RSA DIY with Large Numbers

Conception de la structure Grand Nombre

Notre objet `BigInt` 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
- `sign` : Le signe en `Int` (-1 est négatif, 0 est zéro)

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

Example :

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

Ce qui donne dans l'objet :

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

La raison du type du signe dans `Int` est que, pendant le calcul de `times` (ou `*`), le `sign` est `` `this.sign * other.sign` ``.

Instancier `BigInt`

Constructeur

Nous n'avons qu'un constructeur :

```
class BigInt {
    val mag: UIntArray
    val base: UInt
    val sign: Int

    constructor(mag: UIntArray, base: UInt, sign: Int = 1) {
        this.mag = mag.stripTrailingZero()
        this.base = base
        this.sign = if (this.mag.size == 1 && this.mag.first() == 0u) 0 else
sign  // Sécurité du signe
    }
}
```

Ce qui se réduit à :

```
class BigInt(mag: UIntArray, val base: UInt, sign: Int = 1) {
    val mag: UIntArray
    val sign: Int

    init {
        this.mag = mag.stripTrailingZero()
        this.sign = if (this.mag.size == 1 && this.mag.first() == 0u) 0 else
sign  // Sécurité du signe
    }
}
```

Parser un String (`BigInt.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 le signe en fonction de la présence de signe
- Convertit les charactères en digit en fonction de la base de travail (`radix`)

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

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

Si vous n'êtes pas familier à la déclaration par destructuration : Pair, Control Flow. En résumé, il est possible de faire une assignation parallèle avec `val (a, b) = value1 to value2`.

Egalement, chaque expression de contrôle de flux permet de retourner la dernière valeur. Exemple :

```
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 **statique**, voire plus précisément 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` :

```
class BigInt(mag: UIntArray, val base: UInt, sign: Int = 1) {
    ...
    companion object {
        private const val DEFAULT_BASE_STRING = 10

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

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

Ce qui permet :

```
BigInt.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 :

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

ImASingleton.hello()
```

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

```
class ImNotASingletonBut {
    companion object {  // This is a Singleton
        fun hello() {}
    }
}

ImNotASingletonBut.hello()
```

Comparaison

Implémenter `Comparable<BigInt>`

Parce qu'un nombre est comparable :

```
class BigInt(mag: UIntArray, val base: UInt, sign: Int = 1) : Comparable<BigInt>
{
    override fun compareTo(other: BigInt): Int {
        TODO("Not implemented yer")
    }

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

`compareTo`

L'algorithme est la suivant :

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

Soit, sur Kotlin :

```
override fun compareTo(other: BigInt): Int {
    return when {
        sign < other.sign -> -1 // This is negative, and other is positive
        sign > other.sign -> 1 // This is positive, and other is negative
        sign == -1 && other.sign == -1 -> -1 * this.compareUnsignedTo(other) //
Both are negative.
        else -> this.compareUnsignedTo(other) // Both are positive
    }
}
```

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

```
private fun compareMagnitudeTo(other: BigInt): Int {
    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`

```
override fun equals(other: Any?): Boolean {
    if (this === other) return true  // Instance check
    if (javaClass != other?.javaClass) return false  // Class type check

    other as BigInt  // Cast check

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

    return true
}
```

Opérateurs basique

`unaryMinus` et `unaryPlus`

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

```
operator fun unaryPlus() = this

operator fun unaryMinus() = BigInt(mag, base, -sign)
```

`plus`

Notre implémentation contient quelques conditions pour la sécurité :

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

    if (this.sign != other.sign) {
        // Subtract instead
        val cmp = this.compareUnsignedTo(other)
        if (cmp == 0) return zero
        val result = if (cmp > 0) this.subtractMagnitude(other) else
other.subtractMagnitude(this)
        val resultSign = if (cmp == sign) 1 else -1
        return BigInt(result, base, resultSign)
    }

    val result = this addMagnitude other  // Implémentation

    return BigInt(result, base, sign)
}
```

Allons voir l'implémentation de `addMagnitude`.

```
private infix fun addMagnitude(other: BigInt): 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 = 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 = mag[i] + carry
        result[i] = (sum % base).toUInt()
        carry = sum / base
```

```
            i++
        }
        while (i < other.mag.size) {
            val sum = 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: BigInt): BigInt {
    if (base != other.base) throw NumberFormatException()
    if (this == zero) return BigInt(other.mag, base, -other.sign)
    if (other == zero) return this

    if (this.sign != other.sign) {
        // Add instead
        val result = this.addMagnitude(other)
        return BigInt(result, base, sign)
    }

    val result = this subtractMagnitude other
    val resultSign = if (this < other) -1 else 1

    return BigInt(result, base, resultSign)
}
```

Allons voir `subtractMagnitude`.

```
private infix fun subtractMagnitude(other: BigInt): 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
        carry = if (largest.mag[i] < carry + smallest.mag[i]) {
            sub = largest.mag[i] + (largest.base - carry - smallest.mag[i])
            1u
        } else {
            sub = largest.mag[i] - smallest.mag[i] - carry
            0u
        }
        result[i] = sub.toUInt()
    }

    // Subtract the last part
    for (i in smallest.mag.size until largest.mag.size) {
        var sub: ULong
        carry = if (largest.mag[i] < carry) {
            sub = largest.mag[i] + (largest.base - carry)
            1u
        } else {
            sub = largest.mag[i] - 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 remontrer dans les nombres positifs en faisans `sub += base` et on stocke (`1u`) dans le carry.
- Finir par la dernière partie.

`times`

Rien de surprenant non plus :

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

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

    // Basic multiplication
    for (i in other.mag.indices) {
        var carry = 0uL
        for (j in mag.indices) {
            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 BigInt(result, base, sign * other.sign)
}
```

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): BigInt {
    if (n == 0) return this
    val result = if (n < mag.size) mag.copyOfRange(n, mag.size) else
uintArrayOf(0u)
    return BigInt(result, base, sign)
}
```

`remShl` ou le reste du shift left

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

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

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

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

Note : L'implémentation de `basePowK`.

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

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

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

`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*.

```
operator fun div(other: BigInt): BigInt {
    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

        // If result is the same.
        if (productResult == this || prevMid == mid) {
            return mid
        }
        if (productResult < this) {
            left = mid
        } else {
            right = mid
        }
        prevMid = mid
    }
}
```

En effet, l'algorithme s'applique car les données ($other \times x$ est croissants). 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(): BigInt {
    if (this == zero || this == one) return zero
    val result = mag.copyOf()  // Pour avoir un array de même

    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
        result[i] = result[i] shr 1  // Div by 2
    }

    return BigInt(result, base, sign)
}
```

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: BigInt): BigInt {
    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: BigInt): BigInt {
    var (oldR, r) = this to other
    var (oldT, t) = one to zero

    if (other == one) return zero

    while (r > one) {
        val q = oldR / r

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

        (oldT - q * t).let {  // it = oldT - q * t
            oldT = t
            t = it
        }
    }

    if (t < zero) t += other

    return t
}
```

Notez `.let`, cela permet de faire `(oldR, r) = (r, oldR - q * r)` en séquentiel sans passer par une variable temporaire.

Algorithmie sous la forme de Mongomery

`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: BigInt, n: BigInt, v: BigInt): BigInt {
    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
}
```

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 = BigInt.valueOf("413", 10).toBase2()
    val n = BigInt.valueOf("3233", 10).toBase2()

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

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

    aMgy shouldBe BigInt.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 = BigInt.valueOf("413", 10)
    val n = BigInt.valueOf("3233", 10)

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

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

    aMgy shouldBe BigInt.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 = BigInt.valueOf("413", 10).toBase2()
    val n = BigInt.valueOf("3233", 10).toBase2()

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

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

    aNotMgy shouldBe a
}

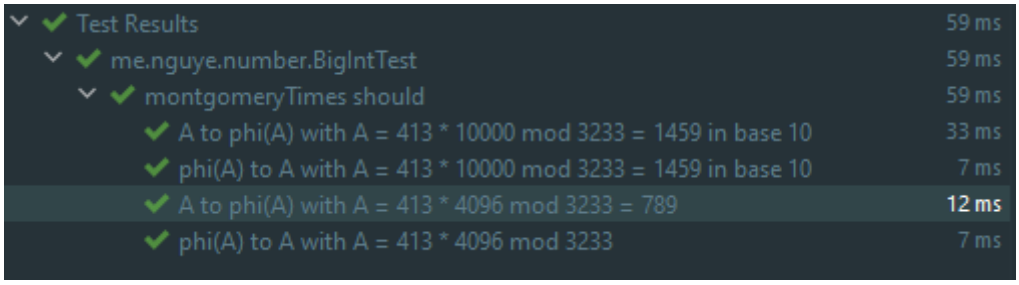
"phi(A) to A with A = 413 * 10000 mod 3233 = 1459 in base 10" {
    val a = BigInt.valueOf("413", 10)
    val n = BigInt.valueOf("3233", 10)

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

    val aMgy = a.montgomeryTimes(rSquare, n, v)
    val aNotMgy = aMgy.montgomeryTimes(BigInt.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 = BigInt.basePowK(1Ou, 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: BigInt, n: BigInt): BigInt {
    if (base != n.base) throw NumberFormatException()

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

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

    val thisMgy = this.montgomeryTimes(rSquare, n, v)

    var p = r - n  // 1 sous forme Montgomery
    for (i in exponentBase2.mag.size - 1 downTo 0) {
        p = p.montgomeryTimes(p, n, v)  // Square
        if (exponentBase2.mag[i] == 1u) {
            p = p.montgomeryTimes(thisMgy, n, v)  // Multiply
        }
    }
    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` afin de mettre sous forme de Montgomery.
- `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 sous forme Montgomery
  for (i in exponentBase2.mag.size - 1 downTo 0) {
      p = p.montgomeryTimes(p, n, v)  // Square
      if (exponentBase2.mag[i] == 1u) {
          p = p.montgomeryTimes(thisMgy, n, v)  // Multiply
      }
  }
  ```

  est l'algorithme square-and-multiply.

  Elle fonctionne de la manière suivante :

  - Si x^k, en mettant k est décomposable en base 2. (exemple si $k = 22 = (10110)_2$) Donc, $x^k = x^{a_n 2^n + \ldots + a_0}$. ($x^{22} = x^{16+4+2} = x^{16} x^4 x^2$)
  - 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}.
        - La deuxième multiplication à la troisième itération va permettre de construire x^4
        - La troisième multiplication à la quatrième itération va permettre de construire x^2
- `p.montgomeryTimes(one, n, v)` est la mise sous la forme "standard".

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 =
10000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000DC000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000002A7B
```

- Attendu :

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

Code / Protocole

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

    val d = BigInt.valueOf("...", workingBase)  // Entrée tronquée
    val c1 = BigInt.valueOf("...", workingBase)  // Entrée tronquée
    val c2 = BigInt.valueOf("...", workingBase)  // Entrée tronquée
    val n = BigInt.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 : 40.2s
```

Résultat satisfaisant.

Test 2 : Base 16 ramené à la base 65536

Paramètres

- Base de travail : 16 (hexadecimal) ramené à 65536
- Entrées :

```
d =
942E315D898EA7934F2B8C233E0529E7D4E32B206679EBBA31D18F803F077C3AC9599226A027
9FACF10B9958507ACF7E2F43811E69E90A4D185E962D211240245FF4FB9873731D0655FE559E
D2FF3C9412B1A64CB3AA510A4F5DAA9C01410AED01482F493545BDE0AE978F972B39DC7691B6
7C06D645A164511EDA0CAB6A68DD
c1 =
2967CB2D53ACF0D909D95BA2D4EA606C3BD8133706E74CE9EE70D8904B30D52ED481BD957F53
3A192DF2AFE1F72FBA4366A6D690C5E0C3D3721A3C68DB0E12494DE52B25F2487C5DE449C73E
5142982877E02088274FE79AFD0C6FE037729B1266F2FA9CC577975611B34D92AE9AAC683979
7F54EB2ABDBB36D1E1D5995A7C2E
c2 =
1AD59925CA4330FE3E7CAB199E04441725CE8641B1DF11C56A4ADB0EA0AEC117DE4045C9EF25
6E6FBBD9CCC35AAB317EBD13E342E3B664369CAAF5E62358D249E939B9D1DA984BFFEE8DE1EE
87993C186FCAB0CBFF867EA69E15AE50A402FBC5818BFA9D077CAEC64F4AC96859961C294CAD
DBC24C2CFEB1E01DFB632ACFFE48
n =
10000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000DC0000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000002A7B
```

- Attendu :

```
m1 = { 123 } (123 en base 65536)
m2 = { 200 } (200 en base 65536)
```

Code / Protocole

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

    val d = BigInt.valueOf("...", workingBase).toBase2PowK(16)  // Entrée
tronquée
    val c1 = BigInt.valueOf("...", workingBase).toBase2PowK(16)  // Entrée
tronquée
    val c2 = BigInt.valueOf("...", workingBase).toBase2PowK(16)  // Entrée
tronquée
    val n = BigInt.valueOf("...", workingBase).toBase2PowK(16)  // 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 : 5.00s
```

Résultat satisfaisant.

Test 3 : Base 10

Paramètres

- Base de travail : 10
- Entrées :

```
d =
104055844167107781248589752608920442121435852413004719607463950823987312821132759166809988685882849660028713452809154247360580185712259277529373755633843749387184470871012615224812078370633557809434904918225321388120741945280206181683834799019246740113882929623211747709365319857181807977455643688718851270877
c1 =
290758915622368535540625991283281595901830289805521010855442627047800535495344185785072368557205674199751632825900477897615920806014961529469521250901567446011587172953825114315233286969047361527760435665801422048859124438317920777841836313982218266646115472398379361989396921782631673388820346076098657311118
c2 =
188437341041754617476200563459776443391918510139452508856334076299360012069243601082493912559943072462589724451930339367648242253435111293532561157305276340533720594607574733025175393648331282672638754090446864553851327686234406135294562430416348736155848001136541789719541731643823803488034757106563337416401
n =
179769313486231590772930519078902473361797697894230657273430081157732675805500963132708477322407536021120113879871393357658789768814416622492847430639477074095512480796227391561801824887394139579933613278628104952355769470429079061808809522886423955917442317693387325171135071792698344550223571732405562649211
```

- Attendu :

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

Code / Protocole

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

    val d = BigInt.valueOf("...", workingBase)  // Entrée tronquée
    val c1 = BigInt.valueOf("...", workingBase)  // Entrée tronquée
    val c2 = BigInt.valueOf("...", workingBase)  // Entrée tronquée
    val n = BigInt.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 : 305s
```

Résultat satisfaisant malgré le temps de traitement élevé.

Test 4 : Base 2

Paramètres

- Base de travail : 2
- Entrées :

d =
100101000010111000110001010111011000100110001110101001111001001101001111001
101110001100001000110011111000000101001010011110011111010100111000110010101
001000000110011001111001111010111011101000110001110100011000111110000000011
111100000111011111000011101011001001010110011001001000100110101000000100111
100111111010110011110001000010111001100101011000010100000111101011001111011
111000101111010000111000000100011110011010011110100100001010010011010001100
010111101001011000101101001000010001001001000000001001000101111111101001111
101110011000011100110111001100011101000001100101010111111110010101011001111
110100101111111100111100100101000001001010110001101001100100110010110011101
101001010001000010100100111101011101101010101001110000000001010000010000101
111011010000000101001000001011110100100100110101010001011011110111100001010
111010010111100011111001011100101011001110011101110001110110100100011011011
011111000000011011010110010001011010000101100100010100010001111011011010000
110010101011011010100110100011011101
c1 =
101001011001111100101100101101010100111010110011111000011011001000010011101101
010101101110100010110101001110101001100000011011000011101111011000000010011001
110111000001101110011101001100111010011111011100111000011011000100100000100101
110011000011010101001011101101010010000001101111011001010101111111010100110001
111010000110010010110111111001010101111111000011111011100101111101110100100000
110110011010100110110101101001000011000101111000001100001111010011011100100000
011010001111000110100011011011000011100001001001001001010011011110010100101010
110010010111110010010010000111110001011101111001000100100111000111001111100101
010001010000010100110000010100001110111111000000010000010001000001001110100110
111110011110011010111111010000110001101111111000000011011101110010100110110000
010010011001101111001011111010100111001100010101110111100101110101011000010000
011011001101001101100100101010111010011010101011000110100000111001011110010101
111111010101001110101100101010101111011011101100110110110100011110000111010101
011001100101011010011111000010111010
c2 =
110101101010110011001001001011100101001000011001100001111111000111110011111010
010101011000110011001111100000010001000100000101110010010111001110100001100100
000011011000111011111000100011100010101101010010010101101101100001110101000000
010101110110000010001011111011110010000000100010111001001111011110010010101100
011100110111111011101111011001110011001100000110101101010101011001100010111111
010111101000100111110001101000010111000111011011001100100001101101001110010101
010101111010111100110001000110101100011010010010010011110100100111001101110000
111010001110110101001100001001011111111111111011101000110111100001111011101000
001111001100100111100000110000110111111100101010110000110010111111111111000011
001111110101001101001111000010101101011100101000010100100000000101111101111000
001011000000110001011111110101001110100000111011111001010111011000110010011100
101001010110010010110100000101100110010110000111000010100101001100101010110111000
110111100001001001100001011001111111010110001111000000000111011111101101100001
100101010110011111111111001001000

```
n =
10000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000010101001111011
```

- Attendu :

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

Code / Protocole

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

    val d = BigInt.valueOf("...", workingBase)  // Entrée tronquée
    val c1 = BigInt.valueOf("...", workingBase)  // Entrée tronquée
    val c2 = BigInt.valueOf("...", workingBase)  // Entrée tronquée
    val n = BigInt.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 : 495s
```

Résultat satisfaisant mais un temps trop élevé.

`Rsa`

Le comportement de RSA est immédiat :

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

    fun encrypt(m: BigInt, e: BigInt, n: BigInt) = 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 = BigInt.valueOf("101011100110",
2).toBase2PowK(k)
                            val d = BigInt.valueOf("110011101",
2).toBase2PowK(k)
                            val n = BigInt.valueOf("110010100001",
2).toBase2PowK(k)
                            val expected = BigInt.valueOf("1000001",
2).toBase2PowK(k)

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

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

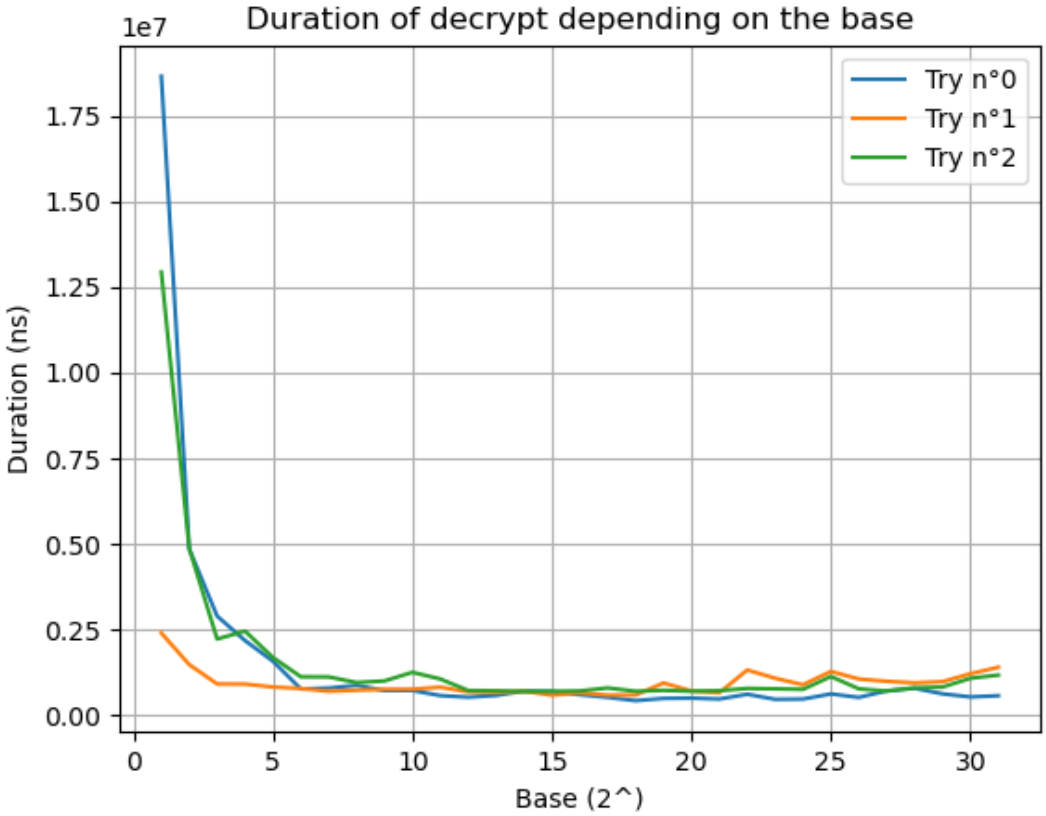
            result shouldBe expected
        }
    }
  }
 }
}
```

Note : Le test est sous format Kotest.

Résultats

```
Base 2^1, m = 1000001, Time elapsed: 23.1ms
Base 2^2, m = 1001, Time elapsed: 9.03ms
Base 2^3, m = 101, Time elapsed: 3.61ms
Base 2^4, m = 41, Time elapsed: 3.44ms
Base 2^5, m = 21, Time elapsed: 2.61ms
Base 2^6, m = {1, 1}, Time elapsed: 3.02ms
Base 2^7, m = {65}, Time elapsed: 939us
Base 2^8, m = {65}, Time elapsed: 2.67ms
Base 2^9, m = {65}, Time elapsed: 1.08ms
Base 2^10, m = {65}, Time elapsed: 951us
Base 2^11, m = {65}, Time elapsed: 1.02ms
Base 2^12, m = {65}, Time elapsed: 933us
Base 2^13, m = {65}, Time elapsed: 1.09ms
Base 2^14, m = {65}, Time elapsed: 823us
Base 2^15, m = {65}, Time elapsed: 857us
Base 2^16, m = {65}, Time elapsed: 883us
Base 2^17, m = {65}, Time elapsed: 913us
Base 2^18, m = {65}, Time elapsed: 823us
Base 2^19, m = {65}, Time elapsed: 694us
Base 2^20, m = {65}, Time elapsed: 947us
Base 2^21, m = {65}, Time elapsed: 775us
Base 2^22, m = {65}, Time elapsed: 1.69ms
Base 2^23, m = {65}, Time elapsed: 1.68ms
Base 2^24, m = {65}, Time elapsed: 1.20ms
Base 2^25, m = {65}, Time elapsed: 1.64ms
Base 2^26, m = {65}, Time elapsed: 1.80ms
Base 2^27, m = {65}, Time elapsed: 1.42ms
Base 2^28, m = {65}, Time elapsed: 1.08ms
Base 2^29, m = {65}, Time elapsed: 1.37ms
Base 2^30, m = {65}, Time elapsed: 968us
Base 2^31, m = {65}, Time elapsed: 923us
```

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
0100010100001010011000001010000111011111100000001000001000100000100111010011
1111100111100110101111111010000110001101111111000000011011101110010100110110
0100100110011011110010111110101001110011000101011101111001011101010110000100
0110110011010011011001001010101110100110101010110001101000000111001011110010
1111111010101001110101100101010101111011011101100110110110100011110000111010
0110011001010110100111110000101110
d =
1001010000101110001100010101110110001001100011101010011110010011010011110010
1011100011000010001100111110000000101001010011110011111010100111000110010101
0010000000110011001111001111010111011101000110001110100011000111110000000001
1111000001110111110000111010110010010101100110010010001001101010000000100111
1001111110101100111100010000101110011001010110000101000001111010110011110111
1110001011110100001110000000100011110011010011110100100001010010011010001100
0101111010010110001011010010000100010010010000000010010001011111111101001111
1011100110000111001101110011000111010000011001010101111111100101010110011110
1101001011111111001111001001010000010010101100011010011001001100101100111010
1010010100010000101001001111010111011010101010011100000000001010000001000010
1110110100000001010010000010111101001001001101010100010110111101111000001010
1110100101111000111110010111001010110011100111011100011101101001000110110110
0111110000000110110101100100010110100001011001000101000100011110110110100000
1100101010110110101001101000110111
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 = BigInt.valueOf("...", 2).toBase2PowK(k)
                            val d = BigInt.valueOf("...", 2).toBase2PowK(k)
                            val n = BigInt.valueOf("...", 2).toBase2PowK(k)
                            val expected = BigInt.valueOf("1111011",
2).toBase2PowK(k)

                            var result: BigInt
                            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
                        }
                    }
                }
            }
        }
    }
```

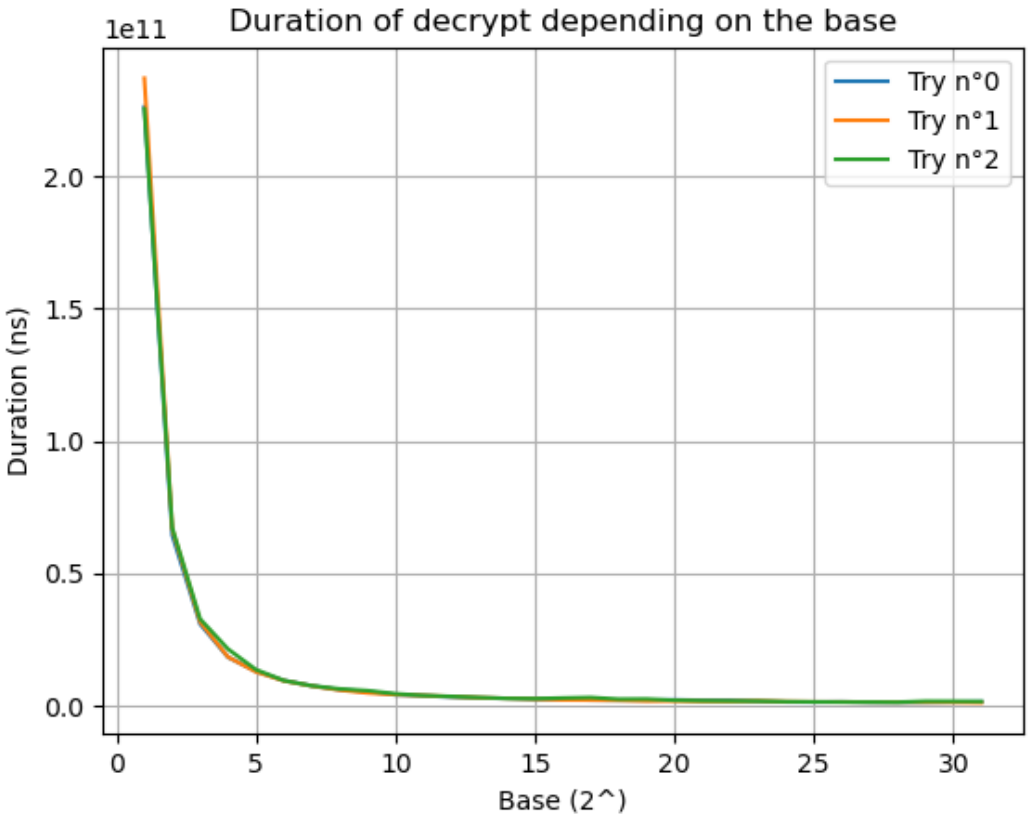
Note : Le test est sous format Kotest.

Résultats

```
Base 2^1, m = 1111011, Time elapsed: 226s
Base 2^2, m = 1323, Time elapsed: 64.4s
Base 2^3, m = 173, Time elapsed: 30.9s
Base 2^4, m = 7b, Time elapsed: 18.3s
Base 2^5, m = 3r, Time elapsed: 13.0s
Base 2^6, m = {59, 1}, Time elapsed: 9.52s
Base 2^7, m = {123}, Time elapsed: 7.53s
Base 2^8, m = {123}, Time elapsed: 6.03s
Base 2^9, m = {123}, Time elapsed: 5.32s
Base 2^10, m = {123}, Time elapsed: 4.16s
Base 2^11, m = {123}, Time elapsed: 3.81s
Base 2^12, m = {123}, Time elapsed: 3.30s
Base 2^13, m = {123}, Time elapsed: 2.95s
Base 2^14, m = {123}, Time elapsed: 2.79s
Base 2^15, m = {123}, Time elapsed: 2.48s
Base 2^16, m = {123}, Time elapsed: 2.38s
Base 2^17, m = {123}, Time elapsed: 2.14s
Base 2^18, m = {123}, Time elapsed: 2.09s
Base 2^19, m = {123}, Time elapsed: 2.09s
Base 2^20, m = {123}, Time elapsed: 2.21s
```

```
Base 2^21, m = {123}, Time elapsed: 2.05s
Base 2^22, m = {123}, Time elapsed: 2.02s
Base 2^23, m = {123}, Time elapsed: 1.69s
Base 2^24, m = {123}, Time elapsed: 1.45s
Base 2^25, m = {123}, Time elapsed: 1.42s
Base 2^26, m = {123}, Time elapsed: 1.49s
Base 2^27, m = {123}, Time elapsed: 1.30s
Base 2^28, m = {123}, Time elapsed: 1.27s
Base 2^29, m = {123}, Time elapsed: 1.31s
Base 2^30, m = {123}, Time elapsed: 1.41s
Base 2^31, m = {123}, Time elapsed: 1.52s
```

En l'exécutant 3 fois :

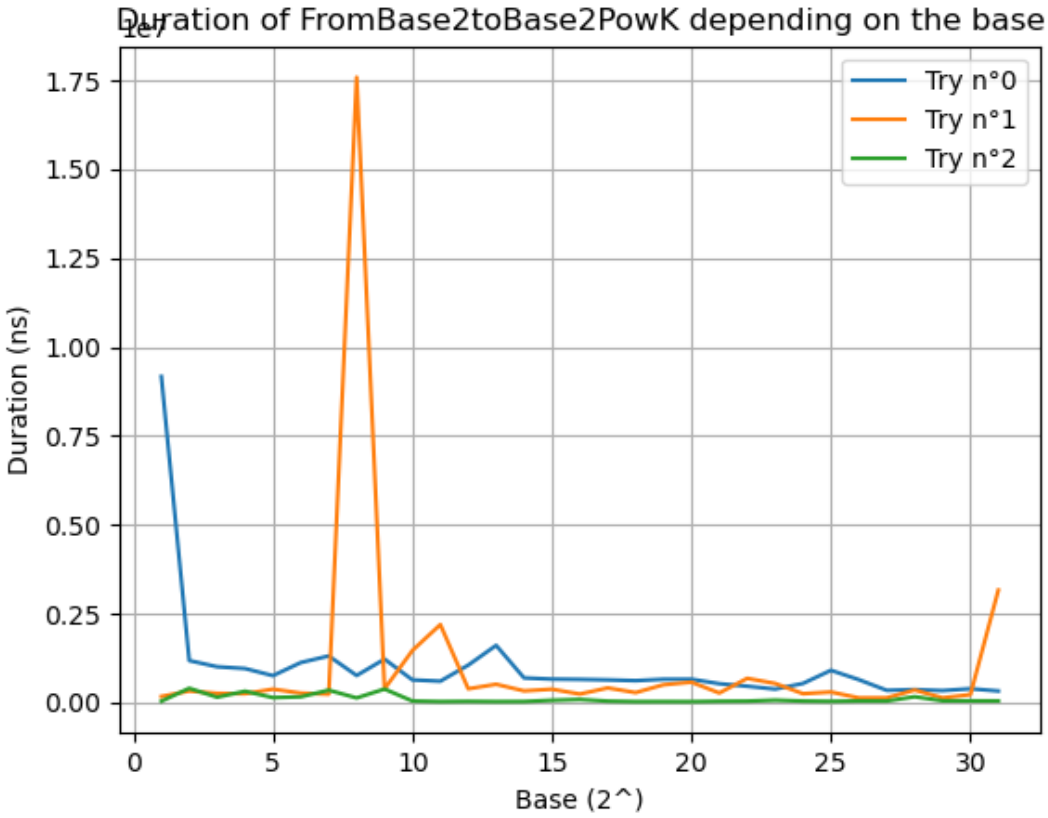


Le résultat est assez claire : plus la base est grande et sous format 2^k, plus les opérations sont rapides.

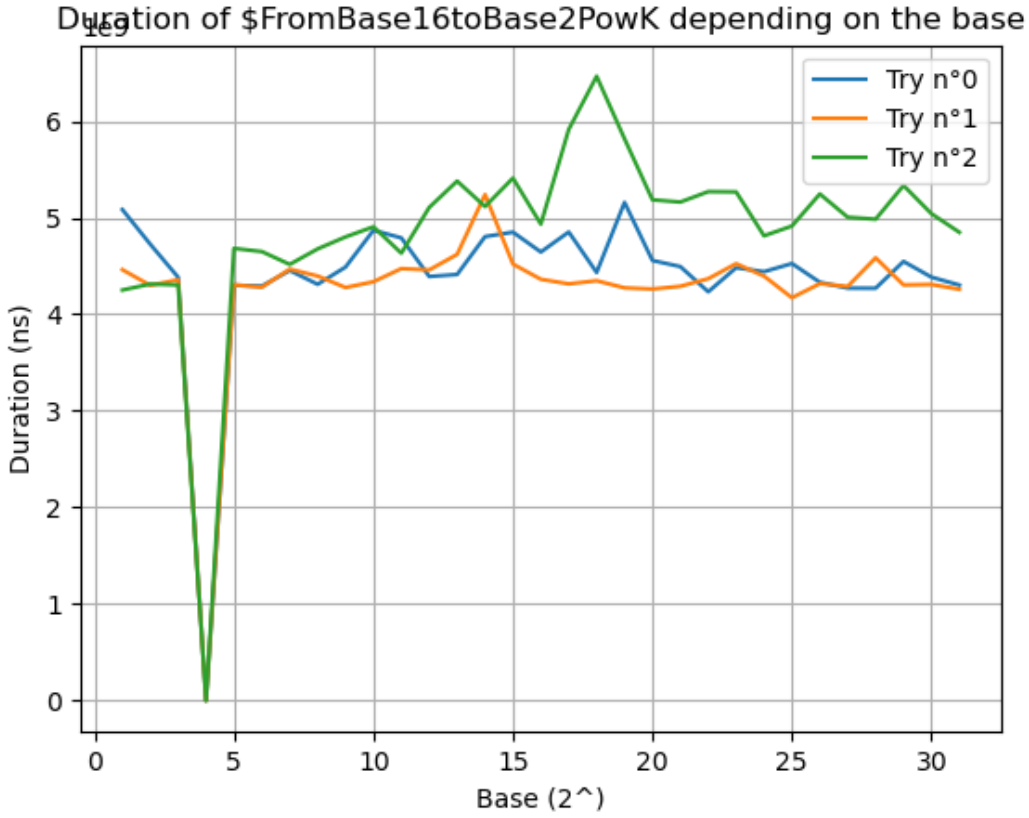
Conclusion

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 :



Par contre, le temps de conversion de la base n vers la base 2^k est assez longue (5s environ pour une conversion d'un mot 1025 bits en base 16 vers une base 2^k, 7s environ pour une conversion d'un mot 1025 bits en base 10 vers une base 2^k). En effet, pour convertir vers la base 2^k, le nombre est d'abord convertit en base 2. Or, cette conversion utilise des divisions et des modulos, ce qui n'est pas optimal.



Une voie de développement serait d'optimiser cette conversion pour mieux supporter les conversions de base de 2^j à 2^k.