ASSIGNMENT REPORT: CLASSICAL ENCRYPTION

Instructor	**PhD. Nguyễn Ngọc Tự**	
Student	Mai Nguyễn Phúc Minh	23520930

1. Affine Cipher

- **Task requirement:** Provide python code to illustrate the solution, using the brute-force method to decrypt a given text (knowing a is a coprime number with 26 and b in range of 0 to 25).
- **Encryption technique:** This encryption uses the following function to encrypt text:

$$E(x) = (ax + b) \bmod m$$

With:

modulus m: size of the alphabet

a and b: key of the cipher (a must be chosen such that a and m are coprime)

- **Building function to decrypt:** based on the theory, we can easily find the way to decrypt the text by using mathematics:

$$D(y) = a * (y - b) \bmod 26$$

Therefore, we apply this function into python code:

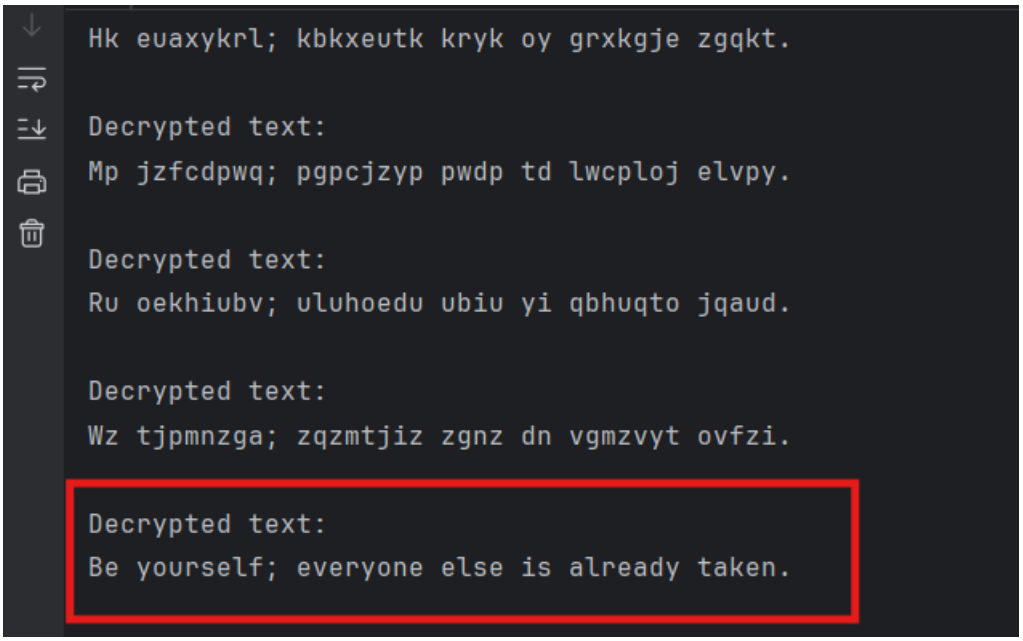
```
def affine_decrypt(text, a, b):
    a_inv = mod_inverse(a, 26)
    if a_inv is None:
        return
    alphabets = string.ascii_uppercase
    result = []
    for char in text:
        if char.isupper():
            y = ord(char) - ord('A')
            x = (a_inv * (y - b)) % 26
            result.append(alphabets[x])
        elif char.islower():
            y = ord(char.upper()) - ord('A')
            x = (a_inv * (y - b)) % 26
            result.append(alphabets[x].lower())
        else:
            result.append(char)
    return ''.join(result)
```

- **Approach:** As the range of the number in a and b is not that big (only about 24*25 = 600 loops), we can simply make 2 loops (one for a and one for b) until we can find that correct key which are used to encrypt:

```
def main():
    encrypted_text = input("\nEnter the encrypted text to decrypt: ")
    for a in range (2,26):
        if mod_inverse(a,26) != None:
            for b in range (25):
                decrypted_text = affine_decrypt(encrypted_text, a, b)
                print("\nDecrypted text:")
                print(decrypted_text)
                #The code under here to see each loop
                # input("\nPress Enter to continue to decryption...")
```

- **mod_inverse:** function to check if the value a is coprime with 26 or not (if valid then it will try b from 1 to 25).

➔ **Testing:** Given encrypted text: *"Pe acgrwenj; eleracxe enwe yw knrekza bkiex."*. Thus, the expectation for this code is to decrypt to the original text ***"Be yourself; everyone else is already taken."***



2. Simple Substitution Cipher

- **Task requirement:** Provide python code to illustrate the solution:
  - Generate random key and encrypt the plain text (about 100 – 200 words)
  - Decrypt without key. Present the details on how to guest the key step-by-step.
- **Encryption technique:** Any character of plain text from the given fixed set of characters is substituted by some other character from the same set depending on a key.
- **Modelling an encrypted function with random key:**

```
def generate_random_mapping():
    """
    Generates a random substitution mapping for uppercase letters.
    Each letter A-Z is mapped to a unique letter (random permutation).
    """
    plain = list(string.ascii_uppercase)
    cipher = plain.copy()
    random.shuffle(cipher)
    mapping = {plain[i]: cipher[i] for i in range(26)}
    return mapping
```

```
def simple_substitution_encrypt(text, mapping):
    """
    Encrypts the input text using the provided substitution mapping.
    Handles both uppercase and lowercase letters while preserving non-alphabet
characters.
    """
    result = []
    for char in text:
        if char.isupper():
            result.append(mapping.get(char, char))
        elif char.islower():
            # Map using uppercase then convert back to lowercase.
            result.append(mapping.get(char.upper(), char.upper()).lower())
        else:
            result.append(char)
    return "".join(result)
```

- **Encryption process:** Running the code and I got an encrypted text with random key: *"Amgewlu gpu cmhuzft pabu, W pufzl f raj efj gpfg epu pfl cfttuh acc gpu rfov ac puz rajczwuhl'e bagazojotu. Rzavuh puz huov. Epu huxuz vhus spfg pwg puz, pu efwl. W sfe 13. Gpu lufl iwzt pfl ruuh f qmhwaz wh pwip eopaat.Gpu twhu ga euu puz ehfvul fzamhl gpu rmwtlwhi. Raje swgp tahi pfwz, sufzwhi gwue gpuj'l razzasul czab gpuwz cfgpuze, fhl iwzte swgp gpwov rtmu ujuepflas ebavul owifzuggue wh gpu nfzvwhi tag. Eabuahu nfeeul f raggtu ac Qfov. Gpuzu suzu ha flmtge gpuzu, qmeg xuzj atl vwle.Epu ftbaeg taavul twvu epu sfe etuunwhi, ukoung gpfg epu sfe gaa egwtt. Gpuzu sfe f nmccwhuee ga puz cfou gpfg lwlh'g euub ymwgu zwipg. Gpuj pfl lzueeul puz caz gpu nzab; gpu ozwhatwhu etuuxue ac puz iash twvu naace ac nwhv oaggah ofhlj. Eabu vwle nzfjul, rmg W oamtlh'g. W qmeg egfzul fg gpu zaeue wh puz oazefiu."*
- **Decryption technique:** To decode the text, we will mainly rely on the common appearance of letters in English and then mapping it to out encrypted text.

```
# Known English letter frequencies (from most frequent to least frequent)
english_frequencies = 'ETAOINSHRDLCUMWFGYPBVKJXQZ'

# Counting letter frequencies in the ciphertext
cipher_counts = Counter(''.join(filter(str.isalpha, cipher_text)))
```

```
# Sorting the dictionary by frequency
sorted_cipher = ''.join([item[0] for item in cipher_counts.most_common()])

# Creating a mapping based on letter frequency
mapping = {}
for i, letter in enumerate(sorted_cipher):
    mapping[letter] = english_frequencies[i]
```

- **Result:** we got the first attempt decoded text:
  - We have done mapping all the letters based on the propose theory but it still illegible. Therefore, deeply analyst through the content, we find that *"FAI"* simply like ***"WAS"*** or *"AE"* like ***"HE"***
  - Keeping do that until we ultimately can fully read the content: *"OUTSIDE THE FUNERAL HOME, I HEARD A BOY SAY THAT SHE HAD FALLEN OFF THE BACK OF HER BOYFRIEND'S MOTORCYCLE. BROKEN HER NECK. SHE NEVER KNEW WHAT HIT HER, HE SAID. I WAS 13. THE DEAD GIRL HAD BEEN A JUNIOR IN HIGH SCHOOL. THE LINE TO SEE HER SNAKED AROUND THE BUILDING. BOYS WITH LONG HAIR, WEARING TIES THEY'D BORROWED FROM THEIR FATHERS, AND GIRLS WITH THICK BLUE EYESHADOW SMOKED CIGARETTES IN THE PARKING LOT. SOMEONE PASSED A BOTTLE OF JACK. THERE WERE NO ADULTS THERE, JUST VERY OLD KIDS. SHE ALMOST LOOKED LIKE SHE WAS SLEEPING, EXCEPT THAT SHE WAS TOO STILL. THERE WAS A PUFFINESS TO HER FACE THAT DIDN'T SEEM QUITE RIGHT. THEY HAD DRESSED HER FOR THE PROM; THE CRINOLINE SLEEVES OF HER GOWN LIKE POOFS OF PINK COTTON CANDY. SOME KIDS PRAYED, BUT I COULDN'T. I JUST STARED AT THE ROSES IN HER CORSAGE."*
- **Manually mapping for the content:**

```
# Optional manual adjustments to improve decryption quality
mapping["W"] = "I"
mapping["F"] = "A"
mapping["S"] = "W" ## FAI --> WAS
mapping["E"] = "S" ## FAI --> WAS
mapping["P"] = "H" ## AE --> HE
mapping["P"] = "H" ## SAE --> SHE
mapping["N"] = "P" ## SLEEVIRG --> SLEEPING
mapping["H"] = "N" ## SLEEVIRG --> SLEEPING
```

```
mapping["Z"] = "R" ## THENE --> THERE
mapping["B"] = "M" ## SEEB --> SEEM
mapping["M"] = "U" ## QWITE --> QUITE
mapping["J"] = "Y" ## THEP --> THEY
mapping["X"] = "V" ## NEJER --> NEVER
mapping["V"] = "K" ## MNEW --> KNEW
mapping["S"] = "W" ## KHAT --> WHAT
mapping["R"] = "B" ## YOY --> BOY
mapping["Q"] = "J" ## KUNIOR --> JUNIOR
mapping["C"] = "F" ## CROM --> FROM
mapping["O"] = "C" ## MOTORUYULE --> MOTORCYCLE
```

3. Polyalphabetic using matrix (Hill Cipher)

- **Task requirement:** Provide python code with the following task:
  - Generate the Encrypt metric (2x2, 3x3, 4x4) for Polyalphabetic
  - Compute Decrypt Matrix (Invert mode 26).
- **Encryption technique:** each block of n letters (considered as an n-component vector) is multiplied by an invertible n × n matrix, against modulus 26. Function to illustrate:

```
def encrypt(key, plaintext, alphabet):
    m = key.shape[0]
    m_grams = plaintext.shape[1]
    ciphertext = np.zeros((m, m_grams)).astype(int)
    for i in range(m_grams):
        ciphertext[:,i] = np.reshape(np.dot(key, plaintext[:,i]) % len(alphabet), m)
    return ciphertext
```

  - *The function above shows how the plaintext is encrypted by computing the **dot products** of the **key matrix** and **plaintext matrix***
- **Decryption technique:** encrypted text simply multiplies by the inverse matrix of the key matrix for gaining the original message. Here are some function to illustrate the process:
  - **Inverse function:**

```
def get_inverse(matrix, alphabet):
    alphabet_len = len(alphabet)
    if math.gcd(int(round(np.linalg.det(matrix))), alphabet_len) == 1:
        matrix = Matrix(matrix)
        return np.matrix(matrix.inv_mod(alphabet_len))
    else:
```

```
        return None
```

- **Decrypt function:**

```
def decrypt(k_inverse, c, alphabet):
    return encrypt(k_inverse, c, alphabet)
```

- **k_inverse**: the inverse matrix of the key
- **c**: the encrypted text (already put into a matrix)

- **Testcase**:
  - **2x2 key**:

```
Insert the text to be encrypted: HELP
Insert the key for encryption: adbc

Key Matrix:
 [[0 3]
 [1 2]]
Key Matrix:
 [[0 3]
 [1 2]]
Plaintext Matrix:
 [[ 7 11]
 [ 4 15]]

The message has been encrypted.

Generated Ciphertext:  MPTP
Generated Ciphertext Matrix:
 [[12 19]
 [15 15]]
```

```
The message has been decrypted.

Generated Plaintext:  HELP
Generated Plaintext Matrix:
 [[ 7 11]
 [ 4 15]]
```

- ***3x3 key:***

```
Insert the text to be encrypted: SAVE ME
Insert the key for encryption: CDBBEFDBB

Key Matrix:
 [[2 3 1]
 [1 4 5]
 [3 1 1]]
Key Matrix:
 [[2 3 1]
 [1 4 5]
 [3 1 1]]
Plaintext Matrix:
 [[18  4]
 [ 0 12]
 [21  4]]

The message has been encrypted.

Generated Ciphertext:  FTXWUC
Generated Ciphertext Matrix:
 [[ 5 22]
 [19 20]
 [23  2]]
```

```
The message has been decrypted.

Generated Plaintext:  SAVEME
Generated Plaintext Matrix:
 [[18  4]
 [ 0 12]
 [21  4]]
```

- ***4x4 key:***

```
Insert the text to be encrypted: You need to run
Insert the key for encryption: IGJFGJFKFIEJKGLE

Key Matrix:
 [[ 8  6  9  5]
 [ 6  9  5 10]
 [ 5  8  4  9]
 [10  6 11  4]]
Key Matrix:
 [[ 8  6  9  5]
 [ 6  9  5 10]
 [ 5  8  4  9]
 [10  6 11  4]]
Plaintext Matrix:
 [[24  4 14]
 [14  4 17]
 [20  3 20]
 [13 19 13]]

The message has been encrypted.

Generated Ciphertext:  BGNYWFBRRZNU
Generated Ciphertext Matrix:
 [[ 1 22 17]
 [ 6  5 25]
 [13  1 13]
 [24 17 20]]
```

```
The message has been decrypted.

Generated Plaintext:  YOUNEEDTORUN
Generated Plaintext Matrix:
 [[24  4 14]
 [14  4 17]
 [20  3 20]
 [13 19 13]]
```

4. Conclusion

- These are some typical types of classical cipher systems and can be breakable in a short time.
- A short comparison between three kinds of cipher systems including its pros and cons:

Comparison / Classical cipher	**Pros**	**Cons**
Affine cipher	+ Simplicity + Flexibility	+ Easy to break (brute-force) when attackers can guess a and b, stimulating a brute-force attack.
Simple Substitution	+ Much more complicated to decrypt the text	+ Vulnerable to frequency analyst
Hill Cipher (Polyalphabetic)	+ The encryption method is much more complicated than two others + Applying mathematics concept to encrypt/decrypt the text	+ Vulnerability to *"Known Plaintext Attack"* + Less complex than modern ciphers like AES and RSA