

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.

Exemple :

```
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  
    }  
}
```

En Kotlin, cela 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 caractères 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 NEGATIVE
        }
        str.first() == '+' -> {
            i++
            UIntArray(str.length - 1) to POSITIVE
        }
        else -> {
            UIntArray(str.length) to POSITIVE
        }
    }

    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
        private const val POSITIVE = 1
        private const val NEGATIVE = -1
        private const val NEUTRAL = 0

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

```

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 { // But, this is a Singleton
        fun hello() {}
    }
}

ImNotASingletonBut.hello()

```

Comparaison

Implémenter Comparable<BigInt>

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

```
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 -> NEGATIVE // This is negative, and other is positive
        sign > other.sign -> POSITIVE // This is positive, and other is negative
        sign == NEGATIVE && other.sign == NEGATIVE -> -
        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 -> NEGATIVE
        this.mag.size > other.mag.size -> POSITIVE
        else -> compareMagnitudeTo(other)
    }
}
```

```

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

```

equals

```

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

    other as BigInt

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

    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) POSITIVE else NEGATIVE
        return BigInt(result, base, resultSign)
    }

    val result = this addMagnitude other

    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: 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: 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) NEGATIVE else POSITIVE

    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
        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.toInt()
  }
  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: BigInt): BigInt {
    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).toInt()
        }
        result[i + mag.size] = carry.toInt()
    }

    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 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 $rem = x - n^k \times \lfloor \frac{x}{n^k} \rfloor$ puisque $remainder = x - other \times quotient$.

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 enchaîner 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* (recherche par dichotomie).

```
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

        when {
            productResult == this || prevMid == mid -> { // Exit condition: mid
                = this / other.
                return mid
            }
            productResult < this -> { // mid < this / other. Too low.
                left = mid // x is 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(): BigInt {
    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 BigInt(result, base, sign)
}

```

Dédution 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'arithmétique modulaire.**

rem ou modulo

Même implémentation de `remSh1`. 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.

Convertisseurs de base

toBase2

Nous utilisons l'algorithme par division.

```
fun toBase2(): BigInt {
    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 BigInt(result, 2u, sign)
}
```


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: 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  
}
```

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

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

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 $n \cdot v \equiv -1 \pmod{r}$.

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 :

✓ Test Results	59 ms
✓ me.nguye.number.BigIntTest	59 ms
✓ montgomeryTimes should	59 ms
✓ A to phi(A) with A = 413 * 10000 mod 3233 = 1459 in base 10	33 ms
✓ phi(A) to A with A = 413 * 10000 mod 3233 = 1459 in base 10	7 ms
✓ A to phi(A) with A = 413 * 4096 mod 3233 = 789	12 ms
✓ phi(A) to A with A = 413 * 4096 mod 3233	7 ms

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(10u, n.mag.size * 2) % n`,
nous utilisons actuellement le modulo !

Egalement pour `modInverse` !

Pour que le calcul de $\text{mod } n$, 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

    // 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 $base^{k-1} \leq n < base^k$ alors $r = base^k$ pour que r soit premier avec n .
- `val rSquare = basePowK(n.mag.size * 2) % n` est $r^2 \bmod n$
- `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,  
 $x^k = x^{a_n 2^n + \dots + a_0}$ . ( $x^{22} = x^{16+4+2} = x^{16} x^4 x^2$ )

- 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  $x^{16}$  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 :

[illegible]

- 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 : 30.4s
```

Résultat satisfaisant.

```
@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 : 2.39s
```

Résultat satisfaisant.

## 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

```
@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 : 282s
```

Résultat satisfaisant malgré le temps de traitement élevé.

## Test 4 : Base 2

### Paramètres

- Base de travail : 2
- Entrées :

```
d =
1001010000101110001100010101110110001001100011101010011110010011010011110010
1011100011000010001100111110000001010010100111100111110101001110001100101011
001000000110011001111001111010111011101000110001110100011000111110000000011
111100000110111110000111010110010010101100110010010001001101010000000100111
1001111110101100111100010000101110011001010110000101000001111010110011110111
1110001011110100001110000001000111100110100111101001000010100100110100011000
0101111010010110001011010010000100010010000000010010001011111111101001111
101110011000011100110111001100011101000001100101010111111100101010110011110
1101001011111111001111001001010000010010101100011010011001001100101100111010
10100101000100001010010011110101110110101010011100000000010100000100001010
1110110100000001010010000010111101001001001101010100010110111101111000001010
1110100101111000111110010111001010110011100111011100011101101001000110110110
0111110000000110110101100100010110100001011001000101000100011110110110100000
110010101011011010100110100011011101

c1 =
1010010110011111001011001011010101001110101100111100001101100100001001110110
0101011011101000101101010011101010011000000110110000111011110110000001001100
1101110000011011100111010011001110100111101110011100001101100010010000010010
1100110000110101010010111011010100100000011011110110010101011111110101001100
1110100001100100101101111100101010111111000011110111001011110111010010000
1101100110101001101101011010010000110001011110000011000011110100110111001000
0110100011110001101000110110110000111000010010010010010100110111100101001010
1100100101111100100100100001111100010111011110010001001001110001110011111001
010001010000101001100000101000011101111100000001000001000100000100111010011
111110011110011010111111010000110001101111110000000110111011100101001101100
0100100110011011110010111110101001110011000101011101111001011101010110000100
0110110011010011011001001010101110100110101010110001101000001110010111100101
1111110101010011101011001010101011110110111011001101101101000111100001110101
0110011001010110100111110000101110

c2 =
1101011010101100110010010010111001010010000110011000011111110001111100111110
0101010110001100110011110000001000100010000010111001001011100111010000110010
0000110110001110111110001000111000101011010100100101011011011000011101010000
0101011101100000100010111110111100100000001000101110010011110111100100101011
01110011011111011101111011001110011001100001101011010101011001100010111111
0101111010001001111100011010000101110001110110110011001000011011010011100101
0101011110101111001100010001101011000110100100100100111101001001110011011100
111010001110110101001100001001011111111111101110100011011110000111101110100
0011110011001001111000001100001101111110010101011000011001011111111111000011
0011111101010011010011110000101011010111001010000101001000000001011111011110
0010110000001100010111111101010011101000001110111110010101110110001100100111
1010010101100100101101000010110011001011000011100001010010100110010101101110
1101111000010010011000010110011111110101100011110000000011101111110110110001
100101010110011111111111001001000
```

- Attendu :

### Code / Protocole

## Résultat

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: 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 : $2790^{413} \bmod 3233$

#### 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..31)) { 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)

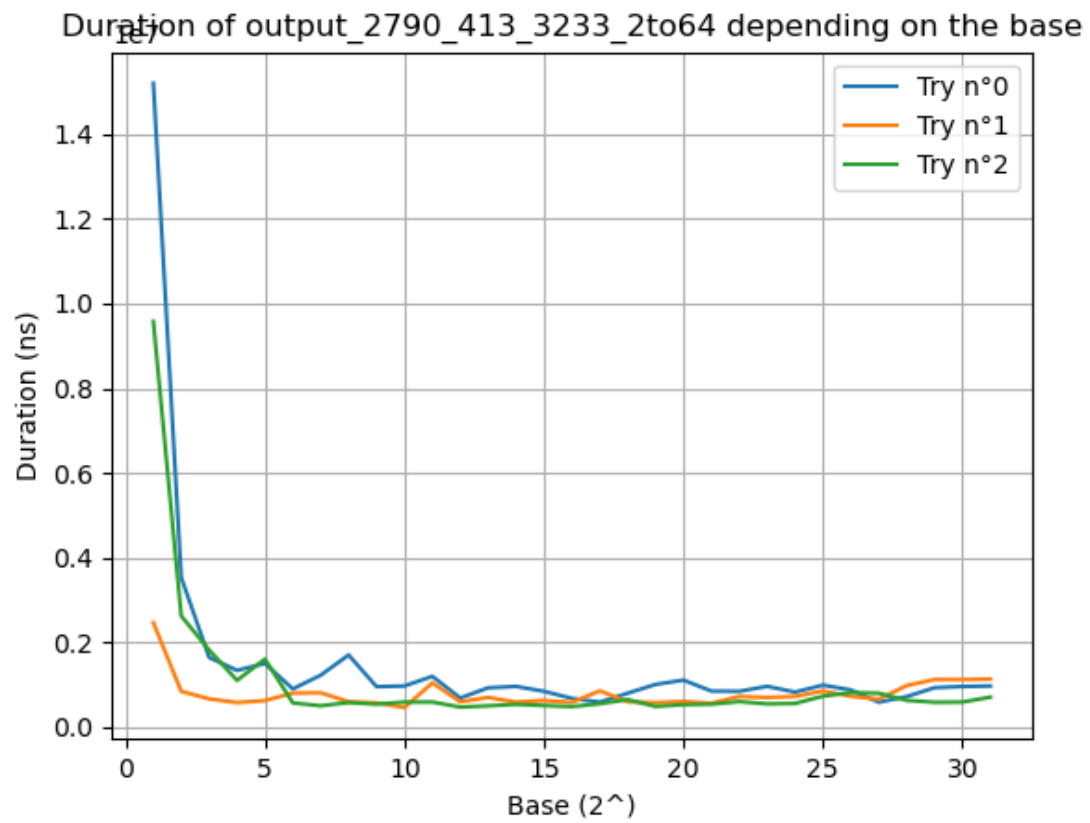
 val n = BigInt.valueOf("110010100001",
2).toBase2PowK(k)

 val expected = BigInt.valueOf("1000001",
2).toBase2PowK(k)

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

En l'exécutant 3 fois :





La première courbe *Try n°0* peut être plus élevée 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 :

[illegible]

- 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..31)) { 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

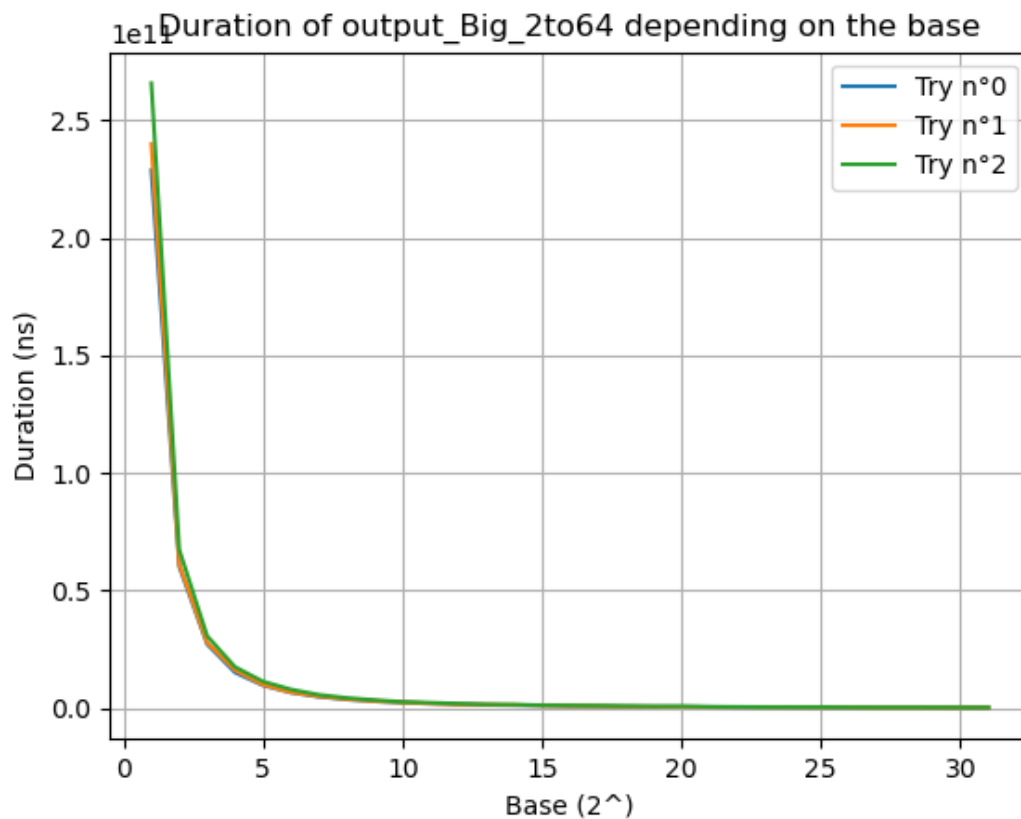
```
Base 2^1, m = 1111011, Time elapsed: 229s
Base 2^2, m = 1323, Time elapsed: 60.3s
Base 2^3, m = 173, Time elapsed: 27.3s
Base 2^4, m = 7b, Time elapsed: 15.3s
Base 2^5, m = 3r, Time elapsed: 9.77s
Base 2^6, m = {59, 1}, Time elapsed: 6.71s
Base 2^7, m = {123}, Time elapsed: 4.92s
Base 2^8, m = {123}, Time elapsed: 3.79s
Base 2^9, m = {123}, Time elapsed: 3.00s
Base 2^10, m = {123}, Time elapsed: 2.40s
Base 2^11, m = {123}, Time elapsed: 2.09s
Base 2^12, m = {123}, Time elapsed: 1.71s
Base 2^13, m = {123}, Time elapsed: 1.47s
Base 2^14, m = {123}, Time elapsed: 1.31s
Base 2^15, m = {123}, Time elapsed: 1.16s
Base 2^16, m = {123}, Time elapsed: 998ms
Base 2^17, m = {123}, Time elapsed: 931ms
Base 2^18, m = {123}, Time elapsed: 750ms
Base 2^19, m = {123}, Time elapsed: 704ms
Base 2^20, m = {123}, Time elapsed: 656ms
```

```

Base 2^21, m = {123}, Time elapsed: 624ms
Base 2^22, m = {123}, Time elapsed: 520ms
Base 2^23, m = {123}, Time elapsed: 484ms
Base 2^24, m = {123}, Time elapsed: 440ms
Base 2^25, m = {123}, Time elapsed: 398ms
Base 2^26, m = {123}, Time elapsed: 400ms
Base 2^27, m = {123}, Time elapsed: 389ms
Base 2^28, m = {123}, Time elapsed: 361ms
Base 2^29, m = {123}, Time elapsed: 342ms
Base 2^30, m = {123}, Time elapsed: 338ms
Base 2^31, m = {123}, Time elapsed: 295ms

```

En l'exécutant 3 fois :

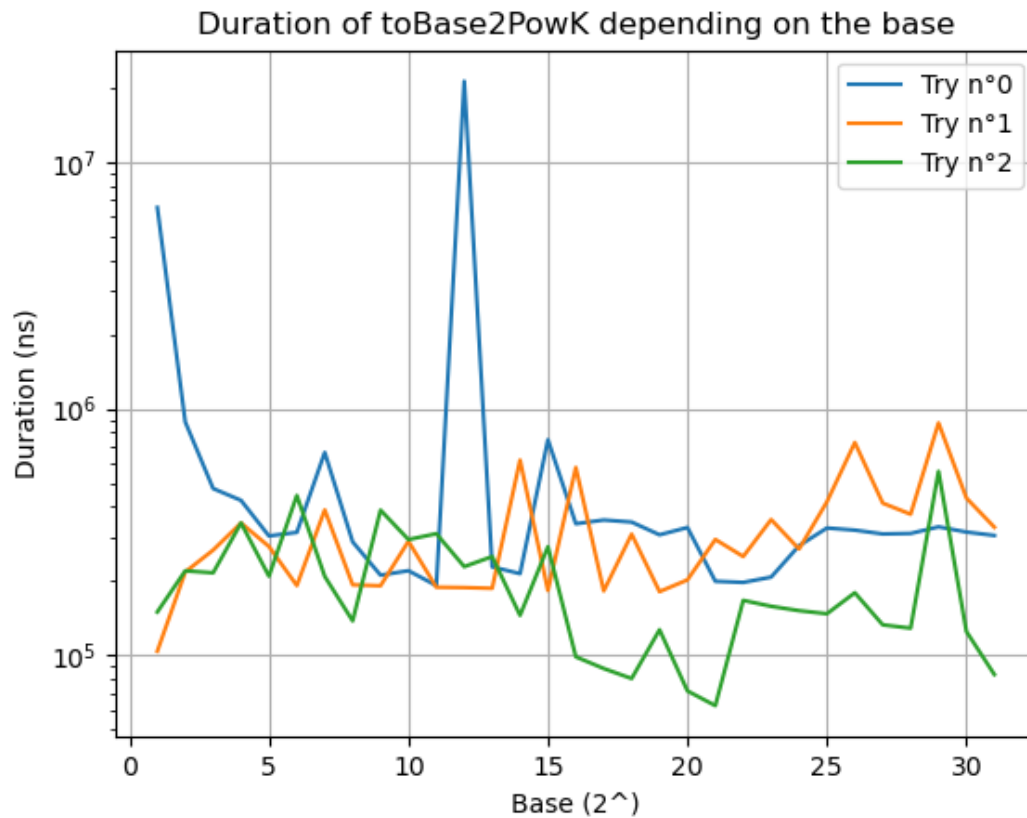


Le résultat est assez claire : plus la base est grande et sous format  $2^k$ , plus les opérations sont rapides.

# Conclusion

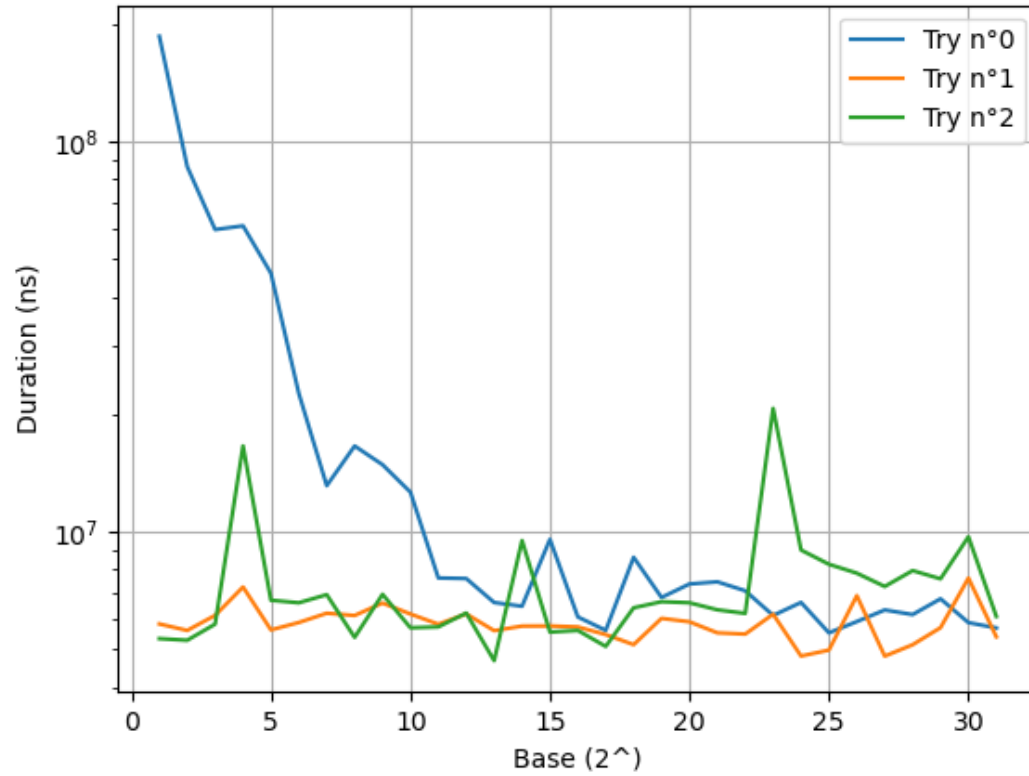
Nous pouvons conclure que la base est grande et sous format  $2^k$ , plus les opérations sont rapides. Cependant, comme nos ordinateurs sont en 64bits,  $2^{31}$  est le maximum (sinon l'implémentation de la multiplication overflow).

De plus, le temps de conversion de la base 2 vers la base  $2^k$  n'est pas très coûteuse (< 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).

Duration of FromBase10toBase2PowK depending on the base



Duration of FromBase16toBase2PowK depending on the base

