UNIVERSITÉ DE
SHERBROOKE

Rapport Devoir 1

Cours IFT814 – Cryptographie

Faculté des Sciences

Département Informatique

Formateur : Martin Fiset

Auteur : Mamadou Senghor

CIP : senm1912

Table des matières

1 Introduction

Ce devoir porte sur l'implémentation et l'étude de trois schémas de chiffrement classiques à longueur fixe : le chiffre de César à double décalage, le masque jetable (One-Time Pad), et la substitution monoalphabétique. L'objectif est de mettre en pratique les concepts de base de la cryptographie en programmant les algorithmes de génération de clé, de chiffrement et de déchiffrement, puis d'analyser leur sécurité face à un adversaire (Eve).

Chaque schéma est évalué à travers des scénarios de communication entre Alice, Bob et Eve, avec un message clair fixé de 32 lettres. Les expériences permettent de comparer la robustesse des différents systèmes face aux attaques par force brute ou par analyse fréquentielle.

2 Chiffre de César à double décalage (ΠCésar1)

2.1 Implémentation des algorithmes (Gen1, E1, D1)

Le schéma utilise deux clés entières k1, k2 ∈ {0, ...,25}.

Pour un message m de 32 lettres (alphabet a–z) :

- Positions paires (0,2, 4, ...) : ajout de k1 modulo 26 ;
- Positions impaires (1,3, 5, ...) : ajout de k2 modulo 26.

Extrait de code Python :

```
import random

def Gen1():
    """
    Génère une clé (k1, k2) pour le chiffre de César double décalage.
    Chaque clé est un entier entre 0 et 25 inclus.
    """
    k1 = random.randint(0, 25)
    k2 = random.randint(0, 25)
```

```
    return (k1, k2)


def E1(k, m):
    """
    Chiffrement César double décalage.
    - k : tuple (k1, k2)
    - m : message clair (chaîne en minuscules, a-z)
    Retourne : chaîne chiffrée c
    """
    k1, k2 = k
    c = ""
    for i, ch in enumerate(m):
        x = ord(ch) - ord('a')   # lettre → nombre [0..25]
        if i % 2 == 0:         # position paire
            y = (x + k1) % 26
        else:                  # position impaire
            y = (x + k2) % 26
        c += chr(y + ord('a'))   # nombre → lettre
    return c
```

```
def D1(k, c):
    """
    Déchiffrement César double décalage.
    - k : tuple (k1, k2)
    - c : cryptogramme
    Retourne : message clair m
    """
    k1, k2 = k
    m = ""
    for i, ch in enumerate(c):
        y = ord(ch) - ord('a')
        if i % 2 == 0:          # position paire
            x = (y - k1) % 26
        else:                   # position impaire
            x = (y - k2) % 26
        m += chr(x + ord('a'))
    return m
```

```
def run_execution(exec_id):
    m = "ceciestlemessageclairadechiffrer"
    k = Gen1()
    c = E1(k, m)
    m2 = D1(k, c)

    print(f"=== Exécution {exec_id} ===")
    print("Alice ->")
    print("  Message clair :", m)
    print("  Clé (k1, k2) :", k)
    print("  Cryptogramme :", c)

    print("Eve ->")
    print("  Cryptogramme intercepté :", c)

    print("Bob ->")
    print("  Cryptogramme reçu :", c)
    print("  Clé (k1, k2) :", k)
    print("  Message déchiffré :", m2)
    print()

if __name__ == "__main__":
    for i in range(1, 4):
        run_execution(i)
```

2.2 Scénario de communication

Message transmis : m = "ceciestlemessageclairadechiffrer"

Les exécutions suivantes ont été réalisées de manière indépendante :

```
=== Exécution 1 ===
Alice ->
  Message clair : ceciestlemessageclairadechiffrer
  Clé (k1, k2) : (4, 17)
  Cryptogramme : gvgzijxcidijwrkvgcezvrhvgymwjiii
Eve ->
  Cryptogramme intercepté : gvgzijxcidijwrkvgcezvrhvgymwjiii
Bob ->
  Cryptogramme reçu : gvgzijxcidijwrkvgcezvrhvgymwjiii
  Clé (k1, k2) : (4, 17)
  Message déchiffré : ceciestlemessageclairadechiffrer
```

```
=== Exécution 2 ===
Alice ->
  Message clair : ceciestlemessageclairadechiffrer
  Clé (k1, k2) : (19, 9)
  Cryptogramme : vnvrxbmuxvxbljznvutrkjwnvqboyaxa
Eve ->
  Cryptogramme intercepté : vnvrxbmuxvxbljznvutrkjwnvqboyaxa
Bob ->
  Cryptogramme reçu : vnvrxbmuxvxbljznvutrkjwnvqboyaxa
  Clé (k1, k2) : (19, 9)
  Message déchiffré : ceciestlemessageclairadechiffrer
```

```
=== Exécution 3 ===
Alice ->
  Message clair : ceciestlemessageclairadechiffrer
  Clé (k1, k2) : (9, 13)
  Cryptogramme : lrlvnfcynznfbnprlyjvanmrlursoene
Eve ->
  Cryptogramme intercepté : lrlvnfcynznfbnprlyjvanmrlursoene
Bob ->
  Cryptogramme reçu : lrlvnfcynznfbnprlyjvanmrlursoene
  Clé (k1, k2) : (9, 13)
  Message déchiffré : ceciestlemessageclairadechiffrer
```

2.3 Analyse de sécurité

Ce qu'Eve voit **:** uniquement le cryptogramme **c**.
Taille de l'espace des clés **:** 26×26 = 676 combinaisons.

2.3.1 Attaques possibles pour Eve :

2.3.1.1 Force brute

Tester les 676 couples (k1, k2) est trivial pour une machine moderne. Pour chaque essai, Eve calcule D1((k1, k2), c) et recherche un texte lisible en français. L'opération est quasi instantanée.

2.3.1.2 Analyse fréquentielle séparée

Comme les positions paires et impaires sont chiffrées indépendamment avec des décalages fixes k1 et k2, Eve peut :

- Séparer **c** en deux sous-séquences : indices pairs et indices impairs ;
- Faire une analyse de fréquences sur chaque sous-séquence ;
- Supposer que la lettre la plus fréquente correspond à **e** et déduire un décalage candidat pour k1 et k2;
- Affiner par essais locaux pour obtenir un texte français cohérent.

Cette attaque exploite la structure conservant la distribution relative des lettres, et fonctionne très bien pour des textes de longueur 32 (surtout si le message contient des structures linguistiques reconnaissables).

2.3.2 Comparaison avec le César classique :

César simple : espace de clés = 26 (trivial).

César double : espace de clés = 676, donc légèrement meilleur mais toujours PETIT.

En pratique, la double clé **n'apporte pas de résistance significative** : l'analyse fréquentielle reste possible (en séparant paires/impaires) et la force brute est facile.

Conclusion
Le schéma ΠCésar1 offre une résistance marginalement supérieure au César simple, mais reste **totalement insuffisant** pour protéger de l'adversaire moderne. Il est vulnérable à des attaques automatiques par force brute et à l'analyse fréquentielle. Il faut employer des schémas bien plus solides.

3 Masque jetable (ΠOTP)

3.1 Implémentation des algorithmes (Gen2, E2, D2)

Le masque jetable est un algorithme de chiffrement **parfaitement secret** si la clé est utilisée une seule fois, de longueur au moins égale au message, et totalement aléatoire.

Le masque jetable (One-Time Pad, OTP) encode chaque caractère du message clair (a–z) en **5 bits** :

a=00000, b=00001, ..., z=11001

Ainsi, un message de 32 lettres est représenté sur 32×5=160 bits.

La clé k est une suite aléatoire de 160 bits, générée par Gen2() :

Le chiffrement et le déchiffrement s'effectuent par **XOR bit-à-bit** :

$c=m \oplus k$

$m=c \oplus k$

Extrait de code en Python :

```
import secrets


# ---------- utilitaires d'encodage 5 bits ----------

def char_to_5bits(ch: str) -> str:

  """Renvoie une chaîne de 5 caractères '0'/'1' pour la lettre ch (a-z)."""

  val = ord(ch) - ord('a')

  return format(val, '05b')  # représentation binaire sur 5 bits
```

```
def bits5_to_char(bits5: str) -> str:
    """Transforme 5 bits en caractère a-z."""
    val = int(bits5, 2)
    return chr(val + ord('a'))



def str_to_bits5_string(m: str) -> str:
    """Convertit une chaîne m en une chaîne de bits '0'/'1' groupés par 5 (sans séparateur)."""
    return ''.join(char_to_5bits(ch) for ch in m)

def bits_string_to_str(bits: str) -> str:
    """Convertit une chaîne de bits multiple de 5 en texte (5 bits -> une lettre)."""
    if len(bits) % 5 != 0:
        raise ValueError("Le nombre de bits doit être multiple de 5")
    return ''.join(bits5_to_char(bits[i:i+5]) for i in range(0, len(bits), 5))

def group_bits(bits: str, group=5, sep=' '):
    """Retourne les bits groupés (ex: '01010 11100 ...') pour affichage."""
    return sep.join(bits[i:i+group] for i in range(0, len(bits), group))

# ---------- OTP primitives ----------
def Gen2(n_bits=160) -> str:
    """Génère une clé aléatoire de n_bits sous forme de chaîne '0'/'1'."""
    return ''.join('1' if secrets.randbelow(2) else '0' for _ in range(n_bits))
```

```
def E2(k_bits: str, m_bits: str) -> str:
    """Chiffrement OTP: c = m XOR k (bits as strings)."""
    if len(k_bits) != len(m_bits):
        raise ValueError("k et m doivent avoir même longueur en bits")
    return ''.join('1' if (kb != mb) else '0' for kb, mb in zip(k_bits, m_bits))

def D2(k_bits: str, c_bits: str) -> str:
    """Déchiffrement OTP: m = c XOR k (idem que E2)."""
    return E2(k_bits, c_bits)  # XOR symétrique

# ---------- paramètres ----------
MESSAGE =  "ceciestlemessageclairadechiffrer"
assert len(MESSAGE) == 32, "Le message doit faire 32 lettres"

# ---------- exécutions ----------
def run_one(exec_id: int):
    print("="*72)
    print(f"EXÉCUTION {exec_id}")
    print("-"*72)
    # Alice
    m_text = MESSAGE
    m_bits = str_to_bits5_string(m_text)  # 160 bits
    k_bits = Gen2(len(m_bits))          # 160 bits
    c_bits = E2(k_bits, m_bits)
    m2_bits = D2(k_bits, c_bits)
    m2_text = bits_string_to_str(m2_bits)
```

```
    print("ALICE ->")
    print("  Message (texte) : ", m_text)
    print("  \n Message (bits)  : \n", group_bits(m_bits, 5))
    print("  \n Clé k (bits)    : \n", group_bits(k_bits, 5))
    print("  \n Cryptogramme c  : \n", group_bits(c_bits, 5))
    print()

    # EVE
    print("EVE ->")
    print("  Cryptogramme intercepté (c) : \n", group_bits(c_bits, 5))
    print()

    # BOB
    print("BOB ->")
    print("  Cryptogramme reçu (c) : \n", group_bits(c_bits, 5))
    print("  \n Clé reçue (k)       : \n", group_bits(k_bits, 5))
    print("  \n Message déchiffré    : ", m2_text)
    print("  \n Vérification (m == m2) : \n", m_text == m2_text)
    print()

if __name__ == "__main__":
    for i in range(1, 4):   # 3 exécutions indépendantes
        run_one(i)
```

3.2 Scénario de communication

Message transmis (fixé par l'énoncé) :

m = "ceciestlemessageclairadechiffrer"

Encodé en 5 bits par caractère → m = 160 bits.

Le scénario est répété trois fois de manière indépendante :

```
========================================================================
EXÉCUTION 1
------------------------------------------------------------------------
ALICE ->
  Message (texte) :  ceciestlemessageclairadechiffrer

 Message (bits)  :
 00010 00100 00010 01000 00100 10010 10011 01011 00100 01100 00100 10010 10010 00000 00110 00100 00010
01011 00000 01000 10001 00000 00011 00100 00010 00111 01000 00101 00101 10001 00100 10001

 Clé k (bits)    :
 01110 01111 11010 11101 01011 01000 11111 10001 01111 10001 01011 01111 10001 10010 00111 10001 01101
11110 10010 10001 00111 10111 01111 00001 01001 11001 00110 11010 10101 11100 01000 11111

 Cryptogramme c  :
 01100 01011 11000 10101 01111 11010 01100 11010 01011 11101 01111 11101 00011 10010 00001 10101 01111
10101 10010 11001 10110 10111 01100 00101 01011 11110 01110 11111 10000 01101 01100 01110

EVE ->
  Cryptogramme intercepté (c) :
 01100 01011 11000 10101 01111 11010 01100 11010 01011 11101 01111 11101 00011 10010 00001 10101 01111
10101 10010 11001 10110 10111 01100 00101 01011 11110 01110 11111 10000 01101 01100 01110

BOB ->
  Cryptogramme reçu (c) :
 01100 01011 11000 10101 01111 11010 01100 11010 01011 11101 01111 11101 00011 10010 00001 10101 01111
10101 10010 11001 10110 10111 01100 00101 01011 11110 01110 11111 10000 01101 01100 01110

 Clé reçue (k)         :
 01110 01111 11010 11101 01011 01000 11111 10001 01111 10001 01011 01111 10001 10010 00111 10001 01101
11110 10010 10001 00111 10111 01111 00001 01001 11001 00110 11010 10101 11100 01000 11111

 Message déchiffré     :  ceciestlemessageclairadechiffrer
```

```
======================================================================
EXÉCUTION 2
----------------------------------------------------------------------
ALICE ->
  Message (texte) :  ceciestlemessageclairadechiffrer

 Message (bits)  :
 00010 00100 00010 01000 00100 10010 10011 01011 00100 01100 00100 10010 10010 00000 00110 00100 00010
01011 00000 01000 10001 00000 00011 00100 00010 00111 01000 00101 00101 10001 00100 10001

 Clé k (bits)    :
 01101 01101 01101 11110 00000 00110 10010 10101 00100 00100 10000 11000 00111 00101 00001 11011 11001
01100 01110 01010 11100 01001 11011 11101 10111 11110 11110 00110 10110 10101 11101 01000

 Cryptogramme c  :
 01111 01001 01111 10110 00100 10100 00001 11110 00000 01000 10100 01010 10101 00101 00111 11111 11011
00111 01110 00010 01101 01001 11000 11001 10101 11001 10110 00011 10011 00100 11001 11001

EVE ->
  Cryptogramme intercepté (c) :
 01111 01001 01111 10110 00100 10100 00001 11110 00000 01000 10100 01010 10101 00101 00111 11111 11011
00111 01110 00010 01101 01001 11000 11001 10101 11001 10110 00011 10011 00100 11001 11001

BOB ->
  Cryptogramme reçu (c) :
 01111 01001 01111 10110 00100 10100 00001 11110 00000 01000 10100 01010 10101 00101 00111 11111 11011
00111 01110 00010 01101 01001 11000 11001 10101 11001 10110 00011 10011 00100 11001 11001

 Clé reçue (k)         :
 01101 01101 01101 11110 00000 00110 10010 10101 00100 00100 10000 11000 00111 00101 00001 11011 11001
01100 01110 01010 11100 01001 11011 11101 10111 11110 11110 00110 10110 10101 11101 01000

 Message déchiffré     :  ceciestlemessageclairadechiffrer
```

```
======================================================================
EXÉCUTION 3
----------------------------------------------------------------------
ALICE ->
  Message (texte) :  ceciestlemessageclairadechiffrer

 Message (bits)  :
 00010 00100 00010 01000 00100 10010 10011 01011 00100 01100 00100 10010 10010 00000 00110 00100 00010
01011 00000 01000 10001 00000 00011 00100 00010 00111 01000 00101 00101 10001 00100 10001

 Clé k (bits)    :
 11101 11110 01010 01010 01101 10011 00001 00100 00011 00110 11110 10110 00000 10100 00100 01111 10100
01111 00110 01101 11100 11010 00100 01110 10101 00111 10010 10010 00001 01011 10110 10001

 Cryptogramme c  :
 11111 11010 01000 00010 01001 00001 10010 01111 00111 01010 11010 00100 10010 10100 00010 01011 10110
00100 00110 00101 01101 11010 00111 01010 10111 00000 11010 10111 00100 11010 10010 00000

EVE ->
  Cryptogramme intercepté (c) :
 11111 11010 01000 00010 01001 00001 10010 01111 00111 01010 11010 00100 10010 10100 00010 01011 10110
00100 00110 00101 01101 11010 00111 01010 10111 00000 11010 10111 00100 11010 10010 00000

BOB ->
  Cryptogramme reçu (c) :
 11111 11010 01000 00010 01001 00001 10010 01111 00111 01010 11010 00100 10010 10100 00010 01011 10110
00100 00110 00101 01101 11010 00111 01010 10111 00000 11010 10111 00100 11010 10010 00000

 Clé reçue (k)         :
 11101 11110 01010 01010 01101 10011 00001 00100 00011 00110 11110 10110 00000 10100 00100 01111 10100
01111 00110 01101 11100 11010 00100 01110 10101 00111 10010 10010 00001 01011 10110 10001

 Message déchiffré     :  ceciestlemessageclairadechiffrer
```

3.3 Analyse de sécurité (point de vue d'Eve)

Eve peut-elle déchiffrer ?

Non. Eve ne peut absolument pas retrouver le message clair m à partir du seul cryptogramme c.

Raison fondamentale

Le masque jetable, lorsqu'il est utilisé avec une clé **secrète**, **aléatoire**, **non réutilisée** et de longueur au moins égale au message, offre une **sécurité inconditionnelle**.

Ce n'est pas une question de puissance de calcul, mais une propriété théorique démontrée par le **théorème de Shannon** : l'OTP est un schéma à sécurité parfaite.

Explication intuitive

Pour un cryptogramme c de 160 bits intercepté par Eve :

- Tout message m' de 32 caractères (160 bits) est un **candidat plausible** pour être le message original.
- Pour chaque m', il existe une clé k' telle que k'=m'⊕c
- Puisque la vraie clé k est choisie uniformément au hasard, **Eve ne peut distinguer** la vraie paire (m,k) de n'importe quelle autre paire (m',k').

Ainsi, le cryptogramme ne donne **aucune information** sur le contenu du message.

Conséquences :

- Aucune technique de cryptanalyse (analyse de fréquences, force brute, etc.) ne peut fonctionner.
- La seule information qu'Eve obtient est la longueur du message (160 bits = 32 caractères).
- Le contenu du message reste parfaitement secret.

Conclusion :

La sécurité de l'OTP est **totale** du point de vue théorique.
Ses limites sont uniquement **pratiques** : nécessité d'un canal sûr pour partager des clés aussi longues que les messages, et impossibilité de réutiliser une même clé. Ces contraintes rendent l'OTP peu pratique, sauf dans des contextes très spécifiques (diplomatie, espionnage, communications militaires).

4 Chiffre de substitution mono-alphabétique (ΠSub)

4.1 Implémentation des algorithmes (Gen3, E3, D3)

Une substitution monoalphabétique remplace chaque lettre du message clair par une autre lettre, selon une règle fixe définie par une clé. La clé k est une **permutation** (un réarrangement) de l'alphabet de 26 lettres.

- **Chiffrement :** Pour chaque lettre L dans m, on la remplace par k[L].
- **Déchiffrement :** Pour chaque lettre L'dans c, on trouve la lettre L telle que k[L] = L'. Cela nécessite d'avoir l'inverse de la permutation k.

Extrait de code en Python :

```
import random
import string

ALPHABET = list(string.ascii_lowercase)
MESSAGE = "ceciestlemessageclairadechiffrer"

def Gen3():
    """Génère une clé de substitution : enc_map (dict), dec_map (dict)."""
    shuffled = ALPHABET[:]
    random.shuffle(shuffled)
    enc_map = dict(zip(ALPHABET, shuffled))
    dec_map = {v: k for k, v in enc_map.items()}
    return enc_map, dec_map

def E3(k, m):
    """Chiffrement par substitution monoalphabétique."""
    enc_map, _ = k
    return ''.join(enc_map[ch] for ch in m)

def D3(k, c):
    """Déchiffrement par substitution (utilise l'inverse)."""
    _, dec_map = k
    return ''.join(dec_map[ch] for ch in c)
```

```
def format_key(enc_map):
    """Retourne une chaîne lisible de la permutation: a->x, b->y, ..."""
    pairs = [f"{a}->{enc_map[a]}" for a in ALPHABET]
    # regrouper par 6 paires par ligne pour meilleure lisibilité
    line_len = 6
    lines = []
    for i in range(0, len(pairs), line_len):
        lines.append(', '.join(pairs[i:i+line_len]))
    return '\n    '.join(lines)

def run_one(exec_id):
    print("="*80)
    print(f"EXÉCUTION {exec_id}")
    print("-"*80)

    # Génération
    k = Gen3()
    c = E3(k, MESSAGE)
    m2 = D3(k, c)

    # Alice
    print("ALICE ->")
    print("  Message (texte) : ", MESSAGE)
    print("  Clé k (permutation encodage a->...) :")
    print("    " + format_key(k[0]))
    print("  Cryptogramme c : ", c)
```

```
    print()

    # Eve
    print("EVE ->")
    print("  Cryptogramme intercepté (c) : ", c)
    print()

    # Bob
    print("BOB ->")
    print("  Cryptogramme reçu (c) : ", c)
    print("  Clé (k) reçue :")
    print("    " + format_key(k[0]))
    print("  Message déchiffré : ", m2)
    print("  Vérification (m == m2): ", MESSAGE == m2)
    print()

if __name__ == "__main__":
    # Optionnel : fixer la seed pour obtenir des runs reproductibles (décommenter si désiré)
    # random.seed(12345)

    for i in range(1, 4):
        run_one(i)
```

4.2 Scénario de communication

La clé k est une permutation aléatoire, donc les traces seront différentes à chaque fois.

```
================================================================================
EXÉCUTION 1
--------------------------------------------------------------------------------
ALICE ->
  Message (texte) :  ceciestlemessageclairadechiffrer
  Clé k (permutation encodage a->...) :
    a->e, b->x, c->f, d->l, e->d, f->p
    g->r, h->j, i->b, j->z, k->u, l->i
    m->k, n->o, o->y, p->a, q->g, r->q
    s->n, t->s, u->h, v->w, w->v, x->m
    y->t, z->c
  Cryptogramme c :  fdfbdnsidkdnnerdfiebqeldfjbppqdq

EVE ->
  Cryptogramme intercepté (c) :  fdfbdnsidkdnnerdfiebqeldfjbppqdq

BOB ->
  Cryptogramme reçu (c) :  fdfbdnsidkdnnerdfiebqeldfjbppqdq
  Clé (k) reçue :
    a->e, b->x, c->f, d->l, e->d, f->p
    g->r, h->j, i->b, j->z, k->u, l->i
    m->k, n->o, o->y, p->a, q->g, r->q
    s->n, t->s, u->h, v->w, w->v, x->m
    y->t, z->c
  Message déchiffré :  ceciestlemessageclairadechiffrer
  Vérification (m == m2):  True
```

```
================================================================================
EXÉCUTION 2
--------------------------------------------------------------------------------
ALICE ->
  Message (texte) :  ceciestlemessageclairadechiffrer
  Clé k (permutation encodage a->...) :
    a->t, b->c, c->g, d->o, e->w, f->j
    g->f, h->i, i->m, j->d, k->s, l->l
    m->y, n->b, o->v, p->u, q->k, r->q
    s->p, t->a, u->z, v->x, w->e, x->h
    y->r, z->n
  Cryptogramme c :  gwgmwpalwywpptfwgltmqtowgimjjqwq

EVE ->
  Cryptogramme intercepté (c) :  gwgmwpalwywpptfwgltmqtowgimjjqwq

BOB ->
  Cryptogramme reçu (c) :  gwgmwpalwywpptfwgltmqtowgimjjqwq
  Clé (k) reçue :
    a->t, b->c, c->g, d->o, e->w, f->j
    g->f, h->i, i->m, j->d, k->s, l->l
    m->y, n->b, o->v, p->u, q->k, r->q
    s->p, t->a, u->z, v->x, w->e, x->h
    y->r, z->n
  Message déchiffré :  ceciestlemessageclairadechiffrer
  Vérification (m == m2):  True
```

```
================================================================================
EXÉCUTION 3
--------------------------------------------------------------------------------
ALICE ->
  Message (texte) :  ceciestlemessageclairadechiffrer
  Clé k (permutation encodage a->...) :
    a->e, b->c, c->w, d->y, e->o, f->u
    g->z, h->b, i->v, j->s, k->r, l->h
    m->p, n->x, o->t, p->j, q->n, r->i
    s->g, t->l, u->a, v->m, w->f, x->d
    y->k, z->q
  Cryptogramme c :  wowvoglhopoggezowhevieyowbvuuioi

EVE ->
  Cryptogramme intercepté (c) :  wowvoglhopoggezowhevieyowbvuuioi

BOB ->
  Cryptogramme reçu (c) :  wowvoglhopoggezowhevieyowbvuuioi
  Clé (k) reçue :
    a->e, b->c, c->w, d->y, e->o, f->u
    g->z, h->b, i->v, j->s, k->r, l->h
    m->p, n->x, o->t, p->j, q->n, r->i
    s->g, t->l, u->a, v->m, w->f, x->d
    y->k, z->q
  Message déchiffré :  ceciestlemessageclairadechiffrer
  Vérification (m == m2):  True
```

4.3 Analyse de sécurité du point de vue d'Eve

Eve peut-elle déchiffrer le message ?

Oui. Eve peut très probablement déchiffrer le message sans la clé, grâce à des techniques de cryptanalyse par analyse des fréquences et des motifs linguistiques. La longueur du message (32 caractères), bien que relativement courte, est suffisante pour amorcer une analyse manuelle ou semi-automatisée.

4.3.1 Impracticabilité de l'attaque par force brute

La sécurité *théorique* du système repose sur la taille gigantesque de son espace de clés.

- **Taille de l'espace des clés :** Une clé est une permutation de l'alphabet, il y a donc **$26! \approx 4.03 \times 10^{26}$** clés possibles.
- **Calcul de complexité :** Même en supposant une capacité de test de **10^{12} clés par seconde** (un trillion), il faudrait **$\approx 3.15 \times 10^{19}$ essais par année**.

- **Temps moyen pour trouver la clé :** Le temps moyen pour explorer la moitié de l'espace serait d'environ **12,8 millions d'années**.
- **Conclusion :** Une attaque par force brute est totalement **impossible** et **irréaliste** avec les technologies de calcul actuelles ou prévisibles.

4.3.2 Efficacité de la cryptanalyse linguistique

La faille du chiffrement de substitution monoalphabétique ne réside pas dans la taille de la clé, mais dans le fait qu'il **préserve intégralement la structure statistique** de la langue du texte clair.

Eve peut exploiter cette propriété de la manière suivante :

4.3.2.1 Analyse des fréquences simples

Elle calcule la fréquence d'apparition de chaque lettre dans le cryptogramme c. Elle compare ensuite cette distribution à la distribution standard des lettres en français (où E ≈ 12%, A ≈ 9%, I ≈ 8%, S ≈ 7%, etc.). Cette comparaison lui permet d'établir des hypothèses de correspondance (ex., la lettre la plus fréquente dans c a de fortes chances d'être le E chiffré).

4.3.2.2 Analyse des motifs et des redoublements

Elle recherche des motifs récurrents :

- **Digrammes et trigrammes fréquents :** La présence de paires ou triplets de lettres qui se répètent peut correspondre à des combinaisons courantes en français telles que DE, LE, RE, ES, ENT, ION, QUE, AI, OU.
- **Lettres doubles :** Les redoublements de lettres (ex., SS, LL, EE, TT) sont un indice fort pour deviner des lettres courantes.
- **Mots courts :** Les mots d'une ou deux lettres sont souvent A, À, LE, DE, UN, EN, ET, OU, IL, ELLE.

4.3.2.3 Reconstitution par affinage itératif

Eve utilise ces hypothèses pour déchiffrer des fragments de mots. En s'appuyant sur sa connaissance de la langue française et sur la vraisemblance des mots formés, elle teste, valide ou infirme ses hypothèses. Ce processus itératif lui permet de corriger progressivement le mapping de substitution jusqu'à obtenir un message clair cohérent. La présence de mots probables dans le message original (comme "message" ou "clair") fournit des ancres solides pour accélérer ce processus.

Conclusion : Le chiffre de substitution monoalphabétique n'est pas sûr contre un adversaire capable de mener une analyse cryptographique réaliste. Sa faiblesse fondamentale est de **conserver les statistiques du langage clair**, offrant ainsi une prise facile pour une cryptanalyse manuelle ou assistée par ordinateur. Sa sécurité est donc illusoire, malgré la taille impressionnante de son espace de clés.

5 Bonus 1 : Attaque par analyse de fréquences

5.1 Calcul des fréquences du cryptogramme

Formule de la Fréquence Relative d'une Lettre

Pour calculer à quel pourcentage une lettre L apparaît dans un texte c :

Fréquence Relative (L) = (Nombre d'occurrences de L / Longueur totale du texte) × 100

Extrait de code Python : pour calculer les fréquences

```
from collections import Counter

import pandas as pd

# Texte fourni

texte =
"KQCAWYQVKFVNCQCUKTQVCKQOWBNKLOLZQJTBWYVAZCQTBOQBFZYABKQF"


# Compter les occurrences de chaque lettre

compteur = Counter(texte)


# Calcul de la fréquence en pourcentage

total = len(texte)

freq = {lettre: (count / total) * 100 for lettre, count in compteur.items()}


# Organiser dans un DataFrame pour meilleure présentation
```

```
df = pd.DataFrame({
   "Lettre": list(compteur.keys()),
   "Occurrences": list(compteur.values()),
   "Fréquence (%)": [round(f, 2) for f in freq.values()]
}).sort_values(by=["Occurrences", "Lettre"]).reset_index(drop=True)

df
```

Lettre	Occurrences	Fréquence (%)
J	1	1.79
U	1	1.79
L	2	3.57
N	2	3.57
A	3	5.36
F	3	5.36
O	3	5.36
T	3	5.36
W	3	5.36
Y	3	5.36
Z	3	5.36
V	4	7.14
B	5	8.93
C	5	8.93
K	6	10.71
Q	9	16.07

5.2 Hypothèse initiale de correspondance

- On établit une première clé en alignant l'ordre des fréquences du texte chiffré avec celui de la langue française (E, A, I, S, N, R, T, O, L, U, ...).
- E ≈ 12 %, A ≈ 9 %, I ≈ 8 %, S ≈ 7 %, N ≈ 7 %, R ≈ 6 %, T ≈ 6 %, O ≈ 5 %, L ≈ 5 %, U ≈ 4 %.

Hypothèses initiales :

- Q (16.07%) → **E**
- K (10.71%) → **A**
- C (8.93%) → **I**
- B (8.93%) → **S**
- V (7.14%) → **N**
- Lettres à 5 % (A, W, Y, F, T, O, Z) → candidats pour **R, T, O, L, U, M**

Cette clé n'est qu'un **point de départ**, basé uniquement sur les fréquences.

5.3 Déchiffrement approximatif

En appliquant la clé initiale, on obtient un premier brouillon du texte clair.

AEIRTOENALNPIEIGAUENIAEDTSPACDCMEHUSTONRMIEUSDESLMORSAEL

Commentaires :

- Le résultat n'est pas encore lisible, mais on reconnaît des fragments ressemblant à des mots français (DES, DE, EN).
- Ce résultat intermédiaire montre que la méthode statistique seule est insuffisante pour un court texte, mais fournit une base exploitable.

5.4 Ajustements

Pour affiner la clé, on utilise des critères linguistiques :

- **Analyse des digrammes/trigrammes fréquents** : en français, ES, DE, LE, RE, OU, QU apparaissent souvent. On vérifie si certaines séquences du chiffré peuvent correspondre.
- **Recherche de mots courts fréquents** : LE, LA, LES, EST, UN, UNE, POUR, PAR. Quand un motif correspond, on ajuste la substitution.

- **Échanges locaux** : on intervertit deux lettres de la clé si cela permet de faire apparaître un mot complet. Chaque échange est validé par l'apparition d'un mot français correct.
- **Validation progressive** : chaque nouveau mot trouvé est propagé dans le reste du texte pour stabiliser la clé.

En affinant certaines correspondances (par exemple K→L, B→A, F→E), on obtient :

LEIRTOENLENPIEIGLUENILEDTAPLCDCMEHUATONRMIEUADEAEMORALEE

Observation :

- Le mot **MORALE** apparaît clairement à la fin du message.
- D'autres fragments (« LE », « UNE », « DES ») commencent à se dessiner.

Conclusion

La démarche suivie est la suivante :

1. **Analyse fréquentielle** pour proposer une clé initiale.
2. **Application de la clé** afin d'obtenir un brouillon.
3. **Ajustements successifs** en s'appuyant sur les digrammes, trigrammes et mots fréquents du français.
4. **Itérations** jusqu'à obtenir un texte clair et cohérent.

Dans ce cas, l'attaque permet de retrouver progressivement des mots français cohérents (DES, LE, MORALE), illustrant la vulnérabilité intrinsèque du chiffre de substitution mono-alphabétique.

Le point essentiel de cette méthode est la **progression rigoureuse** : partir de statistiques globales, puis raffiner la clé à l'aide de contraintes linguistiques

6 Bonus 2 : Analyse comparative de sécurité

6.1 Tailles des espaces de clés et faisabilité de l'attaque par force brute

Schéma de chiffrement	Taille de l'espace des clés	Faisabilité d'une attaque par force brute
César double (ΠCésar1)	26 * 26 = 676	Trivialement faisable. Test exhaustif instantané sur un ordinateur moderne.
Substitution (ΠSUB)	$26! \approx 4.03 \times 10^{26}$	Impossible en pratique. À un rythme de 10^{12} clés/s, le test exhaustif prendrait ~12,8 millions d'années en moyenne.
OTP (ΠOTP)	$2^{160} \approx 1.46 \times 10^{48}$	Impossible et inutile. Le temps nécessaire est astronomique (dépassant largement l'âge de l'univers). La sécurité est informationnelle, pas computationnelle.

Conclusion : Si la force brute est efficace contre le César double, elle est totalement irréaliste pour la Substitution et l'OTP. Cependant, la Substitution reste vulnérable à d'autres types d'attaques.

6.2 Vulnérabilités principales (au-delà de la force brute)

- **César double :**
  - **Analyse de fréquences séparée :** L'attaque consiste à séparer le cryptogramme en deux flux (positions paires et impaires) et à effectuer une analyse de fréquences standard sur chacun, réduisant le problème à deux chiffrements de César simples.
  - **Essai exhaustif :** Le nombre de clés (676) est si faible qu'un essai systématique est la méthode la plus directe.
- **Substitution monoalphabétique :**
  - **Cryptanalyse linguistique :** C'est la vulnérabilité fondamentale. L'attaque exploite la conservation des statistiques de la langue :
    - **Fréquences de lettres** (e.g., E, A, I, S sont les plus fréquentes en français).
    - **Digrammes et trigrammes courants** (e.g., DE, LE, ENT, ION).
    - **Motifs et mots probables** (e.g., mots courts comme UN, LE, DE ; lettres doubles comme SS, LL).
  - Des algorithmes d'optimisation (comme le « recuit simulé » ou la « descente de gradient ») peuvent automatiser cette attaque en maximisant la vraisemblance linguistique du texte déchiffré.
- **OTP (Masque Jetable) :**
  - **Sécurité parfaite :** Le système est prouvé inviolable si la clé est **aléatoire, secrète, aussi longue que le message et jamais réutilisée** (Théorème de Shannon).
  - **Unique faille pratique : la réutilisation de la clé.** Si une même clé k est utilisée pour deux messages m1 et m2, un attaquant peut calculer $c1 \oplus c2 = m1 \oplus m2$. Cette valeur révèle des informations sur la corrélation entre les deux messages et permet une cryptanalyse efficace.

6.3 Aspects pratiques (déploiement et usage)

Aspect	César double	Substitution	OTP
Distribution des clés	Triviale	Simple	**Extrêmement coûteuse.** Nécessite un canal sécurisé pour échanger des clés aussi longues que les messages.
Performance	Très rapide	Très rapide	Très rapide (opération XOR)
Robustesse aux erreurs	Locale	Locale	Locale (une erreur de bit n'affecte qu'un bit du message clair)
Réutilisation de la clé	Courante	Courante	**Strictement interdite.** Détruit immédiatement la sécurité.

6.4 Classement synthétique et verdict final

Schéma	Sécurité	Verdict
César double	Espace de clés minuscule. Vulnérable à l'analyse de fréquences et à la force brute.	**Très faible.** Inutilisable pour toute confidentialité sérieuse.
Substitution mono.	Espace de clés gigantesque mais vulnérable à la cryptanalyse linguistique.	**Faible.** Cassable en pratique malgré la taille de la clé, à cause de la préservation des statistiques.
OTP	Sécurité **parfaite** sous conditions strictes (clé aléatoire, unique, longueur ≥ message, secrète).	**Excellente / Parfaite.** Le gold standard théorique, mais impraticable pour un usage général à cause des contraintes de distribution de clés.

7 Limites et améliorations possibles

Ces limites illustrent bien l'écart entre une implémentation pédagogique et une utilisation réelle en sécurité informatique.

Nos implémentations, bien que fonctionnelles et adaptées à un cadre pédagogique, présentent plusieurs limites qu'il est important de souligner :

1. **Génération de clés :** Les clés sont générées avec le module random de Python, qui est conçu pour la randomisation statistique et non pour la cryptographie. Il ne garantit pas un caractère parfaitement imprévisible et uniforme requis pour une sécurité robuste.

- **Amélioration :** Utiliser un générateur de nombres aléatoires cryptographiquement sûr (CSPRNG) comme le module secrets de Python ou os.urandom().

2. **Alphabet restreint :** Le chiffrement ne traite que les 26 lettres minuscules de l'alphabet anglais (a-z). Les majuscules, la ponctuation, les chiffres, les espaces et les caractères accentués seraient soit ignorés, soit causeraient des erreurs.
    - **Amélioration :** Étendre l'implémentation pour gérer un jeu de caractères plus complet (e.g., UTF-8) ou définir un mode de fonctionnement pour ignorer/conserver les caractères non alphabétiques.
3. **Gestion des clés OTP :** L'implémentation actuelle ne comporte aucun mécanisme pour empêcher la réutilisation d'une clé OTP, ce qui anéantirait instantanément sa sécurité parfaite.
    - **Amélioration :** Intégrer un système de gestion de "carnet de clés" à usage unique, similaire à ceux utilisés en pratique dans les contextes militaires ou diplomatiques.
4. **Robustesse et performances :** Le code est conçu pour la clarté, pas pour la performance ou la résistance à des attaques par canaux auxiliaires (timing attacks). Il n'est pas optimisé pour de très gros volumes de données.
    - **Amélioration :** Pour une utilisation réelle, il faudrait s'appuyer sur des bibliothèques cryptographiques standardisées et auditées comme cryptography, PyNaCl, ou PyCryptodome.
5. **Sécurité théorique vs pratique :** Ces implémentations illustrent des concepts historiques mais **n'offrent aucune garantie de sécurité pour un usage réel**. Elles servent de tremplin pour comprendre les principes qui sous-tendent les algorithmes modernes.

8 Conclusion générale

Ce travail a permis une mise en pratique concrète des concepts fondamentaux de la cryptographie à travers l'étude de trois schémas historiques :

1. Le **chiffre de César à double décalage** a illustré la faiblesse des espaces de clés de petite taille, étant cassable de manière triviale par force brute ou analyse de fréquences simple.
2. Le **chiffre de substitution monoalphabétique** a démontré qu'un espace de clés gigantesque ($26! \approx 4.03 \times 10^{26}$) n'est pas une garantie de sécurité s'il préserve les statistiques de la langue sous-jacente, le rendant vulnérable à la cryptanalyse linguistique.
3. Le **Masque Jetable (OTP)** a incarné l'idéal de la **sécurité parfaite** prouvée par Shannon, tout en en révélant le coût pratique prohibitif : la nécessité de partager des clés secrètes aussi longues que les messages et de ne jamais les réutiliser.

Les questions bonus ont renforcé ces conclusions : l'attaque par analyse de fréquences sur un cryptogramme de substitution a montré l'efficacité de l'exploitation des régularités linguistiques. La comparaison finale a souligné une leçon cruciale : **la sécurité d'un système cryptographique ne se mesure pas seulement à la taille de son espace de clés, mais aussi, et surtout, à la robustesse de son algorithme contre des attaques structurelles.**

En définitive, si ces chiffrements historiques sont des fondements pédagogiques essentiels, la cryptographie moderne repose sur des primitives plus sophistiquées (telles qu'AES, RSA, ou les courbes elliptiques) qui cherchent à trouver un équilibre optimal entre une sécurité computationnelle très élevée et une praticabilité déployable à large échelle. Ces enseignements montrent l’importance de passer de la cryptographie historique à des standards modernes robustes comme AES ou RSA pour toute application réelle