# Observations Numériques autour du crible d'Eratostène

$\pi\lambda$

Octobre 2014
dernière compilation : 12 septembre 2015

**Résumé**

L'objectif de cette séance de travaux-pratiques est de mettre en évidence quelques points sur la correction ou le temps de calcul d'implantations d'algorithmes sur les nombres.

## 1 Arithmétique modulaire sur 64 bits

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 $x^k = 1 \pmod{n}$. Aprés compilation de la source `arithmos.c`, on obtient :

```
mesure> gcc -Wall arithmos.c -o arithmos.exe
mesure> x=2147483648
mesure> n=658812288653553079
mesure> ./arithmos.exe $x $n
ordre de 2147483648 modx 658812288653553079 > 255
mesure> bc <<< "( $x ^ 3 ) % $n"
1
```

1. Il y a un soucis, pourquoi ?
2. Faire un diagnostic !
3. Décrire sommairement une solution pour sortir de ce mauvais pas.

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

```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;
}
```

arithmos.c

## 2 Produit `bc`, `python`

Le script `algo.sh` génére des nombres au hasard de taille croissante, mesure le temps de calcul de leur produit avec `bc`. Les données numériques sont rangées dans un fichier temporaire qui est exploité avec la commande `gnuplot` pour deviner la forme du temps de calcul.

1. Adapter le script `algo.sh` à votre architecture pour analyser le temps de calcul du produit de deux grands entiers en `bc`.
2. Utiliser `ltrace -c` pour vérifier que `bc` n'utilise pas de bibliothèque de calcul.
3. Reprendre ces analyses pour la réduction modulaire.

*homework :*
- Analyser le temps de calcul du produit de deux grands entiers en `python`.
- Reprendre ces analyses pour la multiplication modulaire.

## 3 Crible d'Eratostène

L'implantation `crible.c` du crible d'Eratostène utilise la fonction `getopt` pour traiter les options passées en arguments sur la ligne de commande :

```
NAME

    crible.exe : calcule les valeurs de pi(x)

SYNOPSIS
     ./crible.exe [-p:hl:]
    -p nombre de lignes a  afficher
    -h aide
    -l logarithme de la taille n := 2**l
       ( defaut n=100 )

EXAMPLE

     ./crible.exe -p2
pi(50)=15
pi(99)=25
```

Dans le cours, nous avons établi l'existence de trois constantes positives $\alpha$, $\beta$, et $\gamma$ tel que le temps de calcul du criblage des nombres premiers inférieurs ou égaux à $n$ vaut

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

où

$$\pi(n) = \sharp\{x \mid x \leq n,\ x \text{ premier}\} \quad Ш(n) = \sum_{\substack{p \text{ premier} \\ p \leq n}} \frac{1}{p}.$$

```
function next ( )
{
local    r=$1
         for((k=0; k<step ; k++)); do
            r=$r$(( $RANDOM % 10 ))
         done
      echo $r
}
u=1
v=1
> /tmp/bc.dat
max=${1:-100}
step=100
for((  x=0, t=0; t<max; t++ , x+=step ))
do
      echo -n $x' ' >> /tmp/bc.dat
      (  time bc <<< "z=$u*$v" )  \
            |&  grep user  \
            |  tr  -cd '[0-9\n.]' >> /tmp/bc.dat
         u=$( next $u )
         v=$( next $v )
      echo -ne "\r$t"
done
echo image : bc.png
gnuplot <<PLOT
  set term png
  set output 'bc.png'
  set title    'bc:temps de calcul de la  multiplication'
  set xlabel  'taille  '
  plot  '/tmp/bc.dat' w l, x*sqrt(x)/1E07
  quit
PLOT
```

algo.sh

1. Calculer $\pi(x)$ pour $x < 1000$. Mettre en évidence la relation $\pi(x) = x/\log x$.
2. Mesurer le temps de calcul du crible.
3. Contrôler l'effet des options de compilation -Ox.
4. Mesurer l'impact des améliorations suggérées dans le cours.
5. Contrôler l'effet des options de compilation -Ox.
6. Comparer avec une implantation en `python`.

– Utiliser `gnuplot` pour comparer de $\text{Ш}(n)$ à $\log\log n$.
– Quelle constante $M$ semble vérifier $\text{Ш}(n) - \log\log n = M + \mathrm{o}(1)$ ?
– Retrouver ces résultats sur la toile.

# 4 Mémoire

Pour le crible, on peut économiser de la mémoire en travaillant sur des bits.
– Ecrire une fonction `void biffer(int n, uchar *p)` pour allumer le bit numéro $n$ à partir de l'adresse $p$.
– Ecrire une fonction `int flag(int n, uchar *p)` qui renvoie l'état du bit numéro $p$.
– Implanter un crible économe en mémoire.
– Tester l'efficacité de cette nouvelle approche.

# 5 Hypothèse de Riemann

> *Quand j'étais jeune, j'espérais démontrer l'hypothèse de Riemann. Quand je suis devenu un peu plus vieux, j'ai encore eu l'espoir de pouvoir lire et comprendre une démonstration de l'hypothèse de Riemann. Maintenant, je me contenterais bien d'apprendre qu'il en existe une démonstration.*
>
> André Weil

On note $\pi(x)$ le nombre de nombres premiers inférieurs ou égaux à $x$.
– On note $(p_n)_{0\leq n}$ la suite des nombres premiers. Faire une expérience numérique pour majorer la suite dérivée $p'_n := p_{n+1} - p_n$.
– Utiliser `gnuplot` pour proposer une fonction $e(x)$ susceptible de vérifier :

$$\pi(x) = x/\log x + e(x) + o(1).$$

– Les dessins sont parfois trompeurs !

# 6 progression arithmétique

Pour un entier $n$, $p_n$ désigne le $n$-ième nombre premier. On dit que les premiers forment une suite arithmétique de longueur $k$ de rang $n$ s'il existe un entier $r$ tel que

$$\forall i, \quad 0 \leq i < k, \qquad p_n + ir \text{ est premier}$$

Un résultat récent de B. Green et T. Tao affirme que pour tout entier $k$, il existe $k$ nombres premiers en progression arithmétique. Par exemple, les premiers

$$5, 11, 17, 23, 29$$

forment une suite arithmétique de longueur 5, de raison 6 et de rang 3.

– Ecrire un programme pour déterminer la la progression arithmétique de longueur $k$ de rang minimal.