PREUVES ET ANALYSES DES ALGORITHMES

Résumé. Les étudiants de la deuxième année de licence d'informatique de la faculté des sciences et techniques de l'université de Toulon sont sollicités pour la rédaction de ce document au format LaTeX.

Contributions : Jean-Louis Bosio, Philippe Langevin.

Table des matières

Date: 7 décembre 2010.

FIG. 1. Les origines du mot algorithme.

1. ALGORITHMIQUE

PH. LANGEVIN 21 Octobre 2009

Les organigrammes pour décrire des processus de calculs étaient très à la mode dans les années 70–80. Sans vraiment comprendre pourquoi, je constate qu'ils ont tendance à disparaître des enseignements d'informatique laissant le champ libre aux langages et algorithmes. Algorithme ? une terminologie qui est restée longtemps mystérieuse. Pendant longtemps, on croyait devoir lire dans le mot algorithme le radical grec *arithmos* qui signifie *nombre*. La poursuite étymologique s'enlise à moins de croire en une certaine *douleur des nombres*. En effet, seul le mot grec *algos* (douleur) admet le radical adéquat ! Possible pour les étudiants de premier cycle qui patogent parfois avec les logarithmes [1] mais pas pour les arithméticiens grecs ! Le mathématicien italien du XI-ième siècle, Léonard de Pise, rapporte de ses voyages méditerranéens le *sifr* de Perse. Lecteur des textes mathématiques arabes, Léonard de Pise est à l'origine du mot algorithme. Un terme qu'il aurait utilisé par pour décrire les procédés de calculs qu'il peut lire dans le traité *Kitab al'jabr w'al-muqabala* (règles de restaurations et réductions) écrit par un certain Al-Kwarezmi mathématicien arabe du IX-ième siècle dont le patronyme exact est : Abu Ja'far Mohammed ibn Mûsâ al-Kwarizmi, père de Jafar, Mohammed, fils de Moïse et natif d'al-Kwarezmi. Le Khôresme, ex-Khanat de Khiva, situé sur la partie inférieure du fleuve Amou-Syria est devenue une petite ville de l'Ouzbékistan bien au sud la mer d'Aral.

L'objectif du cours est de fournir une initiation à l'algorithmique : temps de calcul, correction et implantation en `langage C`. Cette micro-introduction est tirée de [4], lien vers le document, voir aussi le volume 1 de l'encyclopédie [1] "the art of computer programming" de D. Knuth.

[1] Quel superbe exemple d'anagramme !

2. Notation asymptotique

Ph. Langevin 3 Novemebre 2009

L'analyse des temps de calcul d'un programme fait apparaître des fonctions définies sur l'ensemble des entiers naturels. En général, ces fonctions ont un comportement assez cahotique. On introduit des *notations asymptotiques* pour décrire grossièrement ces fonctions.

```c
#include <stdio.h>
#include <stdlib.h>
typedef unsigned char uchar;
void crible( uchar* t, int p, int n)
{ int x;
  x = p * p;
  while ( x < n ){
    t[x] = 0;
    x += p;
  }
}
int main( int argc, char*argv[] )
{ int n, s, x;
  uchar* t;
  int cpt=0;
  n = atoi( argv[1] );
  s = atoi( argv[2] );
  t= (uchar*)
      calloc( n , sizeof(uchar));
  for( x = 2; x < n; x++ )
    t[x] = 1;
  for( x = 2; x < n; x++ )
    if ( t[x] )
      crible(t, x, n);
  for( x = 0; x < n; x++ ){
   if ( t[x] ) cpt++;
   if ( 0 == x % s )
     printf("\npi(%d) = %d", x, cpt);
  }
  return 0;
}
```

Le programme `crible.c` est une implantation du célèbre crible d'Ératostène. La représentation graphique Fig. (2) a été réalisée par le `makefile` ci-dessous.

. Fichier makefile

```
max=250
pas=5
all : crible.exe pi.png
crible.exe : crible.c
 gcc -Wall crible.c -o crible.exe
data.txt :
 ./crible.exe $(max) $(pas)  \
  | sed 's/[pif(=)]/ /g' > data.txt
pi.png : plot.cmd data.txt
 gnuplot plot.cmd
```

Avec le fichier de commandes `gnuplot` :

```
set term png
set output 'pi.png'
plot 'data.txt',x/log(x),1.5*x/log(x)
quit
```

On constate que le nombre de premiers inférieurs à x, traditionnellement noté $\pi(x)$ est compris entre deux multiples positifs de $x/\log(x)$. Nous noterons

$$\pi(x) = \Theta\Big(\frac{x}{\log(x)}\Big)$$

qui signifie que $\pi(x)$ est asymptotiquement du même ordre de grandeur que la fonction $\frac{x}{\log(x)}$. En réalité, ces deux fonctions sont équivalentes à l'infini

$$\pi(x) \sim \Theta\Big(\frac{x}{\log(x)}\Big)$$

un résultat conjecturé par Gauss au XVIII-ième siècle, démontré plus tard par Hadamard et De la Vallée-Poussin.

Définition 1. *Soient $f, g\colon \mathbb{N} \to \mathbb{R}$ deux fonctions positives. On dit que f et g sont asympotiquement du même ordre de grandeur s'il existe deux nombres positifs A et B, et un rang n_0 tels que*

$$\forall n \geq n_0, \quad Af(n) \leq g(n) \leq Bf(n).$$

De façon condensée, on note $g = \Theta(f)$.

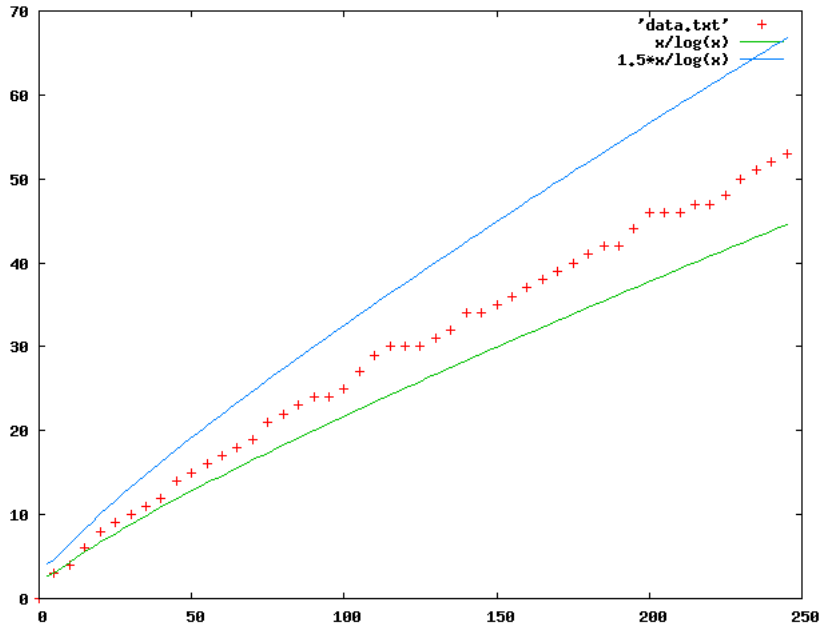

FIG. 2. Graphe de la fonction $\pi(x)$. On constate que $\pi(x) = \Theta\big(x/\log(x)\big)$.

Définition 2. *Soient $f, g\colon \mathbb{N} \to \mathbb{R}$ deux fonctions positives. On dit que f et g sont asympotiquement équivalentes*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1,$$

et on note $g \sim f$.

Il s'agit de deux relations d'équivalence, la seconde est plus fine que la première qui demeure suffisante pour la plupart des applications en algorithmique.

Définition 3. *Soient $f, g\colon \mathbb{N} \to \mathbb{R}$ deux fonctions positives. On dit que g domine f (asymptotiquemen) s'il existe un nombre positif B, et un rang n_0 tels que*

$$\forall n \geq n_0, \quad f(n) \leq Bg(n)$$

On note $f = \mathrm{O}(g)$, ou bien $g = \Omega(f)$.

Considérons l'algorithme de tri TriInsertion, le nombre de comparaisons entre des éément du tableau (ligne 8) à l'etape i est égal à $n-i-1$. Le nombre total de comparaisons est donné par la somme de n termes en progression arithmétique de raison 1 :

$$C(n) = n\frac{n-1}{2} \sim \frac{n^2}{2} = \Theta(n^2)$$

```
TriInsertion( t : table )                          1
variable i, j:indice                               2
debut                                              3
  i←0                                              4
  tantque ( i < n )                                5
    j←i+1                                          6
    tantque ( j < n )                              7
     si ( t[i] < t[j] )                            8
       alors                                       9
          t[i] ↔ t[j]                              10
     fsi                                           11
     j ← j+1                                       12
    ftq                                            13
   i ← i+1                                         14
  ftq                                              15
fin                                                16
```

3. Suite de Fibonacci

Ph. langevin 21 Octobre 2009 La suite de Fibonacci F_n est la suite d'entiers positifs définie par la relation de récurrence d'ordre deux :

$$F_0 = 0, \quad F_1 = 1, \qquad \forall n \geq 2,\ F_n = F_{n-1} + F_{n-2}. \tag{1}$$

On démontre sans difficulté que F_n s'exprime à partir des deux racines l'équation $X^2 = X + 1$ qui sont $\phi \approx 1.618$ et $\hat{\phi} \approx -0.6$:

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) \tag{2}$$

En particulier, F_n croît exponentiellement, plus précisément, F_n est équivalent à $\frac{1}{\sqrt{5}}\phi^n$ au voisinage de l'infini. Par exemple $F_{10} = 55$,

$$F_{100} = 354224848179261915075$$

et F_{1000} est un nombre de 289 chiffres décimaux : 43 466 557 686 937 456 435 688 527 675 040 625 802 564 660 517 371 780 402 481 729 089 536 555 417 949 051 890 403 879 840 079 255 169 295 922 593 080 322 634 775 209 689 623 239 873 322 471 161 642 996 440 906 533 187 938 298 969 649 928 516 003 704 476 137 795 166 849 228 875

```
1  FibRec( n : indice )
2  debut
3     si ( n ≤ 1 ) alors
4      retourner n
5     fsi
6     retourner
7        Fibrec(n-1) + Fibrec(n-2)
8  fin
```

La méthode récursive consiste à appliquer définition récursive. Il s'agit d'une approche naïve (aucun effort !). L'implantation requiert un langage récursif, c'est le cas de la majorité des langages. Si nous notons $R(n)$ le nombre d'étapes pour calculer F_n alors $T(0) = 1$, $T(1) = 1$, et pour tout $n \geq 2$:

$$T(n) = T(n-1) + T(n-2).$$

Le temps de calcul d'une implantation sera proportionnel à F_{n+1}, il est *exponentiel* en n.

```
1  FibIter( n : indice )
2  variable x, y, t: nombre
3  debut
4     x ← 0
5     y ← 1
6     tantque ( n > 0 )
7           t ← x + y
8           x ← y
9           y ← t
10        n ← n-1
11    ftq
12    retourner x
13 fin
```

L'approche itérative demande plus de travail de la part du concepteur. Le nombre d'itérations effectuées pour calculer F_n est proportionnel à n. Le temps de cacul d'une implantation sera de la forme $An + B$, on dit qu'il est *linéaire*.

n	10	20	40	80
$R(n)$	0.00001	0.0018	57	2000 an !
$I(n)$	0.00001	0.00002	0.00004	0.00008

Le tableau ci-dessus montre qu'il ne sera pas possible de calculer F_{80} par la méthode récursive au cours d'une séance de travaux-pratiques...

4. Algorithme additif

Jean-Louis Bosio 23 Novembre 2009

On représente les *grands nombres* par des tableaux de chiffres dans une base B fixée. Un tableau T de taille N représente un nombre

$$a_N B^{N-1} + \ldots + a_1 B^0 < B^N$$

Les chiffes poids faibles sont placés dans les cellules de poids faibles. Avec ces notation, nous avons $T[0] = a_1$.

4.1. **Incrémentation.** Il s'agit d'augmenter de 1 un nombre en gérant les débordements, c'est à dire l'eventualité que tous les chiffres du nombres soient égaux à $BASE - 1$

```
1
2  Algorithme INC(t : nombre)
3  Variable : i : indice
4  debut
5    i ← 0
6    tantque (t[i] = B-1) et (i < N)
7      t[i] ← 0
8      i ← i+1
9    ftq
10   si (N > i) alors
11     t[i] ← t[i]+1
12   fsi
13   retourner N = i
14 fin
```

Un tel algorithme a un temps de calcul variant en fonction des paramètres fournis. On peut donc identifier plusieurs instances d'execution :

(1) **Instance favorable :** Si $a_1 \neq B - 1$ il s'agit d'un cas favorable du point de vue du temps de calcul. Le corps de la boucle n'est pas exécuté et donc le temps de calcul est constant. Le nombre de ces instances est égal à $B^n - 1(B - 1)$.

(2) **Instance défavorable :**
Si $a_n = a_{n-1} = \ldots = a_1 = B - 1$ on a un cas défavorable car on va entrer N fois dans la boucle et on aura un dépassement de mémoire.
Cette instance est unique, ce qui nous amème à utiliser une instance moyenne.

Pour mieux cerner le temps de calcul, on introduit les notions de temps de calcul dans le pire des cas T^+_{inc}, dans le meilleur des cas T^-_{inc}, et la notion de temps de calcul moyen T^μ_{inc}.

$$T^-_{\text{inc}}(N) = \Theta(1), \quad T^+_{\text{inc}}(N) = \Theta(N).$$

Les instance qui provoquent k itérations sont de la forme $a_N \ldots a_{k+1}(B-1)(B-1)(B-1)$ avec $a_{k+1} \neq B - 1$. Il y en a $B^{n-k-1}(B - 1)$, on obtient une formule pour le nombre d'itérations moyen

$$\frac{1}{B^N} \sum_{k=0}^{N} k(B-1)B^{N-k-1} = \frac{B-1}{B^{N+1}} \sum_{k=0}^{N-1} (N-k)B^k \tag{3}$$

Proposition 1. *Le nombre d'itérations moyen est équivalent à* $\frac{1}{B-1}$*, et donc* $T^\mu_{\text{inc}}(N) = \Theta(1)$.

Démonstration. On peut éviter l'étude mathématique de la formule (3) par un bilan comptable. Il faut faire tourner l'algorithme pour toutes les instances, et comprendre le nombre d'itérations total. Imaginons que l'algorithme consomme un jeton pour exécuter les lignes (7) et (8), plus précisément des jetons de rang 1, 2, etc... Il y a B^{N-1} jetons de rang 1, plus généralement, B^{N-k} jetons de rang k et un total de

$$\sum_{k=1}^{N} B^{N-k} = \sum_{k=0}^{N-1} B^k = \frac{B^N - 1}{B - 1}$$

Le nombre de jetons moyen est équvalent à $\frac{1}{B-1}$. □

Exercice 1. *Utiliser le polynôme*

$$P(X) = \sum_{k=0}^{n-1} X^k$$

pour retrouver le résultat précédent. Pour cela, on donnera deux expressions de $XP'(X)$.

Exercice 2. *Ecrire un algorithme pour décrémenter un nombre.*

4.2. **Addition.** Pour additionner deux nombres, nous utilisons l'algorithme connu de tous. Cet algorithme permet d'additioner deux nombres Y et X et de mettre le résultat dans Y. Il gère aussi le débordement (retour 1, si dépassement) en interrogeant la valeur finale de la retenue utilisée pour faire les calculs.

```
1  Algorithme ADD(Y, X : nombre)
2  Variables  t : chiffre
3             i : indice
4  ret : booleen
5  debut
6    ret ← faux
7    i ← 0
8    tantque (i < N)
9      t ← X[i] + Y[i] + ret
10     Y[i] ← t mod B
11     ret ← t div B
12     i ← 1+1
13   ftq
14   retourner ret
15 fin
```

Implantation en language C :

```
#define BASE     13
#define MAX     128
typedef int chiffre;
typedef chiffre nombre[ MAX ];
int add(nombre y, nombre x)
{
    int ret = 0, i = 0, t;
    while (i < MAX) {
        t = x[i] + y[i] + ret;
        y[i] = t % BASE;
        ret = t / BASE;
        i++;
    }
    return ret;
}
```

Pour une taille fixée, le temps de calcul de l'algorithme ADD ne dépend pas des instances : il n'y a pas de cas plus favorable que d'autres.

Proposition 2. *Le temps de calcul de l'algorithme usuel d'addition est linéaire en la taille des nombres.*

Démonstration. Si nous notons α le temps d'exécution des lignes (1)-(7), β celui des lignes (9)-(12) et γ celui des lignes (14)-(17) alors

$$T_{\text{ADD}}(N) = \alpha + \beta N + \gamma = \Theta(N)$$

car les temps α, β, et γ sont tous $\Theta(1)$. □

Exercice 3. *Une implantation de* ADD *calcule un million d'additions sur des nombres aléatoires de taille* 1000 *en 1 minute. Estimer le temps de calcul du même nombre d'additions sur des nombres de taille* 10000.

4.3. **Comparaison.** L'algorithme CMP compare deux nombres, comme dans le cas d'une comparaison de chaines en langage C, il retourne -1, 0 ou 1 si respectivement $Y < X$, $Y = X$, $Y > X$.

```
1  Algorithme CMP(Y, X : nombre)
2  variable i : indice
3  debut
4    i ← n-1
5    tantque (i ≥ 0)
6      si (X[i] > Y[i]) alors
7        retourner -1
8      fin si
9      si (Y[i] > X[i]) alors
10       retourner 1
11     fin si
12     i ← i-1
13   ftq
14   retourner 0
15 fin
```

Le temps de calcul est favorable (proche 0) si les nombres sont différents dés le premier chiffre (celui de plus haut degré, de position $N-1$ dans le nombre). En revanche le cas le plus défavorable est celui où les nombres ne sont différents que de 1, c'est-à-dire si $Y - X = |1|$ et ainsi le temps de calcul est linéaire en N car N comparaisons auront été éffectuées.

On peut en déduire un temps de calcul moyen, qui sera constant (voir algorithme d'incrémentation).

4.4. **Soustraction.** L'algorithme SUB calcule la différence $Y - X$ et met le résultat dans Y, encore une fois, la valeur de la retenue permet de contrôler le bon déroulement de l'opération. Pour un chiffe c, on note $\bar{c}$ le chiffre $B - 1 - c$ i.e. le *complément à* $B - 1$ de c. Il s'appuie sur le résultat :

Proposition 3. *Soit $x = (x_n \dots x_2 x_1)$ un résidu modulo B^n. L'opposé de x modulo B^n est égal à $(\bar{x}_n \dots \bar{x}_2 \bar{x}_1) + 1$ i.e. 1 + le complément à $B - 1$ de x.*

Démonstration. En effet,

$$\begin{aligned}(x_n \dots x_2 x_1) + (\bar{x}_n \dots \bar{x}_2 \bar{x}_1) &= \sum_{i=1}^{n} (x_i + \bar{x}_i) B^{i-1} \\ &= (B-1)(B^0 + B^1 + \cdots + B^{n-1}) = B^n - 1 \equiv -1 \mod B^n\end{aligned}$$

□

```
1  Algorithme SUB(X, Y : nombre)
2  Variables i : indice, ret : booleen,
3  tmp : chiffre
4  debut
5    i ← 0
6    ret ← 1
7    tantque (i < N)
8      tmp ← Y[i] + B-1-X[i] + ret
9      Y[i] ← tmp mod B
10     ret ← tmp div B
11     i ← i+1
12   ftq
13   si (ret != 0) alors
14     retourner erreur
15   fin si
16 fin
```

Implantation en laguage C :

```
#define BASE    10
#define MAX     128
typedef int chiffre;
typedef chiffre nombre[ MAX ];
int sub(nombre y, nombre x)
{
    int ret = 1, i = 0, t;
    while (i < MAX) {
        t = y[i] + BASE - 1 - x[i] + ret;
        y[i] = t % BASE;
        ret = t / BASE;
        i++;
    }
    return ~ ret;
}
```

Le temps de calcul est de la forme $\alpha N + \beta$ car on a N passages dans la boucle (voir algorithme d'addition).

5. Algorithmes multiplicatifs

6. Division euclidienne

7. Fibonacci sur les nombres

Ph. Langevin 21 octobre 2009

Le calcul des termes de la suite de Fibonacci (F_n)

$$F_0 = 0, \quad F_1 = 1, \quad \forall n \geq 2 F_n = F_{n-1} + F_{n-2}$$

pour des rangs élevés ne pose pas de problèmes du point de vue du temps de calcul. Il faut prévoir des registres de tailles convenables, pour cela, on s'appuie sur l'estimation :

$$F_n = \frac{1}{\sqrt{5}}(\phi^n - \hat{\phi}^n) \sim \frac{\phi^n}{\sqrt{5}}$$

avec ϕ qui désigne le nombre d'or i.e. $\phi = \frac{1+\sqrt{5}}{2}$. Par exemple, pour calculer F_n en base B, il suffit d'utiliser des registres de taille $N := \log_B(\phi^n)$ soit $n \log \phi / \log B$.

```
1  Fibonacci( n : indice )
2  variable x, y : nombre
3  debut
4     init( x, 0 )
5     init( y, 1 )
6     tantque ( n > 0 )
7       exc( x, y )
8       add( y, x )
9       n ← n − 1
10    ftq
11    retourner y
12 fin
```

- init initialise un nombre.
- exc échange les valeurs de deux nombres.
- add calcule la somme de deux numbres, le résultat est passé dans le premier argument.

Les temps de calcul de init et add sont linéaires en N (donc linéaires en n). Le temps de calcul de exc dépend du mode d'implantation choisi pour les nombres, il est constant pour des pointeurs, linéaire pour des tableaux. Dans tous les cas le temps de calcul de Fibonacci est quadratique en n.

Proposition 4. *L'algorithme* Fibonacci*(n) calcule le n-ième terme de la suite de Fibonacci en utilisant* $\Theta(n^2)$ *étapes.*

8. Dichotomie

Ph. Langevin 7 Décembre 2010

8.1. Conversion.

La fonction `strtonb` initialise un `nombre` à partir d'une chaine de caractères représentant un nombre.

(1) En langage C, une chaine est pointeur vers des octets. Le contenu de la chaine se lit à partir de cette adresse jusqu'au premier caractère nul qui indique la fin de chaine.

(2) La première boucle recherche la fin de la chaine.

(3) La seconde boucle remplit le tableau des chiffres avec les caractères rencontrés dans la chaine.

(4) La conversion ASCII vers une valeur numérique est réalisée par un décalage de 38 correspondant au code ASCII du caractère ´0´.

(5) `man ascii` pour les détails du codage ascii.

```
nombre strtonb(char *s)
{
    nombre r;
    int i;
    char *ptr;
    r = init(0);
    ptr = s;
    while (*ptr != '\0')
        ptr++;
    i = 0;
    do {
        ptr--;
        r[i] = *ptr - '0';
        i++;
    } while ( s != ptr );
    return r;
}
```

8.2. Calcul d'une racine.

```
1  Racine( z : nombre )
2  variable a, b, m, y : nombre
3  debut
4     a ← 1;
5     b ← z;
6     tantque ( non voisin(a,b) )
7       m ← add( a, b);
8       m ← cdiv( m, 2);
9       y ← prd( m, m);
10      si cmp(y,z) < 0
11         alors
12           a ← m;
13         sinon
14           b ← m;
15      fsi
16    ftq
17    retourner a
18 fin
```

- voisin(a,b) vérifie si les nombres ia et b sont consécutifs : renvoie vrai si $b = a + 1$ et faux sinon.
- add addition de deux grands nombres.
- cdiv quotient de la division par un chiffre.
- prd produit de deux grands nombres.
- cmp comparaison de deux nombres. est passé dans le premier argument.

9. Tri par comparaison

10. Tri linéaire

11. Calcul du PGCD

12. Algorithme d'Euclide

```
1
2
3  #include<stdlib.h>
4  #include<stdio.h>
5  typedef unsigned long long ullong;
6  int iter(ullong a, ullong b)
7  {
8      ullong r;
9      int cpt = 0;
10     while (b > 0) {
11         r = a % b;
12         a = b;
13         b = r;
14         cpt = cpt + 1;
15     }
16     return cpt;
17 }
18
19 int main(int argc, char *argv[])
20 {
21     ullong a, b;
22     int n, max;
23     n = atoi(argv[1]);
24     for (b = 0; b < n; b++) {
25         max = 0;
26         for (a = 0; a < b; a++)
27             if (iter(a, b) > max)
28                 max = iter(a, b);
29         printf("%Ld  %d\n", b, max);
30     }
31     return 0;
32 }
```

. Fichier makefile

```
all : iter.png
iter.exe:iter.c
 gcc -Wall -g iter.c -o iter.exe
iter.png : data.txt
 gnuplot iter.cmd
data.txt : iter.exe
 ./iter.exe 1000 > data.txt
```

```
set term png
set output 'iter.png'
set title "Iterations de l'algorithme d'Euclide"
plot  'data.txt' w l, log(x)/log(1.618)
quit
```

Le théorème de Lamé affirme que le nombre d'itérations de l'algorithme d'Euclide est logarithmique. Un fait illustré par le graphique Fig. (3), obtenu par la présente expérience numérique correspondant au fichier makefile.

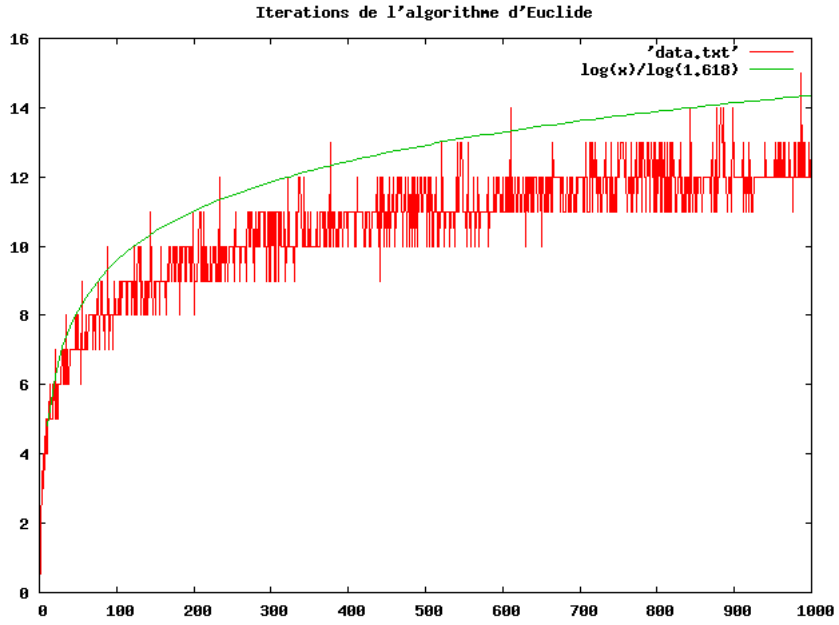

FIG. 3. Nombre d'itérations de l'algogorithme d'Euclide

13. EUCLIDE ÉTENDU

PH. LANGEVIN 21 Octobre 2009

Soient a et b deux entiers naturels. L'objectif de l'algorithme d'Euclide *étendu* est de fournir deux entiers relatifs u et v tels que

$$au + bv = \text{PGCD}(a, b)$$

L'existence de ce couple d'entiers est connu sous le théorème de Bâchet-Bézout.

```
1  EuclideEtendu(a,b:nombre)
2  variable u,v:nombre
3  debut
4      si ( b = 0) alors
5       retourner (1,0)
6      fsi
7      q ← a div b
8      (u,v) ←EuclideEtendu(b, a mod b)
9      t = u - q * v
10     v = u;
11     u = t
12     retourner
13         (u,v)
14 fin
```

```
Euclide(int a, int b, int*u, int*v)      1
{ int q, t;                              2
    if ( b==0 ){                         3
      *u = 1;                            4
      *v = 0;                            5
    } else {                             6
      q = a / b;                         7
      EEtendu(a,b,u,v);                  8
      t = *u - *v * q;                   9
      *u = *v;                           10
      *v = t                             11
    }                                    12
}                                        13
```

La méthode récursive est fondée sur le

TAB. 1. Pile de valeurs. Comment faire remonter les valeurs de u et v ?

a	b	q	u	v
127	13	9		
13	10	1		
10	3	3		
3	1	3		
1	0	?	1	0

Lemme 1. *Soient $0 < b \leq a$ deux entiers naturels, q et r les quotient et reste de la division euclidienne de a par b.*

$$\forall d, \quad d = xb + yr \Longrightarrow au + bv = d$$

avec $u = y$ et $v = x - qy$.

Démonstration. Il suffit de remplacer r par $a - bq$:

$$d = xb + yr = xb + y(a - bq) = ya + (x - qy)b.$$

□

Pour l'implantation en `langage C` sur des scalaires, la transmission de paramètres par adresse permet de récupérer les résultats. Notez bien l'utilisation des adresses u et v. Un exemple d'appel exterieur serait `Euclide(127,13,&x,&y)`. En traçant les variables dans une pile TAB. (1), on peut trouver les valeurs de u et de v.

L'approche itérative, dans sa version la plus naive, consiste à maintenir des relations linéaires sur les restes successifs de l'algorithme d'Euclide. Partant, d'une matrice 2x3 initialisée

$$\begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \end{pmatrix}$$

L'algorithme engendre des combinaisons linéaires de cette matrice en conservant les relations

$$\forall i \in 1, 2, \quad m_{i,1}a + m_{i,2} = m_{i,3}$$

Les valeurs de la troisième colonne de cette matrice ne sont rien d'autre que les restes successifs de l'algorithme d'Euclide, et donc, l'algorithme s'arrête avec

$$\text{PGCD}(a, b) = m_{1,3} = am_{1,2} + bm_{1,2}$$

```
EuclideEtendu(a,b:nombre)                                  1
variable                                                   2
          M: matrice 2x3 nombre                            3
       tmp : matrice 1x3 nombre                            4
         q : nombre                                        5
debut                                                      6
   M[1] ←(1,0,a);                                          7
   M[2] ←(0,1,b);                                          8
   tantque ( M[2,3] > 0 )                                  9
      q ← M[1,3] div M[2,3];                               10
      tmp ← M[1] - q * M[2];                               11
      M[1] ← M[2];                                         12
      M[2] ← tmp                                           13
   ftq                                                     14
   retourner                                               15
         (M[1,1], M[1,2])                                  16
fin                                                        17
```

Cet algorithme est efficace pour trouver les solutions de Bâchet *à la main*. Pour la mise en oeuvre, un code performant est obtenu par la suppression de toutes les variables inutiles.

14. Test de Rabin-Miller

Ph. langevin 9 Novembre 2009

Le test de primalité naif qui consiste à déclarer qu'un nombre z est premier s'il n'est pas divisible par tous les nombres : 2, 3, ... $\sqrt{z}$, est impraticable pour les grands nombres.

```
1  PseudoPremier( k : entier,z : nombre)
2  variable s : entier
3           n : nombre
4           r : booleen
5  debut
6     tant que ( k > 0 )
7      a ← aleas( z )
8      si test(a,z) = faux
9           alors
10            retourner faux
11     fsi
12     k ← k - 1
13    ftq
14    retourner possible
15 fin
```

Il est exponentiel en la taille de z. Typiquement, un tel algorithme réalise qu'un nombre premier de 128 bits est effectivement premier en 2^{64} étapes. Sur une machine standard, il faut s'attendre à un temps de calcul bien supérieur à

$$2^{32}\,\text{secondes} = 136\,\text{années!}$$

Une méthode probabiliste consiste à utiliser une fonction non déterministe test qui renvoie faux ou possible (?).

$$\text{Prob}(\text{test}(z) =? \wedge \neg\text{premier}(z)) \leq q$$

Aprés un nombre suffisant d'essais, on peut se convaincre du caractère premier d'un nombre.

Le test de Rabin et Miller s'appuie sur deux ingrédients algébriques fondamentaux :

Théorème 1 (Fermat)**.** *Un entier $p > 1$ est premier si et seulement si pour tout résidu non nul a, on a :*

$$a^{p-1} \equiv 1 \pmod{p}$$

Démonstration. On peut obtenir se résultat en montrant que, lorsque p est premier, les coefficients binomiaux satisfont

$$C_p^k = \begin{cases} 1, & k = p \text{ ou } k = 0\,; \\ 0, & \text{sinon.} \end{cases}$$

□

Théorème 2. *Un entier impair $n > 2$ est une puissance d'un nombre premier si et seulement si l'équation :*

$$x^2 \equiv 0 \pmod{p} \tag{4}$$

possède exactement 2 solutions, 1 et $p - 1$.

Démonstration. Si n n'est pas une puissance d'un premier, on peut écrire $n = ab$ avec a et b premier entre eux. Il existe u et v tel que $au + bv = 1$. Les éléments $aux + bvy$, avec $x = 1, b-1$ ou $y = 1, a-1$ sont solutions de (4). □

```
1  RabinMillerTest( a, z : nombre)
2  variable s : entier
3           n : nombre
4           r : booleen
5  debut
6     n ← z - 1
7     s ← 0
8     tant que pair( n )
9       n ← n/2
10      s ← s+1
11    ftq
12    t  ← expmod( a, n, z)
13    r  ← vrai
14    repeter
15      r ← minus( t, z )
16      t ← prdmod( t, t, z)
17      s ← s - 1
18    jusqua ( s = 0 ) ou ( t = 1)
19    si ( t = 1 ) alors
20      retourner r
21    fsi
22    retourner faux
23 fin
```

On peut montrer que dans ce cas,

$$\text{Prob}(\text{test}(z) =? \wedge \neg\text{premier}(z)) \leq \frac{1}{4}$$

Le nombre de multiplications effectuées par le test de Rabin et Miller est de l'ordre de $\log_2(z)^3$, en répétant ce test au plus 20 fois, on peut se convaincre du caractère premier d'un nombre de grande taille.

15. Logique de Hoare

Références

[1] Donald E. Knuth. *Art of Computer Programming, Volume 1 : Fundamental Algorithms (3rd Edition) (Art of Computer Programming Volume 1)*. Addison-Wesley Professional, 3 edition, November 1997.

[2] Donald E. Knuth. *Art of Computer Programming, Volume 2 : Seminumerical Algorithms (3rd Edition) (Art of Computer Programming Volume 2)*. Addison-Wesley Professional, 3 edition, November 1997.

[3] Donald E. Knuth. *Art of Computer Programming, Volume 3 : Sorting and Searching (3rd Edition) (Art of Computer Programming Volume 3)*. Addison-Wesley Professional, 3 edition, November 1997.

[4] P. langevin. Une brève histoire des nombres, 2003.