

## Columnar Cipher

### Problem:

The objective is to create a Columnar Cipher, a straightforward encryption method where the plaintext is arranged into a grid of columns, and the letters are then read out according to a specified column order. This task involves developing a program that can both encrypt and decrypt messages using this cipher technique.

### Description:

The Columnar Cipher is a transposition cipher where the plaintext is organized into columns based on a specified key. Each column is then read in a particular order determined by the key, creating the ciphertext. This implementation preserves non-alphabetic characters and requires dynamically adjusting the column arrangement based on the length of the message. The program repeatedly prompts users to either encrypt or decrypt messages until they decide to exit, applying the columnar transposition based on the chosen key.

### Code:

#### 1. Columnar-Cipher-Basic.py

```
import math
import numpy as np

def columnar_matrix(text, key):
    # Determine the number of rows needed in the matrix.

    num_rows = math.ceil(len(text) / len(key))

    # Initialize the matrix with placeholder characters ('_') for empty spaces.
    text_mat = np.full((num_rows, len(key)), '_', dtype='<U1')

    # Populate the matrix with characters from the text.
    for i in range(len(text)):
        row = i // len(key) # Determine the current row index.
        col = i % len(key) # Determine the current column index.
        text_mat[row, col] = text[i] # Place the text character in the matrix.

    return text_mat

def columnar_encrypt(text, key):
    # Create the matrix from the text and key.
    text_mat = columnar_matrix(text, key)

    # Create a list of column indices based on the key and sort them according to the key characters.
    key_order = list(range(len(key))) # Original order of columns.
    key_list = list(key) # Convert the key into a list of characters.
    combined_list = zip(key_list, key_order) # Pair each key character with its column index.
    sorted_key_list, sorted_order = zip(*sorted(combined_list, key=lambda x: x[0])) # Sort columns
    based on key characters.
```

```

# Initialize an empty string to hold the encrypted text.
encrypted_text = ""

# Read the matrix column by column according to the sorted order and concatenate characters to
form the encrypted text.
for i in sorted_order:
    for j in range(math.ceil(len(text) / len(key))):
        encrypted_text += text_mat[j, i]

return encrypted_text

def columnar_decrypt(text, key):
    # Create the matrix for decryption.
    text_mat = columnar_matrix(text, key)

    # Create a list of column indices based on the key and sort them according to the key characters.
    key_order = list(range(len(key))) # Original order of columns.
    key_list = list(key) # Convert the key into a list of characters.
    combined_list = zip(key_list, key_order) # Pair each key character with its column index.
    sorted_key_list, sorted_order = zip(*sorted(combined_list, key=lambda x: x[0])) # Sort columns
    based on key characters.

    # Initialize an empty string to hold the decrypted text.
    decrypted_text = ""
    index = 0 # Index to track the position in the encrypted text.

    # Fill the matrix with the encrypted text characters according to the sorted column order.
    for i in sorted_order:
        for j in range(math.ceil(len(text) / len(key))):
            text_mat[j, i] = text[index]
            index += 1

    # Read the matrix row by row and concatenate characters to form the decrypted text.
    for i in range(len(text)):
        if text_mat[i // len(key), i % len(key)] != '_':
            decrypted_text += text_mat[i // len(key), i % len(key)]

    return decrypted_text

if __name__ == '__main__':
    plain = "Hello world"
    key = "key"

    # Test the functions with sample data.
    matrix = columnar_matrix(plain, key)
    print("Matrix:")
    print(matrix)
    print()

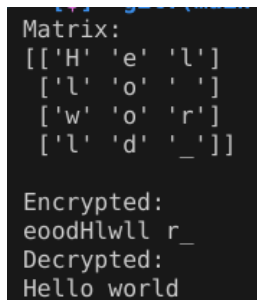
    encrypted = columnar_encrypt(plain, key)
    print("Encrypted:")

```

```
print(encrypted)

decrypted = columnar_decrypt(encrypted, key)
print("Decrypted:")
print(decrypted)
```

### Output:



```
Matrix:
[['H' 'e' 'l' ]
 ['l' 'o' ' ' ]
 ['w' 'o' 'r' ]
 ['l' 'd' ' ' ]]

Encrypted:
eoodHlwll r_
Decrypted:
Hello world
```

### Code Explanation:

#### 1. Import Statements:

- The code imports the ``math`` module for mathematical operations and the ``numpy`` library for array manipulation.

#### 2. Function Definitions:

- ``columnar_matrix(text, key)``:
  - Calculates the number of rows needed based on the length of the text and key.
  - Initializes a matrix with placeholder characters (``_``) to represent empty spaces.
  - Fills the matrix with characters from the text, placing each character into its appropriate position in the matrix.
- ``columnar_encrypt(text, key)``:
  - Uses the ``columnar_matrix`` function to create a matrix of the text.
  - Creates a list of column indices based on the key and sorts these indices according to the alphabetical order of the key characters.
  - Constructs the encrypted text by reading the matrix column by column in the sorted order.
- ``columnar_decrypt(text, key)``:
  - Similar to encryption, it uses the ``columnar_matrix`` function to initialize the matrix.
  - Determines column order based on the key and fills the matrix with the encrypted text characters according to this order.
  - Reads the matrix row by row to reconstruct the original text, skipping placeholder characters.

#### 3. Main Block:

- Defines a sample plaintext and key for testing.
- Demonstrates the matrix creation, encryption, and decryption functions with the provided sample data, printing the matrix, encrypted text, and decrypted text.

**Performance Analysis:****Space Complexity:**

- The space complexity is  $O(n)$ , where  $n$  represents the length of the input text. This is because the matrix created to store the text has dimensions proportional to the length of the text and the key.

**Time Complexity:**

- The time complexity is  $O(n)$ , where  $n$  is the length of the input text. This is due to iterating through the text for both encryption and decryption, as well as sorting the key to determine column order.

## Modified Code:

### 1. Modified-Columnar-Cipher.py

```
import math
import numpy as np
import json
import random
import string

# Define the JSON file to store the keys
KEYS_FILE = 'keys.json'

def generate_random_key():
    alphabet = string.ascii_lowercase
    indices = list(range(len(alphabet)))
    random.shuffle(indices)
    return dict(zip(alphabet, indices))

def save_key_to_json(key):
    with open(KEYS_FILE, 'w') as f:
        json.dump(key, f)

def load_key_from_json():
    try:
        with open(KEYS_FILE, 'r') as f:
            return json.load(f)
    except FileNotFoundError:
        return None

def columnar_matrix(text, key):
    # Determine the number of rows needed in the matrix.
    num_rows = math.ceil(len(text) / len(key))

    # Initialize the matrix with placeholder characters ('_') for empty spaces.
    text_mat = np.full((num_rows, len(key)), '_', dtype='<U1')

    # Populate the matrix with characters from the text.
    for i in range(len(text)):
        row = i // len(key) # Determine the current row index.
        col = i % len(key) # Determine the current column index.
        text_mat[row, col] = text[i] # Place the text character in the matrix.

    return text_mat

def columnar_encrypt(text, key):
    # Create the matrix from the text and key.
    text_mat = columnar_matrix(text, key)

    # Create a list of column indices based on the key and sort them according to the key characters.
    key_order = list(range(len(key))) # Original order of columns.
    key_list = list(key) # Convert the key into a list of characters.
```

```

combined_list = zip(key_list, key_order) # Pair each key character with its column index.
sorted_key_list, sorted_order = zip(*sorted(combined_list, key=lambda x: x[0])) # Sort columns
based on key characters.

```

```

# Initialize an empty string to hold the encrypted text.
encrypted_text = ""

```

```

# Read the matrix column by column according to the sorted order and concatenate characters to
form the encrypted text.

```

```

for i in sorted_order:
    for j in range(math.ceil(len(text) / len(key))):
        encrypted_text += text_mat[j, i]

```

```

return encrypted_text

```

```

def columnar_decrypt(text, key):

```

```

    # Create the matrix for decryption.
    num_rows = math.ceil(len(text) / len(key))
    text_mat = np.full((num_rows, len(key)), '_', dtype='<U1')

```

```

    # Create a list of column indices based on the key and sort them according to the key characters.

```

```

    key_order = list(range(len(key))) # Original order of columns.

```

```

    key_list = list(key) # Convert the key into a list of characters.

```

```

    combined_list = zip(key_list, key_order) # Pair each key character with its column index.

```

```

    sorted_key_list, sorted_order = zip(*sorted(combined_list, key=lambda x: x[0])) # Sort columns
based on key characters.

```

```

    # Initialize an empty string to hold the decrypted text.

```

```

    decrypted_text = ""

```

```

    index = 0 # Index to track the position in the encrypted text.

```

```

    # Fill the matrix with the encrypted text characters according to the sorted column order.

```

```

    for i in sorted_order:
        for j in range(num_rows):
            if index < len(text):
                text_mat[j, i] = text[index]
                index += 1

```

```

    # Read the matrix row by row and concatenate characters to form the decrypted text.

```

```

    for i in range(len(text)):
        if text_mat[i // len(key), i % len(key)] != '_':
            decrypted_text += text_mat[i // len(key), i % len(key)]

```

```

    return decrypted_text

```

```

if __name__ == '__main__':

```

```

    plain = "Hello world"

```

```

    key = "key"

```

```

    # Check if the key file exists; if not, generate a new key and save it.

```

```

    key_mapping = load_key_from_json()

```

```

    if key_mapping is None:

```

```
key_mapping = generate_random_key()
save_key_to_json(key_mapping)

# Map the key to the generated indices.
key = ".join(sorted(key_mapping.keys()), key=lambda k: key_mapping[k]))

# Test the functions with sample data.
matrix = columnar_matrix(plain, key)
print("Matrix:")
print(matrix)
print()

encrypted = columnar_encrypt(plain, key)
print("Encrypted:")
print(encrypted)

decrypted = columnar_decrypt(encrypted, key)
print("Decrypted:")
print(decrypted)
```

**Output:**

[illegible]

### Code Explanation:

### 1. Import Statements:

- The script imports several modules: math for mathematical calculations, numpy for handling arrays, json for JSON file operations, random for generating random values, and string for string operations.

## 2. Function Definitions:

- generate\_random\_key():
  - Generates a random substitution key by shuffling the indices of the alphabet and pairing them with the letters to create a mapping.
- save\_key\_to\_json(key):
  - Saves the generated key into a JSON file for future use.
- load\_key\_from\_json():
  - Loads the key from the JSON file if it exists; otherwise, it returns None.
- columnar\_matrix(text, key):
  - Calculates how many rows are needed for the matrix based on the text length and key length.
  - Initializes a matrix with placeholder characters ('\_') for empty spaces.
  - Fills the matrix with characters from the text, placing them in the correct row and column positions.

- `columnar_encrypt(text, key)`:
  - Uses the `columnar_matrix` function to create the matrix for encryption.
  - Orders the columns based on the alphabetical order of the key characters.
  - Constructs the encrypted text by reading the matrix column by column according to the sorted column order.

- `columnar_decrypt(text, key)`:
  - Similar to encryption, initializes a matrix for decryption.
  - Determines the column order based on the key and fills the matrix with characters from the encrypted text accordingly.
  - Reconstructs the original text by reading the matrix row by row and ignoring placeholder characters.

### **3. Main Block:**

- Defines a sample plaintext and key.
- Checks if a key file exists; if not, generates a new key and saves it to a JSON file.
- Sorts the key based on the generated key mapping.
- Tests the matrix creation, encryption, and decryption functions with the sample plaintext and prints the results.

### **Performance Analysis:**

#### **Space Complexity:**

- The space complexity is  $O(n)$ , where  $n$  is the length of the input text. This is because the matrix created to store the text has dimensions proportional to the length of the text and the key.

#### **Time Complexity:**

- The time complexity is  $O(n)$ , where  $n$  represents the length of the input text. This is due to iterating through the text for both encryption and decryption, along with sorting the key to determine column order.