Exercice D.1 Premier ou pas ? Déterminez si le nombre

$$n = 170141183460469231731687303715884105727$$

est premier ou pas. Vous compléterez le squelette du programme `EPP.java`, qui s'appuie sur la classe **`BigInteger`** (Fig. 27), ou bien celui de `epp.c`, qui fait appel à la librairie GMP (Fig. 28), disponibles dans le répertoire `EPP` de l'archive `RSA.zip`. Vous veillerez à ce que la réponse renvoyée par la fonction **`est_probablement_premier()`** soit correcte avec une probabilité d'erreur inférieure à 10^{-15}.

— En Java, vous utiliserez la méthode **`boolean isProbablePrime(int c)`** des **`BigInteger`** ;
— En C, vous pourrez utiliser la fonction **`int mpz_probab_prime_p(mpz_t n, int r)`** de GMP, décrite succinctement à la page 23.

```java
import java.math.BigInteger;
public class EPP {
  public static void main(String[] args) {
    BigInteger n = new BigInteger("170141183460469231731687303715884105727", 10);
    System.out.print("Le nombre " + n);
    if (est_probablement_premier(n)) System.out.println(" est probablement premier!");
    else System.out.println(" n'est absolument pas premier!");
  }
}
```

FIGURE 27 – Programme `EPP.java` de l'archive `RSA.zip` à compléter

```c
#include "gmp.h"
int main(void){
  mpz_t n;                          // Déclaration de l'entier GMP n
  mpz_init(n);                      // Initialisation de l'entier GMP n
  mpz_set_str(n, "170141183460469231731687303715884105727", 10);
  gmp_printf("Le nombre %Zd", n);   // Affichage de l'entier GMP n, en décimal
  if (est_probablement_premier(n)) printf(" est probablement premier!\n");
  else printf(" n'est absolument pas premier!\n");
  mpz_clear(n);                     // Libération de la mémoire allouée à n
  exit(EXIT_SUCCESS);
}
```

FIGURE 28 – Programme `epp.c` de l'archive `RSA.zip` à compléter

Exercice D.2 Mon premier grand nombre premier La fabrique de clefs RSA se fonde principalement sur la recherche de nombres premiers choisis aléatoirement et gardés secrets. Dans cet exercice, vous devez modifier le code du programme de l'exercice D.1 afin de fabriquer vos propres nombres premiers secrets.

Question 1. Trouvez un nombre premier de 512 bits, c'est-à-dire compris entre 2^{511} et $2^{512} - 1$. Pour cela, tirez des nombres *impairs* de 512 bits de manière aléatoire [4] jusqu'à en trouver un qui soit (très probablement) premier.

Question 2. Complétez votre programme afin qu'il affiche à l'écran :
— le nombre premier obtenu, en décimal,
— le nombre de tentatives nécessaires afin de trouver ce nombre,
— le temps d'exécution nécessaire à ce calcul.

Question 3. Quels sont, en moyenne, sur 10 exécutions successives, le nombre de tentatives nécessaires et le temps de calcul observé pour trouver un nombre premier de 512 bits ? Conservez vos observations dans vos archives personnelles.

4. En Java, pour fabriquer un nombre aléatoire, il suffira de faire appel au constructeur adéquat de la classe **`BigInteger`** comme illustré sur la Figure 31. En C, vous pourrez utiliser les fonctions GMP indiquées sur la page 23 et vous inspirer du code de la Figure 29. Ces deux programmes sont disponibles dans l'archive `RSA.zip`.

Quelques fonctions utiles de GMP

Une documentation sur la librairie GMP est disponible en ligne sur le site de l'UE. Les entiers proposés par GMP sont des variables signées de type **mpz_t**. Chacune de ces variables doit être initilisée à l'aide de la fonction **mpz_init** avant d'être utilisée. La librairie GMP propose également une extension de **printf** notée **gmp_printf** qui permet l'affichage de variables de type **mpz_t**, comme le montre le code de la figure **??**.

Affectation, affichage et comparaisons

int mpz_set_str (mpz_t rop, char *str, int base)
Set the value of rop from str, a null-terminated C string in base $base$. White space is allowed in the string, and is simply ignored. The base may vary from 2 to 62, or if base is 0, then the leading characters are used : $0x$ and $0X$ for hexadecimal, $0b$ and $0B$ for binary, 0 for octal, or decimal otherwise.

char * mpz_get_str (char *str, int base, mpz_t op)
Convert op to a string of digits in base $base$. The base argument may vary from 2 to 62.

int mpz_cmp (mpz_t op1, mpz_t op2)
int mpz_cmp_d (mpz_t op1, double op2)
Compare $op1$ and $op2$. Return a positive value if $op1 > op2$, zero if $op1 = op2$, or a negative value if $op1 < op2$.

Opérations arithmétiques de base

void mpz_add (mpz_t rop, mpz_t op1, mpz_t op2)
void mpz_add_ui (mpz_t rop, mpz_t op1, unsigned long int op2)
Set rop to $op1 + op2$.

void mpz_sub (mpz_t rop, mpz_t op1, mpz_t op2)
void mpz_sub_ui (mpz_t rop, mpz_t op1, unsigned long int op2)
Set rop to $op1 - op2$.

void mpz_mul (mpz_t rop, mpz_t op1, mpz_t op2)
void mpz_mul_si (mpz_t rop, mpz_t op1, long int op2)
void mpz_mul_ui (mpz_t rop, mpz_t op1, unsigned long int op2)
Set rop to $op1 \times op2$.

Arithmétique modulaire

void mpz_mod (mpz_t r, mpz_t n, mpz_t d)
unsigned long int mpz_mod_ui (mpz_t r, mpz_t n, unsigned long int d)
Set r to $n \mod d$. The sign of the divisor is ignored ; the result is always non-negative.

void mpz_tdiv_q (mpz_t q, mpz_t n, mpz_t d)
void mpz_tdiv_r (mpz_t r, mpz_t n, mpz_t d)
void mpz_tdiv_qr (mpz_t q, mpz_t r, mpz_t n, mpz_t d)
Divide n by d, forming a quotient q and/or remainder r.

void mpz_powm (mpz_t rop, mpz_t base, mpz_t exp, mpz_t mod)
void mpz_powm_ui (mpz_t rop, mpz_t base, unsigned long int exp, mpz_t mod)
Set rop to $base^{exp} \pmod{mod}$.

void mpz_gcd (mpz_t rop, mpz_t op1, mpz_t op2)
Set rop to the greatest common divisor of $op1$ and $op2$. The result is always positive even if one or both input operands are negative.

void mpz_gcdext (mpz_t g, mpz_t s, mpz_t t, mpz_t a, mpz_t b)
Set g to the greatest common divisor of a and b, and in addition set s and t to coefficients satisfying $a.s + b.t = g$. The value in g is always positive, even if one or both of a and b are negative. The values in s and t are chosen such that $|s| \leqslant |b|$ and $|t| \leqslant |a|$.

int mpz_probab_prime_p (const mpz_t n, int r)
Determine whether n is prime. Return 2 if n is definitely prime, return 1 if n is probably prime (without being certain), or return 0 if n is definitely non-prime.

This function performs some trial divisions, then r Miller-Rabin probabilistic primality tests. A higher r value will reduce the chances of a non-prime being identified as « probably prime ». A composite number will be identified as a prime with a probability of less than $4^{(-r)}$. Reasonable values of r are between 15 and 50.

```c
#include <time.h>
#include <gmp.h>
int main(void){
  gmp_randstate_t alea;
  gmp_randinit_default(alea);
  gmp_randseed_ui(alea, time(NULL));
  mpz_t x;
  mpz_init(x);
  mpz_urandomb(x, alea, 128);
  gmp_printf("Valeur de x : %Zd\n", x);
}
```

FIGURE 29 – Fabrique d'un entier GMP **x** compris aléatoirement entre 0 et $2^{128} - 1$

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <gmp.h>
mpz_t clair, chiffre, dechiffre, e, n, d;           // Variables globales

void fabrique(void){              // Fabrique d'une paire de clefs RSA (A MODIFIER)
  mpz_set_str(n, "196520034100071057065009920573", 10);
  mpz_set_str(e, "7", 10);
  mpz_set_str(d, "56148581171448620129544540223", 10);
}

int main(void){
  mpz_inits(clair, chiffre, dechiffre, (void *)NULL);
  mpz_init(n);                    // Le module de la clef publique
  mpz_init(e);                    // L'exposant de la clef publique
  mpz_init(d);                    // L'exposant de la clef privée

  mpz_set_str(clair, "4b594f544f", 16);

  /* Affichage du clair à l'aide de la fonction gmp_printf() */
  gmp_printf("Clair              : %Zd\n", clair);

  fabrique();

  /* Affichage des clefs utilisées à l'aide de la fonction gmp_printf() */
  gmp_printf("Clef publique (n) : %Zd\n", n);
  gmp_printf("Clef publique (e) : %Zd\n", e);
  gmp_printf("Clef privée (d)   : %Zd\n", d);

  /* On effectue d'abord le chiffrement RSA du clair avec la clef publique */
  mpz_powm(chiffre, clair, e, n);                    // Calcul du clair chiffré
  gmp_printf("Chiffré            : %Zd\n", chiffre);

  /* On déchiffre ensuite avec la clef privée */
  mpz_powm(dechiffre, chiffre, d, n);              // Calcul du clair déchiffré
  gmp_printf("Clair déchiffré    : %Zd\n", dechiffre);

  mpz_clears(clair, chiffre, dechiffre, n, e, d, (void *)NULL) ;
  exit(EXIT_SUCCESS);
}
```

FIGURE 30 – Programme `rsa_raw.c` à compléter

```java
import java.math.BigInteger;
import java.util.Random;

public class Alea
{
    public static void main(String[] args)
    {
        Random alea = new Random();
        BigInteger x = new BigInteger(128, alea);
        System.out.println ("Valeur de x : " + x);
    }
}
```

FIGURE 31 – Fabrique d'un entier BigInteger **x** compris aléatoirement entre 0 et $2^{128} - 1$

```java
import java.math.BigInteger;

public class RSA_raw
{
  private static BigInteger clair, chiffré, déchiffré ;
  private static BigInteger n ;      // Le module de la clef publique
  private static BigInteger e ;      // L'exposant de la clef publique
  private static BigInteger d ;      // L'exposant de la clef privée

  static void fabrique() {           // Fabrique d'une paire de clefs RSA (A MODIFIER)
    n = new BigInteger("19652003410007105706500992057 3", 10);
    e = new BigInteger("7", 10);
    d = new BigInteger("56148581171448620129544540223", 10);
  }

  public static void main(String[] args)
  {
    clair = new BigInteger("4b594f544f", 16);

    /* Affichage du clair */
    System.out.println("Clair             : " + clair);

    fabrique();

    /* Affichage des clefs utilisées */
    System.out.println("Clef publique (n) : " + n);
    System.out.println("Clef publique (e) : " + e);
    System.out.println("Clef privée (d)   : " + d);

    /* On effectue d'abord le chiffrement RSA du clair clair avec la clef publique */
    chiffré = clair.modPow(e, n);
    System.out.println("Chiffré           : " + chiffré);

    /* On déchiffre ensuite avec la clef privée */
    déchiffré = chiffré.modPow(d, n);
    System.out.println("Déchiffré         : " + déchiffré);
  }
}
```

FIGURE 32 – Programme `RSA_raw.java` à compléter

Nous appellerons comme d'habitude *Alice* la destinatrice du message chiffré et *Bernard* l'émetteur. Le protocole RSA se décompose en trois phases :

1. *Fabrication de deux clefs par Alice*, l'une publique, l'autre privée. Cette première phase se divise elle-même en trois parties :
   (a) Alice choisit deux grands nombres premiers p et q, et calcule $n = p \times q$ et $w = (p-1) \times (q-1)$.
   (b) Alice trouve deux entiers d et e inverses l'un de l'autre modulo w : $d \times e = 1 \pmod{w}$.
   (c) Alice diffuse sa clef publique (n, e) mais garde d pour elle : c'est sa clef privée !
2. *Bernard envoie à Alice un message chiffré avec la clef publique.*
   Bernard chiffre un message x, vu comme un entier compris entre 0 et $n-1$, à l'aide de la clef publique (n, e) connue de tous, en calculant le message chiffré $c = x^e \pmod{n}$, qu'il envoie à Alice.
3. *Alice reçoit le message et le déchiffre grâce à sa clef privée.*
   Alice déchiffre c reçu en calculant $x = c^d \pmod{n}$ à l'aide de la clef privée (n, d) connue par elle seule, et récupère ainsi le message x que Bernard voulait transmettre.

Exercice D.3 Fabrique d'une paire de clefs RSA Vous devez produire votre propre paire de clefs RSA et la tester en modifiant le programme `rsa_raw.c` de la Figure 30 ou du programme `RSA_raw.java` de la Figure 32 (tous deux disponibles dans l'archive `RSA.zip`). Ces programmes chiffrent puis déchiffrent le message clair codé par l'entier $x =$ **`0x4B594F544F`**, qui s'écrit aussi 323 620 918 351 en décimal, à l'aide d'une paire de clefs RSA construite par la fonction **`fabrique()`**.

Question 1. Modifiez la fonction **`fabrique()`** afin de

(a) trouver aléatoirement deux nombres premiers *distincts* p et q de 1024 bits chacun ;

(b) évaluer et afficher $n = p \times q$ et $w = (p-1) \times (q-1)$;

(c) trouver aléatoirement et afficher un entier d compris entre 1 et $w-1$ qui est *inversible modulo* w, c'est-à-dire tel que le PGCD de d et w vaille 1 ;

(d) calculer et afficher l'entier e compris entre 1 et $w-1$ qui est l'inverse de d modulo w ; c'est-à-dire $d \times e = 1 \pmod{w}$ ou encore, de manière équivalente :

$$d \times e + w \times l = 1 \text{ pour un certain entier } l \in \mathbb{Z}.$$

En C, avec GMP, vous utiliserez les fonctions **`mpz_gcd()`**, **`mpz_gcdext()`** et **`mpz_mod()`**. En Java, vous utiliserez les méthodes **`gcd()`** et **`modInverse()`** de **`BigInteger`**.

Question 2. Vérifiez que le code clair est bien égal au code déchiffré lorsque la paire de clefs que vous produisez avec la fonction **`fabrique()`** est utilisée.

```
Code clair        : 323620918351
Clef publique (n) : 160308927667628424313407702107430366 8...373
Clef publique (e) : 65537
Clef privée (d)   : 970850258499499849092749392960298954 8...233
Code chiffré      : 764129241805362696266527887558014201 5...205
Code déchiffré    : 323620918351
```

Exercice D.4 Fabrique d'une paire de clefs RSA conforme aux standards Il s'agit à présent de se conformer à une exigence supplémentaire formulée par de nombreuses autorités. Par exemple, la RFC 4871 stipule que *« the signing algorithm SHOULD use a public exponent of 65537 (section 3.3.1). »* Modifiez le programme précédent afin que la paire de clefs produite soit telle que $e = 65537$.

Pour satisfaire cette condition, recommencez le tirage des nombres premiers p et q jusqu'à ce que 65537 soit inversible modulo w, c'est-à-dire que le PGCD de e et w vaille 1. Vous en déduirez ensuite aisément la valeur de d, puisque d est l'inverse de e modulo w.

Pour gagner du temps de calcul, vous pourrez observer que 65537 est un nombre premier. Ainsi, si e n'est pas premier avec w alors e divise $p-1$ ou $q-1$. Réciproquement, si e divise $p-1$ ou si e divise $q-1$ alors e divise w et e n'est pas premier avec w. Par conséquent, p doit être un nombre premier tel que $p-1$ n'est pas divisible par e. De même pour q. On a alors directement la garantie que e est inversible modulo w.