ et calculez $n + 6$.

Des idées ?

6. Si l'on considère que 2^8 vaut 0 en Int8, ce qui est d'ailleurs le cas :

```

1 > let n = 256 :: Int8
2 > n
3 0

```

Que pensez-vous alors de x et de $2^8 - |x|$ si x est négatif ?

Comment s'écrit $2^8 - |x|$ sur 8 bits ?

Bon, vous commencez à voir ce qui se passe ?

Alors expliquez ce qui se passe ici (illustration en Haskell pour faciliter l'utilisation d'entiers 8 bits) :

```

1 > let n = 13 :: Int8
2 > n * 2
3 26
4 > n * 4
5 52
6 > n * 8

```

```

7  104
8  > n * 16
9  -48
10 > n * 32
11 -96
12 > n * 64
13 64

```

Généralisez, énoncez votre norme de représentation des entiers sur 8 bits, énoncez un « truc » pour trouver facilement de tête le complément à 2^8 .

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, n° 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.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.
- C. S. D. B. UNIVERSITY,
What is computer science?, adresse : <http://www.cs.bu.edu/AboutCS/>, 2015.