# Rapport de TP - ALGO CCP

Rosique Lambert
Master 2 CSI, 2012-2013

18 mai 2011

Sous la direction de Jean-Marc COUVEIGNES

# Table des matières

# Introduction

Rapport de tous nos TP d'Algo CCP, cours du Master 2 CSI de Bordeaux. Les sujets (en général) sont disponibles sur le site du professeur.

Organisation : Le code est suivi d'un commentaire, sous forme de remarque, qui explique certains choix ou l'idée si elle n'est pas évidente.

(Remarque : quelques TP ont été rattrapés le weekend suivant, car j'étais malade ou en entretient. Je m'en excuse, ils sont intégrés au présent document et, évidemment, commentés)

# Chapitre 1

# TP 01

## 1.1 Temps d'Exécution

**Code 1.**
```
print("temps d'execution d'additions \n")
a = random(10^1000) ; b = random(10^1000)
gettime()
for (k=1, 1000000, c = a+b);
t = gettime()
print(t, "ms\n")
print();
```

**Remarque 1.** *Si a et b sont formés de 4 symboles, on obtient comme temps d'exécution (à gauche, le nombre d'opérations) :*

*10 000 -> 20ms ;*

*100 000 -> 144ms ;*

*1 000 000 -> 1.453ms ;*

*10 000 000 -> 14.236ms*

*Travaillons avec 1 000 000 d'additions. Si on augmente de un la taille des chiffres : 1.380ms DONC rajouter un chiffre = multiplier le temps par 10, comme prévu.*

*Remarque : on lance le programme pour un très grand nombre d'additions pour avoir plus de précision (on divise ensuite T par le nombre total d'opérations).*

**Code 2.**
```
print("temps d'execution de multiplications \n")
a = random(10^100000) ; b = random(10^100000)
gettime()
for (k=1, 1000, c = a*b)
t = gettime()
```

```
print(t, "ms\n")
```

**Remarque 2.** *Cette fois-ci, le temps d'exécution vaut 2.873ms*

*On constate que la multiplication est très largement plus coûteuse que l'addition ! (ce qui est parfaitement logique)*

**Code 3.**
```
print("temps d'execution de division \n")
gettime()
a = random(10^100000) ; b = random(10^100000)
for (k=1, 1000000, divrem(a,b))
t = gettime()
print(t, "ms\n")
```

**Remarque 3.** *TEMPS = 4.637ms OU 16.265 (on constate que le temps varie beaucoup suivant les paramètres car certaines opérations peuvent être évitées. Toutefois, en réalisant l'expérience plusieurs fois, on remarque que ca suit le modèle de la multiplication (cela vient de l'algorithme d'Euclide utilisé pour diviser, c'est pour ca que l'on va très vite certaines fois, si Euclide est "court")*

**Code 4.**
```
x=[1000,10000,100000,1000000]
y=[0,0,0,0]
print("Trac\'e de la courbe d'addition")
for(i = 1, length(x), a = random(10^(x[i])) ;
 b = random(10^(x[i])) ; gettime() ;
 for (k=1,100000, c=a+b) ; y[i] = gettime() ; print(y[i]));
plothraw(x,y,1)
```

/* La courbe est légèrement incurvée */ => temps de calcul linéaire en les variables (leur taille)

**Code 5.**
```
x=[100,250,400,600,750,900,1000]
y=[0,0,0,0,0,0,0]
print("Trac\'e de la courbe de multiplication")
for(i = 1, length(x), a = random(10^(x[i])) ;
 b = random(10^(x[i])) ; gettime() ;
 for (k=1,1000000, c=a*b) ; y[i] = gettime() ; print(y[i]));
plothraw(x,y,1)
```

/* La courbe est fortement incurvée ! */ => temps de calcul exponentiel en les variables (leur taille)

**Code 6.**
```
x=[100000,150000,200000, 250000, 300000,350000,400000]
y=[0,0,0,0,0,0,0]
print("Trac\'e de la courbe de division")
for(i = 1, length(x), a = random(10^(x[i])) ;
 b = random(10^(x[i])) ; gettime() ;
 for (k=1,100000, divrem(a,b)) ; y[i] = gettime() ; print(y[i]));
plothraw(x,y,1)
```

/* Pareil que pour la multiplication */

## 1.2 Méthodes de Tri

**Code 7.**
```
rand_list(n,i)={
x = List([]);
for (k=1, n, x=concat(x,[random(10^i)]));
print(x);
}
```

On génère aléatoirement une liste de taille n, de nombres entre 0 et $10^i$

**Code 8.**
```
get_min(l)={
        my (ret, i);
        ret = l[1];
        i = 1;
        for (k=1, length(l), if (ret > l[k], ret = l[k]; i = k));
        return([ret,i]);}

naive_sort(n,i)={
        list = List(rand_list(n,i));
        tri = [];
        for (k=1, n, tri = concat(tri,[get_min(list)[1]]);
          list = listpop(list,get_min(list)[2]));
        return(tri);
}
```

**Remarque 4.**
```
get_min : elle determine l'element minimal d'une liste

naive_sort
```

*: fonction naive de tri, très couteuse en calculs car on parcours de très nombreuses fois la liste. En pratique, on a pu constater un temps d'exécution élevé, par rapport au tri par fusion.*

# Chapitre 2

# TP 02

## 2.1 Produit de polynômes

**Code 9.**
```
prodpol(u,v)={
        my (i, j);
        w = [];
        for (i = 1, length(u), w = concat(w,0);
                for (j = 1, length(v),
                        if (i==1 && j!=length(v), w = concat(w,0));
                        w[i+j-1] = w[i+j-1] + u[i]*v[j]));
        return(w);
}
```

On fait un calcul brutal, extrèmement couteux

## 2.2 Karatsuba

**Code 10.**
```
segmpol(p)={
        a = [];
        b = [];
        n = length(p)/2;
        for ( k = 1 , n,
                a = concat(a,p[k]);
                b = concat(b,p[k+n]);
                );
        return([a,b]);
}
```

```
addlist(a,b)={
        c = [];
        for (i=1, length(a), c = concat(c,a[i]+b[i]));
        return(c);
}
```

```
karat(p,q)={
        my n = length(p)+length(q);
        if (n==2, return(p[1]*q[1]));
        my a = p ;
        my b = q ;
        L1 = karat(segmpol(a)[1],segmpol(b)[1]);
        L2 = karat(segmpol(a)[2],segmpol(b)[2]);
        L3 = karat(addlist(segmpol(a)[1],segmpol(a)[2]),
          addlist(segmpol(b)[1],segmpol(b)[2]));
return([L1,L3,L2]);
}
```

**Remarque 5.** *NOTE : le programme ne marche pas car il faut diviser les moitiés à chaque appel et pas diviser p et q !*

*Karatsuba est une méthode de calcul des produits des polynômes qui est astucieuse car elle utilise quelques produits intermédiaires dont elle se sert plusieurs fois d'affilé, sans avoir à refaire certains calculs du coup. On a plus d'additions mais moins de multiplications donc on y gagne.*

# Chapitre 3

# TP 03

## 3.1 Résolution d'équations

**Code 11.** *M=[2,3,5,7;11,13,17,29];*
*print(matkerint(M));*
*print();*

**Remarque 6.** *Solution :*
*[2 6]*
*[-3 4]*
*[1 5]*
*[0 -7]*
*D'ou on a un espace vectoriel de dim 2 engendré par (2,-3,1,0) et (6,4,5,-7). En effet, la fonction matkerint(M) calcule pour nous les vecteurs propres, et ils sont indépendants (visible directement à la dernière ligne).*
*Remarquons qu'on a solvé le système*
$2x + 3y + 5z + 7t = 0$
*et*
$11x + 13y + 17z + 29t$

**Code 12.** *print(mathnf(M~),1);*

**Remarque 7.** *H = [-19, -16]*
*[ -4, -3]*
*[ 26, 23]*
*[ 0, 1]*
*U = [29, 25]*
*[ -7, -6]*
*On a donc M * U = H*

## 3.2 Nombres premiers

**Code 13.** *P=[]*
```
for(i=10^50,10^50+100000,premier=True;

forprime(p=1,1000,if(i%p==0,premier=False;break));

if(premier&&isprime(i),P=concat(P,i)))
```

**Remarque 8.** *on se sert de la fonction intégrée à Pari-GP pour trouver les nombres premiers entre* $10^{500}$ *et* $10^{500} + 100000$. *Toutefois, cette fonction ne marche pas très bien pour les grands nombres ! donc il vaut mieux recourir à des algorithmes (voir après)*

**Code 14.**
```
crible(A,B,pmax)={
        my(L,n);
        n=0;
        L = vector(B-A+1);
    forprime(p=2,100,  r=(-A)%p+A;  while(r<=B,L[r-A+1]=1;r=r+p));
    for(i=1,length(L),if(L[i]==0,
        if(isprime(i+A-1),/*print(i+A-1)*/;n=n+1;)));
        print(n);
}

crible_correction(A,B,pmax)= {
        my(crible,t,p,n);
        n=0;
        crible = vector(B-A+1);
        forprime(p=2,pmax,
                t=(-A)%p+A;
                while(t<=B,  crible[t-A+1]=1;t=t+p);
     );

    for(t=A,B,if(crible[t-A+1]==0,
        if(ispseudoprime(t),/*print(t);*/n=n+1;);));
        print(n);
}
```

**Remarque 9.** *crible est la version développée en TP*

*crible correction est la version donnée par le prof.*

*Résultats :*

*On trouve 81 nombres premiers (avec les 2 algorithmes) sur un intervalle de 100000, soit un rapport de 81/100000*

*or si on s'en réfère à la répartition des nombres premiers, il devrait y en avoir 1/(500*log(10)) soit environ 87/100000.*

*On constate donc que la loi est bien suivie, du moment qu'on est loin des "petits nombres" !*

# Chapitre 4

# TP 04

## 4.1 Polynômes irréductibles

Trouver un polynôme irréductible de degré 3 dans F2

```
f = (1*x^3 + 0*x^2 + 1*x + 1)*(Mod(1,2))
polisirreducible(f)
```

le pol (vu par la bijection vers $0, 1^4$) (1,1,1,1) ne l'est pas (1,0,1,1) et (1,1,0,1) le sont

```
X = Mod(x,f)
```

on cherche un générateur de F2[X]/f

```
for (i=0 , 7 , print("X^",i,"=",lift(lift(X^i))))
```

grace à cette commande, on voit que (0,0,1,0) est d'ordre 7. Il génère donc tout le groupe car il y a 8 éléments dans notre groupe $(2^3 - 1)$ X génère le groupe $F_2[X]/(x^3 + x + 1)$

Pour évaluer un polynôme : $subst(P, x, 1)$

On construit la table d'exponentielle :

```
l = []
for (i=0 , 6 , l = concat(l,[[i,lift(lift(X^i))]]))
print("Exp du gen ", lift(X), " dans F_2[X]/(", lift(f),")")
for (i = 0, 6, print(l[i+1][1], " donne " , l[i+1][2]))
```

Fonction logarithme discret (inverse de notre exponentielle) g : générateur, A anneau, I idéal, k : entier tel que $k = g^n$ (on doit renvoyer n)

```
logd(A,g,P,k)={
h = Mod(g,(P*Mod(1,A)));
for(i = 0, 2^poldegree(P)-1, if(h^i==k, return(i)));
```

```
print("non-trouv\'e")
}
```

On construit la table du logarithme

```
print("Log du gen ", lift(X), " dans F_2[X]/(", lift(f),")")
for (i=0,6, print( lift(lift(X^i)), " = " , logd(2,x,x^3+x+1,x^i)))
```

Fonction pour déterminer si un élément est générateur. Pour ca, on peut remarquer qu'il faut regarder les puissances premières (ou leurs produits) (qui divisent le cardinal du groupe -1) de notre élément. Si on ne tombe pas sur 1, alors c'est réglé P est à nouveau le quotient et X le polynôme à tester

```
isprimitive(X,P)={
        n = 2^poldegree(P)-1;
        l = divisors(n);
        if( length(l) == 2 , return(1));
        for(k=2, length(l), if(X^(n/l[k])==1, return(0)));
        if(X^n==1, return(1),return(0));
}
```

# Chapitre 5

# TP 05

## 5.1 Exponentiation rapide

**Code 15.** *exp_rapide(x,n,m)={*

```
        my(i,ret,L);
        ret=1;
        n = n%(m-1);
        L=binary(n);

        for(i=1,length(L)-1,if(L[i]==1,ret=ret*x%m);ret=ret*ret%m);
        if(L[length(L)]==1,ret=ret*x%m);

        return(ret);

};

print("2^123456789876543 21 mod 101 = ",
        exp_rapide(2,123456789876543 21,101));
print("");
```

**Remarque 10.** *Algorithme d'exponentiation rapide. Grosso modo, on divise notre exposant soit par 2 soit on fait -1 suivant s'il est pair ou non, car il est "aisé" d'élever au carré et cela va très vite (moins de calculs que la méthode brutale).*

## 5.2 Nombre de Carmichael

**Code 16.**
```
carmichael(N)={

        ret=true;

        for(k=1,N-1,if(gcd(N,k)==1,
                if(Mod(k,N)^(N-1)!=1,ret=false)));

        if(ret==true,print(N,"oui"),
        print(N,"non pas Car."));
        return(ret);
};

carmichael(561);
carmichael(560);
carmichael(2);
carmichael(101);
print("");
```

**Remarque 11.** *Algorithme qui détermine si un nombre est de "Carmichael". Ces nombres sont connus car ils ressemblent à des nombres premiers. Ici, il n'y a qu'un calcul simple à faire pour le vérifier à l'aide des gcd.*

## 5.3 Inversibles d'un groupe

**Code 17.**
```
L=[];
for(i=0,34,if(i%5!=0 && i%7!=0, L=concat(L,i)));
print("Inversibles mod 35 = ",L);
print(length(L)," \'el\'ements dans L et phi(35)= ",(5-1)*(7-1));
print("");
```

**Remarque 12.** *On cherche les inversibles de Z/35Z. Ce sont les nombres premiers avec 35 = 5x7 donc avec 5 et 7. Effectivement, le programme renvoit la bonne liste.*

*On calcule aussi l'inverse d'un élément à l'aide de l'algorithme d'Euclide étendu :*
*12=7*1+5 7=5*1+2 5=2*2+1*
*donc 1=3*12-5*7*
*donc -5*7=1 mod 12*
*L'inverse de 7 mod 12 est -5=7.*

## 5.4 Générateur

**Code 18.**

```
est_generateur(g,N)={
        my(i,L,n);

        n = N-1;
        L = divisors(n);

        if(length(L)==2, return(1));

        for(i=2,length(L), if(Mod(g,N)^(n/L[i])==1,return(false)));
        if(Mod(g,N)^n==1,return(true),return(false));

};

   g=2; N=11;
if(est_generateur(g,N)==true, print(g," gen de (Z/",N,"Z)*"),
        print(g," non gen."));
print("");
```

## 5.5 Tables

/* Ecriture de la table d'exponentielle L'application exp va de Z/10Z dans (Z/11Z)* */

**Code 19.**

```
print("Table d'exponentielle de Z/",N-1,"Z dans (Z/",N,"Z)*")
for(i=0,N-2,print(i," -> ",lift(Mod(g,N)^i)));
print("");
```

/* Ecriture de la table du logarithme L'application exp va de (Z/11Z)* dans (Z/11Z)* */

**Code 20.**

```
\\ Fonction qui retourne n tel que y=g^n mod N
logd(y,g,N)={
        my(i);
        for(i=0,N-1, if(Mod(g,N)^i==Mod(y,N), return(i)));
```

```
};

print("Table de logarithme de (Z/",N,"Z)* dans Z/",N-1,"Z")
for(i=1,N-1,print(i," -> ",logd(i,g,N)));
print("");
```

## 5.6 Algorithme de Miller-Rabin

**Code 21.** *MyMillerRabin(N,x)={*

```
        if(gcd(N,x)!=1,print(x," n'est pas premier \'a ",N);return());

        my(i,m,k);
        m=N-1;
        k=0;
        while(m%2==0, m=m/2; k=k+1);

        \\ Ici, N-1 = 2^k * m

        if(Mod(x,N)^m==1,print(N," PEUT etre premier");return(true));
        for(i=0,k-1,if(Mod(x,N)^(m*2^i)==-1,
                print(N," PEUT etre premier", i," ",m);return(true)));

        print(N," n'est pas premier");
        return(false);
};

N=561;
cpt_faux_temoins = 0;
F=[];
for(k=1,N-1,if(gcd(N,k)==1,
        if(MyMillerRabin(N,k)==true && isprime(N)==0,
          print(k, " = faux temoin");
          cpt_faux_temoins+=1;F=concat(F,k))));
print("");

print("Il y a ",cpt_faux_temoins," faux t\'emoins pour ",N);
print("Ils sont : ",F);
print("");
```

**Remarque 13.** *Cet algorithme est un test de primalité très puissant et répandu, car efficace*

# Chapitre 6

# TP 06

## 6.1 Loi de Réciprocité

**Code 22.**
```
legendre(x,p) = {
        my(ret);
        ret = centerlift(Mod(x,p)^((p-1)/2));
        return(ret);
};

p = nextprime(random(10^100));
q = nextprime(random(10^100));

calcul_reciprocite(p,q) = {
        if(p%4==3 && q%4==3, return(-1), return(1));
        \\ return( (-1)^(((p-1)/2)*((q-1)/2)) );
};
```

**Remarque 14.** *On commence par implémenter le calcul du symbole de Legendre, ce qui est facile avec la formule habituelle*

*Ensuite, on s'assure que la loi de réciprocité quadratique est bien vérifiée*

**Code 23.**
```
{
if(p==q || p==2 || q==2, print("p=q, ou p=2 ou q=2"),
        print("Legendre(p,q) = ", legendre(p,q));
        print("Legendre(q,p) = ", legendre(q,p));
        print("Calcul reciprocite = ", calcul_reciprocite(p,q));
);
```

```
}

print();
```

## 6.2 Restes Chinois

**Code 24.**
```
f(n1,n2,x) = {
        return([Mod(x,n1),Mod(x,n2)]);
};

reciproque_f(n1,n2,x1,x2) = {
        return( Mod(x1*(lift(Mod(n2,n1)^(-1)))*n2
         + x2*(lift(Mod(n1,n2)^(-1)))*n1,n1*n2) );
};

n1=31;
n2=43;
x=876;

ret = f(n1,n2,x);
x1 = lift(ret[1]);
x2 = lift(ret[2]);

print(Mod(x,n1*n2)," ----> ",ret);
print(ret," ----> ",reciproque_f(n1,n2,x1,x2));

print();
```

**Remarque 15.** *Cette fois-ci, on a implémenté le cas général de la fonction des restes chinois (la bijection de Z/n1n2Z vers Z/n1Z x Z/n2Z), ainsi que sa réciproque. L'intérêt est de pouvoir résoudre un système de congruences.*

## 6.3 Racine Carré

Exercice non rattrappé. L'idée était d'implémenter la racine carré modulaire

# Chapitre 7

# TP 07

Soit p=31 (cf feuille du log discret)

## 7.1 Sous-groupe d'ordre 2

**Code 25.**
```
p=31;
print("g\'en\'erateurs␣:");
for(x=1,p-1,ordre=1; while(Mod(x,p)^ordre!=1, ordre=ordre+1);
        if(ordre==2, print(x)););
print();
```

**Remarque 16.** *On montre que notre groupe (Z/31Z)* admet un sous-groupe d'ordre 2 et qu'il n'y en a pas d'autres. L'algorithme ne revoit que le chiffre 30, donc le seul groupe d'ordre 2 est 1,30 = 1,-1*

## 7.2 Sous-groupe d'ordre 3

Vérifier que 3 est un générateur Pour cela on vérifie que $3_i^k! = 1$ avec $k = (p-1)/p_i$ pour tout $p_i$ diviseur premier de p-1

**Code 26.**
```
g=3;
div = divisors(p-1);
gen = true;
for(i=1,length(div),
if(Mod(g,p)^((p-1)/div[i])==1, gen=false; break;));
if(gen==true, print(g, "␣gen."),
 print(g, "␣non␣gen."));
```

```
for(x=1,p-1,ordre=znorder(Mod(x,p)); if(ordre==3, print(x)));
print();
```

**Remarque 17.** *Notre programme, qui cherche à calculer le sous-groupe d'ordre 3 après avoir vérifié qu'il est bien d'ordre 3, nous renvoit comme valeurs 1, 5 et 25. Ceci est évidemment le résultat escompté, puisque* $5 \times 25 = 125 = 4 \times 31 + 1$

## 7.3 Sous-groupe d'ordre 5

**Code 27.**
```
print(Puiss 6e pour elt d'ordre 5");
L = [];
for(x=1,p-1, y = lift(Mod(x,p)^((p-1)/5)); L=concat(L,y) );
L=Set(L);
print(L);
```

# Chapitre 8

# TP 08

## 8.1 Baby step

**Code 28.** *babygiant(p, g, h) = {*

```
my(r,ga,G,B,L,k,valeur,ret,iG,iB);
r = floor(sqrt(p-1));
ga = Mod(g,p)^r;
G = [];
B = [];
for(k=0, r-1, G=concat(G, lift(Mod(ga,p)^k));
   B=concat(B, lift(Mod(h,p)*Mod(g,p)^(-k))) );
L = concat(G,B);

L = vecsort(L);

k = 1;
while(L[k]!=L[k+1] && k<r, k=k+1;);
if(k==r, return(-1););

valeur = L[k];
iG = 0;
iB = 0;
k=1;



while((iG==0 || iB==0) && k<r+1,
```

```
    if(G[k]==valeur, iG=k-1); if(B[k]==valeur, iB=k-1); k=k+1);
    if(iG==0 || iB==0, return(-2));

    ret = iG*r + iB;

    return(ret);
};
```

**Remarque 18.** *Algorithme de Baby step, vu en cours et implémenté à l'aide du poly (on construit le tableau des puissances, et on cherche un pgcd différent de 1. Dès lors qu'on l'a trouvé, on arrete d'avancer dans l'algorithme et il n'y a plus qu'à faire le calcul pour avoir notre valeur cherchée*

## 8.2 Algo de Pollard

**Code 29.**
```
p = 171962010545840643348334056831754301958457563589574256043877110505832165523856261308397965147955578800999455782202456522693290629520826275682227566369411 1;
q=nextprime(10^160);
pollard(n,y) = {

    my(x,k, pgcd);
    x=Mod(y,n);    \\x = Mod(random(n-3)+2,n);
    k = 2;
    pgcd = gcd(lift(x)-1,n);
    while(pgcd==1, x = x^k; k = k+1; pgcd = gcd(lift(x)-1,n));
    return([pgcd, n/pgcd]);
};
```

**Remarque 19.** *la version "random" était celle implémentée au début. On avait pris dans ces valeurs pour éviter les valeurs -1, 0, 1 car le pgcd aurait toujours fait 1 (choix de x inintéressant). Toutefois, il peut aussi y avoir des problèmes si notre x n'est pas un générateur du groupe considéré... du coup, avec cette version il faudrait rajouter un test.*

*Remarque : cet exercice appliqué avec* $p = nextprime(10^10)$ *et* $q = nextprime(10^15)$ *marchait instantanément, tandis qu'avec n'importe quelle autre puissance de 10 "peu éloignée", cela ne donnait rien (en fait, on peut voir que notre nombre doit être bien friable pour éviter que ses facteurs premiers soient trop grands ! (et donc que l'algorithme soit très lent) => si ce n'est pas le cas, cet algorithme ne peut pas s'appliquer (on ne peut pas l'utiliser pour cracker RSA par exemple à cause de ca))*

# Chapitre 9

# TP 09

Absent et non-rattrapé

# Chapitre 10

# TP 10

## 10.1 Friabilité

**Code 30.** *{*
```
friable(n,x)=
local(p);
p=n;
forprime(q=2, x , while( p%q == 0, p = p/q));
return((p==1));
}
```

**Remarque 20.** *On commence par écrire un algorithme qui nous dis si un nombre est friable (ie si les facteurs premiers de n sont inférieurs à x)*

# Conclusion

Tout au long de ce semestre, on a pu programmer de très nombreux algorithmes cruciaux en Algèbre à l'aide de Pari-GP. De plus, on a pu implémenter diverses structures (polynômes, groupes, matrices, etc..) très utiles pour la suite !

Merci.