PREUVE ET ANALYSE DES ALGORITHMES

NICOLAS KARIBBOU

Résumé. Le cours d'algorithmique a pour objectif d'initier l'apprenti informaticien à l'analyse des algorithmes sous trois aspects fondamentaux : logique (correction), complexité (temps de calcul) et mise en oeuvre (langage C, `GNU/linux`).

Préliminaire

Le module de licence *Preuve et Analyse des Algorithmes* comprend 18 heures de cours, 12 heures de travaux-dirigés et 6 séances de travaux-pratiques. Le document comprend 6 sections générales, accompagnées de quelques morceaux choisis d'algorithmique. Les supports écrits ne sont pas finalisés, les étudiants sont invités à compléter le cours officiel, un document au format LaTeX, sous forme de projet `git` accessible au téléchargement quand la grange est ouverte :-)

```
git clone git@mapix.hd.free.fr:paa
```

Dernières modifications

```
    cours.tex 2016-10-16 09:40:40.456931882 +0200
     time.tex 2016-10-16 09:32:42.841621492 +0200
    intro.tex 2016-10-16 09:31:52.936701587 +0200
  euclide.tex 2016-10-16 09:27:14.233822931 +0200
     enum.tex 2016-10-16 09:26:18.637903602 +0200
   macros.tex 2016-10-13 10:56:36.387390415 +0200
    asymp.tex 2016-10-12 19:36:18.429937070 +0200
       bt.tex 2016-07-19 18:33:16.348954236 +0200
     tris.tex 2015-11-04 21:24:12.362879824 +0100
    hoare.tex 2015-10-24 06:49:40.708434351 +0200
   prolog.tex 2015-06-27 17:51:36.985978343 +0200
```

Date: Novembre 2013, dernière compilation :21 novembre 2017.

```
graphe.tex 2014-12-02 10:00:21.902244152 +0100
arbre.tex 2014-10-16 17:01:55.376079070 +0200
dev-arbre.tex 2014-10-16 17:01:07.788082382 +0200
walsh.tex 2014-07-21 10:57:00.907398441 +0200
prime.tex 2014-07-21 10:57:00.909398485 +0200
code.tex 2014-07-21 10:57:00.911398525 +0200
dlx.tex 2013-12-03 15:21:12.403056319 +0100
mul.tex 2013-10-04 13:12:23.174262798 +0200
tfr.tex 2013-10-04 13:12:23.174262798 +0200
```

Contributions

Philippe Langevin.

Table des matières

Préliminaire 1
Dernières modifications 1
Contributions 2
1. Algorithmique 5
1.1. étymologie 5
1.2. syntaxe 6
1.3. analyse 6
1.4. preuve 8
1.5. mise en oeuvre 8
1.6. numération 10
1.7. débordement 12
2. Notation asymptotique 14
2.1. crible d'Eratostène 14
2.2. ordre de grandeur 17
2.3. somme basique 19
2.4. série et intégrale 20
2.5. nombre de Bernoulli 21
2.6. dérivation discrète 22
2.7. exercice 22
3. Algorithme récursif 23
3.1. suite de Fibonacci 23
3.2. algorithme A 24
3.3. dévissage 25
3.4. tri fusion 25
3.5. arbre de récursion 26
3.6. méthode de Karatsuba 27
3.7. Fusion logarithmique 28

3.8. Plus petites valeurs 28
3.9. Elément majoritaire 29
4. Tri rapide 30
4.1. description 30
4.2. décomposition de Lomuto 31
4.3. Temps de calcul 33
4.4. Optimisation 34
4.5. borne minimale 35
4.6. exercice 36
5. Logique de Hoare 36
5.1. formalisme 36
5.2. règle 37
5.3. quelques preuves 37
5.4. exercice 38
6. Algorithme d'Euclide 38
6.1. Plus grand diviseur commun 38
6.2. Temps de calcul 39
6.3. PGCD binaire 39
6.4. Bâchet-Bézout 40
6.5. Euclide étendu 41
7. Enumération 43
7.1. n-uplet 43
7.2. code de Gray 44
7.3. permutation 44
7.4. lien dansant 44
8. Arbre binaire 44
8.1. définition 44
8.2. tableau 44
8.3. pointeur 44
8.4. arbre de recherche 45
8.5. equilibre 46
9. Bit-tracking 46
9.1. bit-à-bit 46
9.2. retour sur traces 48
9.3. permutation 48
9.4. les huit reines 48
9.5. cavalier d'Euler 48
10. Multiplication rapide 48
10.1. produit naif 48
10.2. Karatsuba 48
10.3. interpolation 48
11. Graphe 49

11.1. définition 49
11.2. connexité 50
11.3. Quelques problèmes 50
11.4. Parcours de graphe 52
11.5. Algorithme de Tarjan 52
11.6. Bellman-Ford 54
12. Codages 54
12.1. ascii 54
12.2. format 54
12.3. canal bruité 55
12.4. codage de Hamming 55
12.5. distance 56
12.6. linéarité 57
13. Transformée de Walsh 57
13.1. fonction booléenne 58
13.2. code de Reed-Müller 58
13.3. transformée de Walsh 58
13.4. décodage 59
14. Pseudoprimalité 61
14.1. Calcul modulaire 62
14.2. Théorème de Fermat 63
14.3. ordre 63
14.4. test de Fermat 64
14.5. duplication et addition 64
14.6. exponentiation modulaire 64
15. Transformée de Fourier 65
15.1. racine de l'unité 65
15.2. transformée de Fourier 65
15.3. transformation rapide 66
16. Liens dansants 66
16.1. couverture exacte 66
16.2. algorithme X 66
16.3. implantation 66
Références 66

FIGURE 1. Les origines du mot algorithme.

1. Algorithmique

L'objectif du cours est de fournir une initiation à l'algorithmique : temps de calcul, correction, implantation en `langage C` et expérience numérique sur un système `GNU/linux`.

1.1. **étymologie.** 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 un temps quelque peu mystérieuse. Longtemps, on croyait devoir lire dans le mot algorithme le radical grec *arithmos* qui signifie *nombre*. Mais, 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 XIe siècle, Léonard de Pise, rapporte de ses voyages méditerranéens le *sifr* de Perse. Lecteur des textes

1. Quel d'anagramme !

```
Euclide ( a, b : nombre )
nombre r
debut
  tantque ( b > 0 )
       r := a mod b
       a := b
       b := r
  ftq
  retourner r
fin
```

. Un algorithme ancestral pour calculer le plus grand commun diviseur de deux nombres entiers.

mathématiques arabes, il semblerait bien que Léonard de Pise, alias Fibonacci, soit à l'origine du mot algorithme. Un terme qu'il aurait utilisé pour décrire les procédés de calculs qu'il peut lire dans le traité *Kitab al-mukhtasar fi hisab al-jabr wa-l-muqabala* (règles de restaurations et réductions) écrit par un certain Al-Kwarezmi médecin et mathématicien arabe du IXe 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. Cette micro-introduction est tirée de la généalogie du cryptosystème RSA que je propose dans [6], les détails historiques sont dans le volume 1 de l'encyclopédie *The Art of Computer Programming* de Donald Knuth [3]".

1.2. **syntaxe.** Il existe plusieurs façons de d'écrire un algorithme : organigramme, pseudo-code, langage C, et python trés à la mode ces temps-ci sur les forums. Quelque soit l'approche, il s'agit d'exposer dans un langage clair et concis, prélude à la mise en oeuvre dans un langage de programmation, une méthode de résolution d'un problème. Le plus ancien des algorithmes calcule le PGCD de deux entiers postifs (nombres).

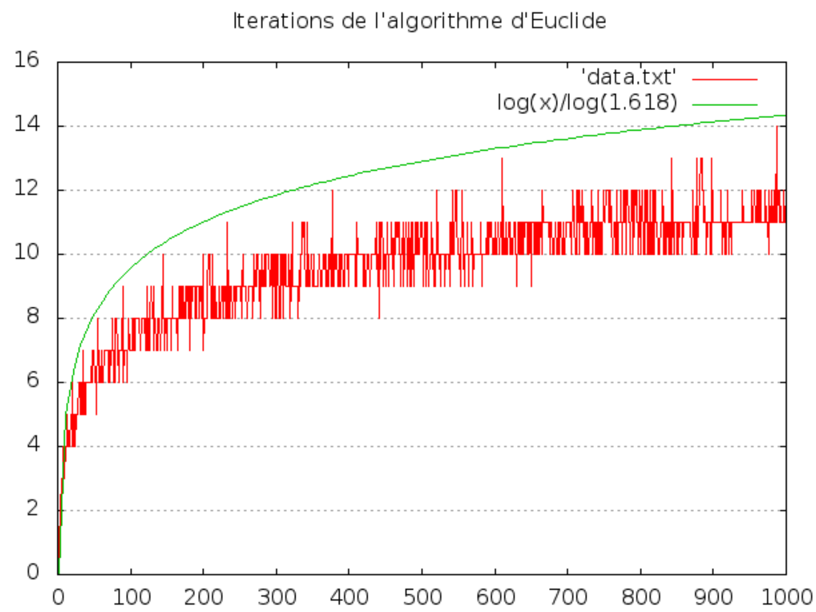
1.3. **analyse.** L'analyse de l'algorithme à pour but de prédire la performance du temps de calcul des implantations. Nous verrons que pour des instances de taille t, le corps de la boucle de l'algorithme d'Euclide est exécuté au plus t fois.

On peut se convaincre de ce fait mathématique en utilisant une implantation en langage C qui calcule $I(a, b)$

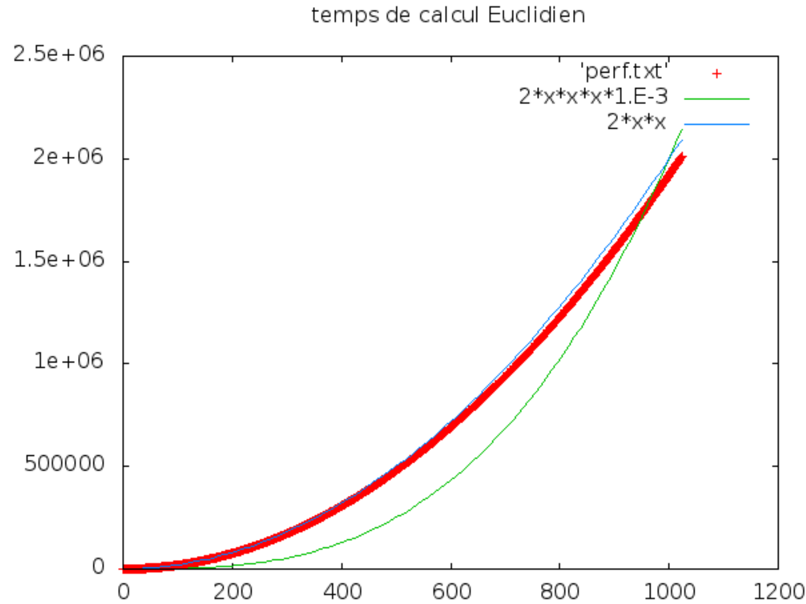
```
1  typedef unsigned int uint
2  uint nbiter ( uint a )
3  { uint r, cpt, scr = 0, x;
4    for ( x = 1; x < a; x++ ){
5           b   = x;
6           cpt = 0;
7           while ( b ) {
8               r = a % b;
9               a = b;
10              b = r;
11          cpt++
12          }
13          if ( cpt > scr ) cpt = scr;
14   }
15   return cpt;
16 }
```

Notons $I(a,b)$ le nombre d'itérations, le graphe de

$$a \mapsto \sup_{0<b\leq a} I(a,b)$$



L'algoritme de réduction modulaire consommant au plus t^2 étapes élémentaires, le temps de calcul de l'algorithme d'Euclide sur des instances de taille t est majoré par une fonction cubique i.e. de la forme At^3.



Une mesure empirique du temps de calcul, avec une implantation multiprécision, montre que l'estimation cubique n'est pas optimale, en réalité, la taille des nombres diminuant au cours des itérations euclidienne. Avec plus de finesse, on peut effectivement montrer que l'algorithme est de complexité au plus quadratique.

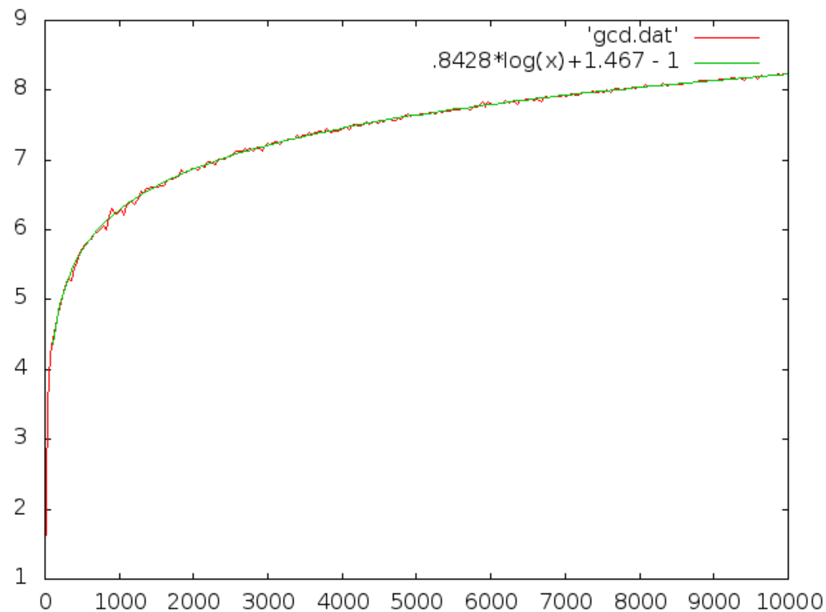
1.4. **preuve.** La preuve de correction algorithmique à s'appuie sur les notions de variant et d'invariant de bouble. Un variant est une grandeur qui décroit à chaque itération sans descendre en dessous d'un certain seuil. Un invariant est une formule logique valide aprés chaque itération. Dans le cas de l'algorithme d'Euclide, la fonction b est un variant, alors que les formules logiques :

$$0 < b \leq a, \quad \text{PGCD}(a, b) = \delta,$$

sont des invariants de la boucle euclidienne. Le variant prouve l'arrêt de la boucle, l'invariant permet d'élaborer une preuve.

1.5. **mise en oeuvre.** Le fichier de définition `arithmos.h`, les sources `arithmos.c` et `gcd.c` illustre la mise en oeuvre d'une expérience numérique maintenue par le fichier `makefile`.

```
1  #!/usr/bin/python
2  import sys, random,  math
3  def iter (  a,  b ):
4   →res  = 0
5   →while (b != 0) :
6   → →r  = a % b
7   → →c  = math.log(a);
8                      res  = res + c*c;
9   → →a,  b = b, r
10  →return res
11
12 l  = int(  sys.argv[1] )
13 a = 2
14 b = 1
15 for r  in range( 1,  l    ) :
16  →print r ,  iter (a,  b)
17  →a = a * 2 + random.randrange(1,2)
18  →b = b * 2 + random.randrange(1,2)
```



D'aprés Knuth,

$$\tau(a) = \frac{1}{\varphi(a)} \sum_{\substack{0 \leq b < a \\ \text{gcd(a,b)=1}}} I(a,b) = \frac{12}{\pi^2} \log(2)\log(a) + C - 1 + \mathrm{O}(a^{-1/6-\epsilon})$$

```
1  OPT=-Wall -g
2  #OPT=-Wall -O2
3  SHELL=/bin/bash
4
5  all  :  gcd.exe
6
7  arithmos.o :  arithmos.c
8          gcc $(OPT) -c arithmos.c
9
10 gcd.exe :  arithmos.o gcd.c
11         gcc $(OPT) arithmos.o gcd.c -o gcd.exe -lm
12
13 gcd.dat :  gcd.exe
14         ./gcd.exe 10000 > gcd.dat
15
16 gcd.png :  gcd.dat
17         echo "set term png; \
18         set output 'gcd.png';\
19         plot 'gcd.dat' w l, .8428*log(x)+1.467 - 1" |
       gnuplot
20 joli      :
21         indent -kr *.c
22 clean     :
23         rm -f gcd.dat
```

. makefile

```
1 #ifndef ARITHMOS
2 #define ARITHMOS
3 typedef long long int nombre;
4 nombre euclide( nombre a, nombre b, int *count);
5 #endif
```

. arithmos.h

où $C \approx 1.467$ désigne la constante de Porter.

1.6. **numération.** Une fois choisie une base $B > 1$, tout nombre entier $z < B^n$ se décompose sous la forme

$$z = (z_n \dots z_2 z_1)_B = \sum_{i=1}^{n} z_i B^{i-1}, \quad 0 \leq z_i < B$$

```c
#include "arithmos.h"

nombre euclide(nombre a, nombre b, int *count)
{
    nombre r;
    *count = 0;
    while (b > 0) {
        r  = a % b;
        a = b;
        b = r;
        (*count)++;
    }
    return a;
}
```

. arithmos.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "arithmos.h"

int main(int argc, char *argv[])
{
    nombre a, b;
    int cpt,  som = 0, phi = 0;
    nombre pas, max;
    max = atol(argv[1]);
    pas = max / 256;
    for (a = 1; a < max; a += pas) {
        som = 0;
        phi = 0;
        for (b = 1; b < a; b++)
            if  (1 == euclide(a, b, &cpt)) {
                som += cpt;
                phi = phi+ 1;
            }
        printf ( "%Ld %.2f\n", a, (float) som / phi );
    }
    return 0;
}
```

. gcd.c

Dans ce cas, z_1 est le sifr des unités, et si z_n n'est pas nul alors z est dit de taille n et z_n est le chiffre dominant de z. On représente un nombre par un tableau de chiffre, pour s'intéresser aux quotient et réduction modulo un chiffre en langage `C` :

```
1 #define TAILLE 1024
2 #define BASE   16
3 #typedef uint sifr
4 #typedef sifr nombre[ TAILLE ]
5 // pour un chiffre  b i.e.  b < BASE
6 sifr     mod( nombre z, sifr b )
7 void     div(  nombre z, sifr b )
8 void   init (  nombre z, int  v )
```

Exercice 1. *Implanter les primitives `mod`, `div` et `init` en précisant le temps de calcul.*

Exercice 2. *Faire une expérience numérique pour mesurer le temps de calcul d'une réduction modulaire en `python`.*

1.7. **débordement.** Soit n un module. L'ordre multiplicatif modulaire d'un entier x premier avec n désigne le plus petit entier f tel que $x^f \equiv 1 \pmod n$. Un tel entier est bien défini à condition que x soit inversible modulo n. En effet, l'entier k variant, la suite $x^k \pmod n$ prenant ses valeurs dans un ensemble fini, il existe $j > i$ tel que :

$$x^j = x^i \pmod n \Longrightarrow x^{j-i} = 1 \pmod n,$$

on peut alors montrer que f divise $j-i$, et c'est bien le plus petit entier qui vérifie $x^k = 1 \pmod n$.

```
⇝ gcc -Wall bord.c -o bord.exe
⇝ x=2147483648
⇝ n=6588122886535533079
⇝ ./bord.exe $x $n
ordre de 2147483648 modx 6588122886535533079 > 255
⇝ bc <<< "( $x ^ 3 ) % $n"
1
```

Exercice 3. *Observer les commandes passées sur le terminal.*

(1) Il y a un soucis, pourquoi ?

(2) Faire un diagnostic !

(3) Comment sortir de ce mauvais pas ?

```c
#include <stdio.h>
#include <stdlib.h>
/*
 * test l'ordre multiplicatif d'un entier modulo un autre
 * status : bug
*/

typedef unsigned long long nombre;

int ordre(nombre x, nombre m)
{
    int res = 1;
    nombre y = x;
    while (y != 1 && res < 255) {
        y = (x * y) % m;
        res++;
    }
    return res;
}

int main(int argc, char *argv[])
{
    nombre x = atoll(argv[1]);
    nombre m = atoll(argv[2]);
    int f = ordre( x, m );
    if ( f < 255 )
       printf ("ordre de %Ld modx %Ld = %d\n", x, m, f);
    else
       printf ("ordre de %Ld modx %Ld > 255\n", x, m);
    return 0;
}
```

. ordre.c

Indication : utiliser `bc -l` *pour calculer le logarithme à base* 2 *de* x *et* n.

2. Notation asymptotique

2.1. **crible d'Eratostène.** 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 chaotique. On introduit des *notations asymptotiques* pour décrire grossièrement ces fonctions.

Définition 1. *Un entier naturel est dit composé s'il s'écrit comme un produit de deux nombres différents de* 1*. Un nombre qui n'est pas composé est dit premier.*

Exercice 4. *Un nombre composé est divisible par un nombre premier, pourquoi ?*

Exercice 5. *Montrer par l'absurde qu'il existe une infinité de nombres premiers.*

Le nombre de premiers inférieurs à x est traditionnellement noté $\pi(x)$. Nous allons faire une expérience numérique pour mettre en évidence une certaine régularité dans la répartiton des nombres premiers.

Le programme `crible.c` est une implantation du célèbre crible d'Eratostène de Cyrène (-IIe) qui permet de déterminer la valeurs de $\pi(10^9)$ sans difficulté. La représentation graphique FG. (2) est réalisée par le fichier `makefile` qui s'appuie sur le fichier de commandes `gnuplot` :

```
set term png
set output 'pi.png'
set ylabel 'pi(x)'
set xlabel 'x'
set title 'distribution des nombres premiers'
plot 'pi.dat',x/log(x),1.25*x/log(x)
quit
```

On constate que le nombre de premiers inférieurs à 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 !

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

```
1  typedef unsigned char uchar;
2  typedef unsigned int uint;
3  /*  base du crible  d'Eratostene. Des ameliorations
4   *  sont possibles , lesquelles  ?                  */
5  uchar* crible ( uint n )
6  {  uint  i , j ;
7     uchar *t = calloc(n, sizeof(uchar));
8     for (i = 2; i < n; i++)
9          if (0 == t[i])
10             for (j = 2*i; j < n; j += i)
11                 t[j] = 1;
12     return t;
13 }
14
15 int main(int argc, char *argv[])
16 {  uint n = 100, i, cpt = 0;
17    uchar *t;
18    int opt, print = 0;
19    char *optstr="p:hl:";
20    while ((opt = getopt(argc, argv, optstr)) != -1) {
21      switch (opt) {
22        case 'p':  print = atoi(optarg);
23                   break;
24        case 'l':  n = 1 << atoi(optarg);
25                   break;
26        case 'h' :
27        default:
28          fprintf (stderr,  "Usage: %s [%s]\n", argv[0], optstr);
29          exit (EXIT_FAILURE);
30      }
31     }
32
33     t = crible( n );
34
35     if ( print ) {
36        print = n / print;
37        cpt = 0;
38        for (i = 2; i < n; i++) {
39          cpt += 1 - t[i];
40          if ( 0 ==  i % print )
41              printf ("pi(%d)=%d\n", i, cpt);
42        }
43        i--;
44        if ( 0 <  i % print  )
45             printf ("pi(%d)=%d\n", i, cpt);
46     }
47     return 0;
48 }
```

```
1  max=16
2  nbp=100
3  cflags =-Wall -g
4
5  all : crible.exe
6
7  crible.exe : crible.c
8          gcc $(cflags) crible.c -o crible.exe
9
10 demo : crible.exe
11         ./crible.exe -l7 -p10
12         /usr/bin/time --format="time=%E %U %S %P
       mem=%M pre=%c flt=%F/%R" ./crible.exe -l$(max
       )
13
14 pi.dat : crible.exe
15         ./crible.exe -p$(nbp) -l$(max)    \
16    | sed 's/[pif(=)]/ /g' > pi.dat
17
18 pi.png : pi.plt pi.dat
19         gnuplot pi.plt
20 clean :
21         rm -f *~ *.exe *.dat
```

. Fichier makefile

un résultat conjecturé par Gauss au XVIIIe, démontré beaucoup plus tard par Hadamard et De la Vallée-Poussin. Le temps de calcul du crible s'écrit :

$$T(n) = \alpha n + \beta\pi(n) + \gamma n\omega(n)$$

où α, β, γ sont des nombres postifs et $\omega(x)$ la somme des inverses des premiers inférieurs à x. On remarque sans difficulté que

$$\omega(x) = \sum_{p\leq n} \frac{1}{p} \leq \sum_{k\leq n} \frac{1}{k} \sim \log n$$

Il en résulte un algorithme rapide dont l'implantattion est davantage limité par la mémoire que par la célérité des processeurs.

Exercice 6. *La compilation* `gcc -S crible.c` *produit un code en assembleur qui permet de visualiser les opérations élémentaires pour prédire les valeurs des constantes cachées :* α, β *et* γ. *Estimer le temps de calcul sur une machine cadencée à 2Ghz.*

```
 1 crible :
 2 .LFB2:
 3  .cfi_startproc
 4  pushl %ebp
 5  .cfi_def_cfa_offset 8
 6  .cfi_offset 5, -8
 7  movl %esp, %ebp
 8  .cfi_def_cfa_register 5
 9  subl $40, %esp
10  movl $1, 4(%esp)
11  movl 8(%ebp), %eax
12  movl %eax, (%esp)
13  call calloc
14  movl %eax, -20(%ebp)
15  movl $2, -12(%ebp)
16  jmp .L2
17 .L6:
18  movl -12(%ebp), %eax
19  movl -20(%ebp), %edx
20  addl %edx, %eax
21  movzbl (%eax), %eax
22  testb %al, %al
23  jne .L3
24  movl -12(%ebp), %eax
25  addl %eax, %eax
26  movl %eax, -16(%ebp)
27  jmp .L4
28 .L5:
29  movl -16(%ebp), %eax
30  movl -20(%ebp), %edx
31  addl %edx, %eax
32  movb $1, (%eax)
33  movl -12(%ebp), %eax
34  addl %eax, -16(%ebp)
35 .L4:
36  movl -16(%ebp), %eax
37  cmpl 8(%ebp), %eax
38  jb .L5
39 .L3:
40  addl $1, -12(%ebp)
41 .L2:
42  movl -12(%ebp), %eax
43  cmpl 8(%ebp), %eax
44  jb .L6
45  movl -20(%ebp), %eax
46  leave
47  .cfi_restore 5
48  .cfi_def_cfa 4, 4
49  ret
50  .cfi_endproc
```

Exercice 7. *Concernant le crible, quelques améliorations sont possibles, lesquelles ?*

Exercice 8. *Donner une implantation du crible basée sur un tableau de bits, donc 8 fois plus économique en mémoire que pour un tableau d'octet.*

Exercice 9. *Utilisez* `gnuplot` *pour vous concaincre de*

$$\omega(x) = \sum_{p<x} \frac{1}{p} \sim \log\log x, \quad \textit{(Euler).}$$

2.2. ordre de grandeur.

Définition 2. *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*

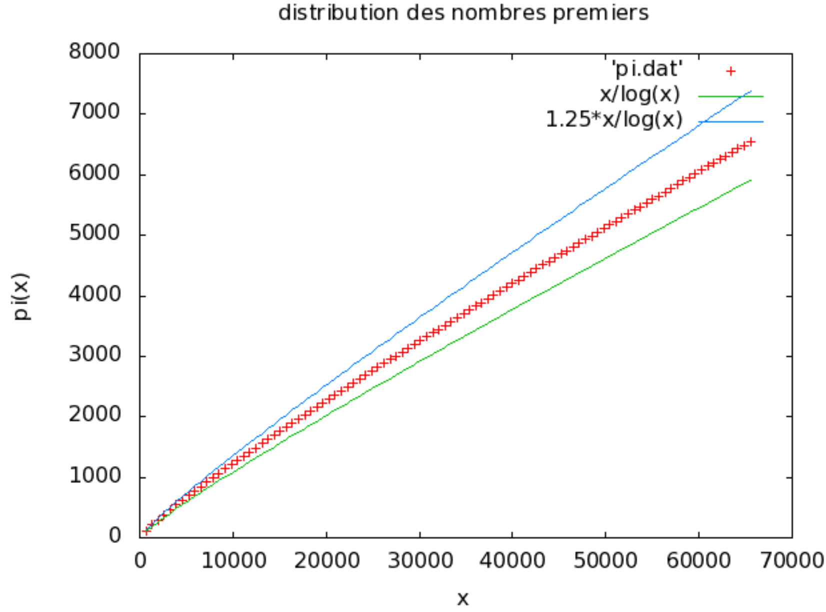


FIGURE 2. Graphe de la fonction $\pi(x)$. On constate que $\pi(x) = \Theta\big(x/\log x\big)$.

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 $g = \Theta(f)$.

Définition 3. *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 4. *Soient $f, g \colon \mathbb{N} \to \mathbb{R}$ deux fonctions positives. On dit que g domine f (asymptotiquement) 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 $g = \Omega(f)$.

```
1  #!/bin/bash
2  if [ $# = 0 ] ; then
3     set 2 1 4 3 6 5 8 7
4  fi
5  tab=($*)
6  n=$#
7  for(( i = 0; i < n; i++)) do
8    for((j = i + 1; j < n; j++ )) do
9       if [[ tab[i] -gt tab[j] ]] ; then
10            let tmp=tab[i]
11            let tab[i]=tab[j]
12            let tab[j]=tmp
13      fi
14   done
15 done
16 set tab
17 echo ${tab[*]}
```

Exercice 10. *Montrer que la relation : f est du même ordre de grandeur que g est une relation d'équivalence sur l'ensemeble des fonctions positives.*

2.3. **somme basique.** Considérons l'implantation `bash` de l'algorithme de tri par sélection, le nombre de comparaisons entre des éléments du tableau à l'etape i est égal à $(n-i)-(i+1)+1$. Le nombre total de comparaisons est donné par la somme de n termes en progression arithmétique de raison 1 :

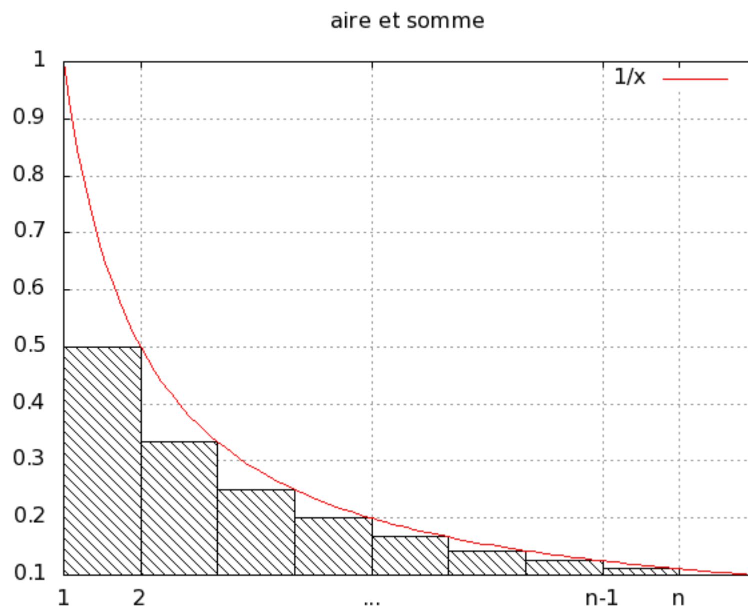
$$C(n) = \sum_{i=0}^{n-1}(n-i-1) = \sum_{0}^{n-1} i = n\frac{n-1}{2} \sim \frac{n^2}{2} = \Theta(n^2) \tag{1}$$

D'une manière générale, pour appréhender les temps de calul des algorithmes de ce cours, il faut absolument maîtriser les deux les formules sommatoires arithmétique et géométrique :

$$1+2+\cdots+(n-1) = \frac{(n-1)n}{2}, \quad q^0+q^1+\cdots+q^{n-1} = \frac{q^n-1}{q-1} \tag{2}$$

Un peu moins connue

$$1^2+2^2+\cdots+(n-1)^2 = \frac{n(n-1)(2n-1)}{6}, \tag{3}$$



et la surprenante relation

$$1^3 + 2^3 + \cdots + (n-1)^3 = (1 + 2 + \cdots + (n-1))^2 \tag{4}$$

2.4. **série et intégrale.** Les résultats de la section précédente sont généraux. En effet, si f est une fonction continue sur $\mathbb{R}+$, positive et croissante ou décroissante mais pas trop alors :

$$\sum_{k=1}^{n} f(k) \sim \int_1^n f(t)\mathrm{d}t \tag{5}$$

La preuve instructive est laissée.

Proposition 1.

$$\sum_{k=1}^{n} \frac{1}{k} \sim \log n \tag{6}$$

$$\sum_{k=1}^{n} k^{s-1} \sim \frac{1}{s} n^s \tag{7}$$

$$\sum_{k=1}^{n} \log k \sim n \log n \tag{8}$$

où s est un entier supérieur à 1.

Démonstration. Facile. □

```
set term png
set output 'sumint.png'
set xzeroaxis linetype 0 linewidth 1.000
set yzeroaxis linetype 0 linewidth 1.000
set title "aire et somme"
set grid
set xtics ("1" 1,"2" 2,"..." 5,"n-1" 8,"n" 9 )
set style rect fc lt -1 fs pattern 4

set object 1 rect from 1,0 to 2,1./2
set object 2 rect from 2,0 to 3,1./3
set object 3 rect from 3,0 to 4,1./4
set object 4 rect from 4,0 to 5,1./5
set object 5 rect from 5,0 to 6,1./6
set object 6 rect from 6,0 to 7,1./7
set object 7 rect from 7,0 to 8,1./8
set object 8 rect from 8,0 to 9,1./9


plot [1:10] 1/x
```

2.5. **nombre de Bernoulli.** Soit $f(X)$ un polynôme de degré s à coefficients positifs, de coefficient dominant γ :

$$f(n) \sim \gamma n^s.$$

Il semble que Jacques Bernoulli ait été l'un des premiers mathématiciens à s'être intéressé aux sommes :

$$S(n,s) = \sum_{k=0}^{n-1} k^{s-1}.$$

Il a établi que $S(n,s)$ est un polyôme de degré s en n.

Proposition 2. *Soit un entier $s > 1$. Il existe un polynôme r de degré s tel que :*

$$S(n,s) = \sum_{k=0}^{n-1} k^{s-1} = \frac{1}{s}n^s + r(n) \sim \frac{1}{s}n^s$$

Démonstration. On peut déterminer r à partir de la formule du binôme de Newton :

$$(a+b)^s = \sum_{i=0}^{s} \mathrm{C}_s^i a^i b^{s-i}.$$

$$\begin{array}{ccccc} n^s & - & (n-1)^s & = & \sum_{k=0}^{s-1} \mathrm{C}_s^k (n-1)^k \\ (n-1)^s & - & (n-2)^s & = & \sum_{k=0}^{s-1} \mathrm{C}_s^k (n-2)^k \\ & & \vdots & & \\ 2^s & - & 1^s & = & \sum_{k=0}^{s-1} \mathrm{C}_s^k 1^k \end{array}$$

En sommant,

$$(n+1)^s = 1 + \sum_{k=0}^{s-1} \mathrm{C}_s^k S(n,k)$$

□

2.6. **dérivation discrète.** Je vous propose astucieux raisonnement emprunté à l'analyse différentielle[2] pour montrer que $S(n,s)$ est un polynôme de degré s en n.

Soit $\mathcal{P}(\mathbb{Q})$ l'espace des fonctions à valeurs dans le corps $\mathbb{Q}$ des rationnels définies sur l'ensemble des entiers naturels. Pour $f \in \mathcal{P}(\mathbb{Q})$, on note f' la dérivée discrète de f, elle est définie par :

$$f'(n) = f(n+1) - f(n)$$

(1) Résoudre l'équation $f' = 0$.
(2) Résoudre l'équation $f' = 1$.
(3) Montrer que $\deg(f') = \deg(f) - 1$.
(4) Montrer que si f' est polynomiale alors f est un polynôme.

2.7. **exercice.**

Exercice 11. *Déterminer le temps de calcul du programme* `loops`*. Procéder à une vérification expérimentale basée sur la commande* `time` *, puis une autre basée sur le profileur* `gprof`*.*

2. suggéré par Jean-Jacques

```
1  #include <stdlib.h>
2
3  int sum=0;
4
5  void add( void) { sum++; }
6  void inc( void) { sum++; }
7
8  int main( int argc, char* argv[] )
9  {
10 int  i, j, k, n;
11
12 n = atoi( argv[1]  );
13
14 for( i  = 0; i  < n; i++ )
15  for( j  = 0; j  < i * i;  j++ )
16     if  ( 0 == j % i )
17         for( k = 0; k < j;  k++ )
18                 add();
19      else inc();
20
21 return 0;
22 }
```

3. Algorithme récursif

3.1. suite de Fibonacci. D'aprés le site Integer Sequences de N. J. A. Sloane, la suite d'entiers 0, 1, 1, 2, 3, 5 trés connue sous le nom de suite de Fibonacci, est parfois appelée suite de Lamé, comme en comprendrons la raison plus loin dans le cours.

Il s'agit d'une suite récurrente d'ordre deux dont le terme général F_n s'obtient par les relations :

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

Les partisans du moindre effort en déduisent le code récursif `ullong fib( int n )`. Avant de vérifier qu'il s'agit d'une de mise en oeuvre désastreuse, je vous suggère d'écrire une fonction itérative linéaire en temps de calcul.

```
1 ullong fib ( int n )
2 {
3 if ( n < 2 ) return n;
4 return fib(n−1)+fib(n−2);
5 }
```

Exercice 12. *Donner une implantation itérative pour calculer le n-ième terme de la suite de Lamé-Fibonacci.*

La version itérative calcule F_{100} en 1μsec, le résultat ne sera pas correct à cause d'un débordement. Il faudrait attendre un siècle pour obtenir le même résultat par la version récursive ! En effet, notons $T(n)$ le nombre d'étape :

$$T(n) = T(n-1) + T(n-2), \quad T(0) = T(1) = 1.$$

autrement dit $T(n) = F_{n+1}$.

Lemme 1. *La suite de Fibonacci est exponentielle. Plus précisément, si ϕ (resp :$\hat{\phi}$) désigne la racine positive (resp :négative) du polynôme $T^2 - T - 1$ alors :*

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

Démonstration. L'induction sur n ne pose aucune difficulté ! □

On peut utiliser `bc -l` pour calculer de nombre d'or, et passer à une application numérique :

$$\phi = \frac{1+\sqrt{5}}{2} = 1.618\ldots \quad \log_2 \phi = .694\ldots$$

et donc $\log_2 T(100) \approx 69\ldots$ c'est beaucoup, non ?

Exercice 13 (débordement)**.** *A partir de quelle valeur de n, le calcul de F_n dépasse la capacité des entiers de 64 bits ?*

3.2. **algorithme A.** Un grand nombre d'algorithmes de type *diviser pour régner* (divide and conquer) suivent le schéma récursif `A` : pour traiter une instance de taille n, les solutions de deux problèmes de taille $n/2$ sont fusionnées par un fonction B, ou encore celui du shéma A_p où ce sont p solutions de taille $n/2$ qui sont fusionnées. Le temps de calcul dépend de celui de B. Pour obtenir une formule, on peut raisonner sur un arbre d'exécution abstrait, ou bien dévisser la récursion pour obtenir une formule sommatoire.

```
void A( int n )
{
  if ( n > 1 ) {
     A( n/2 );
     A( n/2 );
     B( n );
  }
}
```

```
void A_p( int n )
{
  if ( n > 1 ) {
     A( n/2 );
        ...  p fois
     A( n/2 );
     B( n );
  }
}
```

3.3. **dévissage.** Notons T_A le temps de calcul de l'algorithme A. Il dépend du temps de calcul T_B de B. La récursion induit une formule récursive

$$\begin{aligned} T_A(n) &= 2T_A(\frac{n}{2}) + T_B(n) \\ &= 2\big(2T_A(\frac{n}{4}) + T_B(\frac{n}{2})\big) + T_B(n) \\ &= 2^2 T_A(\frac{n}{2^2}) + 2^1 T_B(\frac{n}{2}) + T_B(n) \\ &= 2^k T_A(\frac{n}{2^k}) + 2^{k-1} T_B(\frac{n}{2^{k-1}}) + \cdots + 2^1 T_B(\frac{n}{2}) + T_B(n) \end{aligned}$$

Le nombre d'appels récursifs vaut $k := 1 + [\log_2 n]$. Au final, le temps de calcul prend la forme :

$$T_A(n) = 2^k \gamma + \sum_{i=0}^{k} 2^i T_B(\frac{n}{2^i}) \tag{9}$$

où γ est une constante positive.

3.4. **tri fusion.** Pour trier une table de taille n, on peut trier les $n/2$ premiers éléments, puis le $n - n/2$ suivant et fusionner les résultats au

TABLE 1. Temps de calcul de A

T_B	T_A
constant	linéaire
logarithmique	linéaire
linéaire	$n \log n$

travers d'une table auxiliaire. Il en résulte un algorithme de temps de calcul $n \log_2 n$ quelque peu mémoriphage !

```
TriFusion( t: table, l, r: indice  )
var m : indice
si ( r-l < seuil ) alors
        m := ( r + l ) / 2
 TriFusion( t,  l,  m  )
 TriFusion( t, m+1, l  )
        fusion( t, l, r )
sinon
        TriSelect( t, l, r )
fsi
```

Exercice 14. *Compléter l'algorithme de fusion.*

Exercice 15. *Implanter le tri fusion en déterminant la valeur de seuil optimale.*

Exercice 16. *Implanter le tri fusion sans récursion dans le cas d'un tableau de taille 2^l. Il s'agit de procéder en l étapes. On commence par fusionner les 2^{l-1} tableaux de taille 1, puis les 2^{l-2} tableaux de taille 2 etc...*

3.5. arbre de récursion.

Lemme 2 (berger fou)**.** *Soit f une application surjective d'un ensemble fini E dans un ensemble F. Si le nombre d'antécédants est constant égal à λ alors*

$$|E| = \lambda \times |F|$$

En particulier, dans un arbre binaire, le nombre de noeuds total n est relié au nombre de feuilles f par

$$n = 2f - 1. \tag{10}$$

Exercice 17. *Déterminer le nombre de noeuds internes d'un arbre p-aire en fonction du nombre de feuilles.*

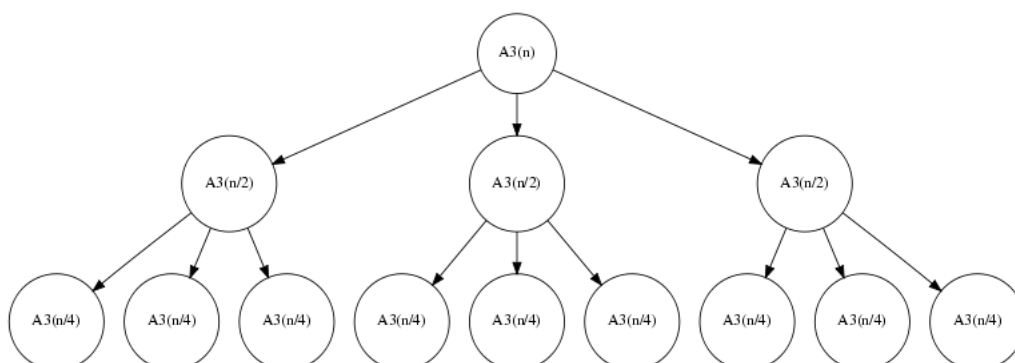


FIGURE 3. Arbre d'exécution de `Ap( n )`. Le coût du niveau l est $p^l \times T_B(\frac{n}{2^l})$

L'observation d'un arbre d'exécution fournie une formule analogue à (9) sur le champ. En effet, il s'agit d'un arbre p-aire avec : une racine au niveau 0, p noeuds de niveau 1, p^l noeuds de niveau l, etc.... Le temps de calcul d'un des p^l noeuds de niveau l est $T_B(\frac{n}{2^l})$. Pour simplifier, on suppose que n est une puissance de 2, la hauteur de l'arbre est $l := \log_2 n$ et le temps de calcul vaut :

$$T_{A_p}(n) = \alpha p^l + \sum_{k=0}^{l} T_B(\frac{n}{2^k})p^k$$

Par exemple, quand B est linéaire :

$$T_{A_p}(n) = \alpha p^l + \beta n \sum_{k=0}^{l} (\frac{p}{2})^k = \alpha p^l + n\frac{(\frac{p}{2})^l - 1}{\frac{p}{2} - 1}$$

Pour finaliser ce calcul, $p^l = (2^{\log_2 p})^l = n^{\log_2 p}$.

3.6. **méthode de Karatsuba.** La méthode de Karatsuba pour calculer le produit de deux polynômes A et A' de taille $2n$ consiste à écrire

$$A = aZ^n + b, \quad A' = a'Z^n + b',$$

où a, a', b et b' sont de tailles n.

$$AA' = abZ^{2n} + (ab' + a'b)Z^n + a'b'.$$

L'astuce algorithmique consiste à remarquer que

$$(ab' + a'b) = (a + a')(b + b') - ab - a'b'$$

et donc une multiplication de taille $2n$ s'obtient par fusion linéaire de trois produit de taille n. Le temps de calcul prend la forme :

$$\Theta(n^{\log_2 3}) \approx n^{1.58496\ldots} \approx n\sqrt{n}.$$

3.7. **Fusion logarithmique.** Le temps de calcul de l'algorithme $A_{2,2}$ à fusion logarithmique fait interbenir la somme

$$\sum_{k=0}^{\log n} 2^k \log(n/2^k) = \log n \sum_{k=0}^{\log n} 2^k \log(n/2^k) - \sum_{k=0}^{\log n} k2^k$$

On peut résoudre cette forme indéterminée par une manipulation classique :

$$\sum_{k=0}^{n-1} kX^k = X\frac{\mathrm{d}}{\mathrm{d}X}\frac{X^n-1}{X-1} = \frac{nX^n(X-1)-X^{n+1}}{(X-1)^2}$$

d'où l'on tire

$$\sum_{k=0}^{\log n} k2^k = n\log n - 2n$$

Au final, le temps de calcul est linéaire.

Exercice 18. *On considère la série*

$$\mu(l) := \sum_{k=1}^{l} k2^{-k} \tag{11}$$

1. *Etudier (D'Alembert) la convergence de $\mu(l)$.*
2. *Vérifier que la série apparaît dans l'analyse du temps de calcul de l'algorithme `A` à fusion logarithmique.*
3. *Vérifier que la série intervient dans l'analyse en moyenne du temps de calcul l'algorithme incrémental des entiers binaires.*
4. *Utiliser une analyse amortie pour compter globalement les jetons distribués par `inc` sur toutes les instances de taille n, en déduire :*

$$\lim_{l\to+\infty} \mu(l) = 2.$$

3.8. **Plus petites valeurs.** On suppose une relation d'ordre sur des objets. Le problème $\textsc{ppv}(k,n)$ consiste à déterminer les k plus petites valeurs d'une table de n objet en tenant compte des multiplicités éventuelles.

1. Combien de comparaisons sont requises pour résoudre $\textsc{ppv}(1,n)$.
2. Donner un algorithme naïf pour résoudre $\textsc{ppv}(2,n)$ avec $2n-3$ comparaisons.

```
inc( z : entier )
variable i : indice
debut
   i := 0
   tanque z[i] = 1
      // jeton
      z[i] := 0
      i := i + 1
   ftq
   z[i] = 1
fin
```

. inc

(3) Donner une solution "diviser pour régner" consommant $\frac{3}{2}n+\mathrm{O}(1)$ comparaisons.

(4) Comment résoudre PPV$(2, n)$ en $n + \log_2 n + 1$ comparaisons.

(5) Montrer que le nombre de comparaisons pour résoudre PPV(k, n) de manière naive est $nk - \frac{(k+1)k}{2}$.

(6) On peut résoudre PPV(k, n) en $n \log_2 n$ comparaisons. Comment ?

(7) A partir de quelle valeur k la dernière suggestion est meilleure que l'approche naïve ?

La section *minimum-comparison selection* de [5] rapporte l'histoire de cette formidable question.

3.9. **Elément majoritaire.** Un élément x est dit majoritaire dans un tableau t de n objets, si le nombre d'occurences de x dans t est strictement supérieur à $\frac{n}{2}$. En particulier, il existe au plus un élément majoritaire dans t !

(1) Ecrire un algorithme pour déterminer l'existence d'un élément majoritaire.

(2) Préciser le temps de calcul.

(3) Utiliser le principe "diviser pour régner" pour obtenir un algorithme en $\mathrm{O}(n \log n)$.

(4) On note x_k les valeurs de t, et n_k le nombre de positions i paires tel que $t[i] = t[i+1] = x_k$. Montrer que si n est pair et x_1 est majoritaire alors

$$n_1 + n_2 + \ldots + n_k < 2n_1.$$

(5) Ecrire un énoncé analogue dans le cas impair.

(6) Déduire des 2 questions précédentes un algorithme linéaire.

(7) Montrer que la fonction `void Major(int t, int n)` détermine l'élément majoritaire en un temps linéaire.

```
1
2 int Major( int t[],  int n )
3 { // invariants : a[i] != a[i+1],  c[j]=c[0]
4   int a[n], c[n];
5   int i=0, j=0, k;
6   a[0]=t[0];
7   for( k = 1; k < n; k++ ){
8       if ( a[i] != t[k] )
9           a[++i]=t[k];
10      else {
11        if ( j > 0 ) {
12            if ( c[j-1] == t[k] ) c[j++]=t[k];
13            else {
14                a[++i] = c[j--];
15                a[++i] = t[k];
16                if ( j > 0 ) a[++i] = c[j--];
17            }
18        } else  c[j++]=t[k];
19      }
20  }
21  if ( j == 0 ) return -1;
22  if ( occ( c[0],  t, 0, n ) > n / 2) return c[0];
23  return -1;
24 }
```

4. Tri rapide

A début des années 60, C.A.R. Hoare [3] a publié un article concernant l'analyse d'une méthode de tri connue sous le nom de `quick sort`, une copie de cet article est sur le site du cours.

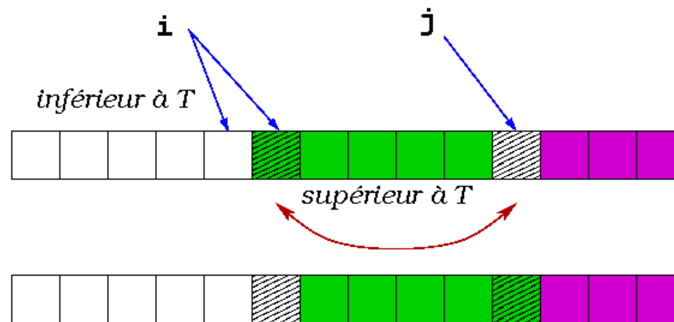
4.1. **description.** L'algorithme présenté ici est tiré de l'ouvrage du bijectioniste Herbert Wilf [10], dont une version est disponible sur la toile à partir des mots clefs : `wilf`, `herbert`, `complexity`,`algorithm`.

Il utilise le principe algorithmique *diviser pour régner* pour décomposer le travail du tri de taille (left - right + 1) en deux tâches de taille respective (left - i) et (i -right). La procédure `split` déplace les éléments

3. Inventeur de la logique du même nom !

```
procedure qksort(x:array; left, right:integer);
{sorts the subarray x[left], . . . , x[right]}
 if right - left >= 1 then
 p := split( x, left, right )
 qksort(x, left, p - 1);
 qksort(x, p + 1, right)
end.{qksort}
```

. quick sort



du sous-tableau et retourne un pivot p vérifiant :

$$\forall k \leq p, \quad x[k] \leq x[p]; \qquad \forall k \geq p, \quad x[k] \geq x[p].$$

Un raisonnement inductif s'appuyant sur la condition qui précéde fournit sur le champ preuve une preuve de correction de l'algorithme.

Il existe plusieurs implantations de recherche de pivot, elles sont toutes linéaires.

4.2. **décomposition de Lomuto.** La décomposition de Lomuto est facile à prouver.

Démonstration. Pour un tableau indexé par les entiers de l à r, elle s'appuie sur l'invariant :

$$x[l] = T, \qquad \forall k \leq i, \quad x[k] < T \qquad \forall k, \quad i < k \leq j, \quad x[k] \geq T.$$

□

```
1 void rapide(char t[], int g, int d)
2 {
3     int i, j, v, x;
4 // algorithme en Langage C, Sedgewick.
5     node(t, g, d);
6     if (g < d) {
7         v = t[d];
```

```
procedure split(x, left, right, i)
T := x[left];
i := left;
for j := left + 1 to right do
   if x[j] < T  then
 i := i + 1
 swap(x[i], x[j])
   fi
done
swap(x[left], x[i])
end. {split}
```

. Lomuto

```
        i = g − 1;
        j = d;
        for (;;) {
            while (t[++i] < v);
            while (t[−−j] > v);
            if (i >= j)
                break;
            x = t[i];
            t[i] = t[j];
            t[j] = x;
        }
        x = t[i];
        t[i] = t[d];
        t[d] = x;
        edge(g, d, i);
        rapide(t, g, i − 1);
        rapide(t, i + 1, d);
    }
}
```

. un exemple de tri rapide

```
rapide.png : rapide.c
 gcc −Wall −g rapide.c
 ./a.out 'PROCEDURE DE PARTITION' > rapide.dot
 dot −Tpng rapide.dot −o rapide.png
```

Exercice 19. *Proposer un invariant pertinent pour la boucle `for` de la procédure de division de `rapide.c`.*

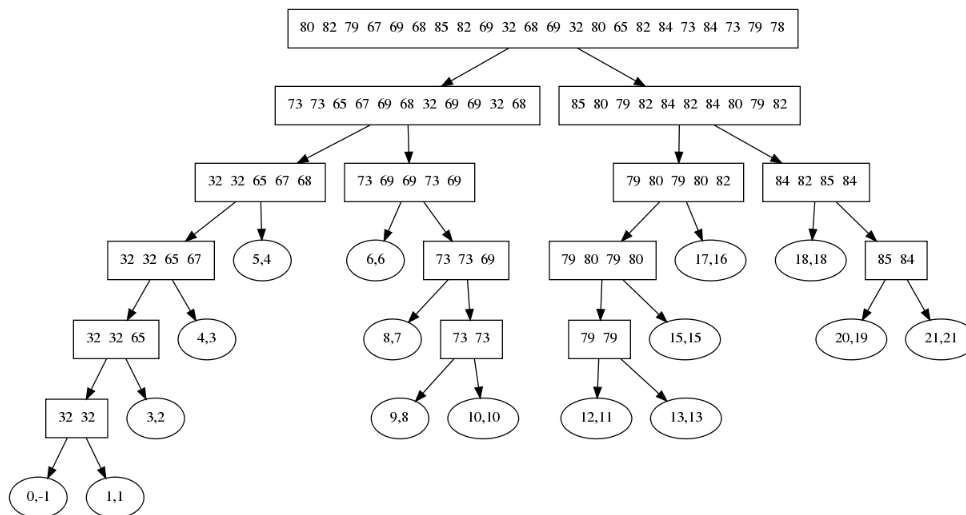


FIGURE 4. L'arbre des divisions du tri rapide en action.

4.3. **Temps de calcul.**

4.3.1. *cas défavorable.* La fonction de coupure retourne toujours un élément extrémal. En notant $L(n)$ le temps de coupure, il existe une constante K telle que

$$T(n) = T(n-1) + L(n) + K \tag{12}$$

Au final, un temps de calcul quadratique.

4.3.2. *fonctionnement idéal.* La fonction de coupure retource un indice médian. En supposant que n est une puissance de 2

$$T(n) = 2T(n/2) + L(n) + K \tag{13}$$

Au final, un temps de calcul de l'ordre de $n \log_2 n$.

4.3.3. *cas moyen.* Notons $C(n)$ le nombre moyen de comparaisons effectuées par `split` satisfait la relation

$$C(n) = (n-1) + \frac{2}{n} \sum_{k=1}^{n-1} C(n-k).$$

On peut deviner la forme de C en supposons une fonction à croissance raisonnable, et en utilisant la technique de comparaison d'une série avec une intégrale

$$C(x) = x + \frac{2}{x} \int_1^x C(t)\mathrm{d}t,$$

qui conduit à l'équation différentielle

$$\begin{aligned}C'(x) &= 1 - \frac{2}{x^2}\int_1^x C(t)\mathrm{d}t + 2C(x)/x \\ &= 1 - \frac{1}{x}(C(x) - x) + 2C(x)/x \\ &= 2 + C(x)/x\end{aligned}$$

dont la solution générale est $2x \log x$ plus un terme linéaire. Cela dit, on peut tout aussi bien s'en tirer par un calcul direct

$$C(n) = (n-1) + \frac{2}{n}\sum_{k=1}^{n-1} C(n-k) \tag{14}$$

$$C(n-1) = (n-2) + \frac{2}{n-1}\sum_{k=1}^{n-2} C(n-1-k) \tag{15}$$

(16)

D'où l'on tire que

$$nC(n) - (n-1)C(n-1) = 2C(n-1) + n(n-1) - (n-1)(n-2)$$

soit

$$nC(n) = (n+1)C(n-1) + 2(n-1)$$

et donc que $F(n) := \frac{C(n)}{n+1}$ satisfait la relation assez conviviale

$$\begin{aligned}F(n) &= F(n-1) + 2\frac{n-1}{n(n+1)} \\ &= F(n-1) + 2(\frac{1}{n} - \frac{1}{n(n+1)})\end{aligned}$$

La série de terme général $\frac{1}{n(n+1)}$ converge, il suit que :

$$C(n) \sim 2nH(n) \sim 2n\log(n) \tag{17}$$

où $H(n)$ désigne la série Harmonique, en particulier, c'est bien un logarithme népérien qui apparaît dans la formule.

4.4. **Optimisation.**

4.4.1. *seuil.* Un appel récursif a toujours un coût : sauvegarde de contexte, empilage et dépilage des paramètres, retour et restauration de contexte. Pour une instance de petite taille, il est péférable d'utiliser une méthode directe : un tri par insertion.

4.4.2. *Elimination de la récursivité.* Un algorithme de tri rapide s'appuie sur un parcours d'arbre binaire. Il est assez facile de supprimer la récursivité c'est-à-dire gérer une pile à la main pour optimiser le code.

4.4.3. *la fonction `qsort`.* Un tri rapide est disponible dans la bibliothèque standard. Vous pouvez vérifier, d'une part, qu'il s'agit bien de la fonction utilisée par la commande `sort`, et que, d'autre part, les deux optimisations précédentes sont mises en oeuvre.

```
tri> which sort
/usr/bin/sort

tri> objdump -T /usr/bin/sort | grep qsort
00000000      DF *UND* 00000000  GLIBC_2.0   qsort

tri> git clone git://sourceware.org/git/glibc.git

tri> grep -iEB3 'def.*thres' glibc/stdlib/qsort.c

/* Discontinue quicksort algorithm when partition
gets below this size.This particular magic number
was chosen to work best on a Sun 4/260. */
#define MAX_THRESH 4
```

4.5. **borne minimale.** A l'image du tri `timesort`, ou plus sérieusement du tri linéaire, un algorithme de tri n'est pas toujours fondé sur la comparaison des objets qu'il doit trier !

```
1 #!/bin/bash
2 for x in $*
3 do
4   ( sleep $x ; echo -n .$x ) &
5 done
```

. timesort

Théorème 1. *Le temps de calcul de d'un algorithme de tri comparatif est de complexité* $\Omega(n \log n)$.

```
QSORT(3)          Linux Programmer's Manual
NAME
        qsort, qsort_r − sort an array
SYNOPSIS
        #include <stdlib.h>

   void qsort(void *base, size_t nmemb, size_t size,
              int (*compar)(const void *, const void *))
   ;
DESCRIPTION
        The  qsort()  function sorts an array with
   nmemb
 elements of size size. The base argument points to
   the
 start of the array.
```

Démonstration. □

Le théorème affirme donc l'impossiblité de trouver un algorithme de tri comparatif plus performant que le tri rapide.

4.6. exercice.

4.6.1. *tri linéaire.* Le tri d'une table de taille n de nombres positifs inférieurs à un entier $m = \mathrm{O}(n)$ est réalisable en temps linéaire.

Il suffit d'utiliser un tableau auxiliaire de taille m pour compter le nombre d'occurrences des valeurs de t.

Exercice 20. *Ecrire une fonction* `void trilin(int* t, int n)` *pour trier un tableau de taille n sachant que la plus grande valeur de t est $\mathrm{O}(n)$.*

5. Logique de Hoare

5.1. **formalisme.** La logique de Floyd-Hoare est une méthode de preuve formelle décrite par Hoare en 1969. Elle étend la logique des prédicats par l'ajout d'un nouveau symbole :

$$p \xrightarrow{S} q, \qquad \text{souvent noté} \quad \{p\}S\{q\}$$

ou p (précondition) , q (postcondition) sont des prédicats logiques et S un programme (action). Le symbole $p \xrightarrow{S} q$ généralise l'implication logique, il se lit : p par le programme S entraine q. Tout le monde peux se convaincre que :

```
1  // example from man qsort
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  static int
7     cmpstringp(const void *p1, const void *p2)
8     {
9   /* The actual arguments to this function are "pointers
10  * to pointers to char", but strcmp(3) arguments are
11  * "pointers to char", hence the following cast plus
12  * dereference
13  */
14    return strcmp(* (char * const *) p1,
15                                * (char * const *) p2);
16    }
17 int main(int argc, char *argv[])
18  {
19  int j;
20
21  if (argc < 2) {
22      fprintf (stderr, "Usage: %s <string>...\n", argv[0]);
23      exit (EXIT_FAILURE);
24      }
25
26  qsort(&argv[1], argc - 1, sizeof(char *), cmpstringp);
27
28  for (j = 1; j < argc; j++) puts(argv[j]);
29        exit (EXIT_SUCCESS);
30  }
```

$$x = \alpha \overset{x:=x+y;y:=x-y;x=x-y}{\longrightarrow} x = \beta$$

Il suffit de tracer la séquence d'affectation (2).

La logique de Hoare permet de formaliser ce type de preuve. On s'intéresse aux séquences de programme $S; P$, à l'affectation des variables $x := t$, à l'alternative $(c)?S : T$ et l'itération $\mathrm{W}(b)S$.

5.2. **règle.**

5.3. **quelques preuves.**

TABLE 2. Traçage d'une séquence d'affectation.

x	y	
α	β	
		x := x+y
$\alpha+\beta$	β	
		y := x-y
$\alpha+\beta$	α	
		x := x-y
β	α	

5.4. **exercice.**

6. Algorithme d'Euclide

6.1. **Plus grand diviseur commun.** On rappelle qu'un entier d est un diviseur d'un entier z s'il existe un entier q tel que $z = qd$. On note PGCD(x, y), ou plus concisément (x, y), le plus grand diviseur commun à deux entiers x et y.

(1) si x n'est pas nul $(x, 0) = x$.
(2) $(x, y) = (y, x)$
(3) $(x, y) = (x + y, y) = (x, y - x)$
(4) $(x, y)z = (xz, yz)$

L'algorithme d'Euclide calcule les restes successifs de la divison euclidienne entre les nombres $r_0 = y$ et $r_1 = x$, jusquà trouver un reste nul

$$\begin{aligned} r_0 &= q_1 r_1 + r_2 \\ r_1 &= q_2 r_2 + r_3 \\ &\vdots \\ r_{n-1} &= q_n r_n + 0 \end{aligned}$$

La stricte décroissance des restes prouve la correction de l'algorithme.

Notons au passage que la preuve de correction implique l'existence de deux entiers u et v tel que

$$xu + yv = (x, y).$$

6.2. **Temps de calcul.**

Théorème 2 (Lamé, 1847)**.** *Le nombre d'itérations de l'algorithme d'Euclide est au plus linéaire en la taille des entrées. Si* $a >= b$ *et* $I(a,b) = n$ *alors :*

$$F_n \leq b, \quad F_{n+1} \leq a.$$

Démonstration. On note $a := r_0$, et $r_1 := b$. Supposons le PGCD des entiers $b \leq a$ calculé en n itérations. On a $r_n \geq 1 \geq F_1$ et $r_{n+1} = 0 \geq F_0$, puis, en remontant le courant au fil de l'eau :

$$\begin{array}{lrrrr}
(18) & r_{n-1} = & q_n r_n + & 0 & \\
(19) & \geq & F_1 + & F_0 = & F_2 \\
(20) & r_{n-2} = & q_{n-1} r_{n-1} + & r_n & \\
(21) & \geq & F_2 + & F_1 = & F_3 \\
(22) & \vdots & & & \\
(23) & r_1 = & q_1 r_2 + & r_3 & \\
(24) & \geq & F_{n-1} + & F_{n-2} = & F_n \\
(25) & r_0 = & q_1 r_2 + & r_3 & \\
(26) & \geq & F_n + & F_{n-1} = & F_{n+1} \\
(27) & & & &
\end{array}$$

□

Le coût d'une réduction modulaire étant quadratique, on en déduit que l'algorithme d'Euclide est de complexité au plus cubique sur les grands entiers. L'expérience montre qu'il est en réalité quadratique. La démonstration est hors programme.

Exercice 21. *Prouver le résultat original de Lamé qui affirme que le nombre d'étapes de l'algorithme d'Euclide exécuté sur deux entiers est majoré par cinq fois le nombre de chiffres décimaux du plus court.*

Exercice 22. *Ecrire un programme en Langage C qui lit un entier* k *sur la ligne de commande pour déterminer des paires d'entiers aléatoires* (a,b) *tel que* $I(a,b) = k$*.*

6.3. **PGCD binaire.** La méthode du PGCD binaire s'appuie sur les relations :

(1) si x n'est pas nul alors $(x,0) = x$.
(2) si x et y sont impairs $(x,y) = (x, \frac{y-x}{x})$.

(3) si x et y sont pair alors $(x, y) = 2(x/2, y/2)$.

(4) si x est impair et y pair alors $(x, y) = (x; y/2)$.

On en déduit sur le champ l'algorithme `pgcdbin` :

```
pgcdbin ( x, y : nombre )
entier  k = 0
debut
  tantque ( pair( x ) & pair( y ) )
      x := x / 2
      y := y / 2
      k := k + 1
  ftq
  tantque ( pair( x ) > 0 )
       x := x / 2
  ftq
  tantque ( y <> 0 )
     tantque pair ( y )
        y := y / 2
     ftq
     si x > y alors x :=: y fsi
     y := y − x
  ftq
  tantque ( k > 0 )
     y := y * 2
     k := k − 1
  ftq
fin
```

. PGCD binaire

Proposition 3. *L'algorithme du PGCD binaire est de complexité quadratique.*

Démonstration. Chaque itération a un coût linéaire. Le nombre d'itération de la boucle principale est majoré par la taille binaire des entiers. □

6.4. **Bâchet-Bézout.** Un sous groupe $\mathfrak{I}$ de $\mathbb{Z}$ est un idéal quand :

$$\forall x \in \mathbb{Z}, \quad \forall i \in \mathfrak{I}, \qquad xi \in \mathfrak{I}.$$

La division euclidienne montre que l'idéal $\mathfrak{I}$ est principal : il existe un unique entier $d > 0$ tel que $=d\mathbb{Z}$.

Proposition 4. *Soient a et b deux entiers. Il existe deux entiers relatifs u et v tels que*

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

Démonstration. On peut démontrer ce résultat de plusieurs manières. Une des plus instructives, consiste à remarquer que les idéaux de $\mathbb{Z}$ sont principaux. L'idéal engendré par a et b est engendré par un entier d, cet élément est divisible par leur PGCD, mais générateur c'est le PGCD. □

Une seconde manière illustre le principe des tiroirs (pigeonhole principle). Dans la liste des $b-1$ résidus modulo b : a, $2a$, ..., $(b-1)a$, le résidu nul ne peut être absent sans contredire la coprimalité, pour la même raison éléments de la liste ne peuvent être identique.

6.5. **Euclide étendu.** Par algorithme d'Euclide étendu, il faut comprendre : un algorithme pour déterminer une solution à l'identité de Bâchet-Bézout.

La récursion fournit une solution rapide à écrire, comme le montre le code `BB()`.

```
1 ullong BB( ullong a, llong *u, ullong b, llong *v )
2 {
3 llong x, y;
4 if ( b == 0 ) { *u = 1; *v = 0; return; }
5 BB( b, & x, a % b, & y);
6 // b x + (a%b) y == d == b x + ( a - (a/b) b ) y
7     *u = y;
8     *v = x - (a/b) y;
9 }
```

Cette approche récursive facile à prouver et à implanter, possède une autre vertue, elle montre que, si b n'est pas nul, la paire (u, v) peut-être choisie avec :

$$au + bv = \text{PGCD}(a, b), \quad |u| < b, \quad |v| < a. \tag{28}$$

Le plus simple est de réécrire l'algorithme en évitant les instances nulles !

```
1 ullong BBNZ( ullong a, llong *u, ullong b, llong *v )
2 {
3 // precondition : a >= b > 0
4 llong x, y;
5 if ( a % b == 0 ) { *u = 0; *v = 1 ; return; }
6 BB( b, & x, a % b, & y);
7 // b x + (a%b) y == d == b x + ( a - (a/b) b ) y
8     *u = y;
```

```
9       *v = x - (a/b) y;
10  }
```

En effet, on part de $a = bq + r$, par induction, BBNZ(), montre que :

$$bx + ry = \text{PGCD}(a, b), \quad |x| \leq r, \quad |y| \leq b.$$

On a bien $|u| = |y| \leq b$.

$$\begin{aligned} |v| &= |x - qy| \\ &\leq |x| + q|y| \\ &\leq r + qy \\ &= a \end{aligned}$$

Une autre façon de procéder, plus efficace pour un humain, consiste à procéder par des réductions successives de triplets $(x, y, z) \in \mathbb{Z}^3$ du plan vectoriel

$$ax + by = z.$$

L'ensemble de ces triplets est stable par addition et multiplication par un entier. En partant des deux lignes $\vec{X} = (1, 0, a)$, $\vec{Y} = (0, 1, b)$, on construit une troisième ligne $\vec{Z} = \vec{X} - q\vec{Y}$, où q est le quotient de a par b. On itère jusquà trouver 0 sur la troisième composante. L'avant dernière ligne s'écrit (u, v, d) avec

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

Exercice 23. *Décrire les solutions de l'équation $au + bv = \text{PGCD}(a, b)$ à partir d'une solution particulière.*

Exercice 24. *Montrer que si a et b sont premiers entre eux, la solution de* EQ. *(28) est unique.*

Exercice 25. *Soit a et b deux entiers naturels premiers entre eux. On considère $S = \{am + bn \mid m, n \in \mathbb{N}\}$ le monoïde engendré par a et b. On définit le genre et le conducteur par :*

$$g(a, b) = \text{card}(\mathbb{N} \setminus S), \quad c(a, b) = 1 + \sup(\mathbb{N} \setminus S).$$

L'objectif de l'exercice est de montrer que $g(a, b) = (a - 1)(b - 1)/2$ et $c(a, b) = (a - 1)(b - 1)$. On note $A = \{0, 1, \ldots, a\}$, $B = \{0, 1, \ldots, b\}$ et $C = \{0, 1, \ldots, 2ab\}$.

(1) Montrer que $(a - 1)(b - 1) - 1$ n'est pas dans S.

(2) Montrer que si $n \geq (a - 1)(b - 1)$ alors il existe une unique paire d'entiers naturels (α, β) telle que :

$$n = \alpha a + \beta b, \quad \alpha < a.$$

On remarquera que les éléments $n - ub$ avec $u \in A$ sont distincts modulo a.

(3) *Déduire la valeur du conducteur $c(a, b)$.*
(4) *On considère l'application $f\colon B \times A \to C$ qui envoie la paire (u, v) sur $au + bv$. Montrer l'image de f est symétrique par rapport à ab. Montrer les éléments de C sont atteints au plus une fois sauf ab qui possède deux antécédants.*
(5) *Déduire de ce qui précède la valeur du genre $g(a, b)$.*

Pour aller plus loin, le lecteur est invité à se documenter sur le problème des pièces de monnaie de Fröbenius, la question 7382 de Sylvester et la conjecture de Wilf.

7. Enumération

7.1. **n-uplet.** Un n-uplet q-aire est représenté par un tableau indexé par $\{1, 2, \ldots, n\}$ dont les valeurs sont $\{1, 2, \ldots, n\}$.

```
1  void upletinc( int q, int n )
2  {
3      int i, x[n + 1];
4      for ( i = 0; i <= n; i++ )
5          x[i] = 1;
6      while ( x[n] == 1 ) {
7          traitement( x, n );
8          i = 0;
9          while ( x[i] == q )
10             x[i++] = 1;
11         x[i]++;
12     }
13 }
```

```
1  void upletrec( int q, int n, int x[] )
2  {
3      int i;
4      if ( n < 0 )
5          traitement( x, n );
6      else {
7          for ( i = 1; i <= q; i++ ) {
8              x[i] = i;
9              upletrec( q, n - 1, x );
10         }
11     }
```

```
12 }
```

7.2. code de Gray.

7.3. permutation.
Une permutation de $\{1, 2, \ldots, n\}$ est représenté par un tableau indexé par les entiers de 1 à n.

La fonction `void permut(int k)` permute les k éléments d'indices 1, 2, $\ldots$, k de la table globale x. Pour garantir la correction de l'algorithme on doit garantir l'invariance de x.

```
1
2 void permut( int k )
3 {
4 int i, t;
5 if ( k == 1 )
6           traitement( );
7 else
8     for( i = 1; i <= k; i++){
9           // isoler x[i]
10          t = x[k]; x[k]=x[i]; x[i]=t;
11          permut(k-1);
12          // invariant
13          t = x[k]; x[k]=x[i]; x[i]=t;
14 }
15 }
```

```
⇝ ./permut.exe 4 | fold -w20
:2341:3241:3421:4321
:2431:4231:4312:3412
:3142:1342:4132:1432
:2413:4213:4123:1423
:2143:1243:2314:3214
:3124:1324:2134:1234
```

7.4. lien dansant.

8. Arbre binaire

8.1. définition.

8.2. tableau.

8.3. pointeur.

8.4. **arbre de recherche.**

8.5. **equilibre.**

9. Bit-tracking

9.1. **bit-à-bit.** Le langage C permet de réaliser des opérations sur les bits des mots. Un exemple élégant et instructif de proprammation bit-à-bit est donné par le calcul du poids de Hamming d'un entier de la figure 5.

```
1 int wt( ullong x )
2 {
3 r = 0;
4 while ( x ) {
5           x&=(x-1);
6           r++;
7 }
8 return r;
9 }
```

```
1 int wt( ullong x )
2 {
3 for( r = 0; x ; r++ )
4           x&=(x-1);
5 return r;
6 }
```

Figure 5. Deux versions du calcul du poids de Hamming d'entier non signé, d'aprés [9] Peter Wegner (1960).

Exercice 26. *Montrer que l'algorithme* `wt` *calcule bien le poids de z. C'est l'exercice 2.09 de la seconde édition du The C Programming Language par Kernighan and Rithie.*

Il est souvent utile de retrouver v à partir de 2^v, ce calcul du logarithme à base deux peut se faire de plusieurs façons. La méthode proposée dans Fig. 6 illustre un mécanisme proche de ce que l'on pourrait rencontrer dans la progammation bitboard des moteurs du jeu des échecs.

Exercice 27. *Proposer différentes fonctions* `int log( ullong x)` *qui retourne le logarithme à base 2 d'un entier de 64 bit sachant que ce dernier de poids 1.*

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  typedef unsigned long long int ullong;
4  int count=0;
5  int magic[67] = { -1, 0, 1, 39, 2, 15, 40, 23, 3, 12,
6      16, 59, 41, 19, 24, 54, 4, 64, 13, 10, 17, 62, 60, 28,
7      42, 30, 20, 51, 25, 44, 55, 47, 5, 32, 65, 38, 14, 22,
8      11, 58, 18, 53, 63, 9, 61, 27, 29, 50, 43, 46, 31, 37,
9      21, 57, 52, 8, 26, 49, 45, 36, 56, 7, 48, 35
10 };
11
12 void traitement(ullong pi[])
13 {
14     int i = 0;
15     printf ("%3d : ", count++);
16     while (pi[i])
17         printf ("%3d", magic[pi[i++] % 67]);
18     putchar('\n');
19 }
```

FIGURE 6. Une utilisation possible d'un fait arithmétique pour calculer le logarithme à base 2 de 2^v : 2 est une racine primitive modulo 67.

Exercice 28.

```
1 int pb( ullong z ) {
2 int r = 0;
3 while ( z ) {
4         r+= (z % 2);
5         z = z / 2;
6 }
7 return r;
8 }
```

Soit m un entier positif. On note $I(\gamma)$ le nombre d'itérations de la boucle de la fonction `int pb(ullong z)` *pour une instance γ du paramètre z, la valeur moyenne sur l'ensemble des entiers de m bits :*

$$\tilde{I}(m) = \frac{1}{2^m} \sum_{\gamma=0}^{2^m-1} I(\gamma).$$

(1) Quel est l'ensemble des instances défavorables.

(2) Quel est ensemble des entiers γ tels que $I(\gamma) = 1$?.

(3) Décrire l'ensemble des entiers γ tels que $I(\gamma) = k$.

(4) Préciser le cardinal de cet ensemble.

(5) *Montrer par récurrence que* $\sum_{k=0}^{m-1} k2^k = 2^m(m-2)+2$.

(6) *Donner une formule du nombre moyen d'itération.*

(7) *Déterminer le nombre réel* α *tel que* $\tilde{I}(m) \sim \alpha m$.

Exercice 29. *Reprendre les questions de l'exercice précédent avec l'algorithme de la figure Fig. 5.*

Exercice 30. *Ecrire un algorithme récursif pour calculer le poids binaire d'un entier* z *de taille* $2m$ *basé sur la relation :*

$$\mathrm{wt}(z) = \mathrm{wt}(x) + \mathrm{wt}(y), \quad z = y2^m + x$$

Reprendre les questions de l'exercice précédent avec ce nouvel algorithme.

9.2. **retour sur traces.**

9.3. **permutation.** Un sous-ensemble de $\{0, 1, \ldots, n-1\}$ avec $n < 64$ peut être représenté par un mot binaire de 64 bits. La boucle qui suit extrait les éléments de la partie représentée par le mot binaire x :

```
1 while ( x ) {
2         t   = x;
3         x &= (x-1);
4         t ^= x;
5         i   = Log( t );
6 }
```

Cette approche est utilisée dans la procédure de backtracking Fig. 7 pour énumérer les permutations de $\{0, 1, \ldots, n-1\}$. Objet que l'on représente par un tableau de n valeurs distinctes.

9.4. **les huit reines.** Le problème des n dames consiste à trouver toutes les configurations de n dames placées sur échiquier $n \times n$ sans que deux dames ne soient sur une même rangée, colonne ou diagonale.

9.5. **cavalier d'Euler.**

10. Multiplication rapide

10.1. **produit naif.**

10.2. **Karatsuba.**

10.3. **interpolation.**

```
1
2 void permut(ullong pi[], ullong l, int n)
3 {
4     ullong x, t;
5     if (n == 0) {
6         traitement( pi );
7         return;
8     }
9     x = l;
10    n--;
11    while (x) {
12        t  = x;
13        x &= (x - 1);
14        t ^= x;
15        pi[n] = t;
16        permut(pi, l ^ t, n );
17    }
18 }
19
20 int main(int argc, char *argv[])
21 {
22    int n = atoi(argv[1]) ;
23    ullong *f = calloc(n + 1, sizeof(ullong));
24    permut( f, (1ULL << n) - 1, n);
25    return 0;
26 }
```

. permutation

FIGURE 7. Un algorithme de backtracking pour énumérer les permutations de $\{0, 1, \ldots, n-1\}$.

11. Graphe

11.1. **définition.** Un graphe orienté est un ensemble de sommets et d'arêtes orientées ou arcs reliant des sommets. Si X désigne l'ensemble des sommets du graphe, les arcs sont représentées par une partie U de $X \times X$. Un chemin de longeur n est une suite de sommets x_0, x_1, ..., x_n tels que :

$$\forall i, \quad 0 \leq i < n \Longrightarrow (x_i, x_{i+1}) \in U,$$

Dans ce cas on dit que x est connecté à y, et on note $x \rightsquigarrow y$. Un chemin est dit simple quand il ne passe jamais plus d'une fois par un

```
typedef unsigned int uint;
int n, count = 0;
void qbit(uint pos, uint left , uint right, int row)
{
    uint x, y;
    if (!row) {
        count++;
        return;
    }
    row--;
    left <<= 1;
    right >>= 1;
    x = pos & (~left) & (~right);
    while (x) {
        y = x;
        x &= (x - 1);
        y = x ^ y;
        qbit(pos & (~y), left | y, right | y, row);
    }
}
```

FIGURE 8. Un algorithme de backtracking pour énumérer les solutions du problème des n dames.

même sommet. Un circcuit est un chemin fermé i.e. $x_0 = x_n$. On définit des notions analogue dans le cas non orienté, chemin est remplacé par chaine, circuit par cycle. Une fonction $f : U \to R$ à valeurs réelles sur les arêtes se prolonge aux chemins par addition. On parle de fonction de coût et un circuit de coût strictement négatif est dit absorbant pour f.

11.2. **connexité.** A condition de poser $x \rightsquigarrow x$ pour tout sommet x, la realion $\rightsquigarrow$ est une relation d'équivalence : réflexive, symétrique et transitive. Les classes d'équivalence sont des composantes connexes. Dans les graphes orientés, en général, la relation n'est pas symétrique, on définit une connexion plus forte en disant que x est équivalent à y quand $x \rightsquigarrow y$ et $y \rightsquigarrow x$. Autrement dit, x et y sont sur un même circuit. Les classes pour cette relation sont des composantes fortement connexes.

11.3. **Quelques problèmes.** La théorie des graphes a été introduite par Léonard Euler. Dans un article publié en 1759, il pose la question

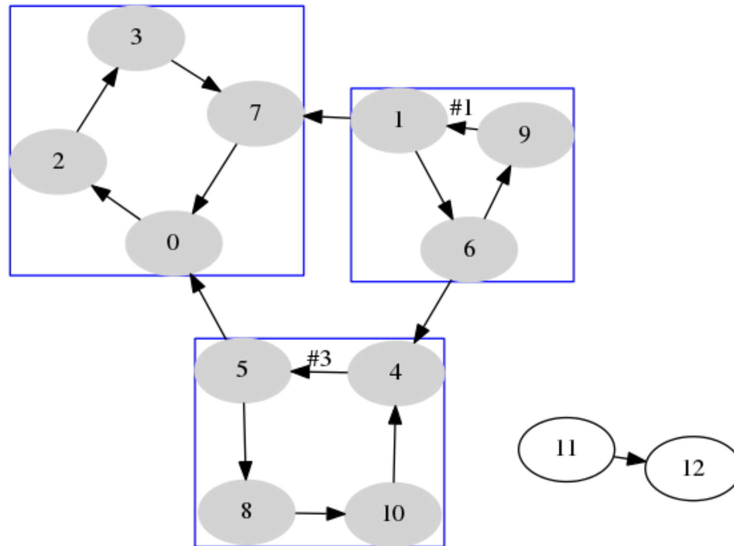


FIGURE 9. Un graphe orienté avec 5 composantes fortement connexes et 2 composantes connexes. Il a été réalisé avec la commande `neato` à partir de la source dg.dot

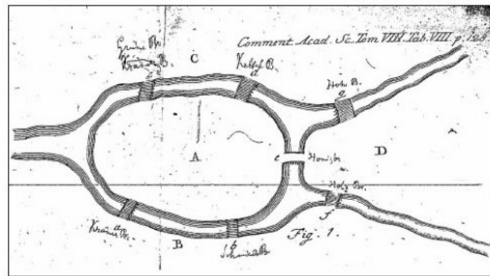


FIGURE 10. Pour des raisons de parité et de degré, il n'existe pas de promenade Eulérienne passant par les sept ponts de la ville de Koenigsberg.

de l'existence d'une promenade passant par les sept pont de la ville de Koenigsberg.

Il s'agit d'un problème algorithmiquement facile : l'existence d'un chemin Eulérien dépend du nombre de sommets pairs. L'algorithme de Dijkstra détermine le plus cours chemin entre deux sommets en un temps polynomial. Le problème du voyageur de commerce est difficile car il décide en particulier de l'existence d'un chemin Hamiltonien, un

```
globale index : entier
proc pp( x, U )
     v[x] := index++
     pour chaque (x,y) dans U
        si v[y] = 0  alors
    // arete (x,y)
             pp( y, U  )
        fsi
     fip


PP( X , U )
index := 0
initialiser v := {0}
pour x dans X
   si 0 == v[x] alors
      // arbre
      pp( x )
  fsi
fip
```

```
cfc-Tarjan( X, U )
globale index : entier
index := 0
pour x dans X
   si ! vu[x] alors
      parcours( x )
   fsi
fip
```

problème dit NP-complet parmi d'autres : colorabilité, isomorphisme de graphe etc. . . .

Exercice 31. *Montrer qu'un graphe connexe est Eulérien si et seulement si le nombre de sommets de degrés pairs est 0 ou 2.*

11.4. **Parcours de graphe.** L'algorithme `PP(X, U)` effectue un parcours en profondeur du graphe (X, U). La valeur $v(x)$ contient l'ordre de visite du sommet x. Les arêtes émises lors de ce parcours constituent une forêt.

11.5. **Algorithme de Tarjan.** L'algorithme de Tarjan (1972) s'appuie sur une pile et un parcours en profondeur pour déterminer les composantes fortement connexes d'un graphe (X, U).

```
parcours( x : sommet  )
index  := index + 1
rep[x] := num[x] := index
empiler( x )
pour x -> y faire
   si num[ y ] = 0
     alors
      parcours( y )
      rep  [ x ] := min( rep[ y ], rep[ x ] )
     sinon
      si num[y] < num[x] alors
         si y est sur la pile
            rep[x] := min ( rep[ x ], num[ y ] )
         fsi
      fsi
   fsi
fip
si rep[x] = num[x] alors
   cpt := 0
   tant que num[ pile ] >= num[ x ]
        tmp := depiler
        cpt := cpt + 1
   ftq
   emettre x, cpt
fsi
```

Proposition 5. *L'algorithme* `cfc-tarjan` *calcule les composante fortement connexes du graphe* (X, U).

Démonstration. La preuve est assez délicate. On s'appuie sur un parcours de graphe en profondeur. Au cours d'un tel parcours, on doit détecter les racines des composantes fortement connexes. Par définition, une racine est un sommet qui minimise l'ordre de visite sur une composante fortement connexe.

$$R(x) = V(x) \iff x \text{ est une racine.}$$

Il s'agit donc de prouver que l'algorithme calcule correctement $R(x)$ dans $rep[x]$. Les sommets sont empilés par numéro croissant, et par construction, si y est empilé avant x alors il existe un chemin de y vers x, un fait qui justifie la seconde mise-à-jour de $rep[x]$. Quand un parcours en profondeur termine, avec $R(x) = rep[x] = V(x)$ alors x est une racine et tous les sommets empilés au-dessus de x constituent la composante fortement connexe de x. □

11.6. Bellman-Ford.

12. Codages

12.1. ascii. Le code ASCII est un code assez ancien. Il normalise les caractères correspondant aux nombres de 7-bits. En effet, lors d'une transmission d'un caractère le 8ième bit est utilisé pour un contrôle de parité par le récepteur.

$$'C' \mapsto 67 = 0x43 = 01000011 \rightsquigarrow 11000011$$

```
ASCII(7)                    Linux Programmer's Manual

NAME
       ascii - ASCII character set encoded in octal,
       decimal, and hexadecimal
DESCRIPTION
  ASCII is the American Standard Code for Information
  Interchange.  It is a 7-bit code.  Many 8-bit codes
(such as ISO 8859-1, the Linux defaultcharacter  set)
contain  ASCII as their lower half.  The international
counterpart of ASCII is known as ISO 646.

The following table contains the 128 ASCII characters.

        Oct    Dec    Hex    Char
        000    0      00     NUL   \0
        101    65     41     A
        006    6      06     ACK   (acknowledge)
        007    7      07     BEL   \a (bell)
        012    10     0A     LF    \n (new line)
        015    13     0D     CR    \r (carriage ret)
```

Une fois n'est pas coutume, le système `MSDOS` et toute sa descendance, contrairement à `unix`, à conserver le saut de ligne ancestral.

```
$ echo 'ascii' | unix2dos  | hexdump -c
0000000    a    s    c    i    i   \r   \n
```

12.2. format. La troisième partie du match Viswanathan Anand, Magnus Carlsen, joué le 12 Novembre 2013 à Chenai, a commencé par les coups : 1. Nf3 d5, 2. g3 g6, 3. c4 dc4. Voilà un codage pour les humains. Il correspond au format algébrique : g1f3 d7d5 g2g3 g7g6 c2c4 d5c4 utilisé dans les moteur d'analyse comme `stockfish`. La suite de

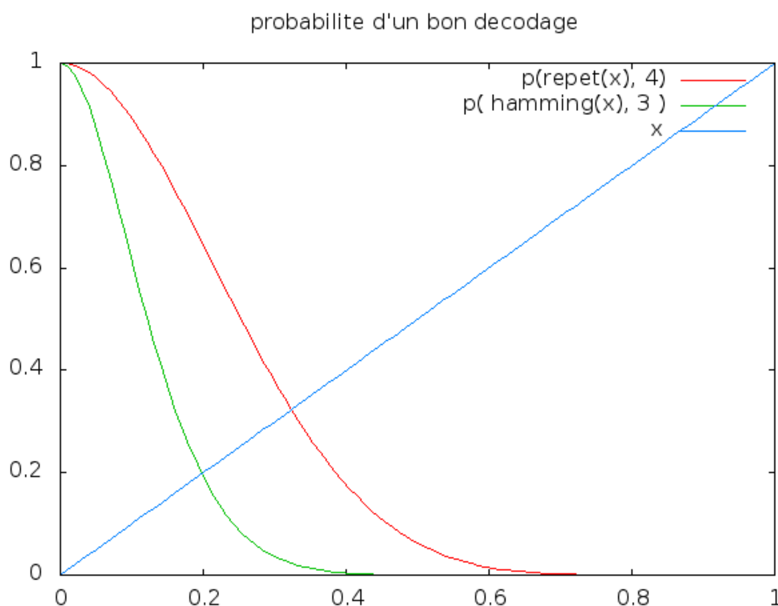


FIGURE 11. Probabilité d'un bon décodage

coups en base octale 60, 52, 36, 34, 61, 62, 66, 65, 21, 23, 34, 23, et le flux binaire correspondant :

011100110001110110110101010001010011011100010011

12.3. **canal bruité.** Dans un canal binaire bruité symétrique, les bits transmis diffèrent des bits reçus, la probabilité de transition est p. Pour traverser un tel canal de communication, il faut introduire de la redondance, par exemple en répétant trois fois chacun des bits. Le récepteur décode l'information par une décision majoritaire, la probabilité de recouvrer chacun des bits transmis est $q^3 + 3q^2p$, au final la probabilité de retrouver un symbole est $(q^3 + 3q^2p)^3$, ou un coup est $(q^3 + 3q^2p)^{12}$.

D'une manière générale, pour un système de codage cappable de corriger e erreur, la probabilité d'erreur sur un symboles aprés décodage est majorée par :

$$\sum_{k=e+1}^{n} \mathrm{C}_k^n q^{n-k} p^k, \quad p + q = 1.$$

12.4. **codage de Hamming.** L'encodage de Hamming consiste à introduire la redondance comme suit

$$(x, y, z, t) \mapsto (x, y, z, t, x + y + t, y + z + t, x + y + t)$$

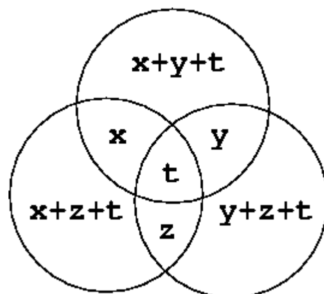


FIGURE 12. Codage de Hamming utilise 3 symboles de parités et 4 symboles d'information.

```
1 int wt( ullong z )
2 {
3 int r = 0;
4 while ( z ) {
5     z &= (z-1);
6     r++;
7 }
8 return r;
9 }
```

FIGURE 13. Une fonction bien utile

Le code de Hamming corrige moins bien que le code à répétition mais il permet un débit de de transmission est 1.71 plus important.

12.5. **distance.** On rappelle que le poids d'un n-uplet binaire est égal au nombre de 1 qui le compose.

Exercice 32. *Que retourne la fonction* `wt` *?*

L'ensemble $\{0,1\}^n$ est muni d'une métrique, la distance de Hamming :

$$\mathrm{d_H}(x,y) = \mathrm{wt}(y-x) = \sharp\{i \mid x_i \neq y_i\}$$

Exercice 33. *Montrer que la distance de Hamming est bien une distance au sens topologique.*

Un code de paramètre $[n,k,d]$ est une partie C de cardinal 2^k dans $\{0,1\}^n$ telle que

$$\forall x,y \in C, \quad x \neq y \Longrightarrow \mathrm{d_H}(x,y) \geq d.$$

Une telle partie permet d'encoder les messages de longueurs k dans des mots de longueurs n de sorte à pouvoir corriger jusqu'à $e = (d-1)/2$ erreurs.

Démonstration. Il suffit de vérifier que tout mot de code c entaché de moins de e erreurs est à distance au plus e d'un et un seul mot de code. □

L'opération de décodage qui consiste à retrouver c à partir d'un mot $c + x$ avec $\mathrm{wt}(x) \leq e$ est une opération algorithmiquement complexe en générale.

12.6. **linéarité.** Un code stable par addition est dit linéaire. Dans ce cas, il s'agit d'un groupe commutatif. Tout système générateur minimal est une base, il existe des matrices génératrices G permettant un encodage linéaire :

$$(x_m, \ldots, x_2, x_1) \mapsto (x_m, \ldots, x_2, x_1)G$$

Dans le cas d'un code linéaire, la distance minimale coincide avec le plus petit poids non nul. L'énumérateur de poids du code est la séquence

$$A_w(C) = \sharp\{c \in C \mid \mathrm{wt}(c) = w\}.$$

Exercice 34. *Prouver l'existence de système générateur.*

Exercice 35. *Ecrire* `int* enum( ullong *G, int k )` *pour calculer l'énumérateur de poids des mots du code engendré par G. Préciser le temps de calcul. Plusieurs solutions possibles : naïve, récursive et basée sur le code de Gray.*

Exercice 36. *Montrer que le code de Hamming est linéaire. Montrer que sa distance minimame est supérieur à 2. Montrer qu'il existe pas de $[7, 4, 4]$-code. Montrer que les boules de rayon 1 centrées sur les mots d'un $[7, 4, 3]$ forment une partition de $\{0, 1\}^7$.*

13. Transformée de Walsh

Dans cette section, m désigne un entier positif. On note X l'ensemble des m-pulets binaires qui est muni des opérations usuelles : disjonction exclusive bit à bit, conjonction bit à bit et produit scalaire.

$$(x \oplus y)_i = (x_i + y_i) \mod 2 \quad (x * y)_i = x_i y_i$$

et

$$x.y = \sum_{i=1}^{m} x_i y_i \pmod 2.$$

13.1. **fonction booléenne.** Une fonction booléenne f est une application de X dans $\{0,1\}$, elle est représentée par un tableau de longueur 2^m, la cellule d'index x contient $f(x)$.

Il s'agit d'une algèbre engendrée par les fonctions coordonnées X_i.

Exercice 37. *Exprimer δ_0 en fonction des X_i.*

Pour un élément $a \in B^m$ et un scalaire $b \in \{0,1\}$, on note $\mathrm{wt}(a,b)$ le poids de Hamming de l'application $\phi_{a,b}$.

Exercice 38. *Montrer que* $\mathrm{wt}(a,0) + \mathrm{wt}(a,1) = 2^m$.

Exercice 39. *Soit $a \neq 0$. Montrer qu'il existe un élément y tel que $a.y = 1$, puis établir une bijection entre $\{x \mid ax = 0\}$ et $\{x \mid ax = 1\}$.*

$$\mathrm{wt}(a,b) = \begin{cases} 0, & \text{a=0 et b=0;} \\ 2^m, & \text{a=0 et b=1;} \\ 2^{m-1} & \text{autre.} \end{cases}$$

13.2. **code de Reed-Müller.** Une application de degré 1 est une forme affine, elle s'écrit :

$$\phi_{a,b} \colon x \mapsto ax + b$$

avec $a \in \{0,1\}^m$, et $b \in \{0,1\}$.

L'ensemble des formes affines sur X forme un code de paramètres :

— longueur 2^m,
— dimension $m+1$,
— distance 2^{m-1}.

Pour la longueur c'est évident. Pour la dimension, il suffit de vérifier que $\phi_{a,b} = \phi_{a',b'}$ si et seulement si $a = a'$ et $b = b'$. La distance de Hamming entre $\phi_{a,b}$ et $\phi_{a',b'}$ n'est autre que le poids de Hamming de $\phi_{a+a'}b + b'$.

13.3. **transformée de Walsh.** Le coefficient de Walsh d'une application booléenne f en un point a de $\{0,1\}^m$ est par définition :

$$f^\star(a) = \sum_{x \in \{0,1\}^m} (-1)^{f(x)+a.x}$$

Il s'agit d'un cas particulier transformation de Fourier. Le coefficient de Fourier-Hadamad d'une application intégrale F en un point a de $\{0,1\}^m$ est par définition :

$$\hat{f}(a) = \sum_{x \in \{0,1\}^m} F(x)(-1)^{a.x}$$

```
1  void Fourier_Hadamard( int * f, int n )
2  {
3  int x,  tmp;
4     if  ( n == 1 ) return;
5     n = n / 2;
6     Fourier_Hadamard( f, n  );
7     Fourier_Hadamard( f + n , n );
8     for( x = 0, x < n  ;  x++ ){
9         tmp = f[ x ];
10        f[  x ]  = tmp + f[ x + n ];
11        f[  x + n ] = tmp - f[ x + n ];
12    }
13 }
```

FIGURE 14. Transformation de Fourier-Hadamard récursive

On peut clairement calculer tous les coefficients de Walsh d'une application intégrale en un temps quadratique. Une manipulation classique permet de ramener ce temps de calcul à $n \log_2 n$.

Convenons de noter $x = (y,t) \in \{0,1\}^{m-1} \times \{0,1\}$, $a = (\alpha,\gamma) \in \{0,1\}^{m-1} \times \{0,1\}$:

$$\begin{aligned}\hat{F}(a) &= \sum_{x\in\{0,1\}^m} F(x)(-1)^{+a.x}\\ &= \sum_{y\in\{0,1\}^{m-1}} F(y,0)(-1)^{\alpha.y} + (-1)^{\gamma} \sum_{y\in\{0,1\}^{m-1}} F(y,1)(-1)^{\alpha.y}\\ &= {F_0}^{\star}(\alpha) + (-1)^{\gamma}{F_1}^{\star}(\alpha)\end{aligned}$$

où $F_0(y) = F(y,0)$ et $F_1(y) = F(y,1)$.

Une analyse du butinage dans la fonction de `walsh` montre qu'il est possible de supprimer la récursion pour obtenir une version rapide `FWT`.

13.4. **décodage.** Soit un mot f du code de Reed-Müller, ϵ un mot erreur, le coefficicient de Walsh de la fonction booléenne $g = f + \epsilon$ vaut :

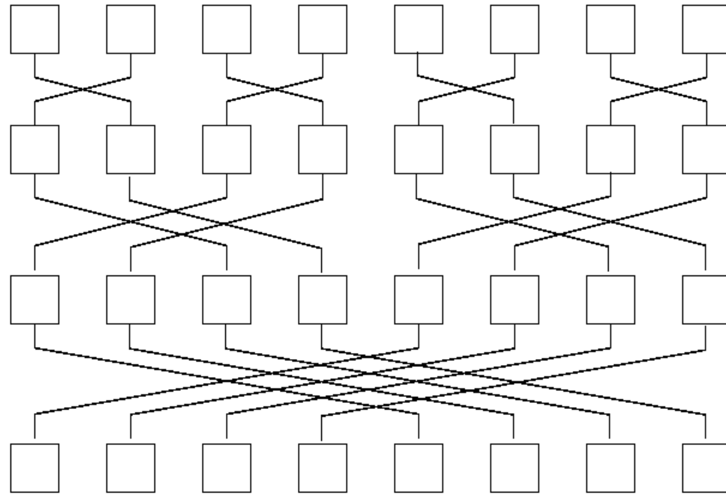


FIGURE 15. Le butinage des cellules d'une table par les papillons de la transformée de Walsh

```
1
2  void fFHT( int * f, int n )
3  { // fast  Fourier Hadamard Transform
4    int q, x, y, t, i, l = 2;
5    while ( l <= n ) {
6      for( q = 0; q < n/l; q++ ){
7          x = q * l;
8          y = x + l / 2;
9          for ( i = 0; i < l/2 ; i++ ) {
10               t = f[x];
11               f[x] = t + f[y];
12               f[y] = t - f[y];
13               x++; y++;
14         }
15         }
16     l = l * 2;
17   }
18 }
```

FIGURE 16. Transformation rapide de Fourier Hadamard

$$
\begin{aligned}
g^\star(a) &= \sum_{x\in\{0,1\}^m} (-1)^{f(x)+\epsilon(x)+a.x} \\
&= \sum_{\epsilon(x)=0} (-1)^{f(x)+a.x} - \sum_{\epsilon(x)=1} (-1)^{f(x)+a.x} \quad = \hat{f}(a) - 2\sum_{\epsilon(x)=1} (-1)^{f(x)+a.x}
\end{aligned}
$$

```
boole PREMIER( nombre z )
{
nombre x = 3;
si pair( z )
   retourner faux
fsi
tantque ( x * x <= z ) {
  si ( z mod x = 0 )
     retourner faux;
  fsi
  x := x + 2;
}
retourner vrai;
}
```

FIGURE 17. Un test de primalité naïf.

Soit w le poids de ϵ. Si $f^\star(a) = 0$ alors est de valeur absolue moindre que $2w$. Si $f^\star(a) = 2^m$ alors est de valeur absolue supérieure à $2^m - 2w$. On déduit de ces petits calculs une méthode de décodage valide dès que :

$$2^m - 2w > 2w \Longrightarrow w < 2^{m-2}$$

Exercice 40. *Laquelle ?*

Exercice 41. *La transformation de Fourier-Hadamard est involutive. Pour le vérifier, établir :*

$$\hat{\hat{F}}(a) = 2^m F(a)$$

14. Pseudoprimalité

Le n-ième nombre de Fermat est par définition qui $F_n = 2^{2^n} + 1$. Au milieu du XVIIe, Fermat remarque que $F_1 = 3$, $F_2 = 5$, $F_3 = 257$ et $F_4 = 65537$ sont premiers. Il conjecture que F_n est toujours premier. Plus tard (1732), Euler montre que $F_5 = 2^{32} + 1 = 4294967297$ n'est pas premier. La tâche n'est pas aisée sans machine à calculer ! L'ojectif de cette section est de faire apparaître le caractère composé des nombre F_n, pour 5, 6, 7 ?, 8 ? ?

Notons qu' il est complètement exclu de tester la divisibilité un grand nombre par un procédé trop élémentaire !

Exercice 42. *Préciser la forme du temps de calcul pour tester la primalité d'un entier de n bit au moyen de la fonction naïve* `premier`.

```
[pl@ou812] for n in {1..7} ; do echo "2^2^$n+1"
   |bc|/usr/bin/time --format "cpu=%U" factor;done
5: 5
cpu=0.00
17: 17
cpu=0.00
257: 257
cpu=0.00
65537: 65537
cpu=0.00
4294967297: 641 6700417
cpu=0.00
18446744073709551617: 274177 67280421310721
cpu=0.00
340282366920938463463374607431768211457:
        59649589127497217 5704689200685129054721
cpu=1544.20
```

14.1. **Calcul modulaire.** Soit n un entier. L'ensemble des résidus modulo n i.e. les restes possibles d'une division euclidienne par n quand il est muni des opérations modulaires

$$x \oplus y := (x+y) \mod n, \quad x \otimes y := (xy) \mod n,$$

forme un anneau noté $\mathbb{Z}/(n)$. Pour tout entier x, il existe un et un seul résidu r modulo n tel que :

$$x = qn + r,$$

nous le noterons $\bar{x}$.

Lemme 3. *Dans un anneau commutatif fini, un élément non nul est un diviseur de zéro ou bien un inversible*

Démonstration. On rappelle que x (non nul) est un inversible (resp :diviseur de zéro) s'il existe un élément (non nul) y tel que $xy = 1$ (resp : $xy = 0$). Pour le lemme, il suffit de considérer la multiplication par x. Un élément inversible est caractérisé par une application surjective, un diviseur de zéro par une application non injective. □

Proposition 6. *L'anneau $\mathbb{Z}/(n)$ est un corps si et seulement si n est nombre premier.*

Démonstration. Si n n'est pas premier alors n est le produit de deux résidus non nul r et s, et donc $r \otimes s = 0$, r et s sont des diviseurs de

zéro. Supposons n est premier. Tout résidu non nul r est premier avec n, par le théorème de Bâchet-Bézout, il existe deux entiers u et v telq que

$$ru + nv = 1, \quad r \otimes \bar{u} = 1.$$

ce qui montre que r est inversible, d'inverse $\bar{u}$. □

14.2. **Théorème de Fermat.**

Théorème 3. *Un entier n est premier si et seulement si pour tout résidu non nul y on a : $y^{n-1} = 1$ modulo n.*

Démonstration. La suffisance est triviale. Pour le reste, plusieurs preuves possibles, en voici une dans l'esprit de Fermat. Soit y un résidu non nul. Comme $x \mapsto y \otimes x$ est une bijection des résidus non-nuls

$$(n-1)! = \prod_{x=1}^{n-1} x = \prod_{x=1}^{n-1} (y \otimes x) = y^{n-1} \otimes (n-1)!$$

d'où $y^{n-1} = 1$. □

14.3. **ordre.** Soit x un élément inversible modulo n, dans la suite x^0, x^1, x^2,etc. . . des se répètent. Désignons par $j > i$, le plus petit entier tel que $x^j = x^i$. Comme x est inversible, nous endéduisons que $i = 0$. L'entier j est l'ordre multiplicatif de x, c'est le plus petit entier positif j vérifiant

$$x^j \equiv 1 \pmod{n}.$$

Exercice 43. *Ecrire une fonction* `ullong ordre(ullong x, ullong n)` *pour déterminer l'ordre de x modulo n.*

Un résultat de Gauss affirme que dans le cas où p est premier, il existe un élément d'ordre $p - 1$. Un tel élément s'appelle un élément primitif modulo p.

Exercice 44. *Ecrire une fonction* `ullong primitif( ullong p)` *pour déterminer un élément primitif modulo p.*

Soit γ un élément primitif modulo p. Tout résidu non nul x modulo p s'écrit d'une et une seule manière sous la forme d'une exponentielle de base γ

$$x = \gamma^k, \quad 0 \leq k < p - 1.$$

L'entier k s'appelle le logarithme discret de x pour la base γ. La détermination de k en fonction de γ est un problème sans solution algorithmique efficace à l'origine du protocole d'échange de clés de Deffie-Hellmann.

```
PRODUIT( nombre y, x, z )
{
indice i = 0;
nombre r = 0;
tantque ( i < N )
   si ( y[i] == 1  ) alors
      add( r, x );
      si ( x > z ) alors
        sub( x, z );
      fsi
   fsi

   fsi
   dbl( x );
   si ( x > z ) alors
      sub( x, z );
   fsi
ftq
retourner r
}
```

FIGURE 18. Calcul du produit yx modulo z par la technique de l'addition et du doublement.

14.4. **test de Fermat.** Le petit théorème de Fermat permet de se convaincre du caractère premier ou composé de la plus part des nombres.

Pour tester un entier n, on choisit un résidu positif x au hasard. On calcul $y := x^{n-1} \mod n$. Si y n'est pas 1 alors n est certainement composé. Sinon il est peut-être premier.

Un nombre déclaré premier 50 fois avec le test de Fermat est presque certainement premier ou bien c'est un nombre de Carmichael i.e. un nombre composé tel que

$$\forall x \in \mathbb{Z}/(n)^*, \quad x^{n-1} = 1 \mod n.$$

Contre toute attente, ce qui vient d'être dit est absolument élémentaire ! Les nombres de Carmichael sont rares mais en nombre infini. . .

14.5. **duplication et addition.** Nous utiliserons cet algorithme de complexité quadratique pour réaliser les multiplications modulaires.

14.6. **exponentiation modulaire.** Une version multiplicative de la technique d'addition/doublement donne un algorithme d'exponentiation.

```
EXPONENTIATION( nombre x, n, z )
{
indice i = 0;
nombre r = 1;
tantque ( i < N  )
   si ( n[i] == 1  )
      PRODUIT( r, x , z)
   fsi
   PRODUIT( x, x, z )
   i := i + 1
ftq
retourner r
}
```

FIGURE 19. Calcul du produit x^n modulo z par la technique du "square and multiply".

15. Transformée de Fourier

15.1. **racine de l'unité.** Soit n un entier ppositif. Dans le corps des nombres complexes, l'équation $X^n = 1$ possède exactement n solutions : les racines n-ième de l'unité

$$x_k = \exp(\frac{2ik\pi}{n}), \quad 0 \leq k < n,$$

qui forment un groupe d'ordre n engendré par la racine primitive standard $\zeta_n = x_1 = \exp(\frac{2ik\pi}{n})$.

Fait 1. *Pour n pair*

$$\zeta_n^2 = \zeta_{\frac{n}{2}}, \quad \textit{en particulier} \quad \zeta_n^{\frac{n}{2}} = -1.$$

Fait 2. *Soit x une racine n-ième,*

$$1 + x + x^2 + \ldots + x^{n-1} = \sum_{i=0}^{n-1} x^i = \begin{cases} n, & x = 1; \\ 0, & x \neq 1. \end{cases}$$

15.2. **transformée de Fourier.** La transformation de Fourier est une application de $\mathbb{C}[X]_n$ dans lui-même :

$$f(X) \mapsto \widehat{f}(X) = \sum_{i=0}^{n-1} f(\zeta_n^k) X^k$$

Fait 3. *L'inverse de la tranformée de Fourier est*

$$f(X) \mapsto f^*(X) = \sum_{i=0}^{n-1} f(\bar{\zeta}_n^k) X^k$$

On décompose le polynôme $f(X) = \sum_{i=0}^{n-1} a_i X^i$ en regroupant la partie paire et la partie impaire :

$$f(X) = g(X^2) + Xh(X^2), \quad g(X) = \sum_{2i<n} a_{2i} X^i \quad h(X) = \sum_{2i+1<n} a_{2i+1} X^i.$$

Fait 4. *On suppose n pair. Pour tout entier k,*

$$f(\zeta_n^k) = g(\zeta_{n/2}^k) + \zeta_n^k h(\zeta_{n/2}^k)$$

et

$$f(\zeta_n^{k+\frac{n}{2}}) = g(\zeta_{n/2}^k) - \zeta_n^k h(\zeta_{n/2}^k)$$

15.3. transformation rapide.

16. Liens dansants

La section fait écho à un article De D. Knuth disponible sur le la toile : *dancing links*. Il concerne les liens qui sont utilisés par le double chainage pour implanter au mieux certains algorithmes impratiquables.

16.1. couverture exacte.

16.2. algorithme X.

16.3. implantation.

Références

[1] T.H. Cormen, C. Leiserson, R. Rivest, and X. Cazin. *Introduction à l'algorithmique*. Science informatique. Dunod, 1994.

[2] Donald Knuth. Dancing links, 2000. http://arxiv.org/abs/cs/0011047.

[3] 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.

[4] 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.

[5] 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.

[6] Philippe Langevin. Une brève histoire des nombres, 2003. http://langevin.univ-tln.fr/notes/rsa/rsa.pdf.

[7] R. Sedgewick. Permutation generation methods, x. https://www.cs.princeton.edu/~rs/talks/perms.pdf.

[8] R. Sedgewick and J. M. Moreau. *Algorithmes en langage C*. I.I.A. Informatique intelligence artificielle. Dunod, 1991.

[9] Peter Wegner. A technique for counting ones in a binary computer. *Communications of the ACM*, 3(5), 1960.

[10] H.S. Wilf. *Algorithms and complexity*. Prentice-Hall international editions. Prentice-Hall, 1986.