Victor Yuyang Zhang (20247653)
Xinning Xu (20213497)

Question 1.1 :

N =
14351633690928181552910415014721024800278971276108690005970534210322078267404628923208243578956328373980574557987343284668088987010788191642824141952083164817391248643164035000086097393530005608928615873757935780597701932955798545493414628255058294246363124569770299851118078700702913956193343355024269304792444038855098349869008076488293410183490802531486146872625342555433476014692353040392452337247768666875256728706020140746463094321823613242377263667518297758570759601601155691750475913144416024025273328296953409286968533893124120478575051974850543903980111976204979608571910659156221711567923658 3

e = 3

c =1101510739796100601351050380607502904616643795400781908795311659278941419415375

Puisque e = 3, on peut alors utiliser l’attaque à faible exposant en prenant la racine e-ième du texte chiffré sur c.

Alors, le programme essaie d’obtenir la e-ième racine, qui retournera l’entier résultat et une valeur booléenne indiquant si on peut extraire une racine n-ième entière de c.

```
mNumber = sympy.integer_nthroot(c, e)
```

Si la racine de c est en effet un entier, il existe une forte chance que la racine de c est alors M le message en forme de chiffre. La racine de c, ici, est représentée comme mNumber[0]. La dernière condition à vérifier est si mNumber[0] est en effet le bon m, qui peut être vérifier avec la formule $c = M^e \, mod \, N$.

```
if c == modular_pow(int(mNumber[0]), int(e), int(N)):
  M = int(mNumber[0]).to_bytes((int(mNumber[0]).bit_length() + 7) // 8, 'big').decode()
  print(M)
else:
  raise Exception("Pas le bon message!")
```

Si la condition est vérifiée, on peut alors décoder le message, qui est **Umberto Eco** pour la question 1.1.

Question 1.2 :

L'attaque à faible exposant ne fonctionne pas pour la question 1.2, puisque l'exposant e de la question 1.2 est 173.

Selon les formules d'encodage et de décodage à clé publique:

$$c = m^e \bmod N$$

$$D(m) = m^d \bmod N.$$

$$D(c) = c^d \bmod N = m$$

$$\Uparrow$$

$$d = (e \bmod (p-1)*(q-1))^{-1}$$

$N = p \cdot q$, p, q sont des nombres premiers.

Alors, on n'a qu'à trouver p et q qui sont des facteurs premiers de N. N est très grand dans la question 1.2. On pourrait utiliser les outils comme Yafu. Ici, nous avons utilisé la calculatrice de factorisations précalculées pour obtenir p et q, pour obtenir d, qui est l'inverse de la division modulaire $e \ mod \ (p-1) * (q-1)$.

Calculatrice en ligne: https://www.alpertron.com.ar/ECM.HTM

```
p = 107150860718626732094842504906000181056140481170553360744375038837
q = 160726291077940098142263757359000271584210721755830041116562558255
phiN = (p-1)*(q-1)
d = modinv(e, phiN)
m = modular_pow(c, d, N)
M = int(m).to_bytes((int(m).bit_length() + 7) // 8, 'big').decode()
print(M)
```

Victor Yuyang Zhang (20247653)
Xinning Xu (20213497)

Par la suite, il faut juste utiliser c, d et N trouvés pour terminer la formule de décodage et ainsi trouver m le message, qui est **Marcel Proust**.

Question 2 :

La question 2 nous demande de décoder un message qui a été chiffré par une variante d'une substitution alphabétique tel que :

- La clé utilisée pour coder un message contient 256 symboles
- Chaque symbole est soit un caractère, soit un bicaractère (une paire de caractères)
- Chaque symbole est associé à une unique séquence de 8 bits

Étant donné un cryptogramme « C », on doit écrire une fonction « decrypt(C) » qui retourne le message clair associé à « C ».

Première tentative :

Notre première idée pour décoder le cryptogramme était d'utiliser une approche statistique de base, c'est-à-dire, analyser l'occurrence des différents symboles dans la langue française et dans le cryptogramme, puis trouver une association entre les deux. Nous avons vu en classe que l'approche statistique est une technique utilisée pour briser la substitution mono-alphabétique, donc elle pourrait nous aider pour le contexte de cette question.

Nous commençons par évaluer des textes français en comptant les caractères et les bicaractères. On fait la même chose pour le cryptogramme : on compile le nombre d'occurrence de chaque groupe de 8 bits. Nous trillons ces deux compteurs tel que le premier élément est le symbole qui apparait le plus souvent. Ensuite, nous créons une clé (dictionnaire) qui relie deux symboles ayant une fréquence relative similaire.

Cependant, cette approche n'est pas suffisante. En effet, nous obtenions une similarité d'environ 0.22% lorsque nous comparons le texte clair avec le message déchiffré obtenue. Cela peut être expliqué par le fait qu'une association effectué selon les fréquences n'est pas assez précise. Par exemple, le 10^{e} symbole crypté peut être associé avec le 14^{e} symbole plutôt qu'avec le 10^{e} symbole du texte clair. En effet, plusieurs symboles ont des fréquences très similaire : les symboles $'r\ '$, $'tr'$, $'et'$, $'ie'$, $'se'$, $'p'$ et $'it'$ ont tous une fréquence d'environ 3500 (la plus petite fréquence est 3485 et la plus grande est 3584).

Deuxième tentative :

Sachant qu'il y avait des erreurs de mapping à cause des fréquences similaires, nous pensions pouvoir trouver un bon mapping en utilisant une approche « force brute » guidé par la clé statistique de la première tentative. Voici la logique derrière cette approche :

1. Parcourir chaque symbole crypté du cryptogramme
2. Déterminer le symbole associé en utilisant la clé statistique. Nous supposons qu'il y a de grandes chances que le symbole réel se retrouve dans un intervalle de -5 à +5 de la position du symbole mappé par fréquence.
3. En commençant par la lettre à $index -= 5$, on construit un string avec le symbole.
4. On continue en analysant le prochain symbole crypté, jusqu'à ce qu'on obtienne un « whitespace »
5. Lorsqu'on a un « whitespace », on vérifie si le mot formé appartient au dictionnaire français. Si non, alors on essaie avec le prochain index
6. On continue cette démarche pour l'entièreté du message crypté

Durant les étapes 3, 4 et 5, nous modifions une clé de sorte qu'elle garantit que le message décrypté contiendra des mots de la langue française (nous avons remarqué que certains mots du texte clair ne sont pas français, donc on accepte quelques erreurs).

Cette approche est plus rapide que tester chaque clé possible ($256!$ configurations), mais ça prend toujours trop de temps.

Troisième tentative :

Cette fois-ci, nous avons décidé d'utiliser un système de score basé sur les bigrammes et les trigrammes (groupe de 2 et 3 symboles) courants de la langue française. Durant le comptage d'occurrences des symboles (expliqué dans la première tentative), nous comptons également l'occurrence des bigrammes et des trigrammes. Ces résultats sont ensuite normalisés pour représenter leur proportion par rapport au texte (c'est pertinent de les normaliser pour ne pas avoir de chiffres astronomiquement grands lors du calcul du score). Comme dans les deux tentatives précédentes, nous calculons naïvement une clé statistique et nous cherchons à l'améliorer en effectuant des changements qui augmente la présence des n-grammes courant dans le texte déchiffré.

Cette méthode est basée sur cet article : https://people.csail.mit.edu/hasinoff/pubs/hasinoff-quipster-2003.pdf

Explication de la fonction « compute_score » (ligne 165) : À partir d'un message décrypté, on veut calculer son score en analysant la présence de bigrammes et trigrammes dans le texte. Les n-grammes les plus fréquents ont un plus grand score puisqu'ils ont une plus grande probabilité d'apparaitre dans un texte français. De cette manière, chaque clé peut être associé à un score : il faut simplement trouver la clé ayant le plus grand score.

```python
# Scoring system using birams and trigrams
def compute_score(text, text_symbols_set, bigram_frequencies, trigram_frequencies):
  score = 0

  text_length = len(text)
  i = 0
  second_previous_symbol = None
  previous_symbol = None

  # Go through the entire text
  while i < text_length:

    # Verify a pair of caracters
    if i + 1 < text_length:
      pair = text[i] + text[i + 1]
      if pair in text_symbols_set:

        # Add to score if bigram or trigram is valid
        if previous_symbol is not None:
          bigram = (previous_symbol, pair)
          if bigram in bigram_frequencies:
            score += bigram_frequencies[bigram]

        if second_previous_symbol is not None:
          trigram = (second_previous_symbol, previous_symbol, pair)
          if trigram in trigram_frequencies:
            score += trigram_frequencies[trigram]

        # Update second_previous_symbol and previous_symbol
        second_previous_symbol = previous_symbol
        previous_symbol = pair

        i += 2              # Evaluate the next pair
        continue

    # Verify a single caracter
    single = text[i]
    if single in text_symbols_set:

      # Add to score if bigram or trigram is valid
      if previous_symbol is not None:
        bigram = (previous_symbol, single)
        if bigram in bigram_frequencies:
          score += bigram_frequencies[bigram]
```

```
      if second_previous_symbol is not None:
        trigram = (second_previous_symbol, previous_symbol, single)
        if trigram in trigram_frequencies:
          score += trigram_frequencies[trigram]

      # Update second_previous_symbol and previous_symbol
      second_previous_symbol = previous_symbol
      previous_symbol = single

    i += 1

  return score
```

Explication de la fonction « optmize_key » (ligne 233 dans student_code.py) : Cette fonction a pour but de trouver une clé qui a le meilleur score. Nous initialisons un score de base avec la clé statistique (clé obtenue dans la première tentative). Ensuite, pour un nombre x d'itérations (nous avons utilisé $x = 50000$), nous interchangeons deux symboles de la clé, puis recalcule le score obtenu par cette clé. On conserve uniquement la clé qui a eu le meilleur score.

```
# Function that optimizes the key, using the bigrams and trigrams to compute a score
def optimize_key(split_cryptogram, initial_key, bigram_frequencies, trigram_frequencies,
ranked_symbols_list, text_symbols_set, iterations=1000):

  current_key = initial_key.copy()
  best_key = current_key.copy()

  sample_size = min(1500, len(split_cryptogram))

  decrypted_text = []
  for crypted_symbol in split_cryptogram[:sample_size]:
    decrypted_text.append(best_key[crypted_symbol])
  decrypted_text = ''.join(decrypted_text)

  best_score = compute_score(decrypted_text, text_symbols_set, bigram_frequencies, trigram_frequencies)

  for _ in range(iterations):

    # slightly randomize the key
    randomized_key = swap_symbols(ranked_symbols_list, current_key.copy(), 15)

    decrypted_text = []
    for crypted_symbol in split_cryptogram[:sample_size]:
      decrypted_text.append(randomized_key[crypted_symbol])
    decrypted_text = ''.join(decrypted_text)

    # Compute score of the randomized key
    score = compute_score(decrypted_text, text_symbols_set, bigram_frequencies, trigram_frequencies)

    # Keep track of the best key / score
    if score > best_score:
      best_score = score
      best_key = randomized_key.copy()
      current_key = randomized_key.copy()

  return best_key, best_score
```