RSA DIY with Large Numbers

Conception de la structure Grand Nombre

Notre objet `BigUInt` stocke :

- `mag` : Un array de magnitude de type `UIntArray` avec les éléments dans la base de travail, en little endian. (ex : `{0, 1, 0, 1}` en base 2 ou `{0, 1}` en base 10)
- `base` : La base de travail

La raison de stocker un tableau de magnitude et une base de travail est que nous allons analyser le String de la valeur.

Example :

```
BigUInt.valueOf("12345", radix=10)
```

Ce qui donne dans l'objet :

```
{
    "mag": {5, 4, 3, 2, 1},
    "base": 10,
}
```

Instancier `BigUInt`

Constructeur

Nous n'avons qu'un constructeur :

```kotlin
class BigUInt {
    val mag: UIntArray
    val base: UInt

    constructor(mag: UIntArray, base: UInt) {
        this.mag = mag.stripTrailingZero()
        this.base = base
    }
}
```

En Kotlin, cela se réduit à :

```kotlin
class BigUInt(mag: UIntArray, val base: UInt) {
    val mag: UIntArray

    init {
        this.mag = mag.stripTrailingZero()
    }
}
```

Parser un String (`BigUInt.valueOf`)

L'algorithme est assez simple, mais ne supporte que les bases 2 à 36 (car, en base 36, contient les caractères 0-9a-z).

En résumé :

- Vérifie s'il existe un signe et attribue en fonction de la présence de signe
- Convertit les charactères en digit en fonction de la base de travail (`radix`)

```kotlin
fun valueOf(str: String, radix: Int = DEFAULT_BASE_STRING): BigUInt {
    var i = 0
    val mag = when {
        str.first() == '-' || str.first() == '+' -> {
            i++
            UIntArray(str.length - 1)
        }
        else -> {
            UIntArray(str.length)
        }
    }

    for (j in mag.size - 1 downTo 0) {
        mag[j] = Character.digit(str[i], radix).toUInt()
        i++
    }
    return BigUInt(mag, radix.toUInt())
}
```

Note : Sur Kotlin, chaque expression de contrôle de flux permet de retourner la dernière valeur. Exemple :

```kotlin
val a = if (condition) {
    println("...")
    32  // This is returned because it is the last line of the expression
} else {
    println("...")
    12  // This is returned because it is the last line of the expression
}
```

Cela permet de remplacer l'opérateur ternaire de Java (`condition ? returnMeIfTrue : returnMeIfFalse`).

Notez que cette méthode est une **méthode factory**. Ce qui signifie qu'elle doit être stocké dans un objet factory ayant un cycle de vie de singleton.

En Kotlin, cela se résume à faire un `companion object` :

```kotlin
class BigUInt(mag: UIntArray, val base: UInt) {
    ...
    companion object {
        private const val DEFAULT_BASE_STRING = 10

        fun valueOf(str: String, radix: Int = DEFAULT_BASE_STRING): BigUInt {
            // ...
        }
    }
}
```

Ce qui permet :

```kotlin
BigUInt.valueOf("12345", 10)
```

`object` sous Kotlin est un singleton et permet de stocker des variables ou méthodes existant sur toute l'application. (En gros, c'est une méthode `static` au sens de C++, c'est-à-dire, partagé entre toutes les instances de classes).

Exemple :

```kotlin
object ImASingleton {
    fun hello() {}
}

ImASingleton.hello()
```

`companion object` associe la classe mère (ici, `BigUInt`) avec l'`object`.

```kotlin
class ImNotASingletonBut {
    companion object {  // But, this is a Singleton
        fun hello() {}
    }
}

ImNotASingletonBut.hello()
```

Comparaison

Implémenter `Comparable<BigUInt>`

Nous faisons cela, parce qu'un nombre est comparable et aidera les implémentations futures :

```
class BigUInt(mag: UIntArray, val base: UInt) : Comparable<BigUInt> {
    override fun compareTo(other: BigUInt): Int {
        TODO("Not implemented yer")
    }

    override fun equals(other: Any?): Boolean {
        TODO("Not implemented yer")
    }
}
```

`compareTo`

L'algorithme est la suivant :

- Comparer la taille des arrays de magnitude
- Si les tailles sont les mêmes, comparer les digits en partant de la fin.

Soit, sur Kotlin :

```
override fun compareTo(other: BigUInt): Int {
    return this.compareUnsignedTo(other)
}
```

```
private fun compareUnsignedTo(other: BigUInt): Int {
    return when {
        this.mag.size < other.mag.size -> -1
        this.mag.size > other.mag.size -> 1
        else -> compareMagnitudeTo(other)
    }
}
```

```
private fun compareMagnitudeTo(other: BigUInt): Int {
    // Check for the first biggest number
    for (i in mag.size - 1 downTo 0) {
        if (mag[i] < other.mag[i]) {
            return -1
        } else if (mag[i] > other.mag[i]) {
            return 1
        }
    }
    return 0
}
```

`equals`

```kotlin
override fun equals(other: Any?): Boolean {
    if (this === other) return true
    if (javaClass != other?.javaClass) return false

    other as BigUInt

    if (this.compareTo(other) != 0) return false

    return true
}
```

Opérateurs basique

`unaryMinus` et `unaryPlus`

Opérateurs `+a` et `-a`. Rien de plus simple :

```
operator fun unaryPlus() = this

operator fun unaryMinus() = this
```

`plus`

Notre implémentation contient quelques conditions pour la sécurité :

```
operator fun plus(other: BigUInt): BigUInt {
    if (base != other.base) throw NumberFormatException()
    if (this == zero) return other
    if (other == zero) return this

    val result = this addMagnitude other

    return BigUInt(result, base)
}
```

Allons voir l'implémentation de `addMagnitude`.

```
private infix fun addMagnitude(other: BigUInt): UIntArray {
    val result = UIntArray(max(mag.size, other.mag.size) + 1)
    var carry = 0uL
    var i = 0

    // Add common parts of both numbers
    while (i < mag.size && i < other.mag.size) {
        val sum: ULong = mag[i] + other.mag[i] + carry
        result[i] = (sum % base).toUInt()
        carry = sum / base
        i++
    }

    // Add the last part
    while (i < mag.size) {
        val sum: ULong = mag[i] + carry
        result[i] = (sum % base).toUInt()
        carry = sum / base
        i++
    }
    while (i < other.mag.size) {
        val sum: ULong = other.mag[i] + carry
        result[i] = (sum % other.base).toUInt()
        carry = sum / base
        i++
    }

    // Add the last carry (if exists)
```

```
    if (carry > 0u) result[i] = carry.toUInt()
    return result
}
```

Notez le `infix` à la déclaration de la fonction. Cela permet `a addMagnitude b` en plus de `a.addMagnitude(b)`.

Ici, il s'agit de l'algorithme de l'addition classique :

- `val sum = mag[i] + other.mag[i] + carry` est assez explicite, nous additionnons digits par digits.
- `result[i] = sum % base` permet d'éviter l'overflow de la base. Si cela overflow (`sum / base > 0`), alors nous mettons cet overflow dans le carry : `carry = sum / base`.

`minus`

Notre implémentation contient quelques conditions pour la sécurité :

```
operator fun minus(other: BigUInt): BigUInt {
    if (base != other.base) throw NumberFormatException()
    if (this == zero) return other
    if (other == zero) return this

    val result = this subtractMagnitude other

    return BigUInt(result, base)
}
```

Allons voir `subtractMagnitude`.

```
private infix fun subtractMagnitude(other: BigUInt): UIntArray {
    val result = UIntArray(max(mag.size, other.mag.size))
    var carry = 0uL

    val (largest, smallest) = if (this.compareUnsignedTo(other) < 0) {
        other to this
    } else {
        this to other
    }

    // Subtract common parts of both numbers
    for (i in smallest.mag.indices) {
        var sub: ULong
        if (largest.mag[i] < carry + smallest.mag[i]) {
            sub = largest.mag[i] + (largest.base - carry - smallest.mag[i])
            carry = 1u
        } else {
            sub = largest.mag[i] - smallest.mag[i] - carry
            carry = 0u
        }
        result[i] = sub.toUInt()
    }

    // Subtract the last part
    for (i in smallest.mag.size until largest.mag.size) {
        var sub: ULong
        if (largest.mag[i] < carry) {
            sub = largest.mag[i] + (largest.base - carry)
            carry = 1u
        } else {
            sub = largest.mag[i] - carry
            carry = 0u
        }
        result[i] = sub.toUInt()
    }
    return result
}
```

L'algorithme de la soustraction **non signé** est également classique et assez similaire à l'addition :

- Partir du plus petit et soustraire les parties communes.
  - `largest.mag[i] - smallest.mag[i] - carry` est assez explicite
  - `if (largest.mag[i] < smallest.mag[i] + carry)` vérifie s'il existe un carry (e.g : "sub est-il négatif ?" ou "sub est-il en underflow ?").
    - Si oui, alors on fait remonter dans les nombres positifs en ajoutant `base` et on stocke (`1u`) dans le carry.
- Finir par la dernière partie.

`times`

Rien de surprenant non plus :

```
operator fun times(other: BigUInt): BigUInt {
    if (base != other.base) throw NumberFormatException()
    if (this == zero || other == zero) return zero

    val result = UIntArray(mag.size + other.mag.size)

    // School case multiplication
    for (i in other.mag.indices) {
        var carry = 0uL
        for (j in mag.indices) {
            // Note: ULong is **necessary** to avoid overflow of other.mag[i] *
mag[j].
            val sum: ULong = result[i + j].toULong() + other.mag[i].toULong() *
mag[j].toULong() + carry
            carry = sum / base
            result[i + j] = (sum % base).toUInt()
        }
        result[i + mag.size] = carry.toUInt()
    }

    return BigUInt(result, base)
}
```

Il s'agit du cas d'école. La seule différence est `carry = result[i + j] / base` et `result[i + j] = result[i + j] % base`.

De la même manière que `plus` et `minus`, `% base` et `/ base` permet d'éviter l'overflow de la base. `% base` va faire que le nombre dépasse pas la base et `/ base` récupère le carry.

`shl` ou littéralement shift left (pas bitwise)

En little-endian, `nombre shl n` divisera le nombre par $base^n$.

L'implémentation est immédiate :

```
infix fun shl(n: Int): BigUInt {
    if (n == 0) return this
    val result = if (n < mag.size) mag.copyOfRange(n, mag.size) else
uintArrayOf(0u)
    return BigUInt(result, base)
}
```

`remShl` ou le reste du shift left

L'algorithme est choisi est le cas d'école :

```
infix fun remShl(k: Int): BigUInt {
    if (k == 0) return zero

    val divResult = this shl k
    return this - basePowK(k) * divResult
}
```

Car le reste de x/n^k est $rem = x - n^k \times \lfloor \frac{x}{n^k} \rfloor$ puisque $\text{remainder} = x - \text{other} \times \text{quotient}$.

Note : L'implémentation de `basePowK`.

```
fun basePowK(base: UInt, k: Int): BigUInt {
    val mag = UIntArray(k + 1).apply {
        set(k, 1u)
    }
    return BigUInt(mag, base)
}
```

Sur Kotlin, `apply` permet enchainer des opérations à la déclaration. Equivalent :

```
fun basePowK(base: UInt, k: Int): BigUInt {
    val mag = UIntArray(k + 1)
    mag[k] = 1u
    return BigUInt(mag, base)
}
```

`div`

`div` est nécessaire d'être implémenté afin de faire l'opérateur `rem` (ou `%`) et de calculer le `modInverse`.

Cependant, `div` est actuellement lourd à implémenter. Par conséquent, **nous essaierons de l'éviter au mieux**.

L'implémentation est l'algorithme de *Binary Search* (recherche par dichotomie).

```
operator fun div(other: BigUInt): BigUInt {
    if (base != other.base) throw NumberFormatException()
    if (other == zero) throw ArithmeticException("/ by zero")
    if (this == one || this == zero) return zero

    var left = zero
    var right = this
    var prevMid = zero

    while (true) {
        val mid = left + (right - left).divBy2()

        val productResult = other * mid

        when {
            productResult == this || prevMid == mid -> { // Exit condition: mid
= this / other.
                return mid
            }
            productResult < this -> { // mid < this / other. Too low.
                left = mid // x if after the middle.
            }
            else -> { // mid > this / other. Too high.
                right = mid // x is before the middle.
            }
        }
        prevMid = mid
    }
}
```

En effet, l'algorithme s'applique car $other \times x$ est strictement croissant et continue. Par conséquent, en appliquant cet algorithme, si $other \times mid = this$ alors $\frac{this}{other} = mid$.

L'implémentation de `divBy2()` est le cas d'école :

```
fun divBy2(): BigUInt {
    if (this == zero || this == one) return zero
    val result = mag.copyOf()

    var carry = 0u
    for (i in mag.size - 1 downTo 0) {
        result[i] = result[i] + carry
        carry = if (result[i] % 2u == 1u) base else 0u // Store carry if
remainder exist
        result[i] = result[i] shr 1 // Div by 2
    }

    return BigUInt(result, base)
}
```

Déduction à propos choix de l'algorithme Binary Search : Le choix de cet algorithme a actuellement un complexité faible : $\mathcal{O}(1)$ si `other` est un multiple de 2. **Cela influencera l'implémentation de l'algorithmie modulaire.**

`rem` ou modulo

Même implémentation de `remShl`. Il s'agit du cas d'école.

```
operator fun rem(other: BigUInt): BigUInt {
    if (base != other.base) throw NumberFormatException()
    if (other == zero) throw ArithmeticException("/ by zero")
    if (this == other || other == one) return zero

    val divResult = this / other
    return this - other * divResult
}
```

`modInverse` avec pgcd(a, n) = 1

Maintenant, que nous avons implémenté `div`, nous pouvons implémenter `modInverse` selon l'[algorithme d'Euclide étendue]() sachant `gcd(this, other) == 1` :

```
infix fun modInverse(other: BigUInt): BigUInt {
    if (other <= one) return zero
    var (oldR, r) = this to other
    var (t, tIsNegative) = zero to false
    var (oldT, oldTIsNegative) = one to false

    while (oldR > one) {
        if (r == zero) // this and other are not coprime
        return zero

        val q = oldR / r

        // (r, oldR) = (oldR - q * r, r)
        (oldR - q * r).let {
            oldR = r
            r = it
        }

        // (t, oldT) = (oldT - q * t, t)
        val qt = q * t
        val tempT = t
        val tempTIsNegative = tIsNegative
        if (tIsNegative == oldTIsNegative) {
            if (oldT > qt) { // oldT - q * t >= 0. Default case.
                t = oldT - qt
                tIsNegative = oldTIsNegative
            } else { // oldT - q * t < 0. We swap the members.
                t = qt - oldT
                tIsNegative = !tIsNegative // Switch the sign because oldT - q * t < 0
            }
        } else { // oldT and t don't have the same sign. The subtraction become an addition.
            t = oldT + qt
            tIsNegative = oldTIsNegative
        }
        oldT = tempT
        oldTIsNegative = tempTIsNegative
    }

    return if (oldTIsNegative) (other - oldT) else oldT
}
```

Convertisseurs de base

`toBase2`

Nous utilisons l'algorithme par division.

```
fun toBase2(): BigUInt {
    if (base == 2u) return this
    val size = ceil((this.mag.size + 1) * log2(base.toDouble()) + 1).toInt()
    val result = UIntArray(size)

    var i = 0
    var num = this
    while (num != zero) {
        result[i] = num.mag[0] % 2u  // num % 2
        num = num.divBy2()
        i++
    }

    return BigUInt(result, 2u)
}
```

`toBase2PowK`

Nous suspectons que l'algorithme sera plus performant avec un array de magnitude en base 2^k.

`toBase2PowK` va permettre de convertir d'un array de magnitude n vers un array de magnitude en base 2^k.

```
fun toBase2PowK(k: Int): BigUInt {
    val newBase = 1u shl k  // 2.pow(k)
    if (base == newBase) return this
    val thisBase2 = this.toBase2()
    val result = UIntArray(size = thisBase2.mag.size / k + 1)

    for (chunkIndex in result.indices) {
        for (offset in 0 until k) {  // k = chunckSize
            // result[chunkIndex] += x * 2.pow(offset)
            // x is a bit. x = thisBase2.mag[chunkIndex * k + offset]
            // If thisBase2.mag[chunkIndex * k + offset] fails, it returns 0u.
            result[chunkIndex] += thisBase2.mag.elementAtOrElse(chunkIndex * k +
offset) { 0u } shl offset
        }
    }

    return BigUInt(result, newBase)
}
```

En résumé, l'algorithme convertit en base 2 puis en base 2^k en convertissant des blocs. Exemple de base 2 vers 2^4 (hexadecimal) :

```
1010 0010
---- ----
   a    2
```

Algorithmie sous la forme de Montgomery

`montgomeryTimes`

$A \otimes B = A \cdot B \cdot r^{-1} \bmod n$

Rien d'extraordinaire. Nous suivons l'implémentation indiqué par l'algorithme de réduction de Montgomery.

```
fun montgomeryTimes(other: BigUInt, n: BigUInt, v: BigUInt): BigUInt {
    val s = this * other
    val t = (s * v) remShl n.mag.size
    val m = s + t * n
    val u = m shl n.mag.size
    return if (u >= n) u - n else u
}
```

v tel que $n \cdot v \equiv -1 \bmod r$. r tel que si $base^{k-1} \leqslant n < base^{k}$ alors $r = base^{k}$.

Notez `remShl` qui signifie "reste de shift left `n.mag.size` fois" soit "reste de $/base^{\text{n.mag.size}}$".

Tests de `montgomeryTimes`

Avant de passer à la suite, il serait intéressant de tester `montgomeryTimes` pour passer sous la forme de Montgomery et afin de montrer l'utilisation de cette méthode.

Nous rappelons que la forme de Montgomery est $\phi(a) = a \cdot r \bmod n$.

Or :

$$\begin{aligned} a \otimes r^2 \bmod n &= a \cdot r^2 \cdot r^{-1} \bmod n \\ &= a \cdot r \bmod n \\ &= \phi(a) \end{aligned}$$

Inversement :

$$\begin{aligned} \phi(a) \otimes 1 &= \phi(a) \cdot 1 \cdot r^{-1} \bmod n \\ &= a \cdot r \bmod n \cdot r^{-1} \bmod n \\ &= a \bmod n \end{aligned}$$

Donc notre test est :

```
"A to phi(A) with A = 413 * 4096 mod 3233 = 789" {
    val a = BigUInt.valueOf("413", 10).toBase2()
    val n = BigUInt.valueOf("3233", 10).toBase2()

    // Convert to base 2
    val r = BigUInt.basePowK(2u, n.mag.size)
    val rSquare = BigUInt.basePowK(2u, n.mag.size * 2) % n
    val v = r - (n modInverse r)

    val aMgy = a.montgomeryTimes(rSquare, n, v)

    aMgy shouldBe BigUInt.valueOf("789", 10).toBase2()
}
```

Note : Le test est sous format Kotest.

Nous pouvons également tester en base 10 :

```
"A to phi(A) with A = 413 * 10000 mod 3233 = 1459 in base 10" {
    val a = BigUInt.valueOf("413", 10)
    val n = BigUInt.valueOf("3233", 10)

    val r = BigUInt.basePowK(10u, n.mag.size)
    val rSquare = BigUInt.basePowK(10u, n.mag.size * 2) % n
    val v = r - (n modInverse r)

    val aMgy = a.montgomeryTimes(rSquare, n, v)

    aMgy shouldBe BigUInt.valueOf("1459", 10)
}
```

Notez v issue de l'identité de Bezout $r \cdot r' - n \cdot v = 1$ soit $n \cdot v \equiv -1 \bmod r$.

Nous pouvons également tester les 2 sens de transformation de Montgomery :

```
"phi(A) to A with A = 413 * 4096 mod 3233" {
    val a = BigUInt.valueOf("413", 10).toBase2()
    val n = BigUInt.valueOf("3233", 10).toBase2()

    val r = BigUInt.basePowK(2u, n.mag.size)
    val rSquare = BigUInt.basePowK(2u, n.mag.size * 2) % n
    val v = r - (n modInverse r)

    val aMgy = a.montgomeryTimes(rSquare, n, v)
    val aNotMgy = aMgy.montgomeryTimes(BigUInt.one(base = 2u), n, v)

    aNotMgy shouldBe a
}

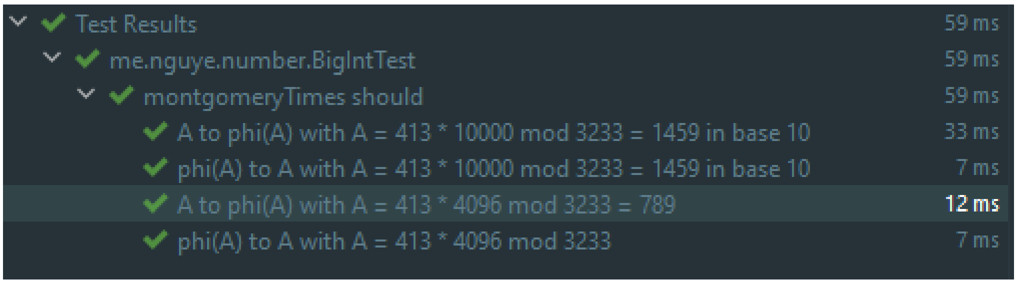
"phi(A) to A with A = 413 * 10000 mod 3233 = 1459 in base 10" {
    val a = BigUInt.valueOf("413", 10)
    val n = BigUInt.valueOf("3233", 10)

    val r = BigUInt.basePowK(10u, n.mag.size)
    val rSquare = BigUInt.basePowK(10u, n.mag.size * 2) % n
    val v = r - (n modInverse r)

    val aMgy = a.montgomeryTimes(rSquare, n, v)
    val aNotMgy = aMgy.montgomeryTimes(BigUInt.one(base = 10u), n, v)

    aNotMgy shouldBe a
}
```

Ce qui donne :



Notez bien que en base 2, le temps est divisé par 3 pour le passage sous forme de Montgomery.

En effet, remarquez la ligne `val rSquare = BigUInt.basePowK(10u, n.mag.size * 2) % n`, **nous utilisons actuellement le modulo !**

Egalement pour `modInverse` !

Pour que le calcul de $\mod n$, ou plus précisément $/n$, soit efficace, il faut que n soit en base 2^k.

`modPow`, exponentiation modulaire avec la réduction de Montgomery

L'algorithme utilisé est *square-and-multiply*.

```
fun modPow(exponent: BigUInt, n: BigUInt): BigUInt {
    if (base != n.base) throw NumberFormatException()

    val exponentBase2 = exponent.toBase2()
    val r = basePowK(n.mag.size)
    val rSquare = basePowK(n.mag.size * 2) % n

    // v*n = -1 mod r = (r - 1) mod r
    val v = r - (n modInverse r)

    // Put this in montgomery form
    val thisMgy = this.montgomeryTimes(rSquare, n, v)

    var p = r - n // 1 in montgomery form
    for (i in exponentBase2.mag.size - 1 downTo 0) {
        p = p.montgomeryTimes(p, n, v) // Square : p = p*p
        if (exponentBase2.mag[i] == 1u) {
            p = p.montgomeryTimes(thisMgy, n, v) // Multiply : p = p * a
        }
    }

    // Return the result in the standard form
    return p.montgomeryTimes(one, n, v)
}
```

Faisons ligne par ligne :

- `val exponentBase2 = exponent.toBase2()` permet exponentiation via l'algorithme square-and-multiply.
- `val r = basePowK(n.mag.size)`, car, r est choisi tel que si $base^{k-1} \leqslant n < base^k$ alors $r = base^k$ pour que r soit premier avec n.
- `val rSquare = basePowK(n.mag.size * 2) % n` est $r^2 \mod n$
- `val v = r - (n modInverse r)` est un coefficient de Bezout issue de $r \cdot r' - n \cdot v = 1$.
- ```
  var p = r - n // 1 in montgomery form
  for (i in exponentBase2.mag.size - 1 downTo 0) {
      p = p.montgomeryTimes(p, n, v) // Square : p = p*p
      if (exponentBase2.mag[i] == 1u) {
          p = p.montgomeryTimes(thisMgy, n, v) // Multiply : p = p * a
      }
  }
  ```

  est l'algorithme *square-and-multiply*.

  Elle fonctionne de la manière suivante :

  - Si x^k, alors k est décomposable en base 2. (exemple si $k = 22 = (10110)_2$) Donc, $x^k = x^{a_n 2^n + \ldots + a_0}$. ($x^{22} = x^{16+4+2} = x^{16}x^4x^2$)
  ```

- Si il y a des 1, alors il faut multiplier, car $x^a \cdot x^b = x^{a+b}$ (plus précisément : $x^{a_i 2^i} \cdot x^{a_j 2^j} = x^{a_i 2^i + a_j 2^j}$).
- Nous utilisons "square" pour multiplier l'exposant par 2 (décaler à gauche les bits de l'exposant), car $x^a \cdot x^a = x^{2a}$ (plus précisément : $x^{a_i 2^i} \cdot x^{a_i 2^i} = x^{a_i 2^{i+1}}$.
- Donc, en reprenant l'exemple :
  - On itère 5 fois. Chaque itération (squaring) multiplie les exposants par 2.
  - La première multiplication à la première itération va permettre de construire x^{16} grâce aux 4 itérations (square) restantes.
  - La deuxième multiplication à la troisième itération va permettre de construire x^4 grâce aux 2 itérations (square) restantes.
  - La troisième multiplication à la quatrième itération va permettre de construire x^2 grâce la dernière itération.

- `p.montgomeryTimes(one, n, v)` est la mise sous la forme "standard" (non-Montgomery).

Tests de `modPow`

Test 1 : Base 16

Paramètres

- Base de travail : 16 (hexadecimal)
- Entrées :

```
d =
942E315D898EA7934F2B8C233E0529E7D4E32B206679EBBA31D18F803F077C3AC9599226A027
9FACF10B9958507ACF7E2F43811E69E90A4D185E962D211240245FF4FB9873731D0655FE559E
D2FF3C9412B1A64CB3AA510A4F5DAA9C01410AED01482F493545BDE0AE978F972B39DC7691B6
7C06D645A164511EDA0CAB6A68DD
c1 =
2967CB2D53ACF0D909D95BA2D4EA606C3BD8133706E74CE9EE70D8904B30D52ED481BD957F53
3A192DF2AFE1F72FBA4366A6D690C5E0C3D3721A3C68DB0E12494DE52B25F2487C5DE449C73E
5142982877E02088274FE79AFD0C6FE037729B1266F2FA9CC577975611B34D92AE9AAC683979
7F54EB2ABDBB36D1E1D5995A7C2E
c2 =
1AD59925CA4330FE3E7CAB199E04441725CE8641B1DF11C56A4ADB0EA0AEC117DE4045C9EF25
6E6FBBD9CCC35AAB317EBD13E342E3B664369CAAF5E62358D249E939B9D1DA984BFFEE8DE1EE
87993C186FCAB0CBFF867EA69E15AE50A402FBC5818BFA9D077CAEC64F4AC96859961C294CAD
DBC24C2CFEB1E01DFB632ACFFE48
n =
10000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000DC00000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000002A7B
```

- Attendu :

```
m1 = 7b (123 en base 10)
m2 = c8 (200 en base 10)
```

Code / Protocole

```kotlin
@ExperimentalTime
@ExperimentalUnsignedTypes
fun main() {
    val workingBase = 16

    val d = BigUInt.valueOf("...", workingBase)  // Entrée tronquée
    val c1 = BigUInt.valueOf("...", workingBase)  // Entrée tronquée
    val c2 = BigUInt.valueOf("...", workingBase)  // Entrée tronquée
    val n = BigUInt.valueOf("...", workingBase)  // Entrée tronquée

    println("decrypting")
    measureTime {
        println(c1.modPow(d, n))
        println(c2.modPow(d, n))
    }.also { println("Time Elapsed : $it") }
}
```

Résultat

```
decrypting
7b
c8
Time Elapsed : 30.4s
```

Résultat satisfaisant.

Test 2 : Base 16 ramené à la base 2^{31}

Paramètres

- Base de travail : 16 (hexadecimal) ramené à 2^{31}
- Entrées :

```
d =
942E315D898EA7934F2B8C233E0529E7D4E32B206679EBBA31D18F803F077C3AC9599226A027
9FACF10B9958507ACF7E2F43811E69E90A4D185E962D211240245FF4FB9873731D0655FE559E
D2FF3C9412B1A64CB3AA510A4F5DAA9C01410AED01482F493545BDE0AE978F972B39DC7691B6
7C06D645A164511EDA0CAB6A68DD
c1 =
2967CB2D53ACF0D909D95BA2D4EA606C3BD8133706E74CE9EE70D8904B30D52ED481BD957F53
3A192DF2AFE1F72FBA4366A6D690C5E0C3D3721A3C68DB0E12494DE52B25F2487C5DE449C73E
5142982877E02088274FE79AFD0C6FE037729B1266F2FA9CC577975611B34D92AE9AAC683979
7F54EB2ABDBB36D1E1D5995A7C2E
c2 =
1AD59925CA4330FE3E7CAB199E04441725CE8641B1DF11C56A4ADB0EA0AEC117DE4045C9EF25
6E6FBBD9CCC35AAB317EBD13E342E3B664369CAAF5E62358D249E939B9D1DA984BFFEE8DE1EE
87993C186FCAB0CBFF867EA69E15AE50A402FBC5818BFA9D077CAEC64F4AC96859961C294CAD
DBC24C2CFEB1E01DFB632ACFFE48
n =
1000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000DC00000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000002A7B
```

- Attendu :

```
m1 = { 123 } (123 en base 2.pow(31))
m2 = { 200 } (200 en base 2.pow(31))
```

Code / Protocole

```kotlin
@ExperimentalTime
@ExperimentalUnsignedTypes
fun main() {
    val workingBase = 16

    val d = BigUInt.valueOf("...", workingBase).toBase2PowK(31)  // Entrée
tronquée
    val c1 = BigUInt.valueOf("...", workingBase).toBase2PowK(31)  // Entrée
tronquée
    val c2 = BigUInt.valueOf("...", workingBase).toBase2PowK(31)  // Entrée
tronquée
    val n = BigUInt.valueOf("...", workingBase).toBase2PowK(31)  // Entrée
tronquée

    println("decrypting")
    measureTime {
        println(c1.modPow(d, n))
        println(c2.modPow(d, n))
    }.also { println("Time Elapsed : $it") }
}
```

Résultat

```
decrypting
{123}
{200}
Time Elapsed : 944ms
```

Résultat très satisfaisant. **Il s'agit du résultat le plus rapide.**

Test 3 : Base 10

Paramètres

- Base de travail : 10
- Entrées :

```
d =
1040558441671077812485897526089204421214358524130047196074639508239873128211
3275916680998868588284966002871345280915424736058018571225927752937375563384
3749387184470871012615224812078370633557809434904918225321388120741945280206
1816838347990192467401138829296232117477093653198571818079774556436887188512
70877
c1 =
2907589156223685355406259912832815959018302898055210108554426270478005354953
4418578507236855720567419975163282590047789761592080601496152946952125090156
7446011587172953825114315233286969047361527760435665801422048859124438317920
7778418363139822182666461154723983793619893969217826316733888203460760986573
1118
c2 =
1884373410417546174762005634597764433919185101394525088563340762993600120692
4360108249391255994307246258972445193033936764824225343511129353256115730527
6340533720594607574733025175393648331282672638754090446864553851327686234406
1352945624304163487361558480011365417897195417316438238034880347571065633374
1640
n =
1797693134862315907729305190789024733617976978942306572734300811577326758055
0096313270847732240753602112011387987139335765878976881441662249284743063947
7074095512480796227391561801824887394139579933613278628104952355769470429079
0618088095228864239559174423176933873251711350717926983445502235717324055626
49211
```

- Attendu :

```
m1 = 123
m2 = 200
```

Code / Protocole

```kotlin
@ExperimentalTime
@ExperimentalUnsignedTypes
fun main() {
    val workingBase = 10

    val d = BigUInt.valueOf("...", workingBase)  // Entrée tronquée
    val c1 = BigUInt.valueOf("...", workingBase)  // Entrée tronquée
    val c2 = BigUInt.valueOf("...", workingBase)  // Entrée tronquée
    val n = BigUInt.valueOf("...", workingBase)  // Entrée tronquée

    println("decrypting")
    measureTime {
        println(c1.modPow(d, n))
        println(c2.modPow(d, n))
    }.also { println("Time Elapsed : $it") }
}
```

Résultat

```
decrypting
123
200
Time Elapsed : 294s
```

Résultat satisfaisant malgré le temps de traitement élevé.

Test 4 : Base 2

Paramètres

- Base de travail : 2
- Entrées :

```
d =
1001010000101110001100010101110110001001100011101010011110010011010011110010
101110001100001000110011111000000101001010011110011110101001110001100101011
0010000001100110011110011110101110111010001100011101000110001111100000000011
1111000001110111110000111010110010010101100110010010001001101010000000100111
1001111110101100111100010000101110011001010110000101000001111010110011110111
1110001011110100001110000001000111100110100111101001000010100100110100011000
0101111010010110001011010010000100010010010000000010010001011111111101001111
1011100110000111001101110011000111010000011001010101111111100101010110011110
1101001011111111001111001001010000010010101100011010011001001100101100111010
1010010100010000101001001111010111011010101010011100000000010100000100001010
1110110100000001010010000010111101001001001101010100010110111101111000001010
1110100101111000111110010111001010110011100111011100011101101001000110110110
01111110000000110110101100100010110100001011001000101000100011110110110100000
110010101011011010100110100011011101
c1 =
1010010110011111001011001011010101001110101100111100001101100100001001110110
0101011011101000101101010011101010011000000110110000111011110110000001001100
1101110000011011100111010011001110100111101110011100001101100010010000010010
1100110000110101010010111011010100100000011011110110010101011111110101001100
1110100001100100101101111100101010111111110000111101110010111110111010010000
1101100110101001101101011010010000110001011110000011000011110100110111001000
0110100011110001101000110110110000111000010010010010010100110111100101001010
1100100101111100100100100001111100010111011110010001001001110001110011111001
0100010100001010011000001010000111011111100000001000001000100000100111010011
1111100111100110101111110100001100011011111110000000110111011100101001101100
0100100110011011110010111110101001110011000101011101111001011101010110000100
0110110011010011011001001010101110100110101010110001101000001110010111100101
1111110101010011101011001010101011110110111011001101101101000111100001110101
0110011001010110100111110000101110
c2 =
1101011010101100110010010010111001010010000110011000011111110001111100111110
010101011000110011001111000000010001000100000010111001001011100111010000110010
0000110110001110111110001000111000101011010100100101011011011000011101010000
0101011101100000100010111110111100100000001000101110010011110111100100101011
01110011011111101110111101100111001100110000110101101010101011001100010111111
0101111010001001111100011010000101110001110110110011001000011011010011100101
0101011110101111001100010001101011000110100100100100111101001001110011011100
1110100011101101010011000010010111111111111101110100011011110000111101110100
0011110011001001111000001100001101111110010101011000011001011111111111000011
0011111101010011010011110000101011010111001010000101001000000000101111011110
0010110000001100010111111101010011101000001110111110010101110110001100100111
1010010101100100101101000010110011001011000011100001010010100110010101101110
1101111000010010011000010110011111110101100011110000000011101111110110110001
100101010110011111111111001001000
```

```
n =
1000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000110111000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000010101001111011
```

- Attendu :

```
m1 = 1111011 (123 en base 2)
m2 = 11001000 (200 en base 2)
```

Code / Protocole

```
@ExperimentalTime
@ExperimentalUnsignedTypes
fun main() {
    val workingBase = 2

    val d = BigUInt.valueOf("...", workingBase)  // Entrée tronquée
    val c1 = BigUInt.valueOf("...", workingBase)  // Entrée tronquée
    val c2 = BigUInt.valueOf("...", workingBase)  // Entrée tronquée
    val n = BigUInt.valueOf("...", workingBase)  // Entrée tronquée

    println("decrypting")
    measureTime {
        println(c1.modPow(d, n))
        println(c2.modPow(d, n))
    }.also { println("Time Elapsed : $it") }
}
```

Résultat

```
decrypting
1111011
11001000
Time Elapsed : 487s
```

Résultat satisfaisant mais un temps trop élevé. Cela peut s'expliquer, car, en base 2, nous utilisons aucune optimisation de l'ordinateur.

`Rsa`

Le comportement de RSA est immédiat :

```
@ExperimentalUnsignedTypes
object Rsa {
    fun decrypt(c: BigUInt, d: BigUInt, n: BigUInt) = c.modPow(d, n)

    fun encrypt(m: BigUInt, e: BigUInt, n: BigUInt) = m.modPow(e, n)
}
```

Benchmark

Après les tests précédents, nous avons remarqué que la base 10 est plutôt lente, mais les bases en 2^k ont l'air d'être rapide selon nos hypothèses. (Car la division euclidienne est basé sur l'algorithme binary search).

Par conséquent, nous allons tester différente base 2^k

Benchmark : $2790^{413} \mod 3233$

Paramètres

- Base de travail : 2^k avec k variant entre $[1, 64]$
- Entrée :

  ```
  c = 2790
  d = 413
  n = 3233
  ```

- Attendu : `m = 65`

Code / Protocole

```
    "(2790, 413, 3233)" when {
        "decrypt" should {
            "returns 65 in base 2^k from 1 to 31" {
                checkAll(Exhaustive.ints(1..3)) { iteration ->
                    val writer =
File("output_2790_413_3233_2to64_$iteration.txt").printWriter()
                    writer.use { out ->
                        checkAll(Exhaustive.ints(1..31)) { k ->
                            val c = BigUInt.valueOf("101011100110",
2).toBase2PowK(k)
                            val d = BigUInt.valueOf("110011101",
2).toBase2PowK(k)
                            val n = BigUInt.valueOf("110010100001",
2).toBase2PowK(k)
                            val expected = BigUInt.valueOf("1000001",
2).toBase2PowK(k)


                            var result: BigUInt
                            measureTime {
                                result = Rsa.decrypt(c, d, n)
                            }.also {
```

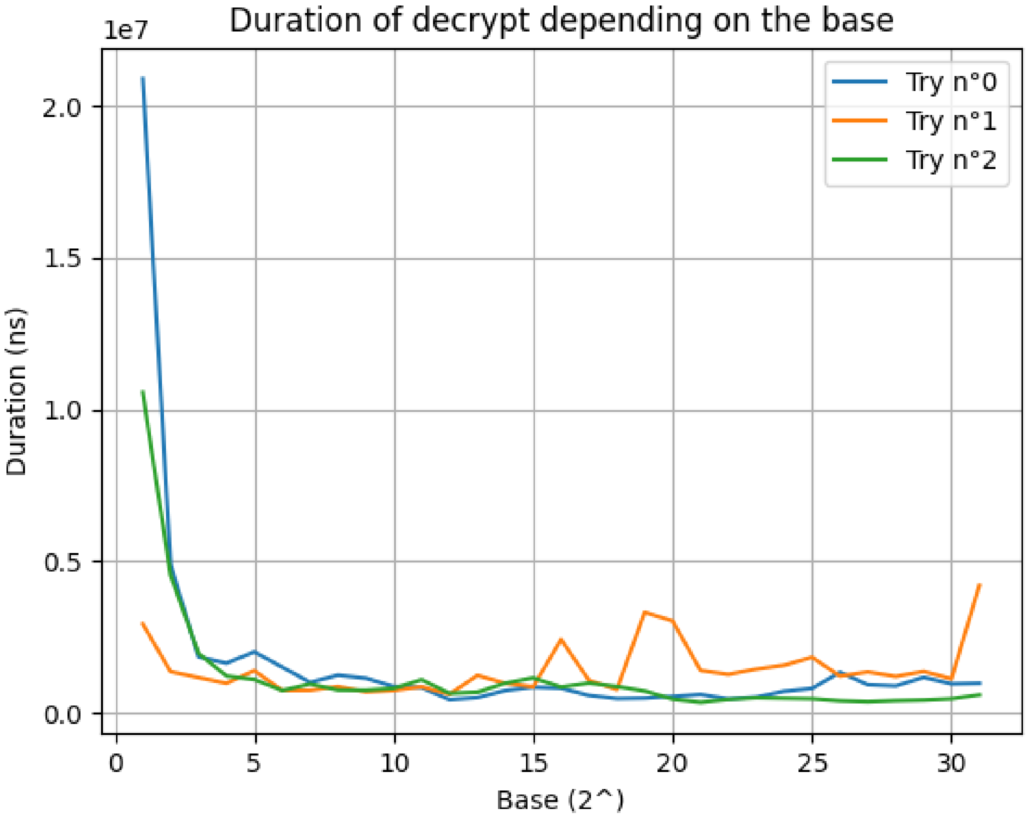
```
                                println("Base 2^$k, m = $result, Time elapsed:
$it")
                                out.println("$k\t${it.toLongNanoseconds()}")
                            }

                            result shouldBe expected
                        }
                    }
                }
            }
        }
    }
```

Note : Le test est sous format Kotest.

Résultats

```
Base 2^1, m = 1000001, Time elapsed: 14.3ms
Base 2^2, m = 1001, Time elapsed: 4.27ms
Base 2^3, m = 101, Time elapsed: 1.59ms
Base 2^4, m = 41, Time elapsed: 1.44ms
Base 2^5, m = 21, Time elapsed: 1.10ms
Base 2^6, m = {1, 1}, Time elapsed: 828us
Base 2^7, m = {65}, Time elapsed: 840us
Base 2^8, m = {65}, Time elapsed: 672us
Base 2^9, m = {65}, Time elapsed: 640us
Base 2^10, m = {65}, Time elapsed: 695us
Base 2^11, m = {65}, Time elapsed: 739us
Base 2^12, m = {65}, Time elapsed: 948us
Base 2^13, m = {65}, Time elapsed: 15.3ms
Base 2^14, m = {65}, Time elapsed: 416us
Base 2^15, m = {65}, Time elapsed: 547us
Base 2^16, m = {65}, Time elapsed: 551us
Base 2^17, m = {65}, Time elapsed: 1.18ms
Base 2^18, m = {65}, Time elapsed: 446us
Base 2^19, m = {65}, Time elapsed: 378us
Base 2^20, m = {65}, Time elapsed: 491us
Base 2^21, m = {65}, Time elapsed: 591us
Base 2^22, m = {65}, Time elapsed: 594us
Base 2^23, m = {65}, Time elapsed: 687us
Base 2^24, m = {65}, Time elapsed: 624us
Base 2^25, m = {65}, Time elapsed: 653us
Base 2^26, m = {65}, Time elapsed: 740us
Base 2^27, m = {65}, Time elapsed: 554us
Base 2^28, m = {65}, Time elapsed: 561us
Base 2^29, m = {65}, Time elapsed: 704us
Base 2^30, m = {65}, Time elapsed: 626us
Base 2^31, m = {65}, Time elapsed: 745us
```

En l'exécutant 3 fois :



La première courbe *Try n°0* peut être plus élevé que les autres du fait que Java est compilé en Just-In-Time.

Benchmark Grand Nombre

Paramètres

- Base de travail : 2^k avec k variant entre $[1, 64]$
- Entrée :

```
c =
1010010110011111001011001011010101001110101100111100001101100100001001110110
0101011011101000101101010011101010011000000110110000111011110110000001001100
1101110000011011100111010011001110100111101110011100001101100010010000010010
1100110000110101010010111011010100100000011011110110010101011111110101001100
1110100001100100101101111100101010111111100001111101110010111110111010010000
1101100110101001101101011010010000110001011110000011000011110100110111001000
0110100011110001101000110110110000111000010010010010100110111100101001010
1100100101111100100100100001111100010111011110010001001001110001110011111001
0100010100001010011000001010000111011111110000000100000100010000010011101001 1
1111100111100110101111110100001100011011111111000000011011101110010100110110 0
0100100110011011110010111110101001110011000101011101111001011101010110000100
0110110011010011011001001010101110100110101010110001101000001110010111100101
1111110101010011101011001010101011110110111011001101101101000111100001110101
01100110010101101001111100001011 10
d =
1001010000101110001100010101110110001001100011101010011110010011010011110010
1011100011000010001100111110000000101001010011110011111010100111000110010101 1
0010000000110011001111001111010111011101000110001110100011000111110000000001 1
1111000000111011111100001110101100100101011001100100100010011010100000000100111
1001111110101100111100010000101110011001010110000101000000111101011001111011 1
1110001011110100001110000000100011110011010011110100100001010010011010001100 0
0101111010010110001011010010000100010010010000000010010001011111111101001111
1011100110000111001101110011000111010000001100101010111111111001010101100111 10
1101001011111111001111001001010000010010101100011010011001001100101100111010
1010010100010000101001001111010111011010101010011100000000010100000100001010
1110110100000000101001000000101111010010010011010101000101101111011110000010 10
1110100101111000111110010111001010110011100111011100011101101001000110110110
0111110000000110110101100100010110100001011001000101000100011110110110100000
11001010101101101010011010001101 1101
n =
1000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000110111000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000010101001111011
```

- Attendu : `m = 1111011 (123 en binaire)`

Code / Protocole

```
    "(Big, Big, Big)" when {
        "decrypt" should {
            "returns a good result in base 2^k from 1 to 31" {
                checkAll(Exhaustive.ints(1..3)) { iteration ->
                    val writer =
File("output_Big_2to64_$iteration.txt").printWriter()
                    writer.use { out ->
                        checkAll(Exhaustive.ints(1..31)) { k ->
                            val c = BigUInt.valueOf("...", 2).toBase2PowK(k)
                            val d = BigUInt.valueOf("...", 2).toBase2PowK(k)
                            val n = BigUInt.valueOf("...", 2).toBase2PowK(k)
                            val expected = BigUInt.valueOf("1111011",
2).toBase2PowK(k)

                            var result: BigUInt
                            measureTime {
                                result = Rsa.decrypt(c, d, n)
                            }.also {
                                println("Base 2^$k, m = $result, Time elapsed:
$it")

                                out.println("$k\t${it.toLongNanoseconds()}")
                            }

                            result shouldBe expected
                        }
                    }
                }
            }
        }
    }
```

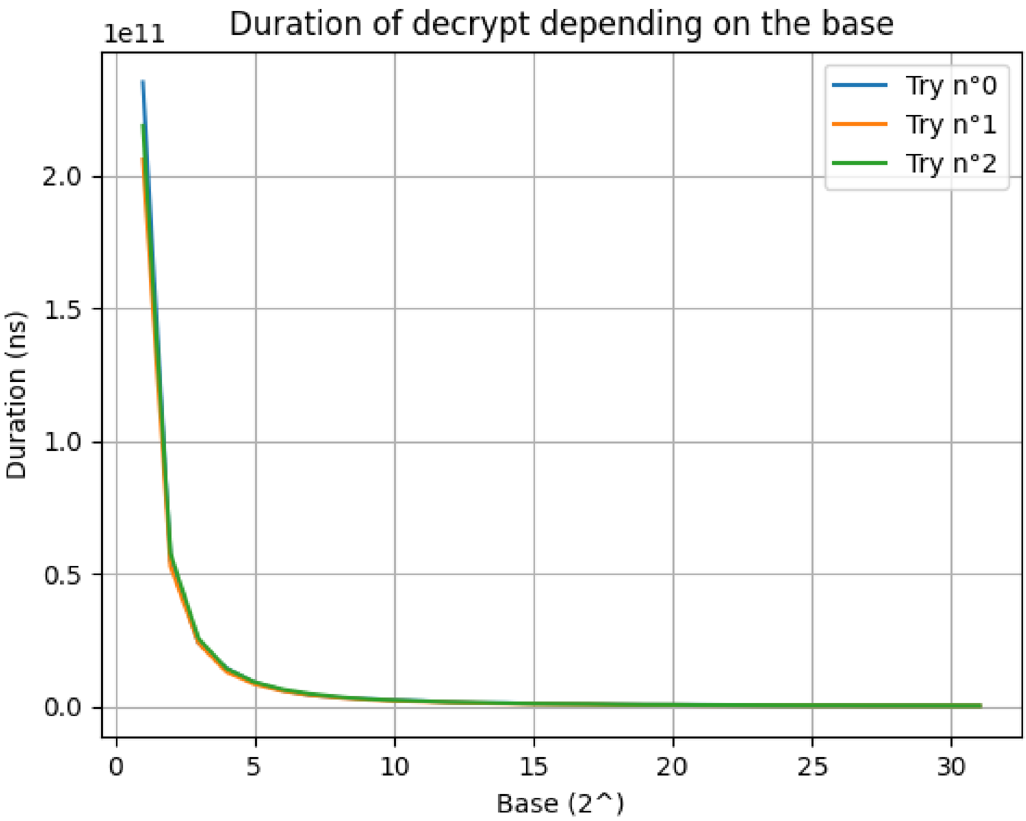
Note : Le test est sous format Kotest.

Résultats

```
Base 2^1, m = 1111011, Time elapsed: 235s
Base 2^2, m = 1323, Time elapsed: 54.8s
Base 2^3, m = 173, Time elapsed: 23.9s
Base 2^4, m = 7b, Time elapsed: 14.4s
Base 2^5, m = 3r, Time elapsed: 8.99s
Base 2^6, m = {59, 1}, Time elapsed: 6.07s
Base 2^7, m = {123}, Time elapsed: 4.29s
Base 2^8, m = {123}, Time elapsed: 3.42s
Base 2^9, m = {123}, Time elapsed: 2.88s
Base 2^10, m = {123}, Time elapsed: 2.42s
Base 2^11, m = {123}, Time elapsed: 2.04s
Base 2^12, m = {123}, Time elapsed: 1.69s
Base 2^13, m = {123}, Time elapsed: 1.36s
Base 2^14, m = {123}, Time elapsed: 1.32s
Base 2^15, m = {123}, Time elapsed: 1.11s
Base 2^16, m = {123}, Time elapsed: 1.02s
Base 2^17, m = {123}, Time elapsed: 931ms
Base 2^18, m = {123}, Time elapsed: 776ms
Base 2^19, m = {123}, Time elapsed: 669ms
Base 2^20, m = {123}, Time elapsed: 626ms
```

```
Base 2^21, m = {123}, Time elapsed: 570ms
Base 2^22, m = {123}, Time elapsed: 503ms
Base 2^23, m = {123}, Time elapsed: 430ms
Base 2^24, m = {123}, Time elapsed: 387ms
Base 2^25, m = {123}, Time elapsed: 364ms
Base 2^26, m = {123}, Time elapsed: 359ms
Base 2^27, m = {123}, Time elapsed: 308ms
Base 2^28, m = {123}, Time elapsed: 305ms
Base 2^29, m = {123}, Time elapsed: 291ms
Base 2^30, m = {123}, Time elapsed: 275ms
Base 2^31, m = {123}, Time elapsed: 260ms
```

En l'exécutant 3 fois :

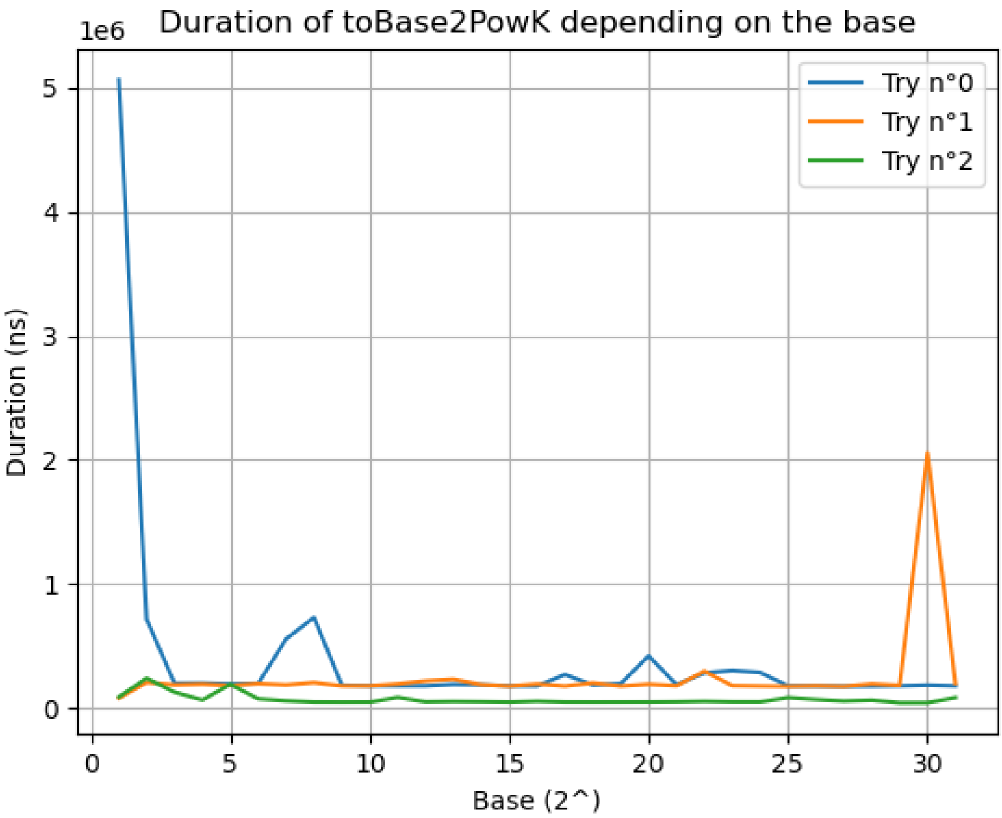


Le résultat est assez claire : plus la base est grande et sous format 2^k, plus les opérations sont rapides.

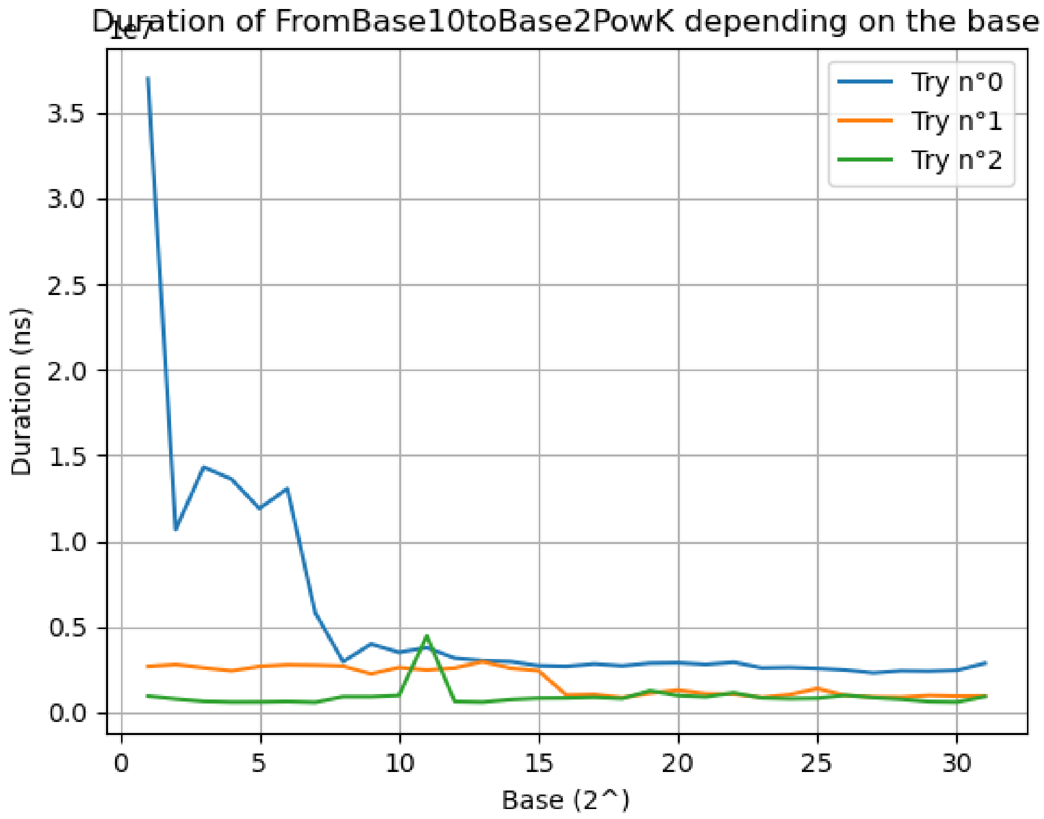
Conclusion

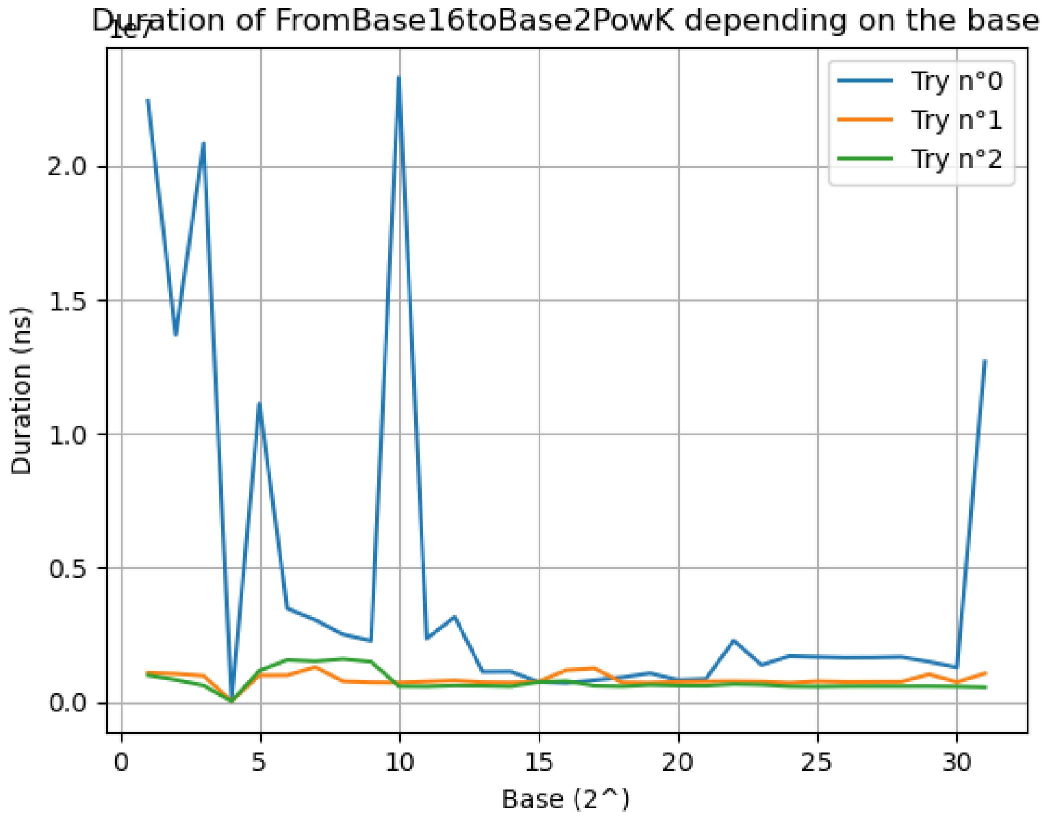
Nous pouvons conclure que la base est grande et sous format 2^k, plus les opérations sont rapides. Cependant, comme nos ordinateurs sont en 64bits, 2^{31} est le maximum (sinon l'implémentation de la multiplication overflow).

De plus, le temps de conversion de la base 2 vers la base 2^k n'est pas très couteuse (< 1 ms pour un mot de 1024 bits) :



Le temps de conversion de la base n vers la base 2^k est également courte (~6 ms depuis la base 10, ~5 ms depuis la base 16 pour un mot de 1024 bits).





Il est donc préférable d'appliquer `.toBase2PowK(31)` avant de faire `modPow`.