UNIVERSITÉ MOHAMMED VI
POLYTECHNIQUE

# Rapport Du Projet

# Cryptographie

**Auteurs:**
Oumar KEITA
Yasmine ED-DYB
Omar KAMI

**Encadrant:**
Pr. Pierre Vincent KOSELEFF

**Date:**
December 10, 2023

# Contents

# Introduction

Le texte présent expose divers aspects de la cryptographie, en se concentrant particulièrement sur la méthode RSA (Rivest, Shamir et Adleman), qui demeure l'une des techniques de cryptographie les plus utilisées de nos jours. La cryptographie RSA repose sur des concepts mathématiques fondamentaux, tels que les nombres premiers, le groupe multiplicatif et la difficulté de factoriser de grands nombres.

La méthode RSA implique la génération de clés publique et privée, où la sécurité repose sur la complexité de certaines opérations mathématiques, notamment l'extraction de racines et la factorisation de nombres premiers. Le texte explore également des vulnérabilités potentielles liées à des choix inappropriés de paramètres, tels que des exponents de chiffrement trop petits.

Une des vulnérabilités discutées concerne le choix d'un petit exposant de chiffrement (e), ce qui pourrait permettre à un attaquant de retrouver le message original avec un coût relativement faible en effectuant des opérations polynomiales en logarithme du module. Une autre faiblesse potentielle réside dans le partage du module RSA entre plusieurs utilisateurs, permettant à l'un d'entre eux de déduire la factorisation du module et ainsi compromettre la sécurité du système.

En se concentrant sur l'aspect de la diffusion large de messages par RSA, le texte soulève des préoccupations liées au choix des paramètres pour optimiser le coût du chiffrement. Il met en garde contre des choix trop agressifs qui pourraient compromettre la sécurité du système.

Dans le cadre de notre projet, l'objectif sera de comprendre comment RSA peut être utilisé pour la diffusion large de messages tout en évitant les pièges discutés dans le texte. Cela implique une analyse minutieuse des paramètres choisis et une évaluation des compromis entre la sécurité et l'efficacité du chiffrement.

# 1 Chiffrement

## 1.1 Principe du chiffrement (mise en puissance)

$$n = pq$$
$$\Phi(n) = \Phi(p) \times \Phi(q) = (p-1)(q-1)$$

On choisit $e$ tel que $(e, \Phi(n)) = 1$.
Chiffrer le message consiste à l'élever à la puissance $e$, donc

$$E(\mu) = \mu^e$$

On obtient donc le message chiffré :

$$C = \mu^e \pmod n$$

Ainsi, calculer le cout du chiffrement RSA d'un message µ revient à calculer le cout de calcul de $\mu^e$.

## 1.2 Exponentiation rapide

Soit $A$ un corps, $x \in A$ et $n \in \mathbb{N}$. Pour calculer $x^n$, on peut utiliser l'algorithme `"naïf"` suivant :
**Principe :**

$$x^n = x \cdot x \cdots x$$

**Algorithme**

Puissance $(x, n)$

$k \leftarrow 0$

$b \leftarrow 1$

tant que $k < n$ :

| $b \leftarrow b.x$

| $k \leftarrow k + 1$

renvoyer $b$.

Le calcul se fait à l'aide de n-1 multiplications, soit une complexité O(n).

On suppose que l'on dispose d'un algorithme binaire $(n)$ qui prend en entrée un entier naturel $n$, et qui renvoie la liste $L = [a_k, \ldots, a_0]$ d'éléments de $\{0, 1\}$ telle que $a_k \neq 0$, $a_i \in \{0, 1\}$ pour tous $i \in [0, k]$ et $n = \sum_{i=0}^{k} a_i 2^i$. L'algorithme suivant prend en entrée un élément $x$ et un entier naturel $n$ et renvoie $x^n$.
**Algorithme d'exponentiation rapide**

expo-rapide $(x, n)$

$L \leftarrow$ binaire $(n)$

$k \leftarrow \text{long}(L) - 1$

$P \leftarrow x$

$a \leftarrow 1$

pour $i$ allant de 0 a $k$

| si $L[k - i] = 1$ :

  | $a \leftarrow aP$

| $P \leftarrow P^2$

renvoyer $a$.

En effet, pour $i \in [0, k]$, notons $a_i, P_i$, etc. les valeurs de $a$, $P$, etc avant le passage $i$ dans la boucle. On a $P_0 = x = x^{(2^0)}$ et $a_0 = 1$. Par récurrence, on a $P_i = x^{(2^i)}$, si $i \in [0, k]$ et

$a_i = \prod_{j<i|a_j=1} P_j = \prod_{j<i|a_j=1} x^{2^j} = x^{\sum_{j<1|a_j=1} 2^j} = x^{\sum_{j<i} a_j 2^j}$. Si on note $a_{k+1}$ la valeur de $a$ après le passage $k$ dans la boucle, on a donc $a_{k+1} = x^{\sum_{j\leqslant k} a_j 2^j} = x^n$.

Pour calculer $x$, on a effectué $k+1$ élévations au carré (pour calculer les valeurs des $P_i$ ), et $k+1$ calculs de produits (pour calculer les valeurs des $(a_i)$). On a donc utilisé $O(\log_2(n))$ produits (les élévations au carré sont des produits), ce qui est beaucoup mieux que $O(n)$.

En comparaison avec l'algorithme naif, il donne beaucoup moins de multiplications à effectuer dans ($O(\log_2(n))$ contre $O(n)$). Si $A = \mathbf{Z}$, le coût de calcul (temps de calcul) d'une multiplication $x.y$ augmente lorsque $x$ et $y$ augmentent. Il n'est donc pas évident a priori que le calcul de $x^{\mathrm{n}}$ soit plus rapide avec l'algorithme d'exponentiation rapide qu'avec l'algorithme naif (car $x^m$ croit avec $m$, si $x \geq 2$). Par contre, si $A = \mathbb{Z}/NZ$, pour $N \in N^*$ le cout du calcul de $ab$, pour $a, b \in \mathbb{Z}/N\mathbb{Z}$ est $O\left(\log_2(N)^2\right)$. Le cout du calcul de $x^n$ pour $n \in \mathbb{N}$ par l'algorithme d'exponentiation rapide est donc $O\left(\log_2(N)^2 \log_2(n)\right)$ alors que celui par l'algorithme naïf est $O\left(\log_2(N)^2 n\right)$.
Or l'exponentiation rapide n'est en aucun cas la méthode optimale

### 1.3 Optimisation du calcul de $x^n$

**Définition** Une chaîne d'additions pour le calcul d'un entier positif $n$ est une suite d'entiers naturels commençant par 1 et se terminant par $n$, et telle que chaque entier de la suite est la somme de deux entiers précédents.

La longueur de la chaîne d'additions, qu'on notera $l(n)$, est le nombre de sommes nécessaires pour exprimer ces entiers ; c'est un de moins que le nombre de termes dans la suite.

**Exemple :** Pour $n = 23$ , une chaine d'additions est $(1, 2, 3, 5, 1, 20, 23)$

En effet :

- $2 = 1 + 1$
- $3 = 2 + 1$
- $5 = 3 + 2$
- $10 = 5 + 5$
- $20 = 10 + 10$
- $23 = 20 + 3$

Les chaînes d'additions peuvent être utilisées pour l'exponentiation avec des exposants entiers en utilisant un nombre de multiplications égal à la longueur d'une chaîne d'addition pour l'exposant. Par exemple, la chaîne d'addition pour 23 conduit à une méthode de calcul de $x^{23}$ en utilisant seulement 6 multiplications.

- $x^2 = x \cdot x$
- $x^3 = x^2 \cdot x$
- $x^5 = x^3 \cdot x^2$
- $x^{10} = x^5 \cdot x^5$
- $x^{20} = x^{10} \cdot x^{10}$
- $x^{23} = x^{20} \cdot x^3$

**Remarque :** La méthode d'exponentiation rapide consiste à former une chaine d'additions, or cette dernière n'est pas forcément optimale.

Ainsi, trouver le nombre minimal de multiplications nécessaires pour calculer $x^n$ revient à trouver la plus courte chaine d'addition, or ceci n'est pas facile ; plus précisément, une variante généralisée du problème, dans laquelle il faut trouver une chaîne pour un ensemble de valeurs, est un **problème NP-complet**. Il n'existe pas d'algorithme connu qui calcule une chaîne d'addition minimale pour un nombre donné en temps raisonnable ou de faible utilisation de la mémoire. En revanche, plusieurs techniques sont connues pour calculer des chaînes relativement courtes, même si elles ne sont pas toujours optimales, dont l'une des plus utilisées est l'exponentiation rapide.

### 1.4 Simulation en SageMath

Code exponentiation rapide:

```
# Methode d'exponentiation modulaire rapide

def modular_exponentiation(M, e, n):
    """
    Effectue l'exponentiation modulaire rapide de 'M' à la puissance 'e' modulo 'n'.

    :param M: int, La base de l'exponentiation.
    :param e: int, L'exposant pour l'opération d'exponentiation.
    :param n: int, Le modulo pour l'exponentiation.

    :return: int, Le résultat de M^e mod n calculé en utilisant l'exponentiation modulaire rapide.

    Utilise la décomposition binaire de l'exposant pour effectuer l'exponentiation de manière efficace.
    """
    # Initialisation
    R = 1

    # Décomposition Binaire de L'Exposant
    binary_e = bin(e)[2:]

    # Itération sur Chaque Bit de L'Exposant
    for bit in binary_e:
        R = (R * R) % n  # Toujours effectuer un carré
        if bit == '1':
            R = (R * M) % n  # Multiplier si le bit courant est 1

    return R
```

On importe les frameworks suivants:

```
# Import libraries
import time as t
import matplotlib.pyplot as plt
from sage.misc.sage_timeit import sage_timeit
```

### 1.4.1 Coût du chiffrement en fonction de $\log(n) = \log(pq)$ :

On génère des valeurs de $p$ et $q$ premiers de l'ordre de $10^i$ et $10^{i+1}$ respectivement avec $30 \leqslant i \leqslant 100$. Ensuite pour chaque valeur de n= $p \cdot q$, on calcule $\mu^e\,[n]$ par l'exponentiation rapide et on mesure le temps d'execution (temps de chiffrement).
Code :

```
c1 = []
mu = 10**40
e = next_prime(10**30)          # on fixe e
for i in range(30,100):             # boucle pour générer n = pq
    p = next_prime(10**i)           #génerer p,q de l'ordre de 10^i, 10^(i+1) respectivement
    q = next_prime((10**(i+1)))
    n = p*q
    phi_n = (p-1)*(q-1)
    execution_time = sage_timeit('modular_exponentiation(mu,e,n)', globals(), preparse=True, number=50)
    execution_time = mean(execution_time.series)
    c1.append((log(n).n(),execution_time))

x_1 = [item[0] for item in c1]    # Log(n)
y_1 = [item[1] for item in c1]    # execution_time

#Courbe 1 : execution_time en fonction de la longuer de n=pq :
plt.figure(1)
plt.scatter(x_1, y_1, marker='o')
plt.xlabel('Longueur de n')
plt.ylabel('Temps de chiffrement')
plt.title('execution_time en fonction de la longuer de n')
plt.grid(True)
plt.show()
```

Simulation : $e = nextprime(10^{30}) \quad \mu = 10^{40}$


Figure 1: Simulation

### 1.4.2 Coût du chiffrement en fonction de $e$

Introduisons la fonction `find_suitable_e_values`, conçue pour pour identifier les valeurs de $e$ qui sont premières par rapport à phi_n, explorant une plage de valeurs comprises entre min_value et max_value.

```python
def find_suitable_e_values(phi_n, min_value, max_value):
    """
    Trouve les valeurs de 'e' appropriées pour le chiffrement RSA dans une plage donnée.

    :param phi_n: int, La valeur de ϕ(n) pour laquelle 'e' doit être premier.
    :param min_value: int, La valeur minimale à partir de laquelle chercher.
    :param max_value: int, La valeur maximale jusqu'à laquelle chercher.

    :return: List[int], Une liste des valeurs 'e' qui sont premières par rapport à 'ϕ(n)'.

    Parcourt seulement les nombres impairs car 'e' doit être impair dans RSA.
    """
    suitable_e_values = []
    for e in range(min_value, max_value, 2):  # Parcourir seulement les nombres impairs
        if gcd(e, phi_n) == 1:
            suitable_e_values.append(e)
    return suitable_e_values
```

Figure 2: Fonction `find_suitable_e_values`.

L étape suivante consiste à visualiser le temps d'encodage en fonction de la valeur de `'e'`, avec e allant de 1 à 65539.

```python
# Initialisation d'une liste vide pour stocker les résultats
res = []

# Boucle sur une série de valeurs 'e' appropriées pour RSA, de 1 à 65539
for e in find_suitable_e_values(phi_n, 1, 10**20):
    bits_count = bin(e).count('1')  # Compte le nombre de bits à 1 dans la représentation binaire de 'e'
    # Mesure le temps d'exécution de l'exponentiation modulaire pour chaque valeur de 'e'
    execution_time = sage_timeit('modular_exponentiation(mu, e, n)', globals(), preparse=True, number=50)
    execution_time = mean(execution_time.series)  # Calcul de la moyenne des temps d'exécution
    # Ajout de la valeur de 'e', du temps d'exécution moyen et du nombre de bits à 1 dans la liste 'res'
    res.append((e, execution_time, bits_count))

# Note: Ce code vise à analyser l'impact de la longueur de 'e' (mesurée par le nombre de bits à 1 dans sa représentation binaire)
# sur le temps d'exécution de l'exponentiation modulaire. Cela permet de comprendre comment la complexité de 'e'
# influence la performance de l'algorithme de chiffrement RSA.
```

```python
# Extraction des données pour le graphique
x_1, y_1 = [item[0] for item in res], [item[1] for item in res]  # Séparation des valeurs de 'e' et des temps d'exécution

# Création du graphique
plt.scatter(x_1, y_1, marker='o')
plt.xlabel('longueur de e')
plt.ylabel('Temps d\'encodage')
plt.title('Graphique de e en fct de temps d\'encodage')
plt.grid(True)
plt.show()
```

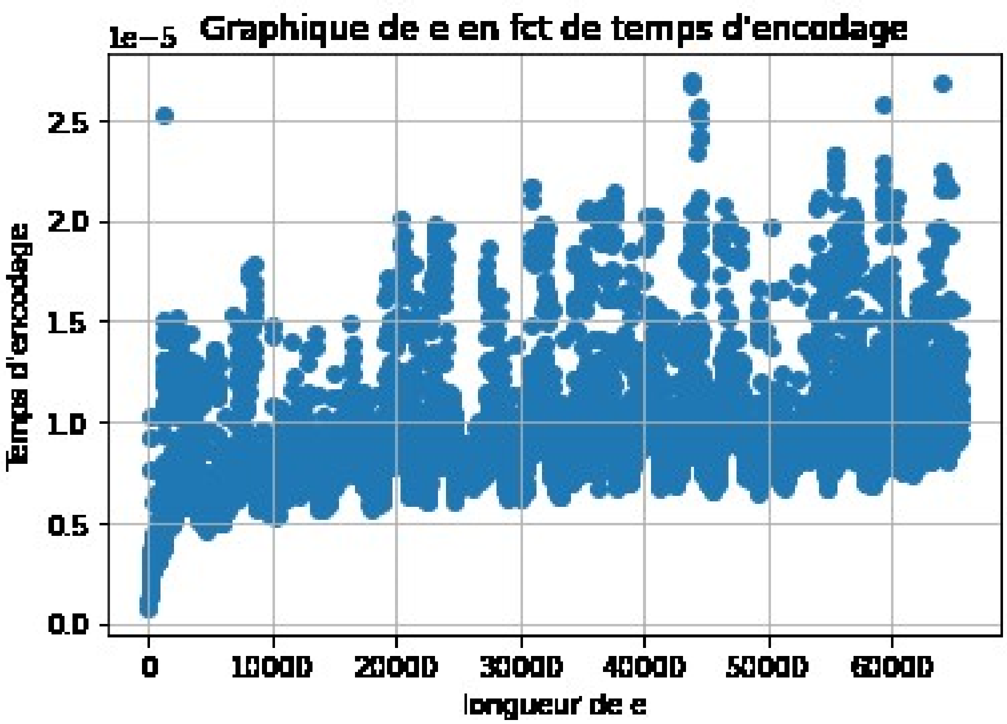



Figure 3: `Graphe représentant le temps d'encadage en fct de e.`

**Discussion**

1. Le graphique suggère qu'il n'y a pas de tendance claire reliant directement la longueur de e et le temps d'encodage, indiquant que d'autres facteurs que la longueur de e peut influencer le temps d'encodage. Des variations significatives dans le temps d'encodage sont observées à travers les différentes valeurs de e, même à longueur de bit similaire.

2. Une des limites de cette simulation est la plage des valeurs de e testées. Il pourrait être instructif d'étendre l'échantillon à un ensemble plus large de valeurs pour évaluer si les tendances observées persistent sur une plus grande échelle.

### 1.4.3 Cout du chiffrement en fonction du $e$ nombre de bits $1$ dans son écriture binaire :

Nous introduisons la fonction 'generate_e_values_prime_with_phi', conçue pour générer une liste de valeurs de 'e' qui sont premières par rapport à 'phi_n' et qui possèdent une longueur binaire spécifique.

```python
def generate_e_values_prime_with_phi(length, phi):
    """
    Génère des valeurs de 'e' avec une longueur binaire spécifique, premières par rapport à 'phi',
    et seulement un exemple par nombre de 1s dans la représentation binaire.

    :param length: int, La longueur binaire des nombres 'e' à générer.
    :param phi: int, La valeur de 'phi' par rapport à laquelle 'e' doit être premier.

    :return: List[int], Liste des valeurs 'e' qui respectent les critères spécifiés.

    Parcourt tous les nombres possibles ayant la longueur binaire spécifiée. Pour chaque nombre,
    il vérifie s'il est premier par rapport à 'phi' et s'il a un nombre unique de 1s dans sa représentation binaire.
    """
    e_values = []
    seen_ones_count = set()  # A set to keep track of the count of 1's we've seen

    # Iterate over all possible binary numbers of the given length
    start = 2**(length - 1)  # Smallest number with 'length' bits
    end = 2**length          # One past the largest number with 'length' bits

    for e in range(start, end):
        ones_count = bin(e).count('1')
        # Check if we have already seen this count of 1's
        if ones_count in seen_ones_count:
            continue  # Skip this number if we have

        # Check if e is prime relative to phi
        if gcd(e, phi) == 1:
            e_values.append(e)
            seen_ones_count.add(ones_count)  # Mark this count of 1's as seen

    return e_values
```

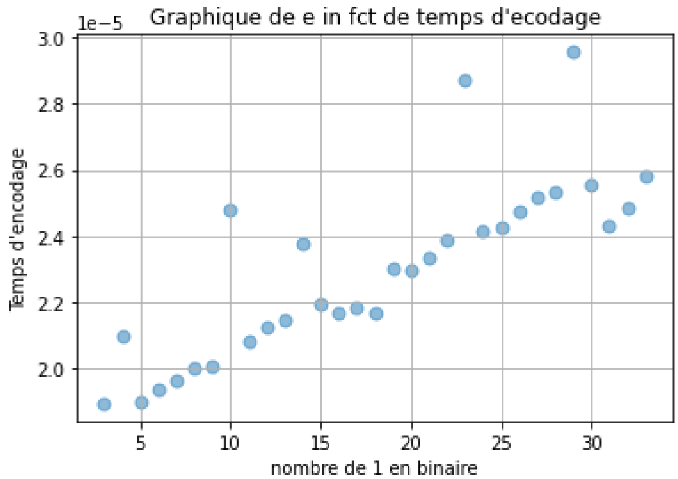
- Cette fonction est mise à l'épreuve avec des valeurs de 'e' dont la représentation binaire est de longueur 34.

```python
time_taken = []
one_count = []
res = []

e_val = generate_e_values_prime_with_phi(34, phi_n)

for e in e_val:
    one_count.append(bin(e).count('1'))
    time_tak = sage_timeit('modular_exponentiation(mu, e, n)', globals(), preparse=True, number=50)
    time_taken.append(mean(time_tak.series))
    res.append([e, bin(e).count('1'), mean(time_tak.series)])
```

- Nous traçons ensuite le temps d'exécution en fonction du nombre de '1' dans la représentation binaire des valeurs de 'e'.

```python
x, y = [item[1] for item in res], [item[2] for item in res]

plt.scatter(x, y, s=50, alpha=0.5)  # s is the size, alpha is the transparency level
plt.xlabel('nombre de 1 en binaire')
plt.ylabel('Temps d\'encodage')
plt.title('Graphique de e in fct de temps d\'ecodage')
plt.grid(True)
plt.show()
```

- Le graphique résultant illustre la relation entre le nombre de '1' en binaire et le temps d'exécution requis. (le temps est en microsecondes)




**Discussion**

1. Le graphique indique une tendance où le temps d'exécution augmente avec le nombre de '1' dans la représentation binaire. Cette observation pourrait suggérer que le temps nécessaire pour exécuter des opérations cryptographiques, telles que l'exponentiation modulaire, est affecté par la densité des '1' dans la représentation binaire de 'e'. Plus le nombre de '1' est élevé, plus le nombre d'opérations de multiplication nécessaires augmente, ce qui peut expliquer l'accroissement du temps d'exécution.

2. Cette analyse est importante pour l'optimisation des algorithmes cryptographiques, en particulier pour ceux qui s'appuient sur des calculs d'exponentiation modulaire rapides et efficaces. Elle souligne la pertinence de choisir des valeurs 'e' avec des caractéristiques binaires qui peuvent réduire le temps de calcul, contribuant ainsi à une mise en œuvre plus efficace des protocoles de sécurité.

# 2 Sécurité

## 2.1 Principe de l'attaque

Si $\mu < \min_{1 \leqslant i \leqslant r} n_i$ (c'est l'hypothèse que l'on a fait à l'origine sur l'encodage du message), et si $e \leqslant r$ alors le pirate peut trouver $\mu$ à partir de $c_1, \cdots, c_r$ en effectuant un nombre d'opérations polynomiales en $\log(n)$.

## 2.2 Preuve de la proposition 1

Dans cette partie, on va montrer que sous les conditions ci-dessus, le pirate peut retrouver le message $\mu$ à l'aide de quelques opérations Soient $n_1, n_2, \cdots, n_r$ deux à deux premiers entre eux.
On pose : $n = n_1 \times \cdots \times n_r$.
On a : $\mu < \min_{1 \leqslant i \leqslant r} n_i$ et $e \leqslant r$.
Quitte à réordonner les $n_i$, on peut supposer que :

$$n_1 \leqslant n_2 \leqslant \cdots \leqslant n_r.$$

Les $r$ messages chiffrés $c_1, \cdots, c_r$ sont tels que :

$$\begin{cases} c_1 & \equiv \mu^e \pmod{n_1} \\ \vdots & \\ c_r & \equiv \mu^e \pmod{n_r} \end{cases}$$

$\mu$ étant le message commun expédié aux $r$ utilisateurs.

Puisque $n_1, n_2, \cdots, n_r$ sont deux deux premiers entre eux, alors d'après le théorème des restes chinois, $\exists! C$, tel que, $C \equiv \mu^e \pmod{n}$.
On a : $n = n_1 \cdots n_r > n_1^r > \mu^r$
Vu que $e \leqslant r \implies \mu^e \leqslant \mu^r < n$.
D'où $\mu^e < n$.
On en déduit que $\mu^e \in \mathbb{N}$ (c'est un vrai entier).
Par conséquent, pour déchiffrer le message et donc pour trouver $\mu$, il suffit au pirate de calculer la racine $e-$ième de $(\mu^e)^{\frac{1}{e}}$.
Le problème se ramène donc au calcul de la racine $e-$ième de $\mu^e$.

## 2.3 Simulation de la proposition 1

Nous introduisons la fonction **trouvermu**, définie comme suit :

```python
def trouver_mu(c_values, n_values, e):
    """
    Trouve la valeur de 'mu' en utilisant le Théorème des Restes Chinois et la racine e-ème.

    :param c_values: List[int], Liste des valeurs de 'c' (chiffrées) dans le cadre du RSA.
    :param n_values: List[int], Liste des modules 'n' correspondant à chaque valeur de 'c'.
    :param e: int, L'exposant utilisé dans le chiffrement RSA (doit être commun à toutes les paires (c, n)).

    :return: int, La valeur de 'mu' obtenue après déchiffrement et extraction de la racine e-ème.

    Utilise le Théorème des Restes Chinois pour trouver 'x' à partir des valeurs 'c_values' et 'n_values'.
    Calcule ensuite la racine e-ème de 'x' pour obtenir 'mu', en supposant que 'x' est une puissance parfaite de 'e'.
    """
    # Utilisation du Théorème des Restes Chinois pour trouver x
    x = crt(c_values, n_values)

    # Calcul de la racine e-ème de x (x est une puissance parfaite de e)
    mu = x.nth_root(e)

    return mu
```

Figure 4: Fonction trouvermu.

Afin d'illustrer son fonctionnement, nous l'appliquons à un exemple spécifique et observons le résultat obtenu :

```python
# Mesure et affiche le temps nécessaire pour exécuter la fonction `trouver_mu`
exe_t = sage_timeit('trouver_mu(C, N, e)', globals(), preparse=True, number=50)
exe_t = mean(exe_t.series)

print(f'▶ mu = {trouver_mu(C, N, e)} with an execution time = {exe_t}')
```

```
▶ mu = 36 with an execution time = 9.439448826014995e-05µs
```

Figure 5: Example pratique

**Existe-t-il d'autres méthodes de calcul de la racine $e$-ième de $\mu^e$ ?**

## 2.4 Quelques méthodes de calcul de la racine $e$-ième de $C = \mu^e$

Il existe plusieurs méthodes de calcul de la racine $e$-ième de $\mu^e$ parmi lesquelles on a :

### 2.4.1 Méthode de Newton-Raphson

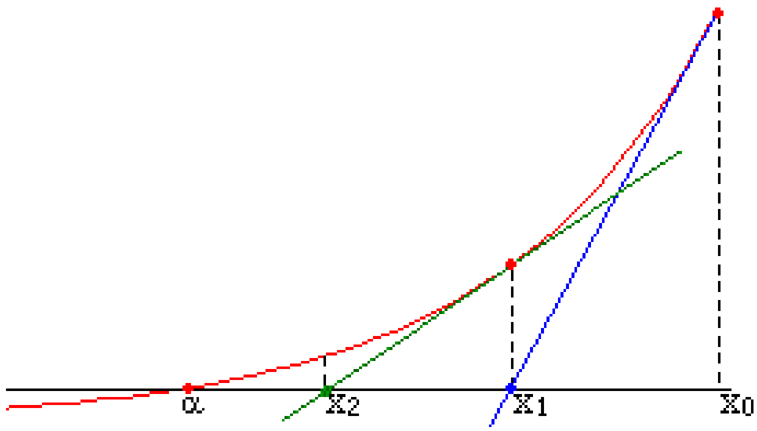



Figure 6: Illustration de la méthode de Newton.

On cherche à résoudre l'équation $f(x) = 0$ où $f$ est une fonction deux fois continument dérivables. Soit $x_o \in D_f$, la tangente à $f$ au point $x_0$ a pour équation :

$$y = f'(x_0)(x - x_0) + f(x_0)$$

L'intersection de cette tangente avec l'axe des abscisses nous donne $x_1$. On repète le procédé avec $x_1$ et on obtient $x_2$ solution de l'équation

$$y = f'(x_1)(x - x_1) + f(x_1) = 0$$

On obtient donc la suite récurrente $(x_n)_{n \in \mathbb{N}}$ définie par :

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

qui, sous de bonnes hypothèses, converge vers une solution $\alpha$ de l'équation $f(x) = 0$ et ce, de façon quadratique.

**La méthode de Newton a une convergence quadratique**

La vitesse de la convergence de la suite $(x_n)_{n\in\mathbb{N}}$ est quadratique.
On suppose que $f \in \mathcal{C}^2$ sur un voisinage de la solution $\alpha$ de l'équation $f(x) = 0$.
On suppose de plus que $\alpha$ est zéro d'ordre 1 de $f$ i.e $f'(\alpha) \neq 0$. D'après la formule de Taylor, on a

$$0 = f(\alpha) = f(x) + (\alpha - x)f'(x) + \frac{(\alpha - x)^2}{2} f''(x)$$

Au bout d'une itération , on obtient

$$N_f(x) - \alpha = x - \frac{f(x)}{f'(x)} - \alpha,$$

ce qui nous permet d'écrire

$$N_f(x) - \alpha = \frac{f''(x)}{2f'(x)}(x - \alpha)^2$$

Pour $I$ compact contenant $x$ et $\alpha$ et inclus dans l'ensemble de définition d $f$, on pose

$$m = \min_{x\in I} |f(x)| \quad M = \max_{x\in I} |f''(x)|$$

Ainsi, on obtient que,

$$\forall x \in I, |N_f(x) - \alpha| \leqslant \frac{M}{2m}|x - \alpha|^2$$

En posant $k = \frac{M}{2m}$, on obtient :

$$|x_{k+1} - \alpha| \leqslant k|x_k - \alpha|^2$$

Par récurrence, on montre que

$$\forall n \in \mathbb{N}, k|x_n - \alpha| \leqslant (k|x_{x_0} - \alpha|)^{2^n}$$

En passant au logarithme népérien , on obtient :

$$\ln |x_n - \alpha| \leqslant 2^n \ln (k|x_0 - \alpha|) - \ln(k)$$

On pose $q = k|x_0 - \alpha|$, alors il vient que :

$$|x_n - \alpha| \leqslant \frac{q^{2^n}}{k}$$

Donc pour $x_0$ tel que $|x_0 - \alpha| \leqslant \frac{1}{k}$, la suite $(x_n)_{n\in\mathbb{N}}$ converge vers $\alpha$ de façon quadratique.

### 2.4.2 Simulation numérique de la méthode de Newton-Raphson

#### Impact d'une initialisation inappropriée

La fonction suivante calcule la racine $e$-ième via la méthode de Newton avec un maximum de 100 itérations :

```python
def newton_method_e_root(x, e, tolerance=1e-10, max_iterations=100):
    """
    Utilise la méthode de Newton pour calculer la racine e-ème d'un nombre 'x'.

    :param x: float, Le nombre dont on cherche la racine e-ème.
    :param e: int, Le degré de la racine.
    :param tolerance: float, Le seuil de tolérance pour l'arrêt de l'algorithme (valeur par défaut : 1e-10).
    :param max_iterations: int, Le nombre maximum d'itérations avant l'arrêt de l'algorithme (valeur par défaut : 100).

    :return: tuple, Retourne un tuple contenant la valeur approchée de la racine, le nombre d'itérations effectuées et l'historic

    Lève une erreur si 'x' est négatif et 'e' est pair, car une racine paire d'un nombre négatif n'est pas définie.
    Utilise une approche itérative pour se rapprocher progressivement de la racine e-ème. La méthode s'arrête lorsque la différer
    """
    if x < 0 and e % 2 == 0:
        raise ValueError("Negative number cannot have an even root")

    y = x
    iterations = 0  # Compteur d'itérations
    history = []  # Pour suivre l'évolution de y à chaque itération

    for _ in range(max_iterations):
        y_prev = y
        y = y - (y**e - x) / (e * y**(e-1))
        history.append(y)
        iterations += 1  # Incrémenter le compteur

        # Vérifier la convergence
        if abs(y - y_prev) < tolerance:
            break

    return y, iterations, history  # Retourner la valeur approchée, le nombre d'itérations et l'historique
```

Figure 7: Méthode de Newton

Nous l'appliquons à un cas simple : $x = 8, e = 3$ :

```python
# Exécuter la fonction pour x = 8 et e = 3 (racine cubique)
racine, iterations, history = newton_method_e_root(8, 3)
racine  #L'algorithme n'a pas pu converger après 100 itération ca mauvaise initialisation

# Exécute la fonction newton_method_e_root pour calculer la racine cubique de 8
racine, iterations, history = newton_method_e_root(8, 3)

# Affiche la valeur de la racine calculée
racine  # Doit afficher le résultat de la racine cubique de 8

# Note:l'algorithme n'ait pas convergé après 100 itérations, ce qui pourrait être dû à une mauvaise initialisation.
# Donc il faut soit ajuster la valeur de départ ou d'augmenter le nombre maximal d'itérations.
```

Figure 8: Application

Et on obtient le résultat suivant :

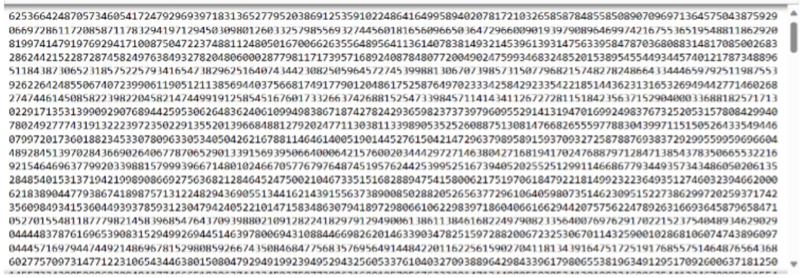


Figure 9: Résultat

**Discussion**

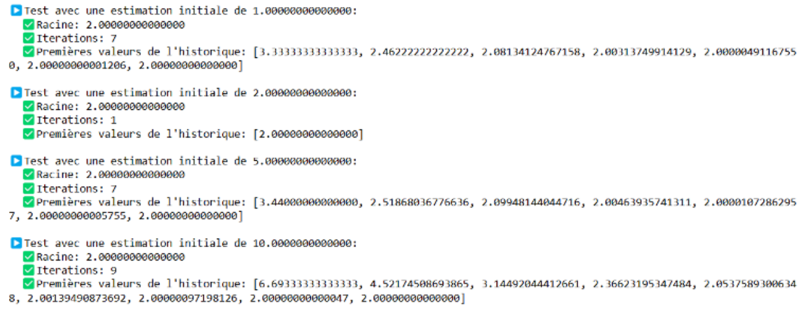
En 100 itérations, le code n'a pas convergé, probablement en raison de l'absence de spécification d'une valeur initiale dans notre appel de fonction, entraînant une initialisation non optimale par le logiciel SageMath.

**Avec différentes initialisations**

```python
def newton_method_e_root_with_ini(x, e, initial_guess=None, tolerance=1e-10, max_iterations=100):
    if x < 0 and e % 2 == 0:
        raise ValueError("Negative number cannot have an even root")

    if initial_guess is None:
        y = x / 2  # Choix d'une valeur initiale par défaut
    else:
        y = initial_guess

    iterations = 0
    history = []

    for _ in range(max_iterations):
        y_prev = y
        y = y - (y**e - x) / (e * y**(e-1))
        history.append(y)
        iterations += 1

        if abs(y - y_prev) < tolerance:
            break

    return y, iterations, history
```

```python
initial_guesses = [1.0, 2.0, 5.0, 10.0]  # Ensemble des estimations initiales à tester

for guess in initial_guesses:
    print(f"▶️Test avec une estimation initiale de {guess}:")
    racine, iterations, history = newton_method_e_root_with_ini(8, 3, initial_guess=guess)
    print(f'''  ✅Racine: {racine}
  ✅Iterations: {iterations}
  ✅Premières valeurs de l'historique: {history[:10]}
''')
```

Figure 10: Simulation avec bonne valeur initiale



```
Test avec une estimation initiale de 1.00000000000000:
  Racine: 2.00000000000000
  Iterations: 7
  Premières valeurs de l'historique: [3.33333333333333, 2.46222222222222, 2.08134124767158, 2.00313749914129, 2.0000049116755
0, 2.00000000001206, 2.00000000000000]

Test avec une estimation initiale de 2.00000000000000:
  Racine: 2.00000000000000
  Iterations: 1
  Premières valeurs de l'historique: [2.00000000000000]

Test avec une estimation initiale de 5.00000000000000:
  Racine: 2.00000000000000
  Iterations: 7
  Premières valeurs de l'historique: [3.44000000000000, 2.51868036776636, 2.09948144044716, 2.00463935741311, 2.0000107286295
7, 2.00000000005755, 2.00000000000000]

Test avec une estimation initiale de 10.0000000000000:
  Racine: 2.00000000000000
  Iterations: 9
  Premières valeurs de l'historique: [6.69333333333333, 4.52174508693865, 3.14492044412661, 2.36623195347484, 2.0537589300634
8, 2.00139490873692, 2.00000097198126, 2.00000000000047, 2.00000000000000]
```

Figure 11: Résultat Simulation avec bonne valeur initiale

**Discussion**

1. En testant plusieurs estimations initiales, nous observons que le choix de cette valeur influence significativement la convergence de l'algorithme vers la racine réelle, et influence le nombre d'itérations avant la convergence

2. Cette simulation démontre l'importance d'une bonne estimation initiale pour la convergence de la méthode de Newton. Sans une valeur initiale appropriée, même avec un nombre élevé d'itérations, l'algorithme peut échouer à converger vers la racine correcte. Il est donc crucial d'ajuster soit la valeur de départ soit le nombre maximal d'itérations pour assurer la précision de la méthode dans la recherche de la racine e-ème.

#### 2.4.3 Décryptage du message par l'algorithme de recherche par dichotomie

On a

$$f(x) = x^e - C$$

Vu que $\mu^e < n$ alors $\mu \in D = \left[1, \lfloor \ln\left(\frac{n}{e}\right) \rfloor\right]$.
On applique donc à $f$ l'algorithme de reherche par dichotomie sur l'intervalle D.

**Principe**

On pose $m = \frac{1+b}{2}$, $b = \lfloor \ln\left(\frac{n}{e}\right) \rfloor$. Si $f(1)$ et $f(m)$ sont de signes opposés alors la solution se trouve dans l'intervalle $[1, m]$. Et on applique le même procédé. Sinon, la solution se trouve dans $[m, b]$ et ainsi de suite.

**La méthode par dichotomie a une convergence logarithmique, plus lente que celle de Newton**

Soit $k$ le nombre total d'opérations à effectuer pour déterminer le message $\mu$ en fonction de la taille $n$ de $D$ qui vaut $n = \lfloor \ln\left(\frac{n}{e}\right) \rfloor - 1$.
Par principe de la dichotomie, $n$ est divisée par 2 à chaque itération de la boucle de recherche. La taille finale de l'intervalle est donc $\left\lfloor \frac{n}{2^k} \right\rfloor$. Le message $\mu$ étant entier, donc la plus petite taille possible de l'intervalle finale est 1, donc $1 \leqslant \frac{n}{2^k}$. Ainsi, on obtient $2^k \leqslant n$ puis finalement $k \leqslant \frac{\ln(n)}{\ln(2)} = \log_2(n)$, ce qui termine la preuve.

# 3 Choix de e :

a. Le choix de $e$ est un compromis entre la sécurité (choix de $e$ très grand) et la performance du chiffrement ($e$ très petit)

b. Pourquoi $e = 3, 257, 65537$ ?

- Nombre de Fermat:
  C'est est un nombre qui s'écrit sous la forme $2^{2^n} + 1$ , si on note $F_n$ le nombre de Fermat d'ordre $n$ : $3, 257et65537$ sont respectivement $F_1$ , $F_3$ et $F_4$. Ainsi, chacune de leurs écritures en base binaire contient uniquement 2 bits 1, ce qui réduit le cout de chiffrement.
- Ce choix sera aussi justifié par le temps d'exécution du chiffrage en fonction du nombre de 1 en représentation binaire

`Remarque:` Le nombre 65537 est considéré comme un choix optimal car il établit un équilibre entre la facilité de chiffrement et la complexité du déchiffrement.