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 de taille maximum $2^{31} = 2147483648$, en little endian.

(Nous ne prenons pas 2^{32} simplement parce qu'il y a des pertes de performances avec de nombreuses conversions vers `ULong`)

Si nous devons analyser un String, celui-ci sera convertit en base 2^{31}

Example :

```
BigUInt.valueOf("2147483648", radix=10)
```

Ce qui donne dans l'objet :

```
{
    "mag": {0, 1},
}
```

Instancier `BigUInt`

Constructeur

Nous n'avons qu'un constructeur :

```kotlin
class BigUInt {
    val mag: UIntArray

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

En Kotlin, cela se réduit à :

```kotlin
class BigUInt(mag: UIntArray) {
    val 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 l'espace en fonction de la présence de signe.
- Convertit les charactères en digit et les stocke dans un array.
- Convertit cet array de base `radix` en base 2^{31}

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

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

    // Convert array in base `radix` to base 2^EXPONENT
    val mag = array.toBase2PowK(radix.toUInt(), EXPONENT)
    return BigUInt(mag)
}
```

Ici, `EXPONENT=31`.

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) {
    ...
    companion object {
        private const val DEFAULT_BASE_STRING = 10
        private const val BASE = 2147483648u  // 2.pow(31)
        private const val EXPONENT = 31

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

Ce qui permet d'utiliser la méthode sans instancier :

```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()
```

Nous avons définie des classes utilitaires pour convertir des arrays d'une base à une autre.

Convertisseur base n vers base 2^k :

```kotlin
/**
 * Convert a array in [radix] to an array in base 2.pow(k)
 *
 * Algorithm Description :
 * -  Convert the source in binary
 * -  Combine chunks of digit into one
 *
 * Self-Explanatory Example with *137 to base 16 (2^4)*:
 * -  137 = 0b10001001
 * -  0b10001001 = 1000 | 1001 = 0x89
 */
@ExperimentalUnsignedTypes
```

```kotlin
fun UIntArray.toBase2PowK(radix: UInt, k: Int): UIntArray {
    val thisBase2 = this.toBase2Array(radix)
    val result = UIntArray(size = thisBase2.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.elementAtOrElse(chunkIndex * k +
offset) { 0u } shl offset
        }
    }

    return result
}
```

Le convertisseur base n vers base 2 :

```kotlin
/**
 * Convert BigInt magnitude array to base 2.
 *
 * Algorithm description: Division method.
 */
@ExperimentalUnsignedTypes
fun UIntArray.toBase2Array(radix: UInt): UIntArray {
    val size = ceil(this.size * log2(radix.toDouble())).toInt()
    val result = UIntArray(size)
    val zero = uintArrayOf(0u)

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

    return result
}
```

Le `divBy2(radix)` :

```kotlin
/**
 * Considering that UIntArray is a array of digit. Return this / 2
 *
 * Algorithm description:
 * `(a_0 + a_1 * base + ... + a_n * base^n)/2` can be developed to
 * `a_0/2 + (a_1 * base)/2 + ... + (a_n * base^n)/2`. If one of the division has
a carry (i.e is impair), then
 *  this carry will pass to the i - 1 th element.
 *
 * So, to summarize:
 * -  Loop from the nth element to the zeroth element
 *    - Add the carry if exist
 *    - Store the new carry if impair
```

```
 *    - Divide by 2 the element
 */
@ExperimentalUnsignedTypes
fun UIntArray.divBy2(radix: UInt): UIntArray {
    val zero = uintArrayOf(0u)
    val one = uintArrayOf(1u)
    if (this.contentEquals(zero) || this.contentEquals(one)) return zero
    val result = this.copyOf()

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

    return result
}
```

Comparaison

Implémenter `Comparable<BigUInt>`

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

```
class BigUInt(mag: UIntArray) : 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`

```
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 en non-signé :

```
operator fun unaryPlus() = this

operator fun unaryMinus() = this
```

`plus`

Notre implémentation contient quelques conditions pour la sécurité :

```
operator fun plus(other: BigUInt): BigUInt {
    if (this == zero) return other
    if (other == zero) return this

    val result = this addMagnitude other

    return BigUInt(result)
}
```

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 % 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 cas d'école:

- `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 (this == zero) return other
    if (other == zero) return this

    val result = this subtractMagnitude other

    return BigUInt(result)
}
```

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 (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)
}
```

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 $base^n$ (où $base = 2^{31}$).

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)
}
```

`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`.

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

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

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

`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 (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 issue de l'implémentation de `UIntArray.divBy2`.

```
fun divBy2() = BigUInt(mag.divBy2(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 (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
}
```

L'implémentation est plus facile avec des nombres signés. Nous n'aurions pas besoin de stocker le signe.

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 "remainder of shift left `n.mag.size` times" 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 * BASE mod 3233 = 882" {
    val a = BigUInt.valueOf("413", 10)
    val n = BigUInt.valueOf("3233", 10)

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

    v shouldBe BigUInt.valueOf("1721706655", 10)
    aMgy shouldBe BigUInt.valueOf("882", 10)
}
```

Note : Le test est sous format Kotest.

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

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

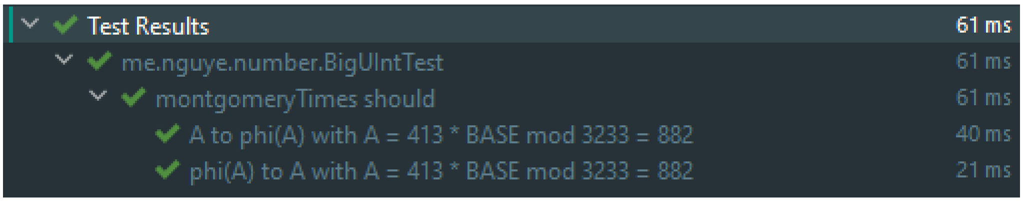
```
"phi(A) to A with A = 413 * BASE mod 3233 = 882" {
    val a = BigUInt.valueOf("413", 10)
    val n = BigUInt.valueOf("3233", 10)

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

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

    aNotMgy shouldBe a
}
```

Ce qui donne :



`modPow`, exponentiation modulaire avec la réduction de Montgomery

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

```
fun modPow(exponent: BigUInt, n: BigUInt): BigUInt {
    val exponentBase2 = exponent.mag.toBase2Array(radix=BASE)
    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.size - 1 downTo 0) {
        p = p.montgomeryTimes(p, n, v) // Square : p = p*p
        if (exponentBase2[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.toBase2Array(radix=BASE)` 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).

Test de `modPow`

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 radix = 10

    val d = BigUInt.valueOf("...", radix)  // Entrée tronquée
    val c1 = BigUInt.valueOf("...", radix)  // Entrée tronquée
    val c2 = BigUInt.valueOf("...", radix)  // Entrée tronquée
    val n = BigUInt.valueOf("...", radix)  // 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 : 277ms
```

Résultat satisfaisant et rapide.

`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)
}
```