Rapport Projet Cryptographie M1 2019/2020

Guichard Erwan 35003608
Hoarau Fabien 35001536

M1 Informatique

35003608@univ-reunion.fr
35001536@univ-reunion.fr

25 mars 2020 - 12 avril 2020

Sommaire

Résumé

Nous allons vous présenter notre application mobile de cryptographie développée sous Android Studio. Dans ce rapport, nous montrerons les différentes méthodes de chiffrement que nous avons choisies.

Introduction

Nous avons opté pour les méthodes de chiffrement et déchiffrements suivants :

— **Atbash**

— **Cesar**

— **Vigenere**

— **Hill**

— **DES**

— **RSA**

Avant de présenter les différentes méthodes de cryptage, nous allons voir des fonctions qui seront utilisées dans toutes les méthodes qui utiliseront la table ASCII étendue.

Nous avons tout d'abord ajouté deux tableau de caractères étendus car Java n'utilise pas la même table ASCII que celle que l'on souhaite utiliser pour ce projet. Elle utilise l'ASCII ANSI alors que nous souhaitons utiliser l'ASCII OEM [1] :

— La première appelée **EXTENDED** contient tous les caractères de la table OEM.

— La deuxième appelée **DECIMALS** contient la valeur décimale en ASCII ANSI de chacun des caractères de la table EXTENDED.

```
1  // Liste des caractères étendus sont forme d'hexa
2  public static final char[] EXTENDED = {          '\u00C7', '\u00FC', '\u00E9', '\u00E2',
3          '\u00E4', '\u00E0', '\u00E5', '\u00E7', '\u00EA', '\u00EB', '\u00E8', '\u00EF',
4          '\u00EE', '\u00EC', '\u00C4', '\u00C5', '\u00C9', '\u00E6', '\u00C6', '\u00F4',
5          '\u00F6', '\u00F2', '\u00FB', '\u00F9', '\u00FF', '\u00D6', '\u00DC', '\u00A2',
6          '\u00A3', '\u00A5', '\u20A7', '\u0192', '\u00E1', '\u00ED', '\u00F3', '\u00FA',
7          '\u00F1', '\u00D1', '\u00AA', '\u00BA', '\u00BF', '\u2310', '\u00AC', '\u00BD',
8          '\u00BC', '\u00A1', '\u00AB', '\u00BB', '\u2591', '\u2592', '\u2593', '\u2502',
9          '\u2524', '\u2561', '\u2562', '\u2556', '\u2555', '\u2563', '\u2551', '\u2557',
10         '\u255D', '\u255C', '\u255B', '\u2510', '\u2514', '\u2534', '\u252C', '\u251C',
11         '\u2500', '\u253C', '\u255E', '\u255F', '\u255A', '\u2554', '\u2569', '\u2566',
12         '\u2560', '\u2550', '\u256C', '\u2567', '\u2568', '\u2564', '\u2565', '\u2559',
13         '\u2558', '\u2552', '\u2553', '\u256B', '\u256A', '\u2518', '\u250C', '\u2588',
14         '\u2584', '\u258C', '\u2590', '\u2580', '\u03B1', '\u00DF', '\u0393', '\u03C0',
15         '\u03A3', '\u03C3', '\u00B5', '\u03C4', '\u03A6', '\u0398', '\u03A9', '\u03B4',
16         '\u221E', '\u03C6', '\u03B5', '\u2229', '\u2261', '\u00B1', '\u2265', '\u2264',
17         '\u2320', '\u2321', '\u00F7', '\u2248', '\u00B0', '\u2219', '\u00B7', '\u221A',
18         '\u207F', '\u00B2', '\u25A0', '\u00A0' };
19
20 // Liste des décimaux des caractères étendus
21 public static final int[] DECIMALS = {     199,   252,   233,   226,
22         228,   224,   229,   231,   234,   235,   232,   239,   238,
23         236,   196,   197,   201,   230,   198,   244,   246,   242,
24         251,   249,   255,   214,   220,   162,   163,   165,  8359,
```

```
25          402,   225,   237,   243,   250,   241,   209,   170,   186,
26          191,  8976,   172,   189,   188,   161,   171,   187,  9617,
27          9618, 9619,  9474,  9508,  9569,  9570,  9558,  9557,  9571,
28          9553, 9559,  9565,  9564,  9563,  9488,  9492,  9524,  9516,
29          9500, 9472,  9532,  9566,  9567,  9562,  9556,  9577,  9574,
30          9568, 9552,  9580,  9575,  9576,  9572,  9573,  9561,  9560,
31          9554, 9555,  9579,  9578,  9496,  9484,  9608,  9604,  9612,
32          9616, 9600,   945,   223,   915,   960,   931,   963,   181,
33           964,  934,   920,   937,   948,  8734,   966,   949,  8745,
34          8801,  177,  8805,  8804,  8992,  8993,   247,  8776,   176,
35          8729,  183,  8730,  8319,   178,  9632,   160 };
```

Ces tables [7] seront donc utilisées par les fonctions qui suivent.

On a tout d'abord la fonction **getExtendChar()** :

```
1  private static int getExtendChar(int code) {
2      for(int i=0; i<DECIMALS.length; i++) {
3          if(DECIMALS[i] == code) {
4              return 128+i;
5          }
6      }
7      return 0;
8  }
```

Elle nous permettra de convertir la valeur décimale de la table ASCII ANSI de Java en celle de la table ASCII OEM que l'on souhaite utiliser. Elle prend donc en paramètre un décimal de la table ASCII de Java, par exemple : 233 (qui représente 'é' avec cette table) et va parcourir la table DECIMALS pour récupérer sa position et par la suite retourner le décimal converti au format de la table ASCII qu'on veut utiliser, ce qui donnera 130.

Nous avons ensuite la fonction **getCharacterByDecimal()** :

```
1  public static String getCharacterByDecimal(int code) {
2      StringBuilder chararter = new StringBuilder();
3      if( code > 127 ) {
4          chararter.append( EXTENDED[code - 128] );
5      } else {
6          String normalCharacter = Character.toString( (char) (code) );
7          String hexaOfNormalChar = "\\x"+ Integer.toHexString( code | 0x100).substring(1);
8          chararter.append(normalCharacter.replaceAll("\\p{C}", "\\"+hexaOfNormalChar) );
9      }
10     return chararter.toString();
11 }
```

Elle nous permettra de récupérer n'importe quel caractère via son code décimal et de le retourner ensuite. Si par exemple le code décimal et supérieur à 127 c'est qu'il s'agit d'un caractère de la table étendue donc on parcourt le tableau EXTENDED pour récupérer le récupérer sinon on récupère le caractère de la table normal avec un *Character.toString((char) code)*. Point particulier,

comme certains caractères de la table normal ne sont pas affichable (comme par exemple le décimal 27 qui correspond à ESC (escape)) il faudra alors retourner un hexadécimal pour pouvoir avoir un visuel du caractère (pour ESC on aura alors \x1b).

Puis la fonction **testHex()** :

```
1  private static boolean testHex(String value) {
2      boolean res;
3      try {
4          new BigInteger(value,16);
5          res = true;
6      } catch (NumberFormatException e) {
7          res = false;
8      }
9      return (res);
10 }
```

Retourne simplement Vrai ou Faux si la chaîne de caractères en paramètre est bien au format hexadécimal (c'est-à-dire, si il contient bien des caractères compris entre 0 et 9 et entre a et f). On utilise un *BigInteger* pour des chaînes de caractères longs.

Et finalement la fonction **getListOfDec()** :

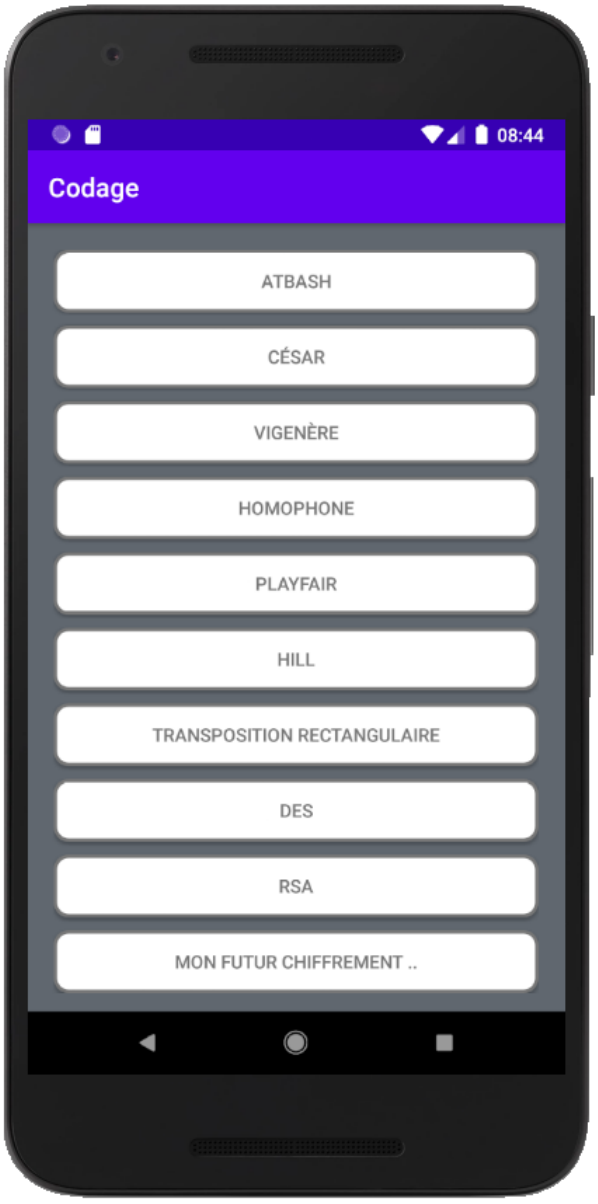
```
1  public static List<Integer> getListOfDec(String mess){
2      List<Integer> listOfDec = new ArrayList<>();
3      for(int j = 0; j < mess.length(); j++) {
4          int codePointKey = mess.codePointAt( j );
5          if(codePointKey > 127)  codePointKey = getExtendChar(codePointKey);
6          if(j<=mess.length()-4) {
7              String i0 = String.valueOf(mess.charAt(j));
8              String i1 = String.valueOf(mess.charAt(j+1));
9              if(i0.equals("\\") && i1.equals("x")) {
10                 String hexaCode = String.format("%s%s", mess.charAt(j+2), mess.charAt(j+3));
11                 if(testHex(hexaCode)) {
12                     codePointKey = Integer.parseInt(hexaCode,16);
13                     j=j+3;
14                 }
15             }
16         }
17         listOfDec.add(codePointKey);
18     }
19     return listOfDec;
20 }
```

Cette fonction qui prend en paramètre une chaîne de caractères retournera une liste contenant les décimaux de chaque caractères du message. Si dans le message il y a deux caractères à la suite qui sont \x et qu'il est suivi de deux caractères qui sont bien des hexadécimaux alors on converti cet hexadécimal en décimal et on saute ses quatre caractères pour continuer d'analyser la chaîne.

Voilà donc les principales fonctions nous ne reviendrons donc plus sur ses parties. Pour la continuité à venir nous verrons juste les points particuliers de chacune des méthodes de cryptage.

1 Les cryptages

Tout d'abord, nous avons réalisé notre projet sur Android Studio et pour pouvoir accéder à toutes nos méthodes de cryptage nous avons donc fait tout d'abord une page principale contenant plusieurs boutons menant à chacune d'entre elles. Notre application se présente de cette manière suivante :



Voyons maintenant les différentes méthodes de cryptage que nous avons réalisée :

1.1 César

```
1  public void cesar(String mess, int key){
2
3      StringBuilder messEncrypt = new StringBuilder();
4      List<Integer> listOfDec = getListOfDec(mess);
5
6      for (int i = 0; i < listOfDec.size(); i++) {
7
8          // ------------------------------------ RECUPERATION DU CODE DECIMAL
9
10         int codePoint = listOfDec.get(i);
11
12         if (key > 255) key= key%256;
13         if (key <0) key = (256-(-key))%256;
14
15         // ------------------------------------ RECUPERATION DU CODE DECIMAL DECALE
16
17         codePointDecal = codePoint + key;
18         if(codePointDecal > 255) codePointDecal = codePointDecal%256;
19
20         // ------------------------------------ AFFICHAGE
21
22         messEncrypt.append(getCharacterByDecimal(codePointDecal));
23     }
24     result.setText(messEncrypt.toString());
25 }
```

La fonction principale de César est assez simple à comprendre :

— On récupère la liste des décimaux du message, on la parcours caractère par caractère et pour chacune d'elles :

1. On vérifie la taille de la clé soit bien comprise entre 0 et 255 pour éviter d'avoir de trop grande valeurs qui sont inutiles (on peut voir à la ligne 13 un cas on la clé est inférieur à 0 c'est parce que dans notre appel à la fonction cesar() pour différencier le décryptage du cryptage nous utilisons une clé négative)
2. On ajoute la clé positive (cryptage) ou négative (décyptage) au décimal du caractère qu'on analyse pour avoir le décimal décalé (il se peut que l'addition des deux crée une valeur supérieure à 255 donc on récupère le reste de la division par 256 pour repartir du décimal 0)
3. On récupère le caractère du décimal décalé et on l'ajoute à une variable résultat

— On retourne la variable résultat contenant l'ensemble des caractères

1.2 AtBash

```
1  public void atbash(String mess){
2
3      StringBuilder messEncrypt = new StringBuilder();
4      List<Integer> listOfDec = getListOfDec(mess);
5
6      for (int i = 0; i < listOfDec.size(); i++) {
7
8          // ------------------------------------ RECUPERATION DU CODE DECIMAL
9
10         int codePoint = listOfDec.get(i);
11         reverseCodePoint = 255 - codePoint;
12
13         // ------------------------------------ AFFICHAGE
14
15         messEncrypt.append(getCharacterByDecimal(reverseCodePoint));
16     }
17     result.setText(messEncrypt.toString());
18 }
```

La fonction principale de AtBash est encore plus simple car elle permet de crypter et décrypter :

— On récupère la liste des décimaux du message à crypter, on la parcours caractère par caractère et pour chacune d'elles :

1. On inverse la valeur du décimal du caractère pour obtenir son opposé
2. On récupère le caractère du décimal inversé et on l'ajoute à une variable résultat

— On retourne la variable résultat contenant l'ensemble des caractères

1.3 Vigenère

```
1  public void vigenere(String mess, String key, int cryptingMethod){
2
3          StringBuilder messEncrypt = new StringBuilder();
4          List<Integer> listOfDecMess = getListOfDec(mess);
5          List<Integer> listOfDecKey = getListOfDec(key);
6
7          for (int i = 0; i < listOfDecMess.size(); i++) {
8
9              // ------------------------------------ RECUPERATION DU CODE DECIMAL
10
11             int codePointMess = listOfDecMess.get(i);
12             int codePointKey = listOfDecKey.get(i%listOfDecKey.size());
13
14             int codePointDecal = codePointMess + cryptingMethod * codePointKey;
15
16             if (codePointDecal > 255) codePointDecal = codePointDecal%256;
17             if (codePointDecal < 0) codePointDecal = 256+codePointDecal;
18
19             // ------------------------------------ AFFICHAGE
20
21             messEncrypt.append(getCharacterByDecimal(codePointDecal));
22         }
23         result.setText(messEncrypt.toString());
```

La fonction principale de Vigenère est de la sorte :

— On récupère la liste des décimaux du message à crypter et du message qui servira de clé, on parcourt la liste du message caractère par caractère et pour chacune d'elles :

1. On récupère le décimal de la clé en faisant en sorte de boucler dans cette liste si la liste des décimaux du message est plus longue que celle de la clé.
2. On ajoute le décimal du message avec celle de la clé qui sera positive si la méthode de cryptage est 1 (cryptage) ou négative si elle est à -1 (décryptage)
3. Comme pour César, on adapte le décimal pour qu'elle reste comprise entre 0 et 255
4. On récupère le caractère du décimal inversé et on l'ajoute à une variable résultat

— On retourne la variable résultat contenant l'ensemble des caractères

1.4 Hill

```
1  public void hill(String mess, int A, int B, int C, int D, int cryptingMethod){
2
3      StringBuilder messEncrypt = new StringBuilder();
4      List<Integer> listOfDec = getListOfDec(mess);
5
6      for (int i = 0; i < listOfDec.size(); i = i + 2) {
7          if(listOfDec.size()%2 != 0) listOfDec.add(35);
8
9          // ------------------------------------------ RECUPERATION DU DECIMAL
10
11         int cpChar0 = listOfDec.get(i);
12         int cpChar1 =listOfDec.get(i+1);
13
14         int determinant = det(pA,pB,pC,pD);
15         if( determinant==0 || (determinant%2 == 0 && determinant%13 != 0) ) {
16             /*notification que la matrice est invalide*/
17         } else {
18             // ===================================== CRYPTAGE
19
20             if(cryptingMethod == 1){
21                 //--------------- Combinaison linéaire du bloc et modulo
22
23                 int linearComb0 = mod(A*cpChar0 + B*cpChar1,256);
24                 int linearComb1 = mod(C*cpChar0 + D*cpChar1,256);
25
26                 // ------------------------------------ AFFICHAGE
27
28                 messEncrypt.append(getCharacterByDecimal(linearComb0));
29                 messEncrypt.append(getCharacterByDecimal(linearComb1));
30
31                 // ================================== DECRYPTAGE
32
33             } else {
34                 //--------------- Multiplication avec la comatrice
35
36                 int[] transpo = transpoComatrice(pA,pB,pC,pD);
37                 int invDet = inverDet(determinant, 256);
38
39                 for(int j = 0; j<transpo.length; j++) { transpo[j] =(transpo[j]*invDet); }
40
41                 int val0 = mod(transpo[0]*cpChar0 + transpo[1]*cpChar1,256);
42                 int val1 = mod(transpo[2]*cpChar0 + transpo[3]*cpChar1,256);
43
44                 // ------------------------------------ AFFICHAGE
45
46                 messEncrypt.append(getCharacterByDecimal(val0));
47                 messEncrypt.append(getCharacterByDecimal(val1));
48             }
49         }
50     }
51     result.setText(messEncrypt.toString());
52 }
```

Pour la méthode de cryptage Hill qui utilise une matrice 2x2 il faut impérativement que cette matrice est un déterminant différent de 0, qu'il soit impair et qu'il n'est pas multiple de 13 car sinon on ne pourra pas décrypter le message.

Il faut aussi que le message soit de taille paire vu que Hill analyse les caractères deux par deux donc si ce n'est pas le cas on insère une nulle (ici nous utiliserons le décimal 35 qui correspond au caractère #)

Si tout est bon le cryptage peut donc être fait et il se décompose ainsi :

— On récupère la matrice 2x2 et la liste des décimaux du message à crypter, on parcourt la liste du message en faisant un couple de deux caractères et pour chacun de ses couples :
  — Ensuite si on souhaite crypter :
    1. On fait les deux combinaisons linéaires du couple avec la matrice en récupérant le modulo 256 pour bien obtenir un décimal entre 0 et 255
    2. On récupère les deux caractères via les deux décimaux obtenus et on les ajoute à une variable résultat
  — Ou bien si on souhaite décrypter :
    1. On fait les deux combinaisons linéaires du couple avec l'inverse de la matrice en récupérant le modulo 256 pour bien obtenir un décimal entre 0 et 255
    2. On récupère les deux caractères via les deux décimaux obtenus et on les ajoute à une variable résultat
— On retourne la variable résultat contenant l'ensemble des caractères

Nous utilisons aussi des petites fonctions qui réalisent des tâches telles que :

— La fonction **mod(value, modulo)** qui retourne juste le modulo d'une valeur qu'elle soit positive ou non
— La fonction **det(a, b, c, d)** qui retourne le déterminant d'une matrice 2x2
— La fonction **transpoComatrice(a, b, c, d)** qui retourne un tableau d'entier contenant les valeurs de la transposée d'une comatrice 2x2
— La fonction **inverDet(value, modulo)** qui retourne l'inverse du déterminant modulo une valeur (dans notre cas cette valeur est 256)

1.5 DES

La méthode de cryptage DES repose sur différentes permutations avec plusieurs tables [5] c'est pourquoi nous ne verrons pas tout en détail mais nous allons juste voir quelques points importants ainsi que quelques détails concernant les différentes fonctions utilisées.

Nous avons réalisé le DES et hexadécimal et en ASCII OEM donc lorsque l'on appuis sur le bouton Crypter / Décrypter on exécute tout d'abord la fonction **getValDES(bloc, cle, cryptingMethod)** qui vérifiera d'abord si on est en mode hexadécimal ou ASCII.

— Si c'est de l'hexadécimal alors on le sépare en bloc de 16 caractères en complétant avec des 0 si un bloc ne comporte pas ses 16 caractères et on exécute la fonction *execDES() (qu'on verra par la suite)* avec chaque bloc.

— Sinon si c'est de l'ASCII :

1. On récupère la liste des décimaux du message et on ajoute des 0 pour que les blocs sont bien de taille 8 bits
2. Avec cette liste on convertit chaque caractère en son octet et on les ajoute dans une variable
3. Avec cette variable, on crée des blocs de 64 bits que l'on convertit en hexadécimal et on exécute la fonction *execDES()* avec chaque bloc.

La fonction principale **execDES(bloc, cle, cryptingMethod)**, qui prend en paramètre une chaîne de caractères et un message clé en hexadécimal ainsi que la méthode de cryptage, va retourner directement le message crypté ou décrypté selon la méthode de cryptage sous forme d'hexadécimal.

```
1  public String execDES(String bloc, String cle, int cryptingMethod) {
2      while(cle.length() < 17) cle += "0";
3
4      // Récuperation des clés
5      String[] keys = getAllKeys(cle);
6
7      // Permutation initiale
8      String permutInitiale = permutation(tablePermutInitiale, bloc);
9
10     // Itération - Feistel
11     if (cryptingMethod == 1) {
12         for (int i = 0; i < 16; i++) {
13             permutInitiale = feistel(permutInitiale, keys[i], i);
14         }
15     } else {
16         for (int i = 15; i > -1; i--) {
17             permutInitiale = feistel(permutInitiale, keys[i], 15 - i);
18         }
19     }
20     // Permutation final
21     String permutFinale = permutInitiale.substring(8, 16) + permutInitiale.substring(0, 8);
22     permutFinale = permutation(tablePermutFinal, permutFinale);
23
24     return permutFinale; \newline
25 }
```

Cette fonction utilise des petites fonctions qui sont :

— La fonction **getAllKey(cle)** qui retourne un tableau contenant les 16 sous-clés qui seront plus tard utilisées

— La fonction **permutation(table, bloc)** qui retourne la chaîne hexadécimale permutée avec la table donnée

— La fonction **feistel(hex, cle, tour)** qui retourne l'hexadécimal itéré avec une des sous-clés avec à un tour donné

Nous utilisons aussi d'autres fonctions dans chacune de ses petites fonctions qui sont :

— La fonction **leftShift(hexa, nbDecalage)** qui permet d'effectuer un décalage de bit à gauche d'une chaîne de caractères selon le nombre souhaité en paramètre (qui dans le cas de DES est de un ou deux bits). Elle est utilisée lors de la création des 16 sous-clés

— La fonction **xor(a, b)** qui retourne le XOR logique entre la valeur a et b utilisée dans *feistel()*

— Les fonctions **hexaToBin(hex)** et **binToHexa(bin)** qui convertissent une chaîne en son autre format

— La fonction **sBox(hex)** qui converti une chaîne hexadécimale de 48 bits en une chaîne de 32 bits. Elle est utilisée dans la fonction *feistel()*

— La fonction **getDecimals(hex)** qui retourne une liste contenant les valeurs décimales d'une chaîne de caractères en hexadécimal qui sera utilisée pour convertir la chaîne hexadécimal en sortie de *execDES()*

1.6 RSA

Un point particulier pour le *chiffrement RSA* est l'utilisation de la classe ***BigInteger***. Au moment de l'étape de chiffrage et déchiffrage, si on utilise de simple *int*, lorsqu'on calcule $m^e mod n$ (pour crypter) et $m^d mod n$ (pour décrypter), on obtient un résultat différent de celui qui correspond au PDF du cours.(Sans doute parce que Java a une limite dans ce type de calcul).
L'utilisation de ***BigInteger*** permet de passer cette limitation.

Conclusion

C'est sur des sites tel que Bibmath [2], Nymphomath [6] ou encore GeekForGeeks [3] que nous avons principalement tourné nos recherches et que nous avons pu développer les différents cas ainsi que comprendre différentes étapes de cryptage qui n'étaient pas très claires de notre point de vue. Le code est disponible sur le GitHub [4].

Références

[1] *ASCII OEM et ANSI.* getcif.

[2] *bibmath.* bibmath.

[3] *GeekForGeeks.* geekforgeeks.

[4] *GitHub du projet.* gitHub.

[5] *Les tables de DES.* Wikipedia.

[6] *Nymphomath.* Nymphomath.

[7] *Table étendue en Java.* stackoverflow.