

-MIS1-
TP1 et TP2

Table des matières

| | |
|--|----|
| TP1 | 2 |
| 1. Test de primalité..... | 2 |
| 1.1. Pseudo-code | 2 |
| 1.2. Code en JAVA..... | 3 |
| 1.3. Résultat de sortie | 3 |
| 2. Décomposition d'un entier nb en facteurs premiers p et q..... | 3 |
| 2.1. Pseudo code..... | 3 |
| 2.2. Code en JAVA..... | 4 |
| 2.3. Résultat de sortie | 4 |
| 3. Algorithme d'Euclide..... | 4 |
| 3.1. Version itérative | 5 |
| 3.1.1. Pseudo code..... | 5 |
| 3.1.2. Code Java | 5 |
| 3.1.3. Résultat de sortie..... | 5 |
| 3.2. Version récursive..... | 6 |
| 3.2.1. Pseudo code..... | 6 |
| 3.2.2. Résultat de sortie..... | 6 |
| 3.3. Algorithme d'Euclide étendu..... | 6 |
| 3.3.1. Code en Java..... | 7 |
| 3.3.2. Résultat de sortie..... | 7 |
| 4. Algorithme d'exponentiation rapide de $z = xb \text{ mod } n$ | 8 |
| 4.1. Pseudo code..... | 8 |
| 4.2. Code Java | 8 |
| 4.3. Résultats de sortie..... | 8 |
| 5. Application au TP | 8 |
| 5.1. Recherche de p et q..... | 8 |
| 5.1.1. Résultat de sortie:..... | 9 |
| 5.1.2. Résultat de sortie:..... | 9 |
| 5.2. Déchiffrage..... | 9 |
| 5.2.1. Pseudo code..... | 9 |
| 5.2.2. Code en JAVA..... | 9 |
| 5.2.3. Résultat de sortie..... | 10 |
| 5.3. Revenir au texte original..... | 10 |
| 5.3.1. Pseudo code..... | 10 |
| 5.3.2. Code en JAVA..... | 10 |
| 5.3.3. Résultat de sortie..... | 11 |
| TP2..... | 13 |
| 1. Générateur de nombre pseudo-aléatoires | 13 |
| 2. Testeur de primalité de type Miller Rabin..... | 13 |
| 2.1. Algorithme de Miller Rabin | 13 |
| 2.2. Code Java | 14 |
| 3. Factorisation en nombres premiers | 15 |
| 3.1. Algorithme | 15 |
| 3.2. Code Java | 15 |
| 4. Fonctions de chiffrement et de déchiffrement RSA..... | 16 |
| Conclusion..... | 18 |

TP1

Questions préliminaires

1. Test de primalité

Ce test permet de déterminer si un entier nb est premier en testant successivement tous ces diviseurs possibles. Après avoir testé le diviseur 2, on teste par récurrence les nombres impairs jusqu'à \sqrt{nb}

1.1. *Pseudo-code*

```
entier nb //nb: nombre à tester
entier div //div: diviseur testé sur nb

si nb < 2
    nb n'est pas premier
finsi
si nb mod div = 0
    nb n'est pas premier
sinon
    si div = 2
        div = 3 //test sur les nombres impairs
    sinon
        si div <  $\sqrt{nb}$ 
            testprim(n, div+2) //appel de la fonction par récurrence
        sinon
            n est premier
        finsi
    finsi
finsi
finsi
```

1.2. Code en JAVA

```
prototype permettant l'appel de la fonction récursive à l'aide d'un seul argument
*/
public static boolean testPrimalite(int nb){return testPrim(nb,2);}

/*
Le test de primalite récursif
*/
public static boolean testPrim(int nb,int div){

    if(nb<2)                //Cas rare pour eviter les valeurs négatives
        return false;
    if(nb%div==0)           //On a trouvé un diviseur, donc le nombre n'est pas premier
        return false;
    else if(div==2)         //Si le diviseur est 2, on passe à 3
        return testPrim(nb,3);
    else if(div<Math.sqrt(nb)) //Sinon on incrémente de deux en deux
        return testPrim(nb,div+2);
    else return true;
}
```

1.3. Résultat de sortie

La fonction retourne un booléen: True si nb est premier, False s'il ne l'est pas.

2. Décomposition d'un entier nb en facteurs premiers p et q

2.1. Pseudo code

```
// Initialisation
entier n
entier div = 2

si n est premier
    il n'y a pas de décomposition possible
finsi

tant que div <  $\sqrt{n}$ 
    si n mod div = 0
        si n/div et div sont premiers
            on a trouvé les 2 entiers
        sinon
            il n'y a pas de décomposition en produit de 2 entiers possible
        finsi
    sinon
        si div = 2
            div = 3                //test sur les nombres impairs
```

```

        sinon div = div + 2
    fin si
fin tant que

```

2.2. Code en JAVA

```

/*
Permet de decomposer un nombre en produit de deux entiers
*/
public static int[] decomposition(int n){
    int div=2;
    int[] erreur={-1,-1};
    if(testPrimalite(n)) return erreur;
    while(div<Math.sqrt(n)){
        if(n%div==0){ // Si j'ai trouvé mon diviseur!
            if(testPrimalite(n/div) && testPrimalite(div)){//Les deux
diviseurs sont ils premiers?
                int[] res={n/div,div};
                return res;
            } else return erreur;
        } else
            if(div==2)//Si le diviseur est 2, on passe à 3
                div++;
            else
                div+=2;//Sinon on incrémente de deux en deux
        }
}

```

2.3. Résultat de sortie

La fonction retourne un tableau de 2 entiers: les 2 entiers solutions si la décomposition est possible, si on ne peut pas décomposer, on retourne -1, -1.

3. Algorithme d'Euclide

L'algorithme d'Euclide permet de calculer rapidement le PGCD de 2 nombres de sorte que:

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

L'algorithme consiste à effectuer les divisions suivantes:

$$r_0 = q_1 * r_1 + r_2 \quad 0 \leq r_2 < r_1$$

$$r_1 = q_2 * r_2 + r_3 \quad 0 \leq r_3 < r_2$$

...

$$r_{m-2} = q_{m-1} * r_{m-1} + r_m$$

$$r_{m-21} = q_m * r_m + 0$$

D'après le théorème, on a bien

$$\begin{aligned}
 \text{PGCD}(r_0, r_1) &= \text{PGCD}(r_1, r_0 - q_1 * r_1) \\
 &= \text{PGCD}(r_1, r_2) \\
 &= \text{PGCD}(r_2, r_3)
 \end{aligned}$$

$$\begin{aligned}
 & \dots \\
 & = \text{PGCD}(r_{m-1}, r_m) \\
 & = r_m
 \end{aligned}$$

Ainsi le PGCD de 2 nombres r_0 et r_1 en suivant l'algorithme d'Euclide est le dernier reste non nul de la division.

3.1. *Version itérative*

3.1.1. Pseudo code

```

entiers a, b //entiers à tester
entier tmp

tant que b non nul
    tmp = a
    a = b
    b = tmp mod b
fin tant que

```

3.1.2. Code Java

```

/*
Implémentation de l'algorithme d'Euclide en itératif
Permet de retourner le pgcd de deux entiers
*/
public static int euclideIteratif(int a, int b){
    if (a < 0)
        a *= -1;

    if (b < 0)
        b *= -1;

    while (b != 0) {
        int tmp = a;
        a = b;
        b = tmp % b;
    }

    return a;
}

```

3.1.3. Résultat de sortie

La fonction retourne l'entier a, PGCD de a et b.

3.2. Version récursive

3.2.1. Pseudo code

```
si b = 0
    le PGCD est a
sinon
    PGCDrecursif(b, a mod b)
finsi
```

```
/*
Calcule le pgcd de deux entiers de manière recursive
*/
public static int pgcdrecursif(int a, int b) {
    if (b == 0) {
        return a;
    }
    else {
        return pgcdrecursif(b, a%b);
    }
}

/*
Implémentation de l'algorithme d'Euclide en récursif
*/
public static int euclideRekursif(int a, int b) {
    if (a < 0)
        a *= -1;
    if (b < 0)
        b *= -1;
    int pgcd=pgcdrecursif(a,b);
    return pgcd;
}
```

Remarque: Pour calculer le pgcd, il faut appeler la fonction euclideRekursif qui teste d'abord les valeurs de a et b puis calcule récursivement en appelant la fonction annexe pgcdrecursif.

3.2.2. Résultat de sortie

La fonction retourne l'entier a, PGCD de a et b.

3.3. Algorithme d'Euclide étendu

L'algorithme d'Euclide étendu permet de calculer l'inverse de b modulo n s'il existe.

Soit $p * q = n$ p et q premiers entre eux

$$\varphi(n) = (p - 1) * (q - 1)$$

$$\text{PGCD}(b, \varphi(n)) = r_m$$

et l'inverse de b est calculé par l'équation suivante

$$x_k = x_{k-2} - q_{k-1} * x_{k-1} \quad \text{avec } x_0 = 0 \text{ et } x_1 = 1$$

3.3.1. Code en Java

```
/*
Implémentation de l'algorithme d'Euclide étendu
*/
public static double euclideEtendu(int n, int b){
    int[] decompose=decomposition(n);
    int phy=(decompose[0]-1)*(decompose[1]-1);
    int phySav=phy;
    Vector tabQ=new Vector();
    int i=0;
    int reste=1;
    while(reste!=0){
        tabQ.add( new Integer((int)Math.floor(phy/b)) );
        reste=phy%b;
        phy=b;
        b=reste;
        i++;
    }

    double res= euclidePartie2(tabQ,i);
    if(res<0)
        res=res+phySav;

    return(res);
}

/*
Calcul de la clé de déchiffrement
*/
public static double euclidePartie2(Vector tabQ,int rangMax){
    if(rangMax==0)
        return 0.0;
    else if(rangMax==1)
        return 1.0;
    else{

        Integer tmp=(Integer) tabQ.elementAt(rangMax-2);
        int tmp2 = tmp.intValue();
        double res=euclidePartie2(tabQ,rangMax-2)-
        (tmp2)*euclidePartie2(tabQ,rangMax-1);
    }
}
```

3.3.2. Résultat de sortie

La fonction retourne un entier qui est l'inverse de b.

4. Algorithme d'exponentiation rapide de $z = x^b \bmod n$

4.1. Pseudo code

```
entiers n et b // données
entier z = 1

tant que n non nul
    si n impair
        z = z * b
    fin si
    b = b * b
    n = n / 2
fin tant que
```

4.2. Code Java

```
/*
Implémentation de l'algorithme de Hörner
*/
public static int horner(int n, int b){
    int z = 1;
    while(n != 0){
        // si n est impair, on multiplie resultat par b
        if((n % 2) == 1){
            z = z * b;
        }
        b = b*b;
        n = n/2;
    }
    return z;
}
```

4.3. Résultats de sortie

La fonction l'entier z résultat de $z = x^b \bmod n$.

5. Application au TP

5.1. Recherche de p et q

On a $n = 18923$, on cherche les nombres p et q tels que $n = p * q$.

On applique la fonction de décomposition

5.1.1. Résultat de sortie:

n = 18923 p = 127 q = 149

On cherche ensuite $\varphi(n) = (p-1) * (q-1)$
 $= 126 * 149 = 18648$

Enfin, l'exposant de déchiffrement a est calculé grâce à l'algorithme d'Euclide étendu

5.1.2. Résultat de sortie:

a = 5797

5.2. Déchiffrage

**Il s'agit à présent de déchiffrer le texte suivant:
12423 11524 7243 7459 14303 6127 10964 16399**

On utilise la technique du déchiffrage par exponentiation modulaire.

5.2.1. Pseudo code

```
entiers a , b, n, code                  //données
entier i, z = 1                         // initialisation
conversion de b en binaire
pour i=0à longueur de b_binaire
    z=z2 mod n
    si b_binaire[i]=1
        z = z * code mod n
    finsi
fin pour
```

5.2.2. Code en JAVA

```
/*
  Déchiffrage d'un mot encodé par exponentiation modulaire
*/
public static int dechiffre(String code,int b, int n){
    int a=(int) euclideEtendu(n,b);
    int z = 1 ;
    int message=Integer.parseInt(code);
    String b_bin ;
    // convertir un entier en string binaire
    b_bin = Integer.toBinaryString((int)a);
    //algorithme d'exponentiation rapide qui permet de trouver la valeur déchiffrée
    for (int i=0; i< (b_bin.length());i++) {
        z = (int) Math.pow(z,2) % n;
        if (b_bin.charAt(i)=='1'){ // si b[i] = 1
            z = (z * message) % n;
        }
    }
    return(z);
}
```

5.2.3. Résultat de sortie

Le texte déchiffré devient:

5438 1364 2925 14571 14303 5746 8805 4588

5.3. Revenir au texte original

Maintenant que le texte est déchiffré, il faut le transformer pour que le texte apparaisse. Pour cela, on effectue des divisions successives sur chaque nombre pour donner un mot de 3 lettres.

5.3.1. Pseudo code

```
entier nb //donnée
chaîne_caractere : mot
pour i = 2 à 0
    lettre=nb/partie_entiere(26i)
    nb = nb mod 26i
    mot=mot+lettre
fin pour
```

Remarque: on utilise la variable origine pour effectuer la conversion des lettres de l'alphabet Unicode.

On inclut cette fonction qui prend entre autres pour paramètre le texte en entier: les mots s'ajoutent les uns à la suite des autres.

5.3.2. Code en JAVA

```
/*
  Decodage d'une phrase en connaissant n et b
*/
public static String decoderPhrase(String code, int b, int n){
    String[] tableauChiffre;
    String decode="";
    tableauChiffre=code.split(" ");
    for(int i=0;i<tableauChiffre.length;i++)
        decode+=lettrifie(dechiffre(tableauChiffre[i],b,n));
    // System.out.println("Phrase decodee: "+decode);
    return(decode);
}
```

```

/*
Transforme un nombre de la forme n°lettre*26^2+n°lettre*26^1+n°lettre*26^0 en un mot de
trois lettres
*/
public static String lettrifie(int nb){
    String mot="";
    char origine = 'A';
    // System.out.println("LETTRIFIE "+nb);
    for(int i=2;i>=0;i--){
        int lettre=nb/((int) Math.floor(Math.pow(26,i)));
        nb=nb%((int) Math.pow(26,i));
        char l=(char)(lettre+origine);
        mot+=l;
    }

    return mot;
}

```

5.3.3. Résultat de sortie

On obtient le texte:

IBE CAM EIN VOL VED INA NAR GUM

Remarque: Nous avons implémenté les fonctions qui permettent de recoder le message.

```

//Encode un mot numéroté selon les conventions décrites dans la fonction numerote
public static String encode(int mot,int n, int e){
    String res="";
    BigInteger m=new BigInteger(""+mot);
    m=m.modPow(new BigInteger(""+e),new BigInteger(""+n));
    mot=m.intValue();
    res="" + mot;
    return res;
}

//Crypte une phrase en la decomposant par blocs de 3 lettres, puis en encodant chaque
bloc
public static String crypte(String message, int n, int e){
    String cryp="";
    Vector mots=numerote(message);
    for(int i=0;i<mots.size();i++){
        // System.out.println(((Integer)(mots.elementAt(i))).intValue());
        cryp+=""+encode(((Integer)(mots.elementAt(i))).intValue(),n,e)+" ";
    }
    return cryp;
}

```

```

/*
Transforme un texte en une suite de nombres
de la forme n°lettre*26^2+n°lettre*26^1+n°lettre*26^0
trois par trois
*/
public static Vector numerote(String message){
    int mot=0;
    char origine = 'A';
    Vector mots=new Vector();
    int cpt=0;
    for(int i=0;i<message.length();i++){
        if(cpt%3==2 || i==message.length()-1){
            mot+=(int)Math.pow((double)26,(double)(2-cpt))*(message.charAt(i)-
origine);

            mots.add(new Integer(mot));
            mot=0;
            cpt=0;
        }else{
            mot+=(int)Math.pow((double)26,(double)(2-cpt))*(message.charAt(i)-
origine);

            cpt++;}
    }

    return mots;
}

```

TP2

L'objectif de ce TP est de programmer en Java des fonctions cryptographiques en utilisant entre autres la classe `BigInteger`.

1. Générateur de nombre pseudo-aléatoires

On utilise le constructeur

```
public BigInteger(int numBits, Random rnd)
```

avec numBits la taille maximale du nouveau `BigInteger`

rnd source aléatoire à employer pour calculer le nouveau `BigInteger`.

Il génère aléatoirement un `BigInteger` uniformément réparti entre 0 et $2^{\text{numBits}} - 1$

```
//Genere une nombre aléatoire très grand
public static BigInteger bigRandom(int nbBits){
    Random rnd=new Random();
    BigInteger tmp= new BigInteger(nbBits,rnd);
    return tmp;
}
```

2. Testeur de primalité de type Miller Rabin

2.1. Algorithme de Miller Rabin

écrire $p-1=2^Q \cdot m$ tel que m est impair

choisir a tel que $1 \leq a \leq p-1$

$b \leftarrow a^m$

si $b \equiv 1 [p]$, i.e. $B \bmod p = 1$ alors n est premier

pour $i=0$ à $Q-1$ faire

 si $a^{m \cdot 2^i} \equiv -1 [p]$ alors p est premier fin si

fin pour

2.2. Code Java

```
//Trouve les deux diviseurs du nombre donné en entrée
public static BigInteger [] getqm(BigInteger p){
    p = p.subtract(BigInteger.ONE);
    BigInteger two = new BigInteger("2");
    BigInteger neg = new BigInteger("-1");
    BigInteger [] rt = { BigInteger.ZERO, BigInteger.ZERO }; // rt = {q, m}
    if (p.mod(two).compareTo(BigInteger.ZERO) != 0){
        rt[0] = neg; rt[1] = neg;
        return rt;
    }
    BigInteger divisor = p.divide(two);
    BigInteger counter = BigInteger.ONE;

    while (divisor.mod(two).compareTo(BigInteger.ZERO)==0){
        counter = counter.add(BigInteger.ONE);
        divisor = divisor.divide(two);
    }
    rt[0] = counter; rt[1] = divisor;
    return rt;
}
```

```
//Mise en place de l'algorithme millerRabin qui permet de determiner la primalité d'un
//entier de grande taille.
public static boolean millerRabin(BigInteger pval) {
    BigInteger [] qandm = getqm(pval);
    BigInteger qval = qandm[0];
    BigInteger neg = new BigInteger("-1");
    if (qval.compareTo(neg)==0) return false;

    BigInteger bval = bigRandom(pval.bitLength());

    BigInteger mval = qandm[1];
    BigInteger two = new BigInteger("2");

    BigInteger pminusone = pval.subtract(BigInteger.ONE);

    if (bval.modPow(mval, pval).compareTo(BigInteger.ONE)==0) return true;
    BigInteger j = BigInteger.ZERO;
    BigInteger indexval = mval;
    while (j.compareTo(qval)<0)
    {
        if (pminusone.compareTo(bval.modPow(indexval, pval))==0) return true;
        indexval = indexval.multiply(two);
        j = j.add(BigInteger.ONE);
    }
    return false;
}
```

Le plus grand nombre premier sur 512bits est

13407807929942597099574024998205846127479365820592393377723561443721764030073546
976801874298166903427690031858186486050853753882811946569946433649006083527.

3. Factorisation en nombres premiers

3.1. Algorithme

On utilise le même principe que celle créée dans le TP1.

// Initialisation

entier n

entier div = 2

tant que $div < \sqrt{n}$

 si $n \bmod div = 0$

 si n/div et div sont premiers (test avec millerRabin)

 on a trouvé les 2 entiers

 sinon

 il n'y a pas de décomposition en produit de 2 entiers possible

 finsi

 sinon

 si div = 2

 div = 3

 //test sur les nombres impairs

 sinon div = div + 2

 finsi

finsi

3.2. Code Java

```
//Decomposition d'un nombre en un produit de deux entiers premiers
```

```
public static BigInteger [] decomposition(BigInteger n){
    BigInteger div=new BigInteger("2");
    BigInteger[] erreur={new BigInteger("-1"),new BigInteger("-1")};
    while((div.multiply(div)).compareTo(n)==-1){
        if((n.mod(div)).compareTo(BigInteger.ZERO)==0){
            if(millerRabin(n.divide(div)) && millerRabin(div)){//CAS OK
                BigInteger[] res={n.divide(div),div};
                return res;
            }else return erreur;
        }else div=div.add(BigInteger.ONE);
    }
}
```

fin tant que

Ainsi, on obtient pour 831802500: 150151 et 123433

et 18533588383 ne se décompose pas en deux entiers premiers (dans ce cas, le programme renvoie -1, -1).

4. Fonctions de chiffrement et de déchiffrement RSA

Le chiffrement ou encodage consiste à transformer le texte en une suite de nombres de la forme $n^{\circ}\text{caractere} * 127^7 + n^{\circ}\text{caractere} * 127^6 + \dots + n^{\circ}\text{caractere} * 127^0$, 8 par 8. On rappelle que la table ASCII est constitué de 127 caractères. Nous l'avons implémenté par 3 fonctions, la première « numerote » effectue la conversion, la deuxième « encodage » encode toute la phrase, enfin la troisième « encrypte » crypte la phrase par $C = m^e[n]$.

```
/*
Transforme un texte en une suite de nombres
de la forme

*/
public static Vector numerote(String message){
    BigInteger mot=BigInteger.ZERO;
    char origine = 'A'-'A';
    Vector mots=new Vector();
    int cpt=0;
    for(int i=0;i<message.length();i++){
        if(cpt%8==7||i==message.length()-1){
            //On est en fin de bloc de 8 lettres, ou à la fin du texte
            BigInteger puiss=new BigInteger("127");
            BigInteger car=new BigInteger(""+(int)(message.charAt(i)-origine));
            BigInteger toto=mot;
            mot=encrypt(mot.add(puiss.pow(7-cpt).multiply(car)));
            mots.add(new BigInteger(""+mot));
            mot=BigInteger.ZERO;
            cpt=0;
        }else{
            //On est en train de parcourir les lettres et de les ajouter
            BigInteger puiss=new BigInteger("127");
            BigInteger car=new BigInteger(""+(int)(message.charAt(i)-origine));
            mot=(mot.add(puiss.pow(7-cpt).multiply(car)));
            cpt++;}
    }
    return mots;
}

//Prend une phrase en entrée, et ressort une phrase codée et cryptée
public static String encodage(String phrase){
    String res="";
    Vector mots=numerote(phrase);
    for(int i=0;i<mots.size();i++){
        res+=mots.elementAt(i)+" ";
    }
    return res;
}

//Encrypt un message selon les clés
public static BigInteger encrypt(BigInteger message)
{
    //Encrypter c'est c = m^e mod n
    return message.modPow(e, n);
}
```

```

public static String lettrifie(String message){
    char origine = 'A'-'A';
    String mot="";
    BigInteger nb=new BigInteger(message);
    //Je decrypte le nombre
    BigInteger bloc=decrypt(nb);
    BigInteger lettre;
    //Pour chacun, je decode en RSA
    for(int i=7;i>=0 && bloc.compareTo(BigInteger.ZERO)==1;i--){
        BigInteger puiss=new BigInteger("127");
        //je prends mon caractere
        lettre=bloc.divide(puiss.pow(i));
        //Je prepare le tour suivant
        bloc=bloc.mod(puiss.pow(i));
        //Je continue
        char l=(char)(lettre.intValue()+origine);
        //Et je le rajoute au mot de 8 caracteres
        mot+=l;
    }
    return mot;
}

//Separe les blocs de codes cryptés et les met sous forme de lettres
public static String decodage(String phrase){
    String[] tableauChiffre;
    String decode="";
    tableauChiffre=phrase.split(" ");
    for(int i=0;i<tableauChiffre.length;i++)
        decode+=lettrifie(tableauChiffre[i]);
    return(decode);
}

public static BigInteger decrypt(BigInteger message)
{
    //Decrypter c'est  $m = c^d \bmod n$ 

```

Le déchiffrement ou décryptage consiste à passer d'un nombre crypter à une phrase décodée. Ainsi, on décrypte par la fonction « decrypt » qui utilise la formule de déchiffrement $m = C^d [n]$. La fonction « lettrifie » convertit le nombre en lettres et la fonction déchiffre la totalité du nombre.

Conclusion

Ce second TP nous a permis d'approfondir les notions de cryptographies et les applications développées dans le premier TP en introduisant notamment la classe `BigInteger` qui permet de gérer des grands nombres, essentiels pour le système RSA. Nous avons ajouté en plus quelques fonctions pour mettre en évidence les possibilités offertes par la cryptographie RSA.