Arithmétique et cryptographie

1. Première séance (TP1)

1.1. Test de primalité

L'algorithme est le suivant (test naïf) :

```
entrée n
si n mod 2 = 0 et n ≠ 2
   n n'est pas premier
sinon pour (d = 3 ; d² < n ; d = d + 2)
      si n mod d = 0
         n n'est pas premier
      finsi
   finpour
   n est premier
finsi
```

Une fois codé en C, cela donne :

```
bool estPremier(int n){
        bool result=true;
        if (n%2==0&&n!=2){
                result=false;
        }
        else
        {
                for(int i=3;i*i<=n;i=i+2){
                        if (n%i==0){
                                result=false;
                        }
                }
        }
        return result;
}
```

La fonction retourne un résultat de type booléen : VRAI si le nombre n est premier, FAUX si n n'est pas premier.

1.2. Décomposition en facteurs premiers

Elle est effectuée de la manière suivante :

```
void decompose(int n){
        int j,k=0,ntemp=0;
        int fin=n;
```

```
        if (estPremier(n)){
                facteur[0][0]=n;
                facteur[0][1]=1;
        }else{
                for (int i=2;i<fin;i=i+1){
                        if (estPremier(i)) {
                                if (n%i==0){
                                        ntemp=n;
                                        j=0;
                                        while(ntemp%i==0){
                                                j=j+1;
                                                ntemp=(int)ntemp/i;
                                        }
                                        facteur[k][0]=i;
                                        facteur[k][1]=j;
                                        k=k+1;
                                        fin=fin/i+1;
                                }
                        }
                }
        }
}
```

Afin de regrouper les facteurs par puissance, on utilise une structure de type tableau à deux dimensions, qui fait correspondre à chaque facteur premier trouvé sa puissance (incrémentée chaque fois que le même facteur est trouvé). Ainsi on obtient un résultat de type $z = x^a * y^b \ldots$

Remarque : cette fonction fait appel à la précédente fonction « estPremier ».

1.3. Algorithme d'Euclide

L'algorithme d'Euclide se base sur le théorème suivant :

$$\forall n \in Z \text{ on a } pgcd(a,b) = pgcd(a,b + an)$$

Il consiste à effectuer la suite de divisions suivantes :

$r_0 = q_1 * r_1 + r_2 \qquad 0 \leq r_2 < r_1$
$r_1 = q_2 * r_2 + r_3 \qquad 0 \leq r_3 < r_2$
…
$r_{m-2} = q_{m-1} * r_{m-1} + r_m$
$r_{m-1} = q_m * r_m + 0$

$pgcd(r_0,r_1) = pgcd(r_1,r_0 - q_1r_1) = pgcd(r_1,r_2) = pgcd(r_2,r_1 - q_2r_2) = pgcd(r_2,r_3) = \ldots = pgcd(r_{m-1},r_m) = r_m$

Ainsi le pgcd de deux nombres r_0 et r_1 en suivant l'algorithme d'Euclide est le dernier reste non nul de la division.

On obtient le résultat grâce à la fonction suivante :

```
int pgcd(int a, int b){
	int r0=a;
	int r1=b;
	int r2=r0%r1;
	int q=(int)r0/r1;
	while (r2!=0){
		r0=r1;
		r1=r2;
		q=(int)r0/r1;
		r2=r0%r1;
	}
	return r1;
}
```

Cette fonction calcule le pgcd de a et b par divisions successives pour arriver à r_1.

1.4. Algorithme d'Euclide généralisé

Cet algorithme, extension du précédent, permet de calculer l'inverse d'un nombre et de trouver x et y dans la relation de Bezout (a*x + b*y = 1) :

```
void pgcdEtendu(int a,int b){
	int compteur=2,q[50],x[50],y[50];
	x[0]=0;
	x[1]=1;
	y[0]=1;
	y[1]=0;
	int ind=2;
	int r0=a;
	int r1=b;
	int r2=r0%r1;
	int qo=(int)r0/r1;
	q[0]=0;
	q[1]=qo;
	while (r2!=0){
		compteur++;
		r0=r1;
		r1=r2;
		qo=(int)r0/r1;
		q[ind]=qo;
		ind++;
		r2=r0%r1;
	}
	if (compteur>1)
	{
		for(int i=2;i<compteur;i++){
			x[i]= x[i-2] - q[i-1]*x[i-1];
			y[i]= y[i-2] - q[i-1]*y[i-1];
		}
	}
	xx=x[compteur-1];
	yy=y[compteur-1];
	ordre=compteur-1;}
```

Cette fonction applique l'algorithme d'Euclide généralisé sur deux nombres a et b et nous renvoie les valeurs suivantes :
- *l'ordre, soit le rang du dernier reste non nul*
- *les x,y correspondant au rang donné*

Remarque : dans le cas de cette fonction le rang ne peut dépasser 50.

1.5. Exponentiation rapide

Pour calculer rapidement une exponentielle du type a^x mod n, on utilise l'algorithme de Hörner :

on écrit x en base 2 : $x = x_{p-1}*2^{p-1} + x_{p-2}*2^{p-2} + \dots + x_0 2^0$
entrée a,x,n
sortie a^x mod n
si $x_{p-1} = 1$
  r = a
sinon r = 1
finsi
pour i de p – 2 à 0 faire
  $r = r^2$ mod n
  si $x_i = 1$
    r = r*a mod n
  finsi
finpour

On a donc besoin des deux fonctions suivantes pour réaliser ce calcul :

```
void decimalToBinary(int n){
        int bintmp[100];
        int ntmp = n;
        int ind = 0;
        while (ntmp!=0){
                bintmp[ind]=ntmp%2;
                ntmp = int(ntmp/2);
                ind++;
        }
        for(int i=ind-1;i>=0;i--){
                binaire[ind-1-i]=bintmp[i];
        }
        tailleBinaire=ind;
}
```

```
int horner(int a,int x,int n){
	decimalToBinary(x);
	int r,tmp;
	if (binaire[0]==1){
		r=a;
	}else{
		r=1;
	}
	for (int i=1;i<tailleBinaire;i++){
		if (binaire[i]==1){
			tmp=r*r*a;
		}else{
			tmp=r*r;
		}
		r=tmp%n;
	}
	return r;
}
```

2. Seconde séance (TP2)

Un texte est codé en utilisant le chiffrement RSA. On associe d'abord à chaque lettre de la langue un nombre de {0,1,…,25} puis on transforme chaque groupe de 3 lettres en un nombre par la règle suivante :

$$DOG \rightarrow 3 * 26^2 + 14 * 26^1 + 6 * 26^0 = 2398$$
$$CAT \rightarrow 2 * 26^2 + 0 * 26^1 + 19 * 26^0 = 1371$$

On chiffre ensuite chaque nombre (représentant chacun un groupe de 3 lettres) par le chiffrement RSA. On choisit ici comme données publiques :

$$n = 18923 \text{ et } b = 1261$$

2.1. Trouver les nombres p et q tels que n = p*q, puis le nombre φ(n) et enfin l'exposant de déchiffrement a

Pour ce faire, on va faire utiliser certaines fonctions précédemment réalisées :
- *« pgcd » pour décomposer n = p*q*
- *« pgcdEtendu » (Euclide généralisé) pour déterminer a*

```
void decompose(int n){
	int j,k=0,ntemp=0;
	int fin=n;
	if (estPremier(n)){
		facteur[0][0]=n;
		facteur[0][1]=1;
	}else{
		for (int i=2;i<fin;i=i+1){
			if (estPremier(i)) {
				if (n%i==0){
					ntemp=n;
					j=0;
					while(ntemp%i==0){
						j=j+1;
						ntemp=(int)ntemp/i;
					}
					facteur[k][0]=i;
					facteur[k][1]=j;
					k=k+1;
					fin=fin/i+1;
				}
			}
		}
	}
}
```

On obtient donc n = 18923 = 127 * 149.
On calcule alors φ(n) = (p-1) * (q-1) = 126 * 148 = 18648.

On applique l'algorithme d'Euclide généralisé sur φ(n) et b pour trouver a :

```
void pgcdEtendu(int a,int b){
	int compteur=2,q[50],x[50],y[50];
	x[0]=0;
	x[1]=1;
	y[0]=1;
	y[1]=0;
	int ind=2;
	int r0=a;
	int r1=b;
	int r2=r0%r1;
	int qo=(int)r0/r1;
	q[0]=0;
	q[1]=qo;
	while (r2!=0){
		compteur++;
		r0=r1;
		r1=r2;
		qo=(int)r0/r1;
		q[ind]=qo;
		ind++;
		r2=r0%r1;
	}
	if (compteur>1)
	{
		for(int i=2;i<compteur;i++){
			x[i]= x[i-2] - q[i-1]*x[i-1];
			y[i]= y[i-2] - q[i-1]*y[i-1];
		}
	}
	xx=x[compteur-1];
	yy=y[compteur-1];
	ordre=compteur-1;
}
```

Finalement a = 5797.

2.2. Déchiffrer ensuite le texte chiffré suivant :

12423 11524 7243 7459 14303 6127 10964 16399

Pour acquérir correctement le message à déchiffrer, on utilise le code suivant :

```
while((entree=getch())!=13){
	printf("%c",entree);
	if (entree==' '){
		message[ind2]=atoi(msg1);
		msg1 = new char[10];
		ind=0;
		ind2++;
	}else{
		msg1[ind]=entree;
		ind++;
```

```
        }
}
message[ind2]=atoi(msg1);
```

On décrypte alors chaque nombre (4/5 chiffres) du message par $M = C^a \bmod n$ où C représente chaque nombre ; on applique ensuite l'algorithme de Hörner pour calculer chaque M. Les nombres décryptés deviennent :

5438 1364 2925 14571 14303 5746 8805 4588

2.3. Revenir ensuite au texte original en inversant le procédé préliminaire

Pour cela, on procède par divisions successives de chaque nombre du texte décrypté par des puissances décroissantes de 26 (de 2 à 0). Ceci est automatisé grâce à la fonction suivante :

```
void decode(int *t,int taille,int a,int n){
        _int64 dechiffre,rdechiffre;
        char l1;
        char l2;
        char l3;
        for(int i=0;i<=taille;i++){
                //dechiffrage RSA
                dechiffre=horner(t[i],a,n);
                //decodage
                l1=(int)(dechiffre/(26*26)+65);
                rdechiffre=dechiffre%(26*26);
                l2=(int)(rdechiffre/26+65);
                l3=rdechiffre%26+65;
                printf("%c%c%c ",l1,l2,l3);
                mot[i][0]=l1;
                mot[i][1]=l2;
                mot[i][2]=l3;
        }
}
```

Après le décryptage RSA puis le déchiffrage des lettres à partir des nombres (fonction « decode »), on obtient le texte suivant :

IBE CAM EIN VOL VED INA NAR GUM

Remarque : pour vérifier ce résultat, nous avons réalisé une fonction « code » qui permet de réencoder les lettres pour être sûr que l'on retombe sur le message initial, le cryptage RSA étant réalisé par $C = M^b \bmod n$:

```
void code(int taille,int b,int n){
        int nb;
        _int64 chiffre;
        for(int i=0;i<=taille;i++){
                nb = (mot[i][0]-65)*26*26 + (mot[i][1]-65)*26 + (mot[i][2]-65);
                chiffre = horner(nb,b,n);
                printf("%d ",chiffre);}}
```

3. Conclusion

Ce programme permet de mettre en application les différents éléments vus en TD, notamment les algorithmes d'Euclide et de Hörner, afin de comprendre le mécanisme de cryptage/décryptage d'un message simple.

Néanmoins la taille des entiers à manipuler devient vite problématique. Si l'on souhaite utiliser des clés (n,b) de l'ordre d'une dizaine de chiffres ou plus (ce qui reste très en dessous de la plupart des clés de cryptage actuellement utilisées), le programme n'arrive plus à calculer ce que l'on souhaite, ou alors met un temps très important à effectuer ce calcul.

C'est pourquoi les algorithmes utilisés actuellement pour sécuriser des données (ex : sur Internet) calculent la même chose mais sont bien plus optimisés que ce nous venons de proposer.

ANNEXE
Code source C

```
#include <stdio.h>
#include <string.h>
#include <iostream>
#include <conio.h>
#include <math.h>
#include <windows.h>

// teste si un nb est premier
bool estPremier(int);
// décompose en facteurs premiers
void decompose(int);
// calcule le pgcd d'un nombre
int pgcd(int,int);
// calcule les coefficients x et y (Euclide généralisé)
void pgcdEtendu(int,int);
// convertit un entier décimal en un entier binaire
void decimalToBinary(int);
// applique la méthode de Hörner pour calculer a^x mod n
_int64 horner(int,int,int);
// décodage d'une chaîne
void decode(int*,int,int,int);
// recodage d'une chaîne
void code(int,int,int);

// résultat de décompose()
int facteur[100][2];
//résulats de decimalToBinary()
int binaire[100];
int tailleBinaire=0;
//résultats de pgcdEtendu();
int xx,yy,ordre;
char mot[50][3];

// MAIN
void main(){
	int n,b,ind=0,ind2=0;
	int *message = new int[50];
	char *msg1 = new char[10];
	char entree=' ';
	char reload='O';
	while(reload=='O'||reload=='o')
	{
		printf("\n************MI51 TP02**************\n\n");
		printf("Entrer un entier n : ");
		scanf("%d",&n);
		printf("Entrer un entier b : ");
		scanf("%d",&b);
		printf("Entrer le message crypte : ");
		//mot à décoder
		while((entree=getch())!=13){
```

```
                        printf("%c",entree);
                        if (entree==' '){
                                message[ind2]=atoi(msg1);
                                msg1 = new char[10];
                                ind=0;
                                ind2++;
                        }else{
                                msg1[ind]=entree;
                                ind++;
                        }
                }
                message[ind2]=atoi(msg1);
                //détermination de la clé secrète
                decompose(n);
                if (facteur[2][0]==0&&facteur[0][1]==1&&facteur[1][1]==1){
                        printf("\n\nn = %d * %d",facteur[0][0],facteur[1][0]);
                        int phi = (facteur[0][0]-1)*(facteur[1][0]-1);
                        printf("\nphi(n) = %d",phi);
                        pgcdEtendu(phi,b);
                        printf("\na = %d\n",xx);
                        decode(message,ind2,xx,n);
                        printf("\nmessage original : ");
                        code(ind2,b,n);
                }else{
                        printf("\n\nn n'est pas decomposable en un produit de nombres
premiers.");
                }


                printf("\n\nNouvel essai ? (O/N)\n");
                scanf("%s",&reload);
                if (reload=='O'||reload=='o'){
                        system("cls");
                        memset(message,0,sizeof(message));
                        ind2=0;
                        ind=0;
                        msg1 = new char[10];
                }
        }
}

// teste si un nombre est premier
bool estPremier(int n){
        bool result=true;
        if (n%2==0&&n!=2){
                result=false;
        }
        else
        {
                for(int i=3;i*i<=n;i=i+2){
                        if (n%i==0){
                                result=false;
                        }
                }
        }
        return result;
}
```

// décompose n en facteurs premiers

```
void decompose(int n){
	int j,k=0,ntemp=0;
	int fin=n;
	if (estPremier(n)){
		facteur[0][0]=n;
		facteur[0][1]=1;
	}else{
		for (int i=2;i<fin;i=i+1){
			if (estPremier(i)) {
				if (n%i==0){
					ntemp=n;
					j=0;
					while(ntemp%i==0){
						j=j+1;
						ntemp=(int)ntemp/i;
					}
					facteur[k][0]=i;
					facteur[k][1]=j;
					k=k+1;
					fin=fin/i+1;
				}
			}
		}
	}
}
```

// calcule le pgcd d'un nombre

```
int pgcd(int a, int b){
	int r0=a;
	int r1=b;
	int r2=r0%r1;
	int q=(int)r0/r1;
	while (r2!=0){
		r0=r1;
		r1=r2;
		q=(int)r0/r1;
		r2=r0%r1;
	}
	return r1;
}
```

// calcule les coefficients x et y (Euclide généralisé)

```
void pgcdEtendu(int a,int b){
	int compteur=2;
	int q[50];
	int x[50];
	x[0]=0;
	x[1]=1;
	int y[50];
	y[0]=1;
	y[1]=0;
	int ind=2;
	int r0=a;
	int r1=b;
	int r2=r0%r1;
```

```
	int qo=(int)r0/r1;
	q[0]=0;
	q[1]=qo;
	while (r2!=0){
		compteur++;
		r0=r1;
		r1=r2;
		qo=(int)r0/r1;
		q[ind]=qo;
		ind++;
		r2=r0%r1;
	}
	if (compteur>1)
	{
		for(int i=2;i<compteur;i++){
			x[i]= x[i-2] - q[i-1]*x[i-1];
			y[i]= y[i-2] - q[i-1]*y[i-1];
		}
	}
	xx=x[compteur-1];
	yy=y[compteur-1];
	ordre=compteur-1;
}
```

// convertit un entier décimal en un entier binaire

```
void decimalToBinary(int n){
	int ntmp = n;
	int ind = 0;
	while (ntmp!=0){
		binaire[ind]=ntmp%2;
		ntmp = int(ntmp/2);
		ind++;
	}
	tailleBinaire=ind;
}
```

// applique la méthode de Hörner pour calculer a^x mod n

```
_int64 horner(int a,int x,int n){
	decimalToBinary(x);
   _int64 r;
	if (binaire[tailleBinaire-1]==1){
		r=a;
	}else{
		r=1;
	}
	for (int i=tailleBinaire-2;i>=0;i--){
		if (binaire[i]==1){
			r=(r*r)%n;
			r=(r*a)%n;
		}else{
			r=(r*r)%n;
		}
	}
	return r;
}
```

// décodage d'un message codé puis chiffré RSA

```
void decode(int *t,int taille,int a,int n){
   _int64 dechiffre,rdechiffre;
        char l1;
        char l2;
        char l3;
        printf("\nmessage decrypte avec RSA : ");
        for(int i=0;i<=taille;i++){
                // décryptage RSA
                dechiffre=horner(t[i],a,n);
                printf("%d ",dechiffre);      // affichage des nombres décryptés
        }

        printf("\ntexte decode : ");
        for(i=0;i<=taille;i++){
                // décryptage RSA
                dechiffre=horner(t[i],a,n);
                // décodage des lettres
                l1=(int)(dechiffre/(26*26)+65);
                rdechiffre=dechiffre%(26*26);
                l2=(int)(rdechiffre/26+65);
                l3=rdechiffre%26+65;
                printf("%c%c%c ",l1,l2,l3); // affichage des lettres décodées
                mot[i][0]=l1;
                mot[i][1]=l2;
                mot[i][2]=l3;
        }
}
```

// recodage complet d'une chaîne de texte décodée pour vérification

```
void code(int taille,int b,int n){
        int nb;
        _int64 chiffre;
        for(int i=0;i<=taille;i++){
                nb = (mot[i][0]-65)*26*26 + (mot[i][1]-65)*26 + (mot[i][2]-65);
                chiffre = horner(nb,b,n);
                printf("%d ",chiffre);
        }
}
```