Aufgaben

11. Zeigen sie mit Hilfe der auf Slide 52 hergeleiteten Formel, dass der Shift Cipher, wenn alle 26 Schlüssel z gleich wahrscheinlich sind, perfekt sicher ist - für einbuchstabige Plaintexte.

Shift Cipher ist perfekt sicher wenn $P(X = x) = P(X = x|Y = y)$ gilt.

bei einem einbuchstabige Plaintext X gibt es 26 möglichkeiten, selbiges gilt für die Ciphertexte und Schlüssel. Somit wissen wir:

- $P(X = x) = \frac{1}{26}$
- $P(Y = y) = \frac{1}{26}$
- $P(K = k) = \frac{1}{26}$
- $P(Y = y|X = x) = \frac{1}{26}$ da es unter annahme eines Buchstaben x trotzdem noch 26 mögliche Schlüssel gibt.

$P(X = x|Y = y) = \frac{P(X=x)*P(Y=y|X=y)}{P(Y=y)}$
$P(X = x|Y = y) = \frac{\frac{1}{26}*\frac{1}{26}}{\frac{1}{26}}$
kürzen
$P(X = x|Y = y) = \frac{1}{26} = P(X = x)$

---

13. Implementieren sie die in der VO besprochene short Key XOR Verschlüsselung (Text wird über ascii-Nummern binär dargestellt und mit ensprechendem “binärem” Text Key XOR verschlüsselt, variable Key-länge für Experimente erforderlich). Bestimmen sie mit der in der VO besprochenen “Counting Coincidences” Methode die Länge des jeweils verwendeten Keys.

Zuerst strecken wir den Schlüssel auf die Länge des Plaintexts:

```
def pad_key(plaintext, key):
    return (key * (len(plaintext) // len(key) + 1))[:len(plaintext)]
```

Dann wird das XOR auf jedes Bit des Plaintext/Schlüssels ausgeführt (a/b), so wird verschlüsselt.

```
def xor_encrypt(plaintext, key):
    padded_key = pad_key(plaintext, key)
    ciphertext = bytes([a ^ b for (a, b) in zip(plaintext.encode(), padded_key.encode())])
    return ciphertext
```

Zum entschlüsseln führen wir das ganze einfach rückwärts aus, XOR des Schlüssels auf den Ciphertext:

```
def xor_decrypt(ciphertext, key):
    padded_key = pad_key(ciphertext.decode(), key)
    decrypted_text = bytes([a ^ b for (a, b) in zip(ciphertext, padded_key.encode())])
    return decrypted_text.decode()
```

Für den zweiten Teil müssen wir zuerst den Ciphertext bit für bit mit einer verschobenen version von sich selbst XORen und zählen, wie oft die Operation 0 ergibt, also wie viele stellen gleich sind:

```
def count_coincidences(ciphertext, shift):
    count = 0
    for i in range(len(ciphertext) - shift):
        if (ciphertext[i] ^ ciphertext[i + shift]) == 0:
            count += 1
    return count
```

Um dann die key länge zu finden probieren wir sämtliche shift Längen ≤ der Länge des Ciphertext durch und berechnen jeweils die Anzahl der gleichen stellen mit der vorigen funktion. Wenn der Prozentsatz der gleichbleibenden Stellen den Threshold (6.65%) übersteigt können wir davon ausgehen ein vielfaches der Schlüssellänge gefunden zu haben.

```
def determine_key_length(ciphertext, threshold=0.0665):
    n = len(ciphertext)
    percentages = []
    for shift in range(1, len(ciphertext)):
        coincidence_count = count_coincidences(ciphertext, shift)
        percentage = coincidence_count / (n - shift)
        percentages.append((shift, percentage))

        if percentage > threshold:
            return shift, percentage, percentages
    return None, None, percentages
```

Das ganze funktioniert an sich auch gut:

```
plaintext = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam nec odio et
odio fermentum fermentum."
key = "key"
ciphertext = xor_encrypt(plaintext, key)

key_length, coincidence_rate, all_percentages = determine_key_length(ciphertext)

if key_length:
    print(key_length)

>> 3
```

Das ganze funktioniert umso besser, desto länger der Ciphertext und desto kürzer der Schlüssel.

---

14. Fortsetzung Aufgabe 13.): Bestimmen sie mit der in der VO besprochenen Methode mit Iteration über die Länge der Keys unter Berechnung der Hamming Distanz (Slide 65, 2. Verfahren) die Länge des Keys in dem in Aufgabe 13.) realisierten short Key XOR Verschlüsselungsverfahren.

Zuerst brauchen wir eine Funktion welche die Hamming-Distanz zwischen zwei Byte Arrays berechnet:

```
def hamming_distance(bytes1, bytes2):
    return sum(bin(b1 ^ b2).count('1') for b1, b2 in zip(bytes1, bytes2))
```

Dann eine, welche jeweils für eine bestimmte Schlüssel länge den Ciphertext in dementrsprechend lange Blöcke aufteilt und dann die hamming distanz nebeneinanderliegender blöcke vergleicht. Die Hammingdistanz wird auf die Schlüssel Länge normiert und der durchschnitt dieser Werte ausgegeben:

```
def normalized_hamming_distance(ciphertext, key_size):
    blocks = [ciphertext[i:i + key_size] for i in range(0, len(ciphertext), key_size)]

    distances = []

    for i in range(0, len(blocks) - 1):
        if len(blocks[i]) == key_size and len(blocks[i+1]) == key_size:
            distance = hamming_distance(blocks[i], blocks[i+1])
            distances.append(distance / key_size)
    return sum(distances) / len(distances) if distances else float('inf')
```

Nun muss nur noch für jede möglich Schlüssel Länge von 1 bis der Länge des Ciphertexts durchprobiert werden und die Länge gewählt werden, bei der nie normierte Hammingdistanz durchschnittlich am geringsten war:

```
def find_key_length(ciphertext):
    key_sizes = range(1, len(ciphertext))
    normalized_distances = [(key_size, normalized_hamming_distance(ciphertext, key_size))
for key_size in key_sizes]
    return min(normalized_distances, key=lambda x: x[1])
```

Dies funktioniert oft gut und gibt die tatsächliche Schlüssellänge aus, in vielen Fällen wird aber auch ein Vielfaches der tatsächlichen Schlüssellänge errechnet. Z.B.:

```
plaintext = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam nec odio et
odio fermentum fermentum."

key = "thisisakey"
ciphertext = xor_encrypt(plaintext, key)

key_length= find_key_length(ciphertext)

if key_length:
    print(key_length)

>>> (10, 2.433333333333333)
```

Repository

- [Repo Link]