Julian Ray-Constanty
Sacha Trouvé

G5B

R3.09 – Cryptographie
TP 1 et 2

# Sommaire

Julian Ray-Constanty G5B
Sacha Trouvé

# Introduction

Ce projet a pour objectif de développer, en langage Rust, un système de chiffrement et déchiffrement utilisant le chiffrement de Vigenère, avec une entrée de texte pouvant être fournie soit manuellement, soit à partir de fichiers. Nous avons également implémenté la méthode de Kasiski, permettant d'analyser un texte chiffré afin d'estimer la longueur de la clé de chiffrement.

# Chiffrement et déchiffrement de Vigenère

# Prétraitement du texte

Le texte est traité avant d'être soumis à l'algorithme. En effet chaque caractère est transformé en une valeur numérique grâce à la table ASCII ; chaque caractère possède son code décimal. Cette méthode à l'avantage de pouvoir aussi gérer les caractères spéciaux (sauf les caractères à accents.) ainsi que de différencier les caractères alphabétiques minuscules et majuscules.

# Algorithme de chiffrement

Tout d'abord on redimensionne la clé pour qu'elle ait la même longueur que le message, puis on traite les caractères ASCII imprimables (entre ' ' et '~').

Ensuite, on applique cycliquement les règles suivantes pour chaque caractère du message :

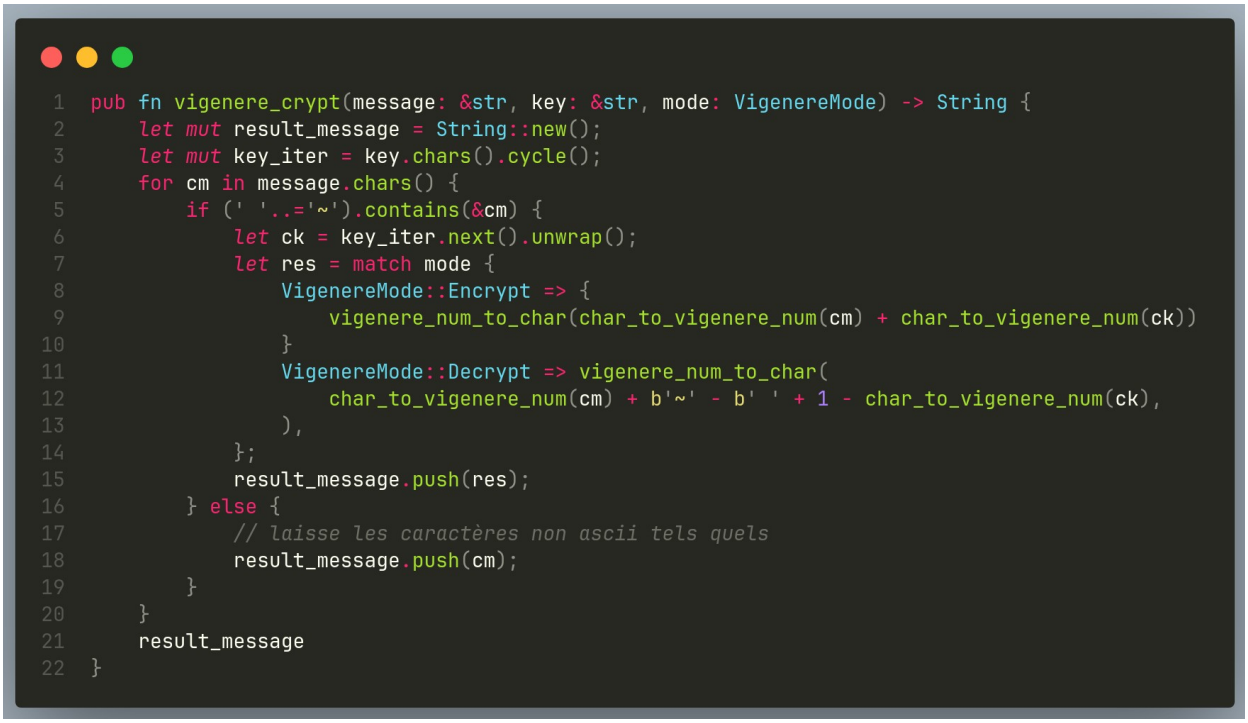
- si le caractère n'est pas imprimable on le garde tel quel, sinon :
- conversion du caractère du message et de la clé en code ASCII
- addition des codes de ces deux caractères (modulo 95 pour ne pas dépasser) en un nouveau caractère

les caractères sont ensuite combiné en une chaine de caractères.

# Algorithme de déchiffrement

Le déchiffrement inverse l'opération ; il s'agit du même procédé mais au lieu d'additionner un caractère du message et un caractère de la clé, on effectue une soustraction.

# Code principal :



```rust
pub fn vigenere_crypt(message: &str, key: &str, mode: VigenereMode) -> String {
    let mut result_message = String::new();
    let mut key_iter = key.chars().cycle();
    for cm in message.chars() {
        if (' '..='~').contains(&cm) {
            let ck = key_iter.next().unwrap();
            let res = match mode {
                VigenereMode::Encrypt => {
                    vigenere_num_to_char(char_to_vigenere_num(cm) + char_to_vigenere_num(ck))
                }
                VigenereMode::Decrypt => vigenere_num_to_char(
                    char_to_vigenere_num(cm) + b'~' - b' ' + 1 - char_to_vigenere_num(ck),
                ),
            };
            result_message.push(res);
        } else {
            // laisse les caractères non ascii tels quels
            result_message.push(cm);
        }
    }
    result_message
}
```

# Méthode de Kasiski

La méthode de Kasiski est une technique permettant de retrouver la longueur probable de la clé utilisée dans un chiffrement de Vigenère.

# Étape 1 : Détection des répétitions

Le texte chiffré est parcouru pour identifier des séquences répétées (souvent de 3 lettres ou plus).
Ces séquences sont enregistrées dans une structure de type table de hachage, associant la séquence répétée aux positions où elle apparaît dans le texte.

# Étape 2 : Calcul des distances

Pour chaque séquence répétée, les distances entre occurrences successives sont calculées. Ces distances reflètent l'espacement entre réutilisations de la même portion de clé. Elles sont stockées dans des vecteurs afin de pouvoir être analysées globalement.

# Étape 3 : Détermination de la longueur de la clé

Les distances obtenues sont utilisées pour calculer des PGCD.
La longueur de clé probable correspond aux diviseurs qui apparaissent le plus fréquemment.
Cette estimation permet ensuite d'affiner l'analyse cryptographique du texte chiffré.

# Code principal :

```rust
pub fn kasiski(message: &str) {
    // Filtrer le message pour ne garder que les caractères ASCII imprimables
    let filtered: String = message
        .chars()
        .filter(|&c| (' '..='~').contains(&c))
        .collect();
    let repet = build_repet_table(&filtered);

    if repet.is_empty() {
        println!("Echantillon trop petit pour Kasiski");
        return;
    }

    let first_dist = repet[0].1;
    let mut candidats = divisors(first_dist);

    for &(_, dist) in repet.iter().skip(1) {
        let mut temp: Vec<usize> = Vec::new();
        for &cand in &candidats {
            let g = compute_gcd(cand, dist);
            if g > 1 {
                temp.push(g);
            }
        }

        temp.sort_unstable();
        temp.dedup();

        if !temp.is_empty() {
            candidats = temp;
        }
    }

    candidats.sort_unstable();
    candidats.dedup();

    if candidats.is_empty() {
        println!("?");
    } else {
        println!("Tailles de clé possibles : {candidats:?}");
    }
}
```

## Jeux de test

Plusieurs tests unitaires et fonctionnels ont été mis en place pour valider le projet :

- **Test de chiffrement/déchiffrement** : vérifier que le texte original est bien retrouvé après un cycle complet.
- **Test de normalisation du texte** : s'assurer que la conversion en majuscules et la suppression des caractères non alphabétiques sont correctes.
- **Test de la méthode de Kasiski** : vérifier que la longueur de clé retrouvée correspond bien à la clé utilisée lors du chiffrement.
- **Test sur fichiers** : confirmer que la lecture et l'écriture de fichiers texte fonctionne correctement avec de grandes entrées.

## Conclusion

Ce projet démontre l'utilisation efficace de Rust pour implémenter un chiffrement classique comme le Vigenère et pour mettre en œuvre une attaque cryptographique (méthode de Kasiski). L'approche combinant structures de données performantes (Vec, HashMap) et gestion stricte des types de Rust offre une robustesse importante.

Les résultats des tests confirment la validité des implémentations et montrent que le projet peut être utilisé à la fois pour l'apprentissage et comme base pour des applications plus complexes de cryptanalyse.