# Sécurité Informatique

HOARAU Julien

April 2020

## 1 Introduction

Le But du projet est de créer une application mobile sur la plateforme Android ou swift basé sur la Cryptographie. En effet, nous avons plusieurs algorithmes à coder comme : Atbash, César, Vigenère, Hill, le carré de Polybe, Playfair, la Transposition rectangulaire, le DES (Data Encryption Standard), et le RSA. Dans ce projet, nous allons voir l'ensemble de ses algorithmes codé sur la plateforme iOS avec leurs difficultées et futur amélioration. Certain de ses Algorithmes, sont appliqués sur la table *ASCII étendu* .
Tout au long de ce projet, nous utilisons la table ASCII étendu : ISO/IEC 8859-1 Nous allons voir dans un premier temps les différents Algorithmes avec leur principe de fonctionnement et les difficultés que j'ai rencontré. Puis nous allons voir les difficultés rencontrées en général.

## 2 Sommaire

# 3 Atbash

## 3.1 Principe

Atbash est un chiffrement par subtitution monoalphabétique simple pour l'alphabet hébreu. Cette méthode consiste a subtitué la premiere lettre de l'alphabet (a) avec la dernière (z) puis la deuxième lettre avec l'avant dernière (y) ect ...

## 3.2 Dans le code

Dans l'application, Atbah est appliquée sur la table ASCII étendu.
En effet, l'algorithme se base s'appuie sur les 256 valeurs de la table.

```
func atbash(t : [Int]) -> [Int]{//modifie les valeur ascii (atbash)
        print("Valeur Ascii du message : ",t)
        var tab : [Int] = []
        for i in t {
            let x = 256 - 1 - i //func atbash : f(x) = taille du tableau - 1 - index
            tab.append(x)
        }
        return tab
    }
```

Cette fonction, codée sous le langage, nous renvoie la valeur opposée dans la table ASCII étendu.
On remarque que la fonction manipule des entiers, effectivement, il y a une fonction qui permet de stocker dans un tableau les valeurs entières des caractères saisie :

```
    func getAsciiInt(m : String) -> [Int]{
        print("message clair : ",m)
        var tab : [Int] = []
        let mess = Array(m)
        var i = 0
        while i <= (mess.count - 1){
            if mess[i] == "\\" && i < mess.count - 1 {/
                if mess[i+1] == "x" {
                    print("Détection de l'Hexa")
                    var hex : String = ""
                    i += 2 //on saute le \x
                    var j = 0
                    while j != 2 {
                        print("lettre hexa : ",mess[i])
                        hex.append(mess[i])
                        i += 1
                        j += 1 //compteur pour l'hexadécimal
                    }
                    tab.append(Int(hex, radix: 16)!)
                }
            }else{
                tab.append(Int(UnicodeScalar(String(mess[i]))!.value))
                i += 1
            }
        }
        return tab
    }
```

On peut remarquer que cette fonction stocke chaque caractère par sa valeur entière correspondante dans la table ASCII. De plus, on s'aperçoit qu'une partie de la fonction permet de détecter la présence de l'hexadécimal représenté par :

```
\x
```

et deux caractères qui se suivent (00 à ff : 0 à 256 valeurs)

Il existe aussi une autre fonction qui permet de faire le fonctionnement inverse de GetAsciiInt, c'est la fonction getAsciiChar. Elle permet de faire la liaison entre le Symbole et sa valeur Entière. On notifie aussi qu'elle détecte les valeurs hexadécimale.

### 3.3 Difficultées :

Les difficultés rencontrées étaient de pouvoir lire la valeur entière du caractère saisies, mais aussi de connaître le caractère entrée selon la valeur décimal.
De plus, le format des valeurs Hexadécimales ont été créer pour être conforme à celle de l'énoncé.

# 4 César

## 4.1 Principe

Le chiffrement de César est une méthode de chiffrement très simple. Le texte chiffré s'obtient en remplaçant chaque lettre du texte clair original par une lettre à distance fixe, toujours du même côté, dans l'ordre de l'alphabet. Pour les dernières lettres (dans le cas d'un décalage à droite), on reprend au début. Par exemple avec un décalage de 3 vers la droite, A est remplacé par D, B devient E, et ainsi jusqu'à W qui devient Z, puis X devient A etc.

## 4.2 Dans le code

```
func CrypCesar(cle : Int, t: [Int]) -> [Int]{
    var tab : [Int] = []
    let modulo = 256
    for i in t {
        tab.append((i+cle)%modulo)
    }
    return tab
}

func DecrypCesar(cle : Int, t: [Int]) -> [Int]{
    var tab : [Int] = []
    let modulo = 256
    for i in t {
        tab.append((i-cle)%modulo)
    }
    return tab
}
```

Nous voyons au-dessus les deux fonctions principales afin de coder et de décoder un message. Dans le ViewController vous pouvez apercevoir que l'algorithme est appliqué sur la table ASCII mais aussi que certaines fonctions sont identique à celle de Atbash.
Effectivement les fonctions GetAsciiInt et getAsciiChar seront présent dans tous les algorithmes manipulant des caractères dans la table ASCII.

## 4.3 Difficultés :

Je n'ai pas rencontré de difficulté particulière pour cet algorithme.

# 5 Vigenère

## 5.1 Principe :

Le chiffre de Vigenère est un système de chiffrement par substitution polyalphabétique mais une même lettre du message clair peut, suivant sa position dans celui-ci, être remplacée par des lettres différentes, contrairement à un système de chiffrement mono alphabétique comme le chiffre de César (qu'il utilise cependant comme composant).
Ce chiffrement introduit la notion de clé. Une clé se présente généralement sous la forme d'un mot ou d'une phrase. Pour pouvoir chiffrer notre texte, à chaque caractère nous utilisons une lettre de la clé pour effectuer la substitution. Évidemment, plus la clé sera longue et variée et mieux le texte sera chiffré. Il faut savoir qu'il y a eu une période où des passages entiers d'œuvres littéraires étaient utilisés pour chiffrer les plus grands secrets.

## 5.2 Dans le code :

Pour chiffrer un message, l'algorithme de Vigenère s'appuie sur une clé comme mentionnée ci-dessus. Nous pouvons avoir plusieurs cas lors de la saisie du message et de la clé.
Effectivement, nous pouvons avoir le cas :

— La taille du message est identique a la taille de la clé.
On a aucune modification à faire.

— La taille du message est supérieur a la taille de la clé.
On répète la clé pour quelle a la même taille que le message.

```
func repeatKey(m : String, k : String) -> String{
    var rk : String = k //clé qui va etre répété
    if m.count > k.count{
        let tm = Array(m)
        let tk = Array(k)
        var i = tk.count
        while i < tm.count {
            rk.append(tk[i % tk.count])
            i += 1
        }

    }
    return rk
}
```

Les fonctions utilisées pour déchiffrer et chiffrer :

```
func CryptVigenere(m : [Int], k : [Int]) -> [Int]{//les deux tableaux ont la meme taille

    var tab : [Int] = []
    for i in 0..<k.count {

        let x = (m[i] + k[i]) % 256 //Taille de la table ascii étendu
        //print("Décalage : abs(",m[i]," - ",k[i],") % 256 = ",abs(m[i] - k[i]))
        tab.append(x)
    }
    return tab
}
```

```
func DecryptVigenere(m : [Int], k : [Int]) -> [Int]{//les deux tableaux ont la meme taille
    var tab : [Int] = []
    for i in 0..<k.count {
        let x = (m[i] - k[i]) % 256
        tab.append(abs(x))
    }
    return tab
}
```

## 5.3 Difficultés :

Je n'ai pas rencontré de difficulté particulière pour cet algorithme.

# 6 Carré de Polybe

## 6.1 Principe :

Le principe est assez simple, il consiste à ordonner les lettres de l'alphabet en ordre alphabétique dans un tableau carré de 6 cases de côté dont chaque ligne et chaque colonne sont numérotées, de gauche à droite et de haut en bas. Considérant que l'alphabet français comporte 26 lettres et que le carré possède seulement 36 cases, on peut rajouter les 10 chiffres (0 à 9)

Ensuite, pour chiffrer un mot, il faut trouver la paire de numéros correspondants à chaque lettre. Le premier chiffre est le numéro de la colonne et le second celui de la rangée.

## 6.2 Dans le code :

Cet algorithme n'est pas appliqué sur l'ASCII, mais que sur 36 caractères : "abcdefghijklmnopqrstuvwxyz1234567890"

```
func CreateSquarePolybe() -> [Array<String>] {//Créer le tableau Polye de 36 charactère de a à
   let alpha = "abcdefghijklmnopqrstuvwxyz1234567890"
   //print(alpha.count)
   let tab = Array(alpha)
   var t : [Array<String>] = []
   var index = 0
   for _ in 0...5{
       var x : [String] = []
       for _ in 0...5{
           x.append(String(tab[index]))
           index += 1
       }
       t.append(x)
   }
   //print(t)
   return t
}
```

La fonction Createsquare permet de créer le carré de polybe. Puis nous avons d'autres fonctions qui permettent de lire le tableau.

- GetNumber prend en paramètre le tableau de polybe et un caractère, elle retourne la ligne puis la colonne.
- GetLetter prend en paramètre un x et qui va correspondre a la ligne puis a la colonne du tableau.

GetNumber prend en paramètre le tableau de polybe et un caractère, il retourne la ligne puis la colonne. La fonction va retourner le caractère correspondant dans le tableau.

## 6.3 Difficultés :

Je n'avais pas de difficulté particulière pour cet algorithme

# 7 Playfair

## 7.1 Principe :

Le Chiffre de Playfair ou Carré de Playfair est une méthode manuelle de chiffrement symétrique qui fut la première technique utilisable en pratique de chiffrement par substitution polygrammique. Il consiste à chiffrer des paires de lettres (des digrammes), plutôt que des lettres seules comme

dans les chiffrements par substitutions poly-alphabétiques tels que le chiffre de Vigenère.
Le chiffre de Playfair utilise un tableau de 6*6 lettres, contenant un mot clé ou une phrase. La mémorisation du mot clé et de 4 règles à suivre suffisent pour utiliser ce chiffrement.

### 7.2 Dans le code :

On remarquera que certaines fonctions sont similaires comme dans le code du carré de polybe. En effet, nous avons les mêmes fonctions pour parcourir la table.
En revanche, la création du tableau est similaire à celle de Polybe. La différence est qu'on complète le tableau avec la clé tout évitant les occurrences.

```
func CreateTablePlayfair(k : String) -> [String]{
    let key = Array(k)
    let alpha = Array("abcdefghijklmnopqrstuvwxyz1234567890")
    var tab : [String] = []
    var i = 0
    while i < key.count {
        //élimine les occurences
        if !tab.contains(String(key[i])) && key[i] != " "{
            tab.append(String(key[i]))
        }
        i += 1
    }
    //print("tab with only key : \(tab)")
    var j = 0
    while j < alpha.count {
        //éliminé les espaces et les récurrences
        if !tab.contains(String(alpha[j])) && alpha[j] != " "{
            tab.append(String(alpha[j]))
        }
        j += 1
    }
    //print("tab : \(tab)")
    return tab
}
```

### 7.3 Difficultés :

Je n'avais pas de difficulté particulière pour cette algorithme

## 8 Hill

### 8.1 Principe :

Il consiste à chiffrer le message en substituant les lettres du message, non plus lettre à lettre, mais par groupe de lettres.l permet ainsi de rendre plus difficile le cassage du code par observation des fréquences. Les lettres sont regroupées par lettre de 2,puis transcrit par leur valeur entière dans la table Ascii. La clé est une matrice 2,2 qui permettra de chiffre le message.

### 8.2 Dans le code :

La première fonction que nous allons voir est la fonction CanBeUseHill. Cette fonction a pour but de vérifier si la matrice est conforme afin qu'elle puisse être utilisée pour chiffrer et déchiffrer par la suite. Concrètement on regarde si le pgcd de : (a*d - b*c) est égale 1. Si oui la matrice est

conforme, sinon on arrête l'algorithme.
Ensuite nous allons nous pencher sur la fonction qui permet de déchiffrer :

```swift
func DecryptHill(k : [Int], x1 : Int, x2 : Int) -> [String] {
    let a = k[0]
    let b = k[1]
    let c = k[2]
    let d = k[3]
    //var tab : [String] = []

    let nb_inv = a*d - b*c
    print("Inverse(nb)  : \(nb_inv) =  \(GetInverseMod(k: nb_inv, mod: 256))")
    if GetInverseMod(k: nb_inv, mod: 256) != 0 {
        let inv = GetInverseMod(k: nb_inv, mod: 256)

        var a1 = (d * inv) % 256
        var b1 = (-b * inv) % 256
        var c1 = (-c * inv) % 256
        var d1 = (a * inv) % 256

        //operation pour rester dans les positif
        if a1 < 0 {
            a1 = (a1 + 256) % 256
        }
        if b1 < 0 {
            b1 = (b1 + 256) % 256
        }
        if c1 < 0 {
            c1 = (c1 + 256) % 256
        }
        if d1 < 0 {
            d1 = (d1 + 256) % 256
        }
        print("valeur originial = a: \(a) ,b: \(b) ,c: \(c) ,d: \(d) ")
        print("valeur modifier = a: \(a1) ,b: \(b1) ,c: \(c1) ,d: \(d1) ")
        let x = (a1*x1 + b1*x2)%256 //table ascii étendu
        let y = (c1*x1 + d1*x2)%256


        return getAsciiChar(t: [x,y])
    }


    return ["Error"]
}
```

Dans cette fonction, on peut apercevoir l'utilisation d'une autre fonction GetInversMod. Elle permet d'avoir l'inverse du modulo afin de pouvoir décrypter le message chiffré sur 255.

## 8.3 Difficultés :

La plus grande difficulté était de comprendre le code et de pouvoir l'appliqué sur la table ASCII étendu.En effet, j'ai du j'ai cherché un algorithme qui permettait de calculer l'inverse modulaire puis l'adapter aux langage Swift.

# 9 Transposition Rectangulaire

## 9.1 Principe :

Une transposition rectangulaire consiste à écrire le message dans une grille rectangulaire, puis à arranger les colonnes de cette grille selon un mot de passe donné (le rang des lettres dans l'alphabet donne l'agencement des colonnes)

## 9.2 Dans le code :

Dans ce code, le tableau est inspiré du code du playfair. En effet on insére la clé plus le message mais tout en laissant les occurences.

```
func createTable(message: String, key : String) -> [Array<String>]{
    var tab : [Array<String>] = []
    var messageInTable : [String] = []
    let k = Array(key)
    let m = Array(message)

    //on concatène la clé et le message tout en supprimant les espaces
    for i in 0..<key.count{
            messageInTable.append(String(k[i]))
    }
    for i in 0..<m.count{
            messageInTable.append(String(m[i]))
    }

    for _ in 0..<(key.count - (message.count % key.count)){
        messageInTable.append(" ") //on rajoute des espace pour completer le tableau
    }

    //=======================
    var t : [String] = []
    for i in 0..<messageInTable.count{//ordorne les characteres dans les bonnes case du tableau
        //print(messageInTable[i]," => ",i%3)
        if i % key.count == key.count - 1 {
            t.append(messageInTable[i])
            tab.append(t)
            t = []
        }else{
            t.append(messageInTable[i])
        }
    }
    //print(tab)
    return tab
}
```

Dans ce code, nous voyons qu'on créer un tableau de tableau de type String, dont la taille est égale à la taille de la clé. On peut remarquer dans le code, que si le message ne remplit pas toutes les cases du tableau, alors on le remplissait avec un espace.

## 9.3 Difficultés :

Lors de la mise en place de cet algorithme, j'ai rencontré quelque difficulté comme par exemple :

— La gestion des cases vides dans le tableau
— Gérer les occurrence afin que n'apparaissent pas les mêmes colonnes pour deux colonnes différentes.
— La retranscription du tableau, précisément gérer les lignes avec des cases vides

# 10 DES

## 10.1 Principe :

Le Data Encryption Standard est un algorithme de chiffrement symétrique (chiffrement par bloc) utilisant des clés de 56 bits.L'algorithme DES transforme un bloc de 64 bits en un autre bloc de 64 bits. Il manipule des clés individuelles de 56 bits, représentées par 64 bits (avec un bit de chaque octet servant pour le contrôle de parité). Ce système de chiffrement symétrique fait partie de la famille des chiffrements itératifs par blocs, plus particulièrement il s'agit d'un schéma de Feistel (du nom de Horst Feistel à l'origine du chiffrement Lucifer)... https://fr.wikipedia.org/wiki/Data_Encryption_Standard

## 10.2 Dans le code :

Dans cette partie, nous allons résumer l'action de chaque fonction utilisée dans l'algorithme. La fonction prepareBloc permet de créer un ensemble de 64 bit chiffrer à partir du message. Pour réaliser cela, elle s'appuie sur plusieurs petites fonctions permettant de transformer le message en binaire sur 64bits.
Les fonctions qui permettent de faire cela sont :

- getAsciiInt (Retourne un tableau d'entier, dont chaque valeur correspond à une lettre.
- getBinary Retourne la conversion du tableau d'entier en tableau binaire.
- ajustOctet ajuste le nombre binaire pour être dans le format d'un octet.
- ajustXblock ajuste l'ensemble des octets afin qu'il soit composé de 64 bits.
- concatOctetToBloc concatène le tableau afin de retourner une chaîne de caractère.

Ensuite nous allons voir les fonctions qui gèrent la clé :

- DiverseKey (retourne les 16 sous clé de K selons les sous fonctions)

```
func diversKey(k:String) -> [String]{//transforme la clé en 16 sous clé
    print("\n==============================|| Préparation de la clé ||============================\n")
    print("PHASE 1 :\n")
    print("clé = \(k)\n")
    let key_bin = getKey(key: k)
    print("key modifier : ",key_bin)
    let tablePC = getTablePC1()
    let key_pi = P(key_binaire: key_bin, PC1: tablePC)
    //print(key_pi,": key PC1 (",key_pi.count,")")
    //let new_key_pi = PermuInit(key_binaire: key_bin, PC: tablePC) //8 bit en trop
    //print("====================\n",key_pi,"\n",new_key_pi)
    let CD = C0D0(pc1: key_pi)
    let C0 = CD[0]
    let D0 = CD[1]

    let tCs = getCs(C0: C0)
    let tDs = getDs(D0: D0)
    let PC2 = getTabPC2()
    let table_Ki = getKi(Ci: tCs, Di: tDs, PC2: PC2)

    return table_Ki
}
```

```
func cryptDes(message:String, key:String) -> String{
    let table_Ki = diversKey(k: key)
    //print("\n==============================|| Fin de la préparation de la clé ||============================\n")
    print("\n==============================||      Préparation du texte      ||============================\n")
    print("PHASE 2 :\n")
    let x = prepareBloc(m: message)
    var ens_message = ""
    for i in 0..<x.count{
        print("====================================* Bloc \(i+1) *====================================")
//      let x = prepaMessage(m: message)
        //let i = 0
        let Y = P(key_binaire: x[i], PC1: getTableP())
        let G0D0 = getG0D0(Y: Y)

        let G_0 = G0D0[0]
        let D_0 = G0D0[1]
        let z = feistel(Ki: table_Ki, G_0: G_0, D_0: D_0)
        let result = P(key_binaire: z, PC1: getTablePInv())
        print("message coder en binaire: ", result)
        //print(cut8(m: result))
        let res_t = cut8(m: result)//résultat binaire
        var r_int : [Int] = []
        for i in res_t{
            r_int.append(Int(i,radix: 2))
            //print("ent = ",Int(i,radix: 2))
        }
        let t = getAsciiChar(t: r_int)
        var mess = ""
        for i in t{
            mess.append(i)
        }
        ens_message.append(mess)
    }
    print("message encoder = ", ens_message)
    return ens_message
}
```

### 10.3 Difficultés :

J'ai eu beaucoup de mal à comprendre l'algorithme du DES. J'ai passé la plupart de mon temps dans la phase de débug afin de trouver les potentielles erreurs de mon code. Il s'avérer aussi que certains tableaux donné par le site Bibmath comportais plusieurs erreurs. Pour terminer, je n'ai pas réussi à debuger et à corriger la partie pour déchiffrer.

# 11 RSA

## 11.1 Principe :

Le chiffrement RSA est un algorithme de cryptographie asymétrique, très utilisé dans le commerce électronique, et plus généralement pour échanger des données confidentielles sur Internet. Tous les calculs se font modulo un nombre entier n qui est le produit de deux nombres premiers. Le petit théorème de Fermat joue un rôle important dans la conception du chiffrement.

Les messages clairs et chiffrés sont des entiers inférieurs à l'entier n2. Les opérations de chiffrement et de déchiffrement consistent à élever le message à une certaine puissance modulo n (c'est l'opération d'exponentiation modulaire).

## 11.2 Dans le code :

```
func expmod(a:Int,y:Int,n:Int)->Int{//func recursive qui calcule l'exposant modulaire
    if y % 2 == 0{
        return expmod(a: ((a*a)%n), y: Int(y/2), n: n) % n
    }
    else{
        if y != 1 {
            return (a%n)*expmod(a: ((a*a)%n), y: Int((y-1)/2), n: n) % n
        }else{
            return a%n
        }
    }
}
```

Cette fonction va nous servir à calculer la puissance modulaire, en effet, nous verrons dans le code qu'elle va servir pour coder et décoder.

```
func crypter(m : Int,e: Int, n: Int)->Int{
    //print("Crypter : \(m)^\(e) % \(n) = \(expmod(a: m, y: e, n: n))")
    return expmod(a: m, y: e, n: n)
}

func decrypter(m: Int,d: Int,n: Int)->Int{
    //print("Décrypter : \(m)^\(d) % \(n) = \(expmod(a: m, y: d, n: n)))")
    return expmod(a: m, y: d, n: n)
}
```

Nous allons voir ensuite comment la fonction qui permet le découpage du nombre en question.

```
func selectNumber(m: String, x:Int)-> [String]{//divise la chaine pour avoir un un bloc Int < n
    //var number = ""
    var tab : [String] = []
    var ms = m
    var s = m //var temp
    //print(m)
    while ms.count > 0{
        if Double(s)! < Double(x) {
            tab.append(s)//ajout de s dans le tableau
            ms = String(ms.dropFirst(s.count))//màj
            s = ms
        }else{
            s = String(s.dropLast(1))//on décrémente d'une lettre
        }
    }
    return tab
}
```

Cette fonction ci-dessus permet de voir comment le nombre est découpé pour qu'il soit inférieur à n.

```swift
func detecZ(m:[String])-> [String]{
    var num = ""
    var tab : [String] = []
    for i in m {
       let t = Array(String(i))
        if t[0] == "0"{//si le premier élément est égale 0 alors on le sauvegarde
            var indice = 0
            while indice < t.count  {
                if t[indice] == "0" && indice != t.count-1{//evite la duplication d'un zero si c'est que de 0
                    num.append("0")//ajout le nb de 0 qu'on detect dans le nombre
                }
                indice += 1
            }
            tab.append(num)//ajoutes les 0 dans le tableau
            num.removeAll()//RàZ
        }else{
            tab.append("")//on ajoute un élément vide pour conserver le rang pour identifier le nombre dans la table
        }
    }
    //print("table de 0 = \(tab)")
    return tab
}
```

Cette fonction nous servira à détecter les zéros inutiles afin qu'ils soient sauvegardés avant d'être perdus lorsque la chaîne de caractère sera transformée en Entier.

```swift
func addZero(z:[String],n:[Int])->[String]{
    var tab : [String] = []
    for i in 0..<z.count{
        if !z[i].isEmpty{
            tab.append(z[i]+String(n[i]))
        }else{
            tab.append(String(n[i]))
        }
    }
    //print(tab)
    return tab
}
```

Cette fonction permet prendre les 0 stockés dans la fonction detectZero afin de bien les placée à la sortie du calcul. Cela permet de conserver le nombre de chiffres et ensuite d'avoir une bonne restitution du message lors de la phase de décryptage.

## 11.3 Difficultés :

J'ai rencontré le plus les difficultés dans le langage Swift que pour comprendre l'algorithme. En effet, lorsque j'utilisais la puissance avec un exposant trop grand ($> 60$), les valeurs retournées était arrondie. De plus, il avait un problème lors de la conversion chaîne de caractère vers un Entier. Précisément, lorsque nous avons "00" en chaîne de caractère, il sera traduit en 0 en Entier.

## 12 Conclusion

Pour conclusion, je dirais que la plupart des algorithmes ont été simples pour programmer. En effet, j'avais déjà fait la plupart des exercices en python. De plus, il faut noter que je n'ai pas la même table ASCII que celle de l'énoncé.

## 13 Bibliographie


— https://www.ascii-code.com/?fbclid=IwAR0jwjK80nSHUvUrkpcmGVm0dMY4Tv2MQCRuesuB2FbhOqtLZnZSsFBZC
— http://www.bibmath.net/crypto/index.php?action=affiche&quoi=moderne/des
— https://fr.wikipedia.org/wiki/Constantes_du_DES
— https://www.univers-ti-nspire.fr/files/pdf/05-exponentiation_modulaire-TNS21.pdf
— https://www.youtube.com/watch?v=aYI9B9ra_P4
— https://fr.wikipedia.org/wiki/Chiffrement_RSA#Fonctionnement_détaillé
— https://fr.wikipedia.org/wiki/Algorithme_d%27Euclide