FACULTY OF ENGINEERING AND TECHNOLOGY
BACHELOR OF TECHNOLOGY

Information and Network Security (303105376)

SEMESTER VII
Computer Science and Engineering Department

Lab Manual

CERTIFICATE

Mr. ***Dinesh Tak*** *with Enrollment No.* ***2203051050175*** *has successfully completed his laboratory experiments* ***Information and Network Security (303105376)*** *from the department of* ***Computer Science and Engineering*** *during the academic year* ***2025-2026.***

Date of Submission ……………… **Staff In charge ………………..**

Head of Department………………

TABLE OF CONTENT

PRACTICAL NO. : 1

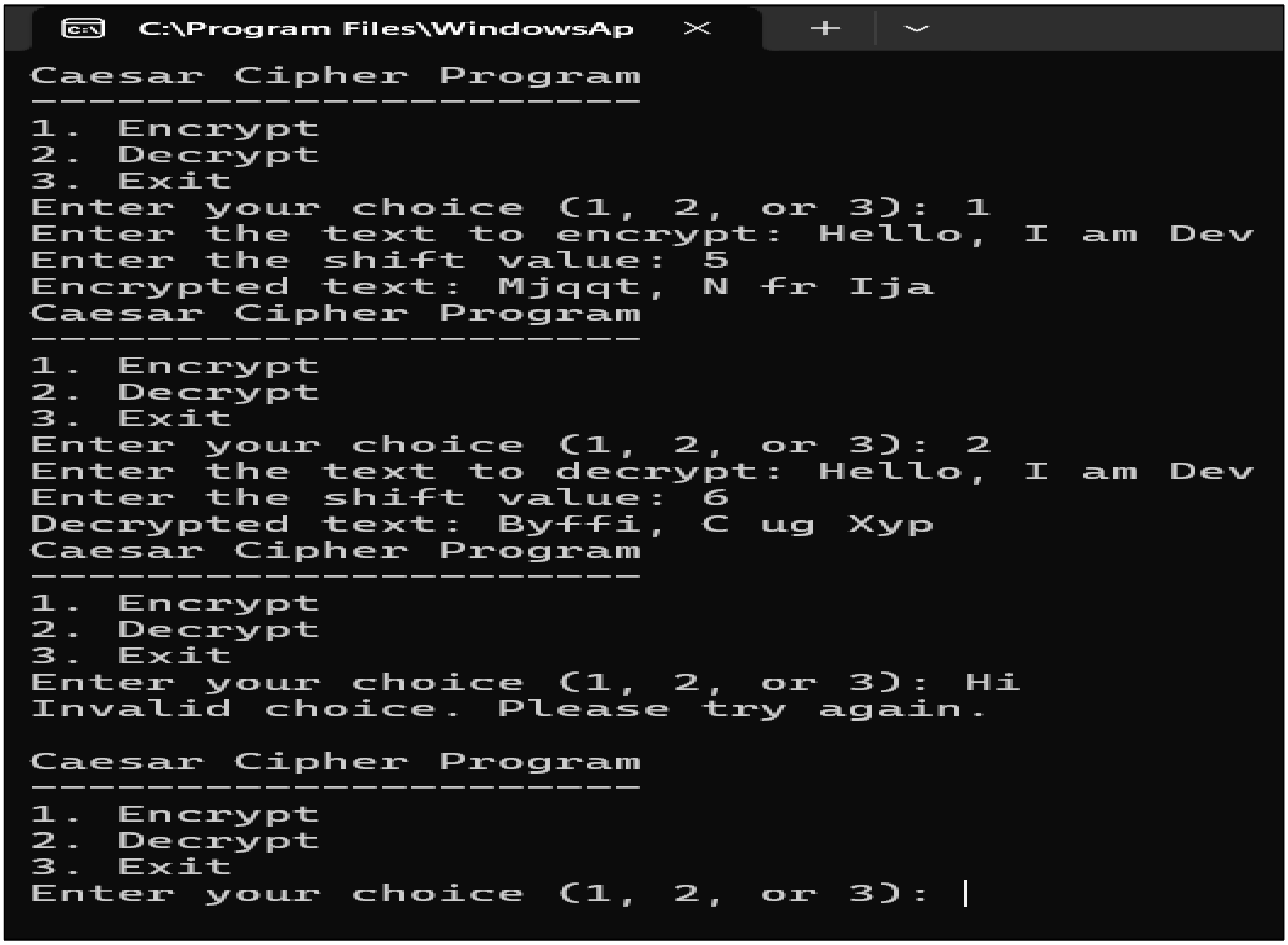
Aim : Implement Caesar cipher encryption – decryption.

Description : The Caesar Cipher is one of the oldest and simplest encryption techniques, named after Julius Caesar, who allegedly used it to communicate with his generals. This substitution cipher works by shifting each letter in the plaintext a fixed number of positions down the alphabet, determined by a key. The Caesar Cipher is easy to implement and understand, making it a popular introduction to cryptography. The Caesar Cipher is a type of substitution cipher where each letter in the plaintext is shifted a fixed number of positions down the alphabet. This shift is determined by a key, which is a number that specifies how many positions each letter should be moved.

Code :

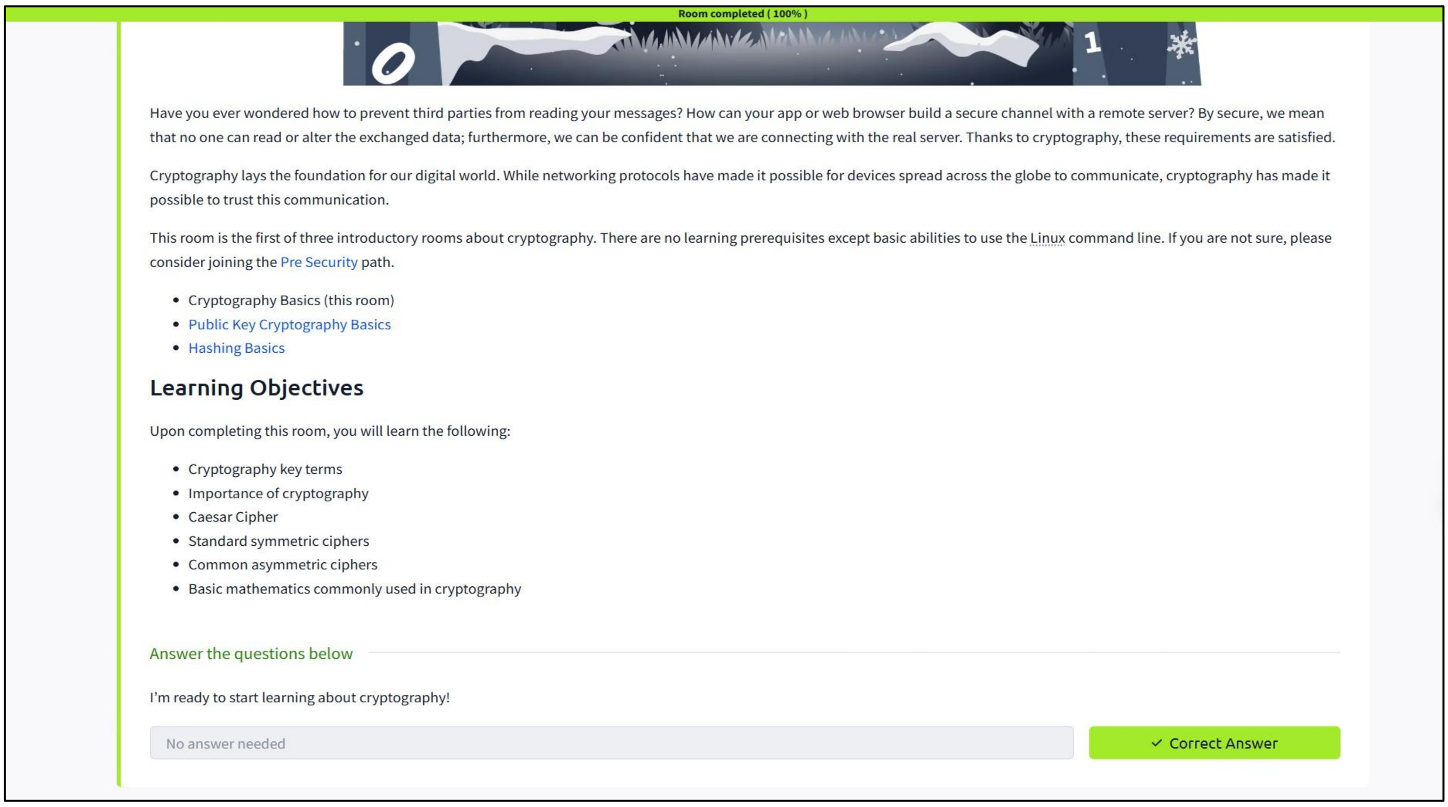
```
def caesar_encrypt(text, shift):
    encrypted_text = ""
    for char in text:
        if char.isalpha():
            ascii_offset = 65 if char.isupper() else 97
            encrypted_char = chr((ord(char) - ascii_offset + shift) % 26 + ascii_offset)
            encrypted_text += encrypted_char
        else:
            encrypted_text += char
    return encrypted_text
def caesar_decrypt(text, shift):
    decrypted_text = ""
    for char in text:
        if char.isalpha():
            ascii_offset = 65 if char.isupper() else 97
            decrypted_char = chr((ord(char) - ascii_offset - shift) % 26 + ascii_offset)
            decrypted_text += decrypted_char
        else:
            decrypted_text += char
    return decrypted_text
def main():
    while True:
        print("Caesar Cipher Program")
        print("-------------")
        print("1. Encrypt")
        print("2. Decrypt")
        print("3. Exit")
        choice = input("Enter your choice (1, 2, or 3): ")
        if choice == "1":
            text = input("Enter the text to encrypt: ")
            shift = int(input("Enter the shift value: "))
            encrypted_text = caesar_encrypt(text, shift)
            print("Encrypted text:", encrypted_text)
        elif choice == "2":
            text = input("Enter the text to decrypt: ")
            shift = int(input("Enter the shift value: "))
            decrypted_text = caesar_decrypt(text, shift)
            print("Decrypted text:", decrypted_text)
        elif choice == "3":
            print("Exiting the program...")
            break
        else:
            print("Invalid choice. Please try again.\n")
if _name___== "_main_":
    main()
```

OUTPUT :



```
Caesar Cipher Program
---------------------
1. Encrypt
2. Decrypt
3. Exit
Enter your choice (1, 2, or 3): 1
Enter the text to encrypt: Hello, I am Dev
Enter the shift value: 5
Encrypted text: Mjqqt, N fr Ija
Caesar Cipher Program
---------------------
1. Encrypt
2. Decrypt
3. Exit
Enter your choice (1, 2, or 3): 2
Enter the text to decrypt: Hello, I am Dev
Enter the shift value: 6
Decrypted text: Byffi, C ug Xyp
Caesar Cipher Program
---------------------
1. Encrypt
2. Decrypt
3. Exit
Enter your choice (1, 2, or 3): Hi
Invalid choice. Please try again.

Caesar Cipher Program
---------------------
1. Encrypt
2. Decrypt
3. Exit
Enter your choice (1, 2, or 3): |
```

TRYHACKME.COM – > CYBER SECURITY BASICS


Room completed (100%)

Have you ever wondered how to prevent third parties from reading your messages? How can your app or web browser build a secure channel with a remote server? By secure, we mean that no one can read or alter the exchanged data; furthermore, we can be confident that we are connecting with the real server. Thanks to cryptography, these requirements are satisfied.

Cryptography lays the foundation for our digital world. While networking protocols have made it possible for devices spread across the globe to communicate, cryptography has made it possible to trust this communication.

This room is the first of three introductory rooms about cryptography. There are no learning prerequisites except basic abilities to use the Linux command line. If you are not sure, please consider joining the Pre Security path.

- Cryptography Basics (this room)
- Public Key Cryptography Basics
- Hashing Basics

Learning Objectives

Upon completing this room, you will learn the following:

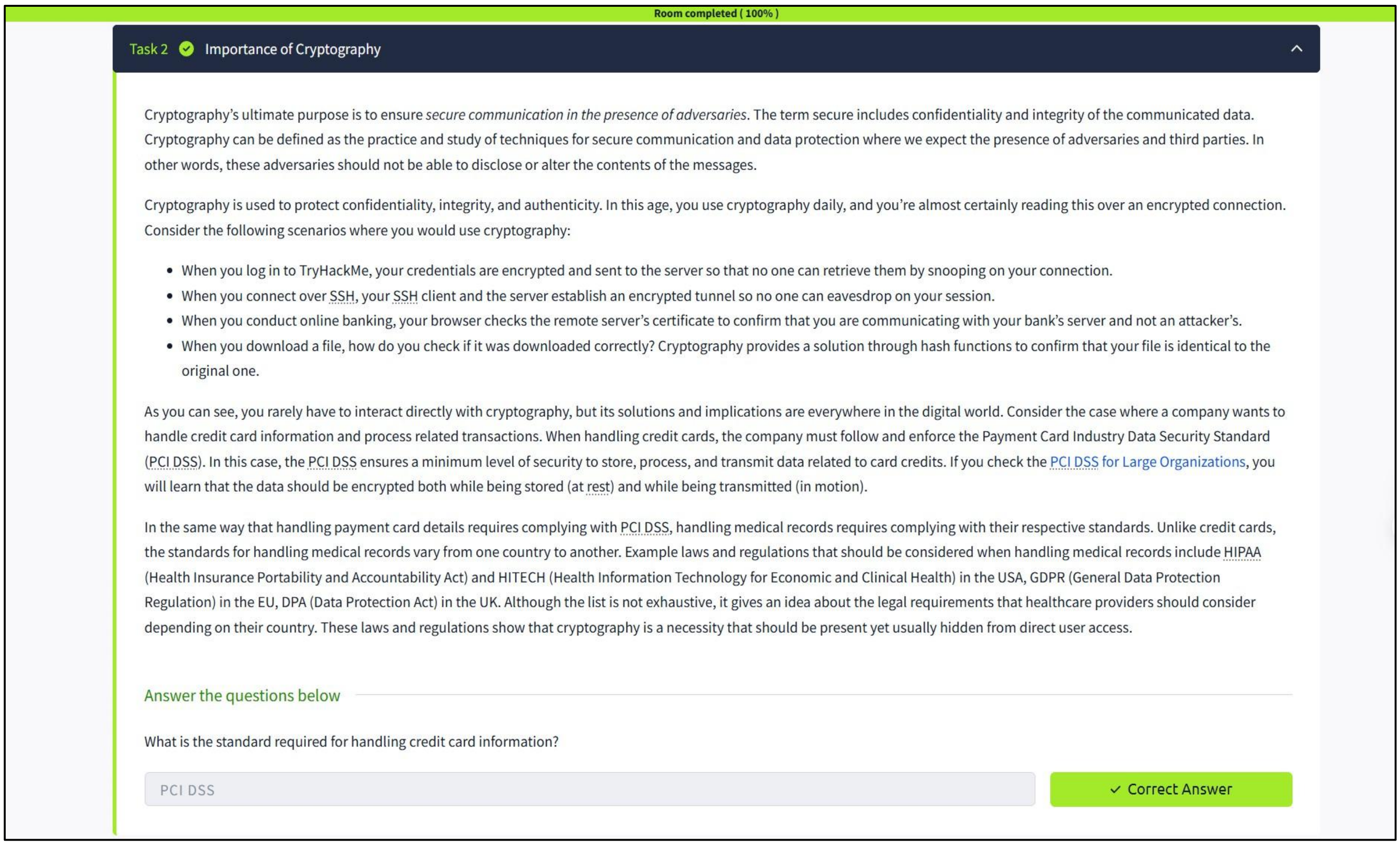
- Cryptography key terms
- Importance of cryptography
- Caesar Cipher
- Standard symmetric ciphers
- Common asymmetric ciphers
- Basic mathematics commonly used in cryptography

Answer the questions below

I'm ready to start learning about cryptography!

No answer needed

✓ Correct Answer


Room completed (100%)

Task 2 Importance of Cryptography

Cryptography's ultimate purpose is to ensure *secure communication in the presence of adversaries*. The term secure includes confidentiality and integrity of the communicated data. Cryptography can be defined as the practice and study of techniques for secure communication and data protection where we expect the presence of adversaries and third parties. In other words, these adversaries should not be able to disclose or alter the contents of the messages.

Cryptography is used to protect confidentiality, integrity, and authenticity. In this age, you use cryptography daily, and you're almost certainly reading this over an encrypted connection. Consider the following scenarios where you would use cryptography:

- When you log in to TryHackMe, your credentials are encrypted and sent to the server so that no one can retrieve them by snooping on your connection.
- When you connect over SSH, your SSH client and the server establish an encrypted tunnel so no one can eavesdrop on your session.
- When you conduct online banking, your browser checks the remote server's certificate to confirm that you are communicating with your bank's server and not an attacker's.
- When you download a file, how do you check if it was downloaded correctly? Cryptography provides a solution through hash functions to confirm that your file is identical to the original one.

As you can see, you rarely have to interact directly with cryptography, but its solutions and implications are everywhere in the digital world. Consider the case where a company wants to handle credit card information and process related transactions. When handling credit cards, the company must follow and enforce the Payment Card Industry Data Security Standard (PCI DSS). In this case, the PCI DSS ensures a minimum level of security to store, process, and transmit data related to card credits. If you check the PCI DSS for Large Organizations, you will learn that the data should be encrypted both while being stored (at rest) and while being transmitted (in motion).

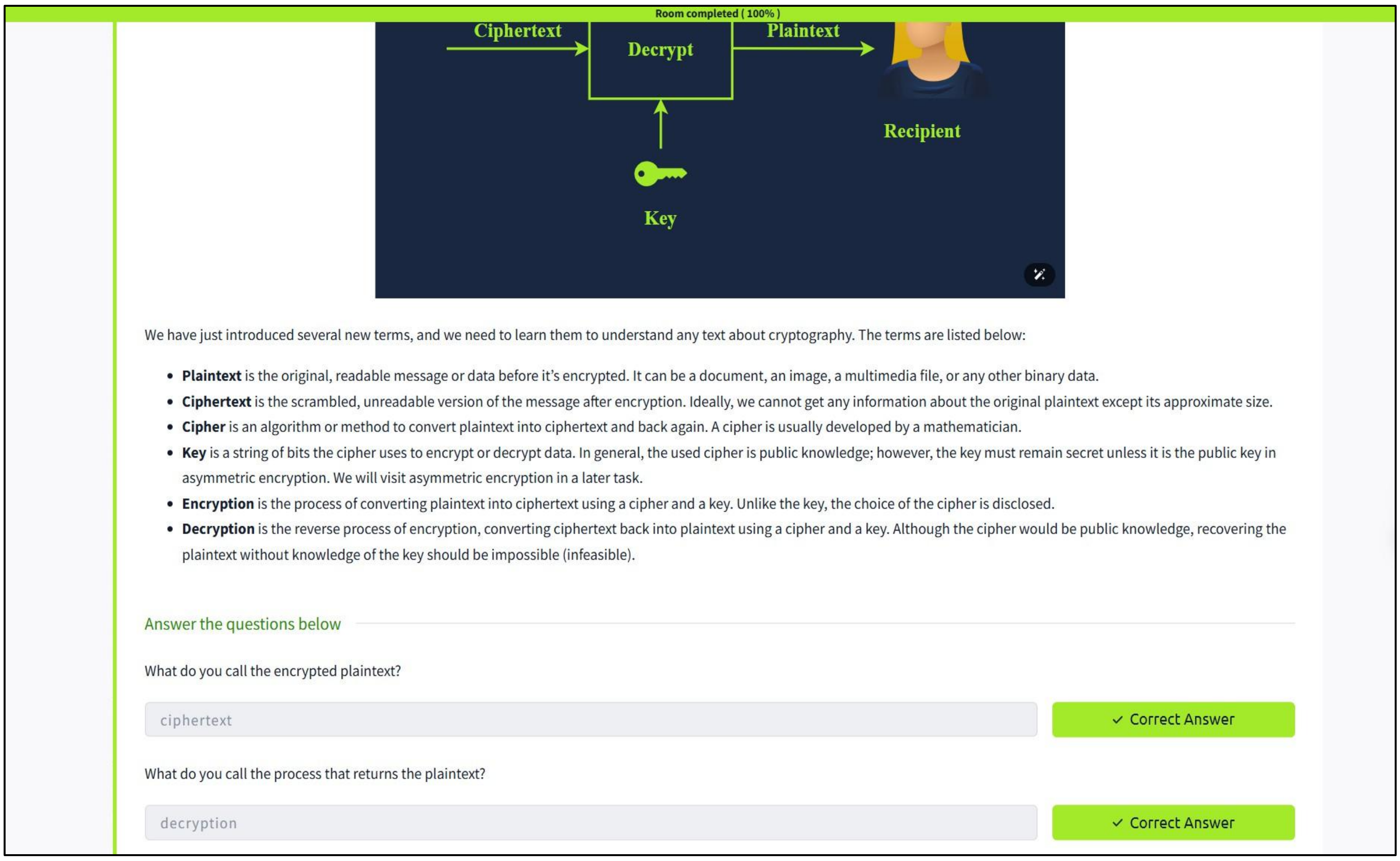
In the same way that handling payment card details requires complying with PCI DSS, handling medical records requires complying with their respective standards. Unlike credit cards, the standards for handling medical records vary from one country to another. Example laws and regulations that should be considered when handling medical records include HIPAA (Health Insurance Portability and Accountability Act) and HITECH (Health Information Technology for Economic and Clinical Health) in the USA, GDPR (General Data Protection Regulation) in the EU, DPA (Data Protection Act) in the UK. Although the list is not exhaustive, it gives an idea about the legal requirements that healthcare providers should consider depending on their country. These laws and regulations show that cryptography is a necessity that should be present yet usually hidden from direct user access.

Answer the questions below

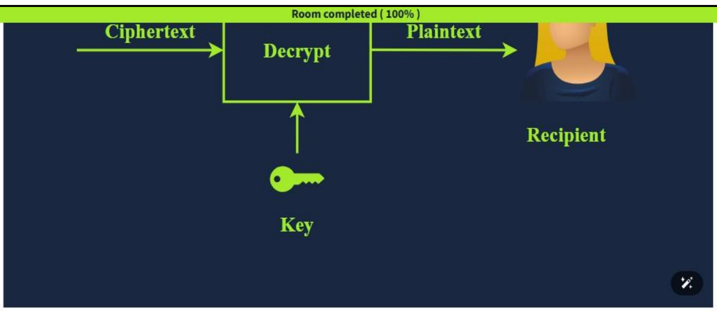
What is the standard required for handling credit card information?

PCI DSS

✓ Correct Answer



Room completed (100%)



We have just introduced several new terms, and we need to learn them to understand any text about cryptography. The terms are listed below:

- **Plaintext** is the original, readable message or data before it's encrypted. It can be a document, an image, a multimedia file, or any other binary data.
- **Ciphertext** is the scrambled, unreadable version of the message after encryption. Ideally, we cannot get any information about the original plaintext except its approximate size.
- **Cipher** is an algorithm or method to convert plaintext into ciphertext and back again. A cipher is usually developed by a mathematician.
- **Key** is a string of bits the cipher uses to encrypt or decrypt data. In general, the used cipher is public knowledge; however, the key must remain secret unless it is the public key in asymmetric encryption. We will visit asymmetric encryption in a later task.
- **Encryption** is the process of converting plaintext into ciphertext using a cipher and a key. Unlike the key, the choice of the cipher is disclosed.
- **Decryption** is the reverse process of encryption, converting ciphertext back into plaintext using a cipher and a key. Although the cipher would be public knowledge, recovering the plaintext without knowledge of the key should be impossible (infeasible).

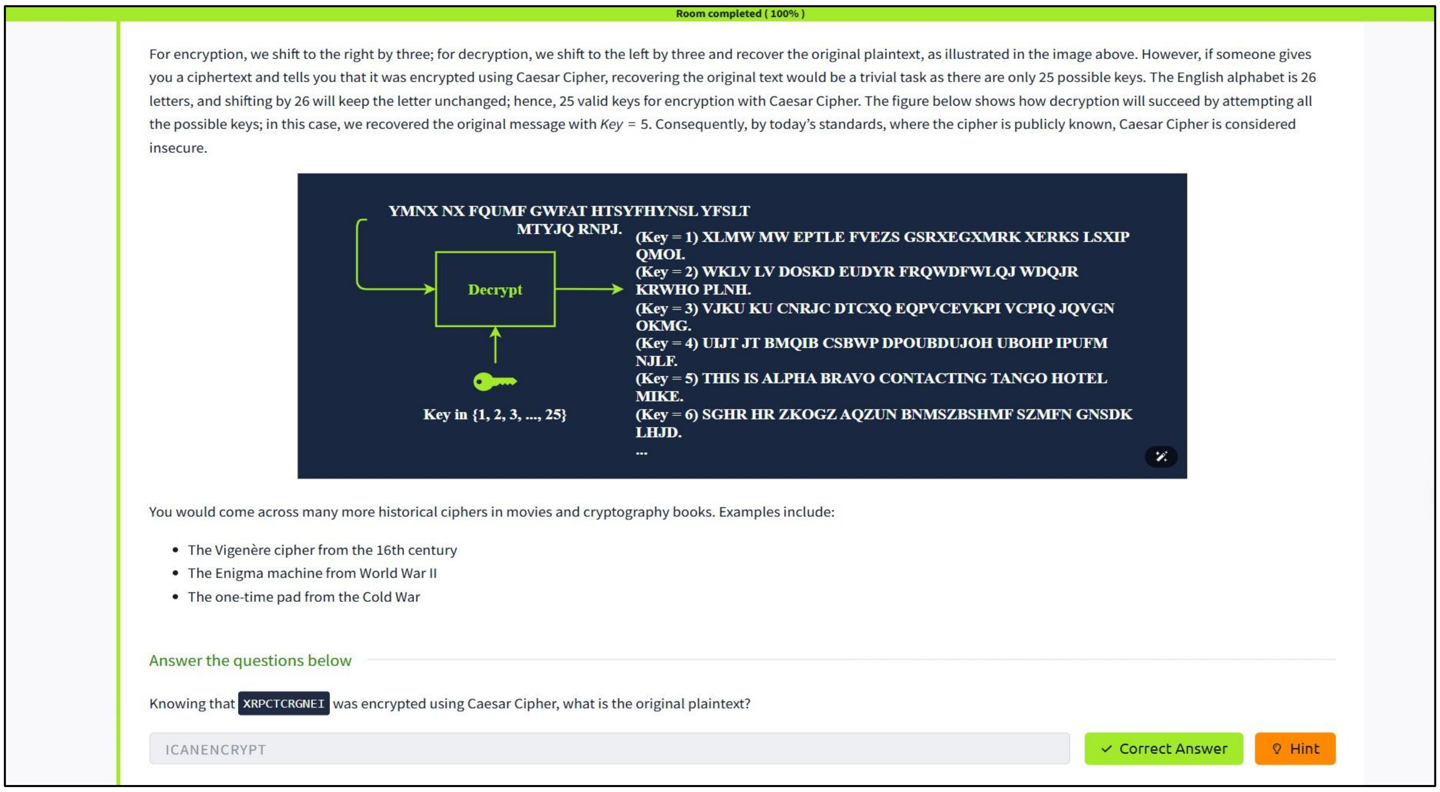
Answer the questions below

What do you call the encrypted plaintext?

ciphertext — ✓ Correct Answer

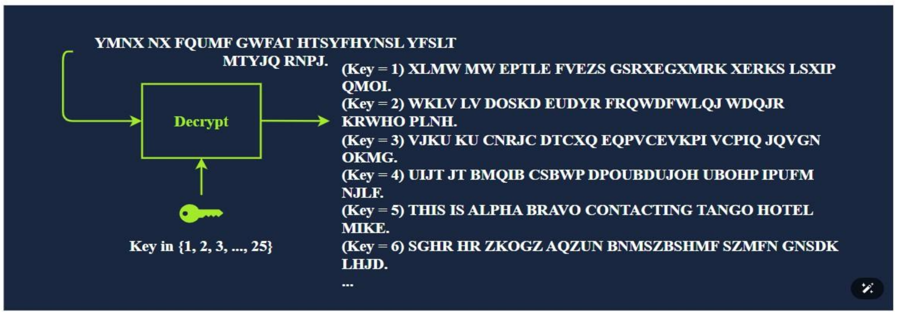
What do you call the process that returns the plaintext?

decryption — ✓ Correct Answer



Room completed (100%)

For encryption, we shift to the right by three; for decryption, we shift to the left by three and recover the original plaintext, as illustrated in the image above. However, if someone gives you a ciphertext and tells you that it was encrypted using Caesar Cipher, recovering the original text would be a trivial task as there are only 25 possible keys. The English alphabet is 26 letters, and shifting by 26 will keep the letter unchanged; hence, 25 valid keys for encryption with Caesar Cipher. The figure below shows how decryption will succeed by attempting all the possible keys; in this case, we recovered the original message with *Key* = 5. Consequently, by today's standards, where the cipher is publicly known, Caesar Cipher is considered insecure.


YMNX NX FQUMF GWFAT HTSYFHYNSL YFSLT MTYJQ RNPJ.
Decrypt
Key in {1, 2, 3, ..., 25}
(Key = 1) XLMW MW EPTLE FVEZS GSRXEGXMRK XERKS LSXIP QMOI.
(Key = 2) WKLV LV DOSKD EUDYR FRQWDFWLQJ WDQJR KRWHO PLNH.
(Key = 3) VJKU KU CNRJC DTCXQ EQPVCEVKPI VCPIQ JQVGN OKMG.
(Key = 4) UIJT JT BMQIB CSBWP DPOUBDUJOH UBOHP IPUFM NJLF.
(Key = 5) THIS IS ALPHA BRAVO CONTACTING TANGO HOTEL MIKE.
(Key = 6) SGHR HR ZKOGZ AQZUN BNMSZBSHMF SZMFN GNSDK LHJD.
...

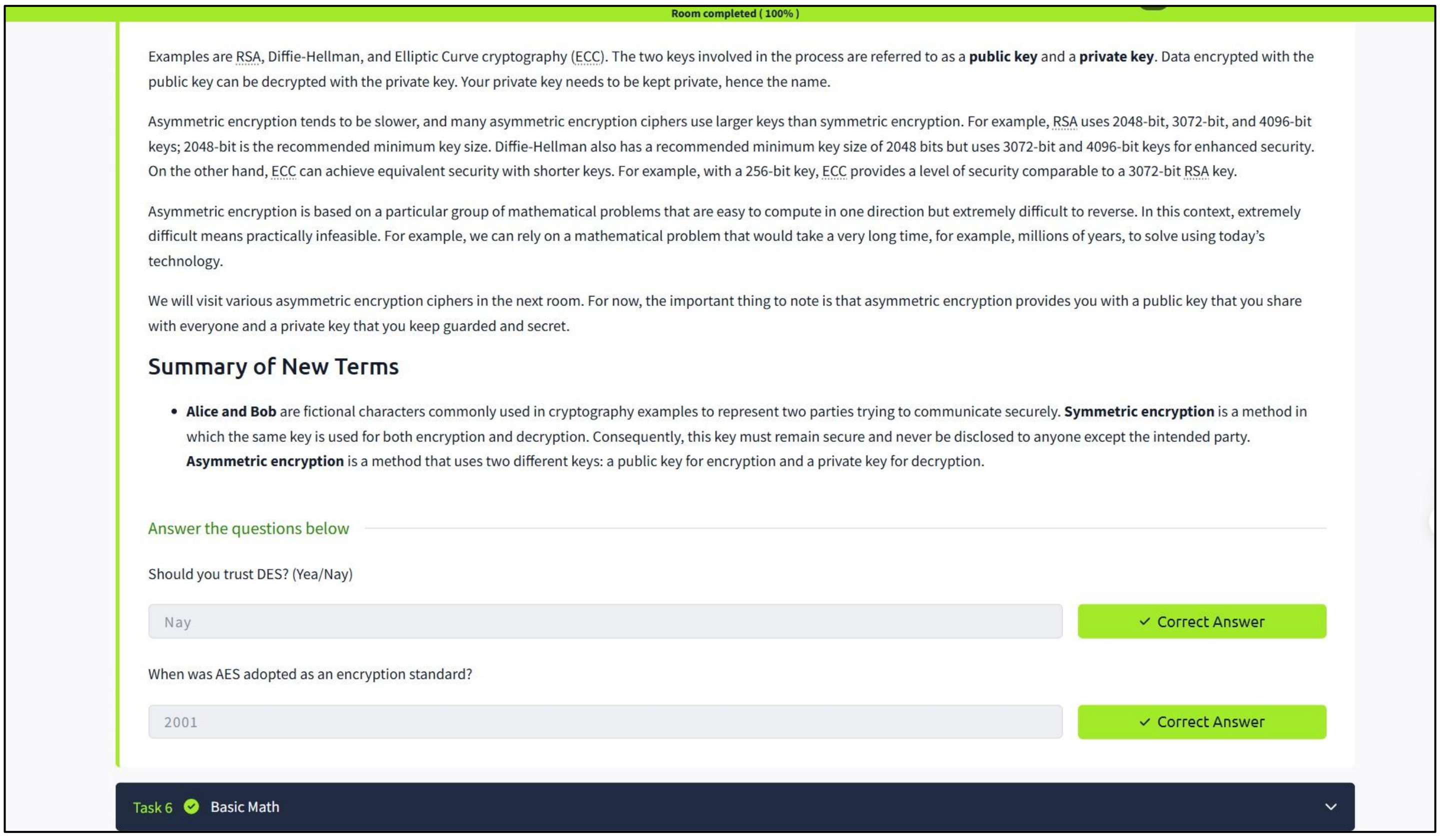
You would come across many more historical ciphers in movies and cryptography books. Examples include:

- The Vigenère cipher from the 16th century
- The Enigma machine from World War II
- The one-time pad from the Cold War

Answer the questions below

Knowing that `XRPCTCRGNEI` was encrypted using Caesar Cipher, what is the original plaintext?

ICANENCRYPT — ✓ Correct Answer — Hint


Room completed (100%)

Examples are RSA, Diffie-Hellman, and Elliptic Curve cryptography (ECC). The two keys involved in the process are referred to as a **public key** and a **private key**. Data encrypted with the public key can be decrypted with the private key. Your private key needs to be kept private, hence the name.

Asymmetric encryption tends to be slower, and many asymmetric encryption ciphers use larger keys than symmetric encryption. For example, RSA uses 2048-bit, 3072-bit, and 4096-bit keys; 2048-bit is the recommended minimum key size. Diffie-Hellman also has a recommended minimum key size of 2048 bits but uses 3072-bit and 4096-bit keys for enhanced security. On the other hand, ECC can achieve equivalent security with shorter keys. For example, with a 256-bit key, ECC provides a level of security comparable to a 3072-bit RSA key.

Asymmetric encryption is based on a particular group of mathematical problems that are easy to compute in one direction but extremely difficult to reverse. In this context, extremely difficult means practically infeasible. For example, we can rely on a mathematical problem that would take a very long time, for example, millions of years, to solve using today's technology.

We will visit various asymmetric encryption ciphers in the next room. For now, the important thing to note is that asymmetric encryption provides you with a public key that you share with everyone and a private key that you keep guarded and secret.

Summary of New Terms

- **Alice and Bob** are fictional characters commonly used in cryptography examples to represent two parties trying to communicate securely. **Symmetric encryption** is a method in which the same key is used for both encryption and decryption. Consequently, this key must remain secure and never be disclosed to anyone except the intended party. **Asymmetric encryption** is a method that uses two different keys: a public key for encryption and a private key for decryption.

Answer the questions below

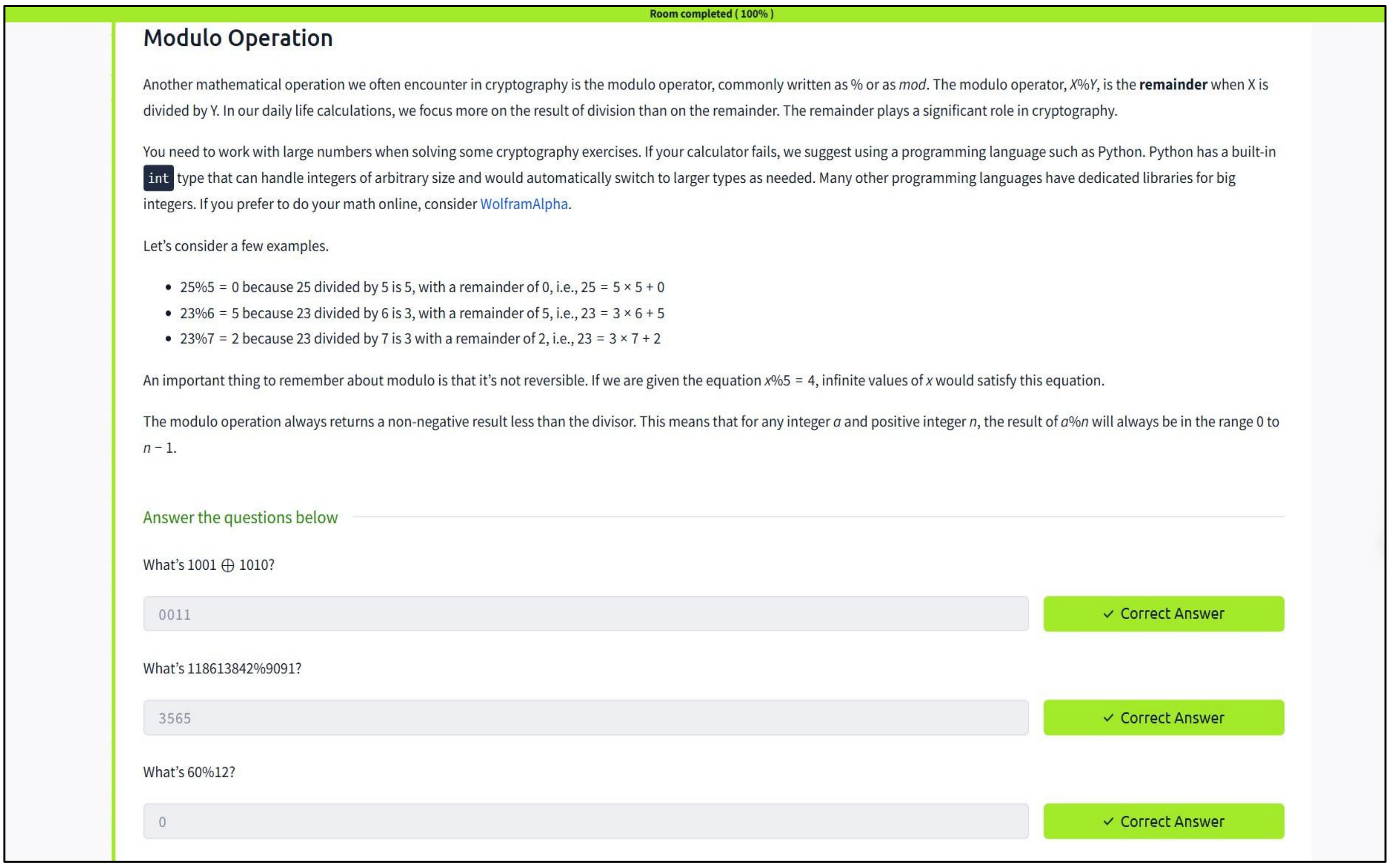
Should you trust DES? (Yea/Nay)

Nay — ✓ Correct Answer

When was AES adopted as an encryption standard?

2001 — ✓ Correct Answer

Task 6 ✓ Basic Math


Room completed (100%)

Modulo Operation

Another mathematical operation we often encounter in cryptography is the modulo operator, commonly written as % or as *mod*. The modulo operator, *X%Y*, is the **remainder** when X is divided by Y. In our daily life calculations, we focus more on the result of division than on the remainder. The remainder plays a significant role in cryptography.

You need to work with large numbers when solving some cryptography exercises. If your calculator fails, we suggest using a programming language such as Python. Python has a built-in `int` type that can handle integers of arbitrary size and would automatically switch to larger types as needed. Many other programming languages have dedicated libraries for big integers. If you prefer to do your math online, consider WolframAlpha.

Let's consider a few examples.

- 25%5 = 0 because 25 divided by 5 is 5, with a remainder of 0, i.e., 25 = 5 × 5 + 0
- 23%6 = 5 because 23 divided by 6 is 3, with a remainder of 5, i.e., 23 = 3 × 6 + 5
- 23%7 = 2 because 23 divided by 7 is 3 with a remainder of 2, i.e., 23 = 3 × 7 + 2

An important thing to remember about modulo is that it's not reversible. If we are given the equation *x*%5 = 4, infinite values of *x* would satisfy this equation.

The modulo operation always returns a non-negative result less than the divisor. This means that for any integer *a* and positive integer *n*, the result of *a*%*n* will always be in the range 0 to *n* − 1.

Answer the questions below

What's 1001 ⊕ 1010?

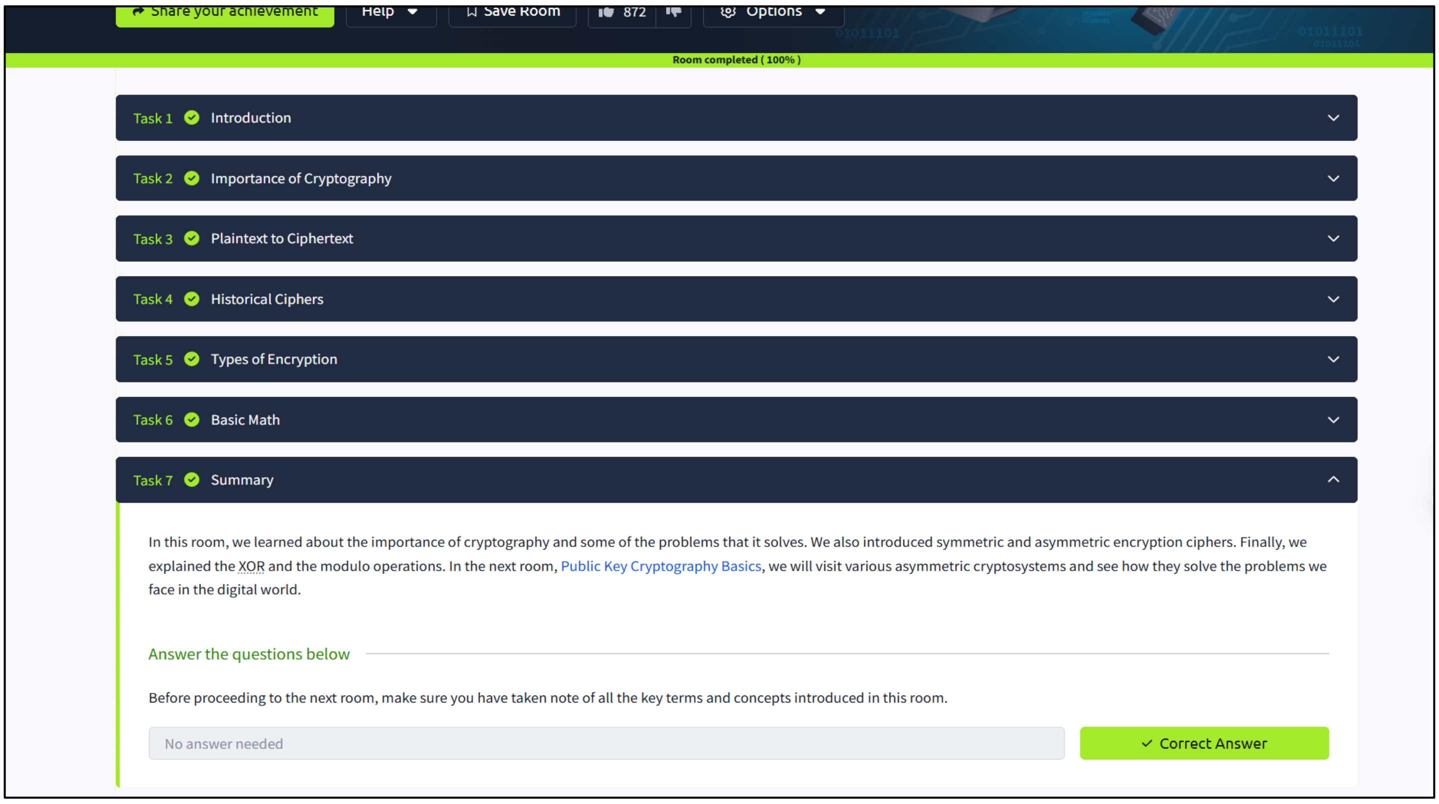
0011 — ✓ Correct Answer

What's 118613842%9091?

3565 — ✓ Correct Answer

What's 60%12?

0 — ✓ Correct Answer



Task 1 Introduction

Task 2 Importance of Cryptography

Task 3 Plaintext to Ciphertext

Task 4 Historical Ciphers

Task 5 Types of Encryption

Task 6 Basic Math

Task 7 Summary

In this room, we learned about the importance of cryptography and some of the problems that it solves. We also introduced symmetric and asymmetric encryption ciphers. Finally, we explained the XOR and the modulo operations. In the next room, Public Key Cryptography Basics, we will visit various asymmetric cryptosystems and see how they solve the problems we face in the digital world.

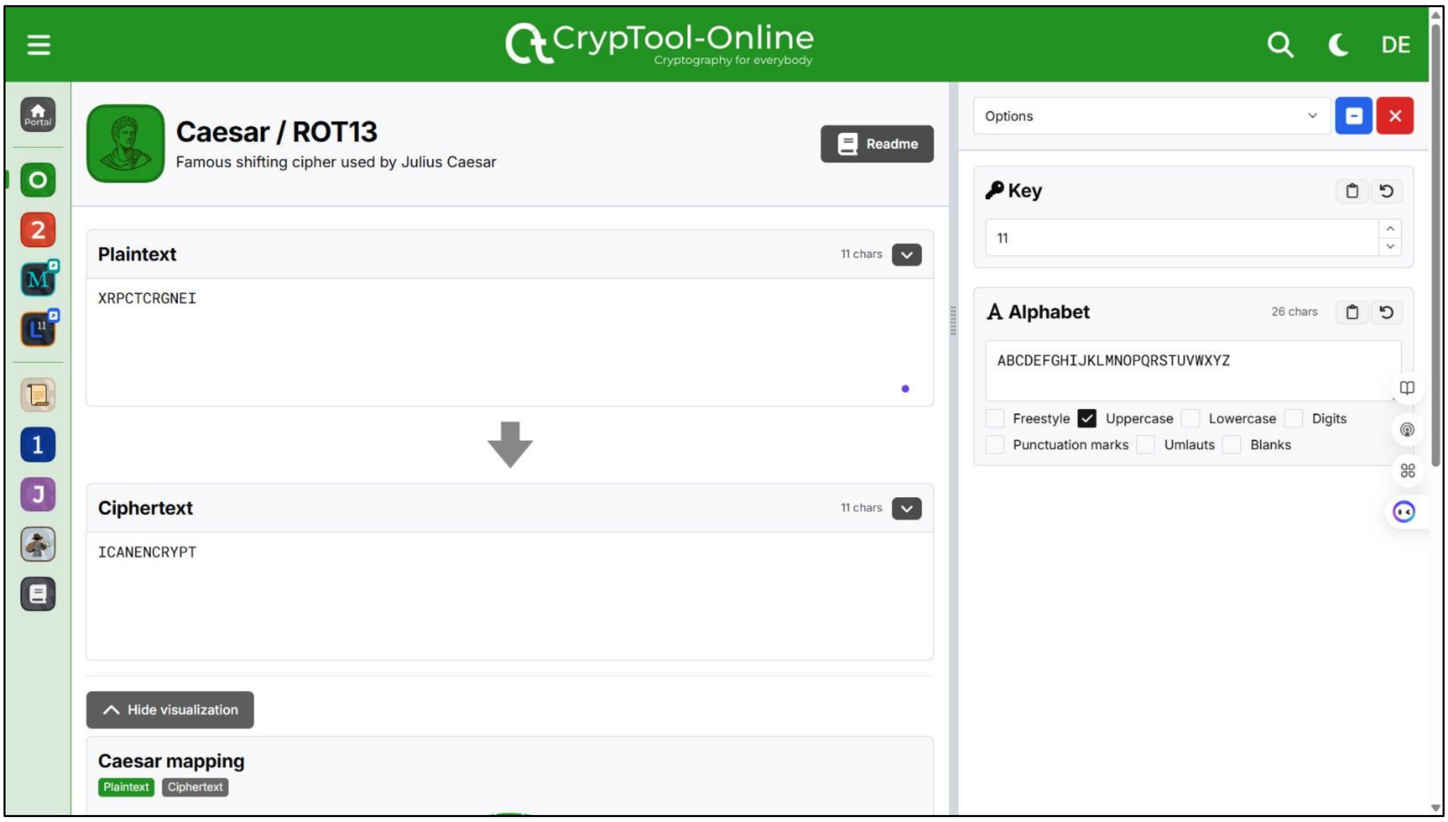
Answer the questions below

Before proceeding to the next room, make sure you have taken note of all the key terms and concepts introduced in this room.

No answer needed

✓ Correct Answer

CRYPTOOL SOFTWARE USE :



Caesar / ROT13
Famous shifting cipher used by Julius Caesar

Plaintext (11 chars)
```
XRPCTCRGNEI
```

Ciphertext (11 chars)
```
ICANENCRYPT
```

Key: 11

Alphabet (26 chars)
```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Freestyle [✓ Uppercase] Lowercase Digits Punctuation marks Umlauts Blanks

PRACTICAL NO. : 2

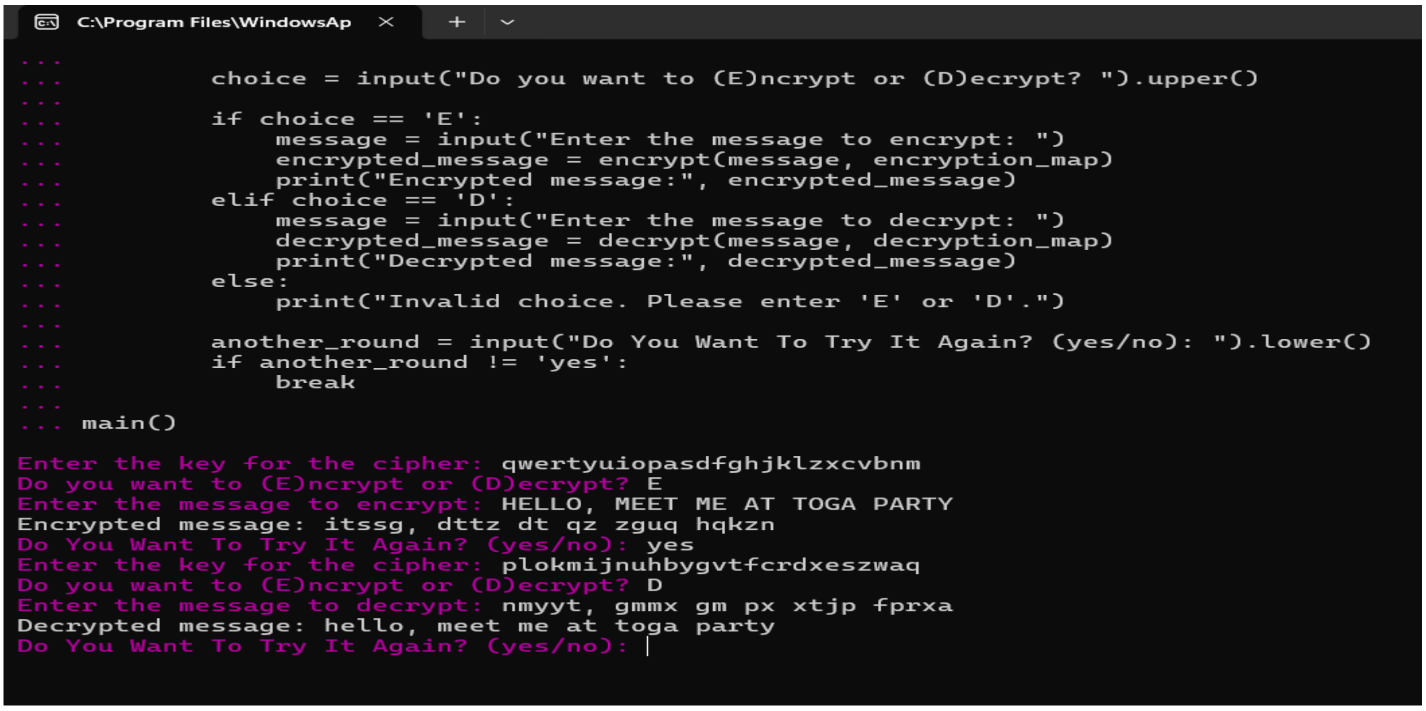
Aim: To implement mono-alphabetic cipher for encryption – decryption.

Description: A mono-alphabetic cipher is a type of substitution cipher where each letter in the plaintext is replaced by a fixed letter in the ciphertext. This substitution is consistent throughout the encryption process, meaning that a particular letter in the plaintext will always be replaced by the same letter in the ciphertext.

CODE:

```
import string

def create_cipher_mapping(key):
    alphabet = string.ascii_lowercase
    key = key.lower()
    key_unique = ''.join(sorted(set(key), key=key.index))  # Remove duplicates and maintain order
    remaining_chars = ''.join([c for c in alphabet if c not in key_unique])
    cipher_alphabet = key_unique + remaining_chars
    encryption_map = dict(zip(alphabet, cipher_alphabet))
    decryption_map = dict(zip(cipher_alphabet, alphabet))
    return encryption_map, decryption_map

def encrypt(message, encryption_map):
    encrypted_message = ""
    for char in message.lower():
        if char in encryption_map:
            encrypted_message += encryption_map[char]
        else:
            encrypted_message += char  # Keep non-alphabetic characters as is
    return encrypted_message

def decrypt(message, decryption_map):
    decrypted_message = ""
    for char in message.lower():
        if char in decryption_map:
            decrypted_message += decryption_map[char]
        else:
            decrypted_message += char  # Keep non-alphabetic characters as is
    return decrypted_message

def main():
    while True:
        key = input("Enter the key for the cipher: ")
        encryption_map, decryption_map = create_cipher_mapping(key)

        choice = input("Do you want to (E)ncrypt or (D)ecrypt? ").upper()

        if choice == 'E':
            message = input("Enter the message to encrypt: ")
            encrypted_message = encrypt(message, encryption_map)
            print("Encrypted message:", encrypted_message)
        elif choice == 'D':
            message = input("Enter the message to decrypt: ")
            decrypted_message = decrypt(message, decryption_map)
            print("Decrypted message:", decrypted_message)
        else:
            print("Invalid choice. Please enter 'E' or 'D'.")

        another_round = input("Do You Want To Try It Again? (yes/no): ").lower()
        if another_round != 'yes':
            break

main()
```

OUTPUT:



```
C:\Program Files\WindowsAp
...
...         choice = input("Do you want to (E)ncrypt or (D)ecrypt? ").upper()
...
...         if choice == 'E':
...             message = input("Enter the message to encrypt: ")
...             encrypted_message = encrypt(message, encryption_map)
...             print("Encrypted message:", encrypted_message)
...         elif choice == 'D':
...             message = input("Enter the message to decrypt: ")
...             decrypted_message = decrypt(message, decryption_map)
...             print("Decrypted message:", decrypted_message)
...         else:
...             print("Invalid choice. Please enter 'E' or 'D'.")
...
...         another_round = input("Do You Want To Try It Again? (yes/no): ").lower()
...         if another_round != 'yes':
...             break
...
... main()

Enter the key for the cipher: qwertyuiopasdfghjklzxcvbnm
Do you want to (E)ncrypt or (D)ecrypt? E
Enter the message to encrypt: HELLO, MEET ME AT TOGA PARTY
Encrypted message: itssg, dttz dt qz zguq hqkzn
Do You Want To Try It Again? (yes/no): yes
Enter the key for the cipher: plokmijnuhbygvtfcrdxeszwaq
Do you want to (E)ncrypt or (D)ecrypt? D
Enter the message to decrypt: nmyyt, gmmx gm px xtjp fprxa
Decrypted message: hello, meet me at toga party
Do You Want To Try It Again? (yes/no): |
```

CRYPTANALYSIS OF MONO – ALPHABETIC CIPHER

FREQUENCY ANALYSIS

In Mono-alphabetic cipher, rather than just shifting the alphabet it could shuffle (jumble) the letters arbitrarily. Each plaintext letter maps to a different random ciphertext letter, hence key is 26 letters long.

The sender and the receiver decide on a randomly selected permutation of the letters of the alphabet. With 26 letters in alphabet, the possible permutations are 26!

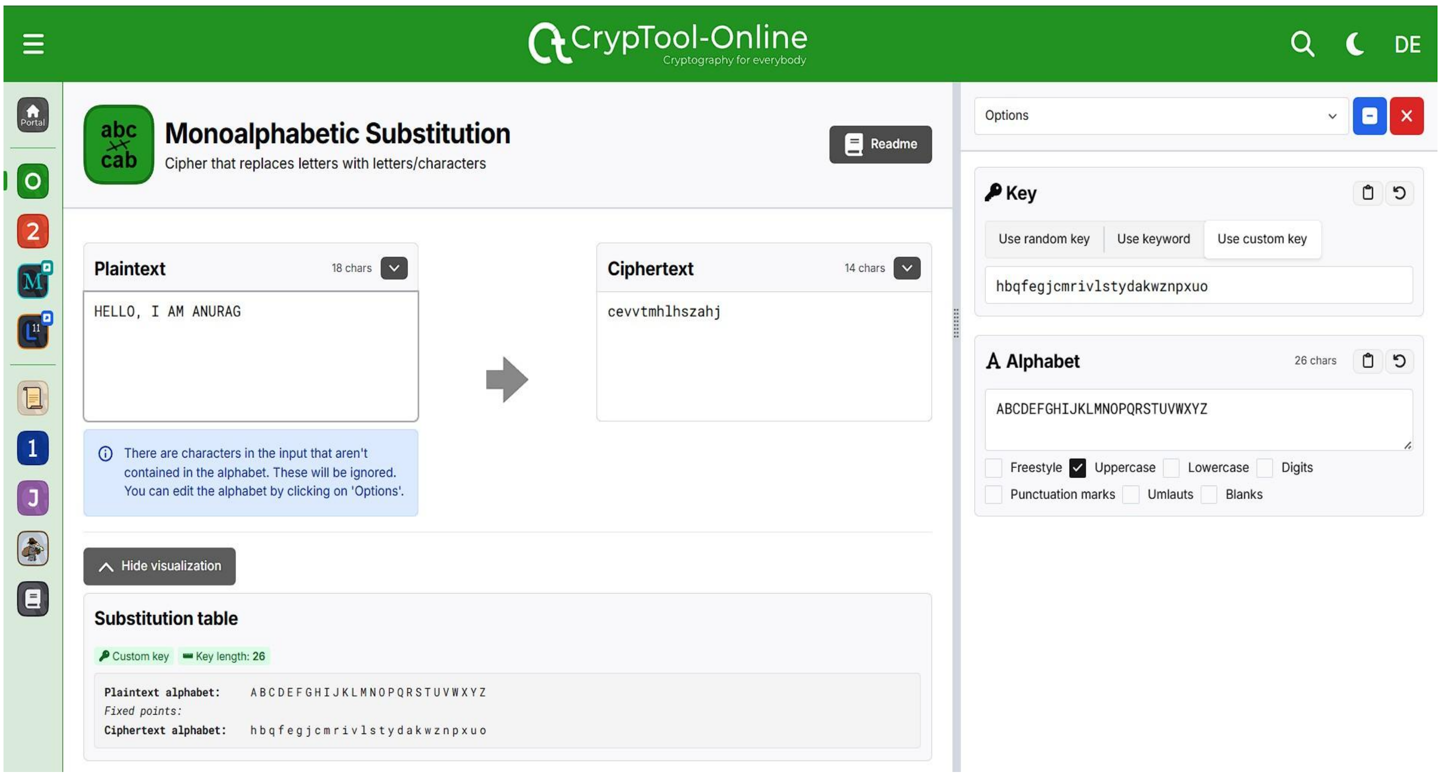
In English Language, letters are not equally commonly used, in English E is by far the most common letter – followed by T,R,N,I,O,A,S. Other letters like Z,J,K,Q,X are fairly rare. In Mono-alphabetic cipher, it has tables of single, double & triple letter frequencies for various languages.

STEPS OF CRYPTAALYSIS OF MONO – ALPHABETIC CIPHER

1. mono-alphabetic substitution ciphers do not change relative letter frequencies.
2. calculate letter frequencies for ciphertext.
3. compare counts/plots against known values.
4. for monoalphabetic must identify each letter– tables of common double/triple letters help.

If the cryptanalyst knows the nature of the plaintext, then the analyst can exploit the regularities of the language. The relative frequency of the letters can be determined and compared to a standard frequency distribution for English. If the message were long enough, this technique alone might be sufficient, but relatively for the short message, we cannot expect an exact match.

CRYPTOOL SOFTWARE :


CrypTool-Online
Cryptography for everybody
DE
Portal
Monoalphabetic Substitution
Cipher that replaces letters with letters/characters
Readme
Options
Hide visualization

Plaintext 18 chars

```
HELLO, I AM ANURAG
```

Ciphertext 14 chars

```
cevvtmhlhszahj
```

There are characters in the input that aren't contained in the alphabet. These will be ignored. You can edit the alphabet by clicking on 'Options'.

Key: Use random key | Use keyword | Use custom key

```
hbqfegjcmrivlstydakwznpxuo
```

Alphabet 26 chars

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

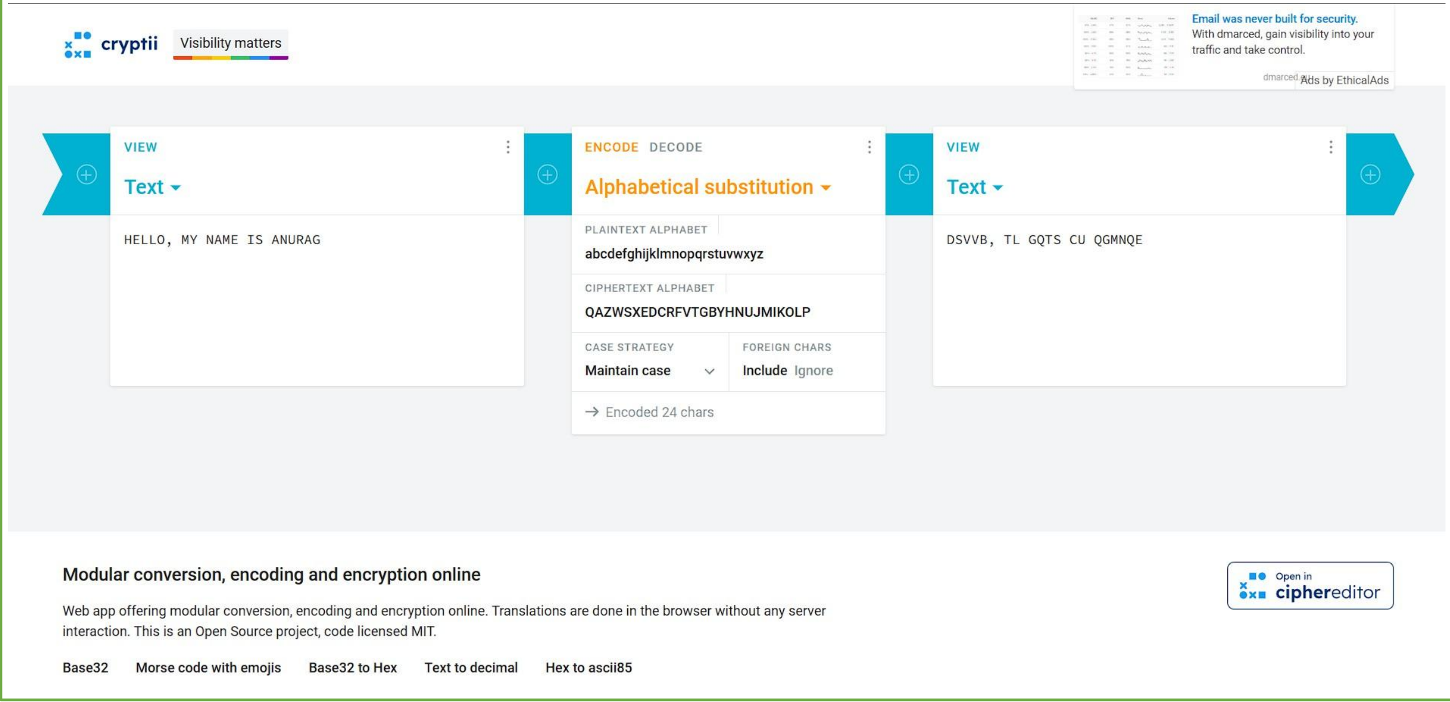
Freestyle [✓ Uppercase] Lowercase Digits Punctuation marks Umlauts Blanks

Substitution table

Custom key · Key length: 26

```
Plaintext alphabet:   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Fixed points:
Ciphertext alphabet:  h b q f e g j c m r i v l s t y d a k w z n p x u o
```

CRYPTII TOOL SOFTWARE:



cryptii Visibility matters

Email was never built for security. With dmarced, gain visibility into your traffic and take control. Ads by EthicalAds

VIEW Text

```
HELLO, MY NAME IS ANURAG
```

ENCODE DECODE
Alphabetical substitution
PLAINTEXT ALPHABET: abcdefghijklmnopqrstuvwxyz
CIPHERTEXT ALPHABET: QAZWSXEDCRFVTGBYHNUJMIKOLP
CASE STRATEGY: Maintain case
FOREIGN CHARS: Include Ignore
→ Encoded 24 chars

VIEW Text

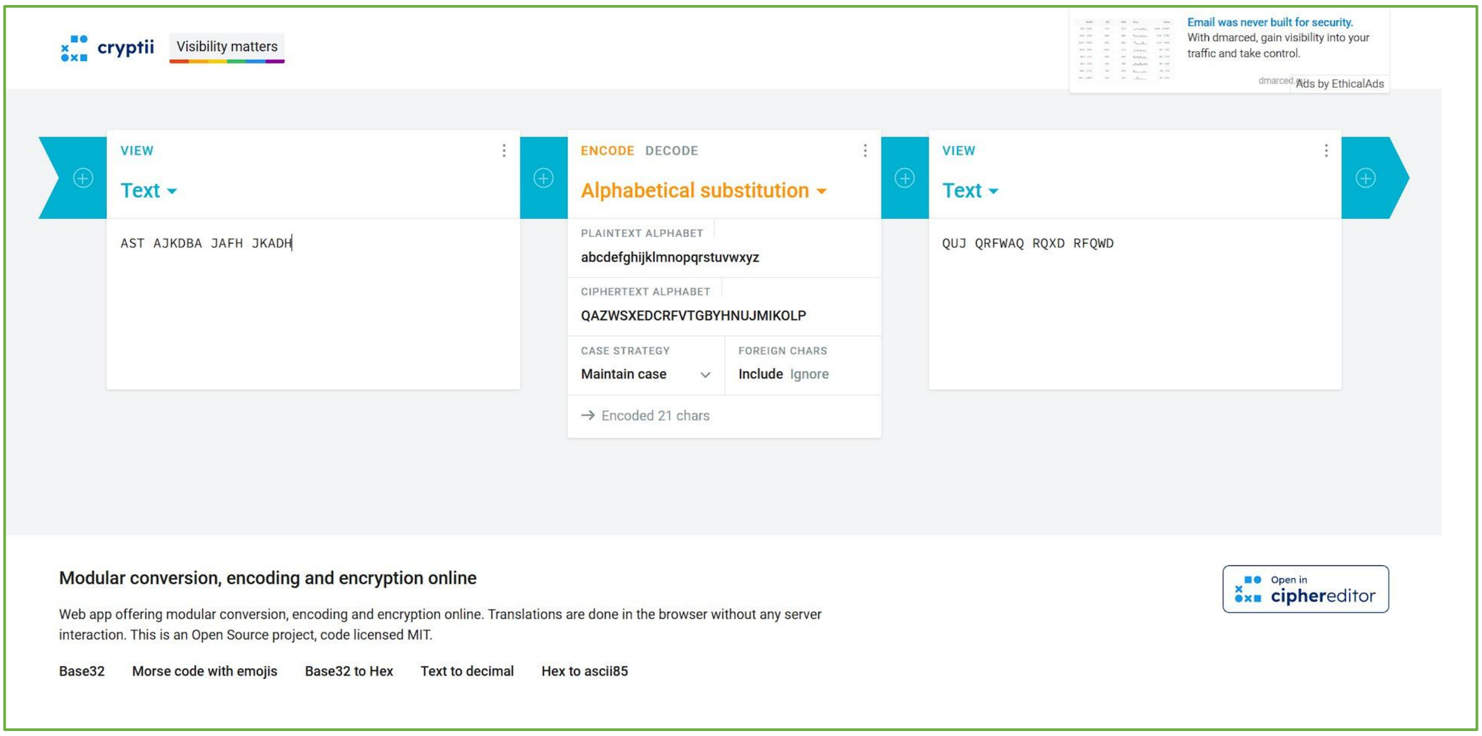
```
DSVVB, TL GQTS CU QGMNQE
```

Modular conversion, encoding and encryption online

Web app offering modular conversion, encoding and encryption online. Translations are done in the browser without any server interaction. This is an Open Source project, code licensed MIT.

Base32 Morse code with emojis Base32 to Hex Text to decimal Hex to ascii85

Open in ciphereditor



cryptii Visibility matters

Email was never built for security. With dmarced, gain visibility into your traffic and take control. Ads by EthicalAds

VIEW Text

```
AST AJKDBA JAFH JKADH
```

ENCODE DECODE
Alphabetical substitution
PLAINTEXT ALPHABET: abcdefghijklmnopqrstuvwxyz
CIPHERTEXT ALPHABET: QAZWSXEDCRFVTGBYHNUJMIKOLP
CASE STRATEGY: Maintain case
FOREIGN CHARS: Include Ignore
→ Encoded 21 chars

VIEW Text

```
QUJ QRFWAQ RQXD RFQWD
```

Modular conversion, encoding and encryption online

Web app offering modular conversion, encoding and encryption online. Translations are done in the browser without any server interaction. This is an Open Source project, code licensed MIT.

Base32 Morse code with emojis Base32 to Hex Text to decimal Hex to ascii85

Open in ciphereditor

PRACTICAL – 3:

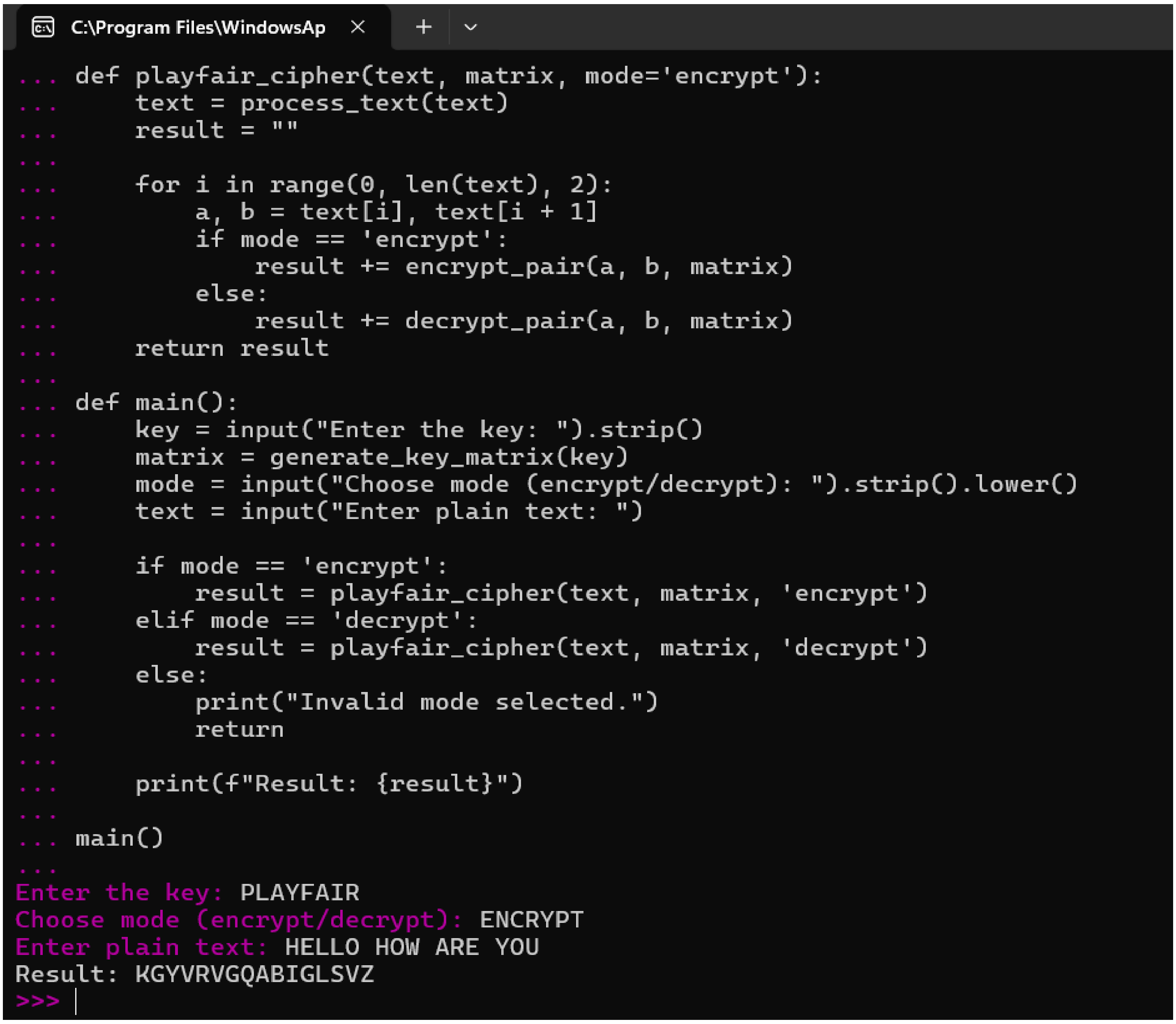
Aim: Implement Playfair cipher encryption-decryption.

Description: The Playfair cipher encrypts pairs of letters (digraphs), instead of single letters as in the simple substitution cipher. It uses a 5x5 matrix of letters constructed using a keyword. The same letter cannot appear twice in the matrix. Typically, 'J' is combined with 'I'. Rules for encryption and decryption depend on the positions of the letters in the matrix.

CODE:

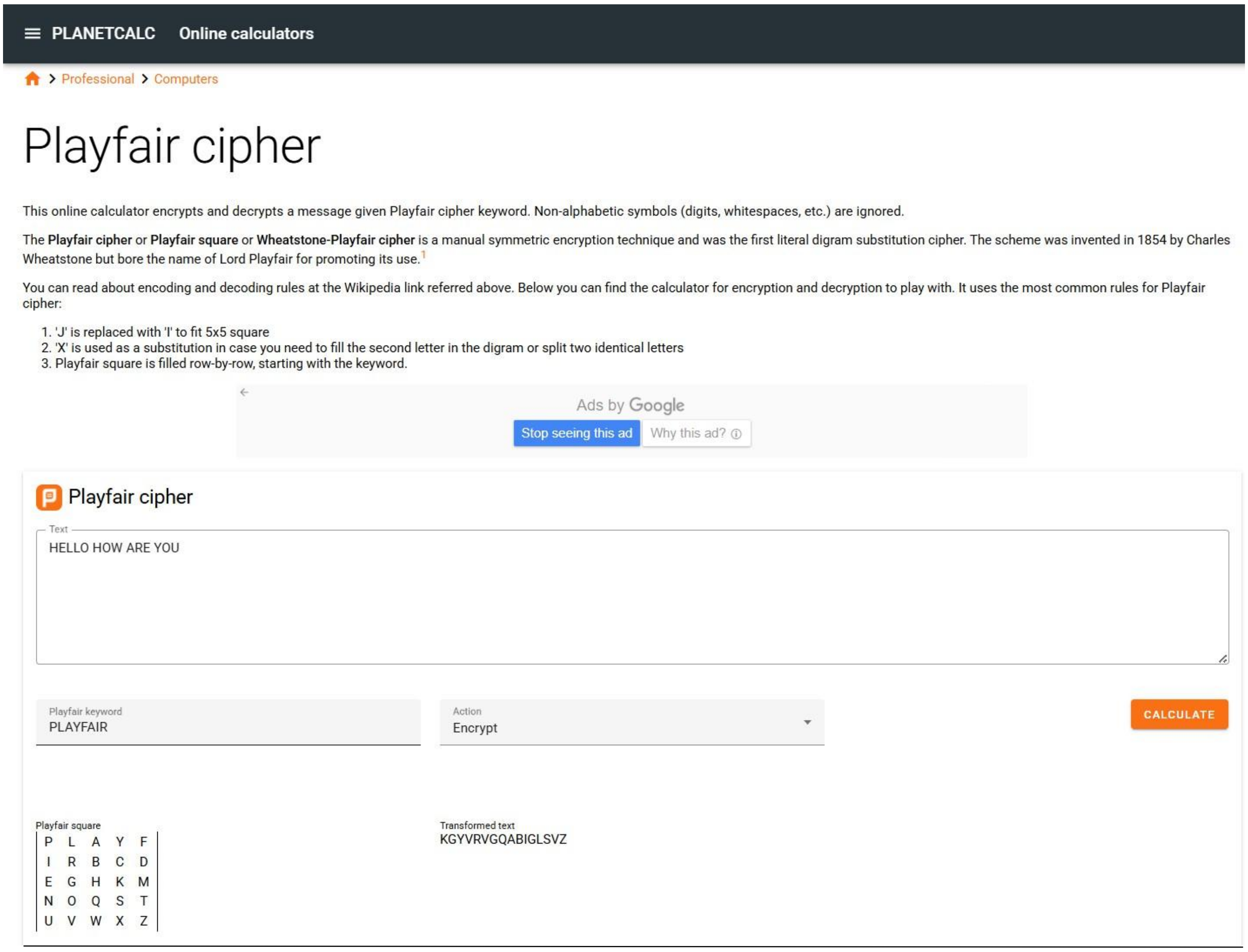
```
def generate_key_matrix(key):
    key = key.upper().replace("J", "I")
    key_matrix = []
    used = set()

    for char in key:
        if char not in used and char.isalpha():
            used.add(char)
            key_matrix.append(char)

    for char in "ABCDEFGHIKLMNOPQRSTUVWXYZ":
        if char not in used:
            used.add(char)
            key_matrix.append(char)

    return [key_matrix[i:i + 5] for i in range(0, 25, 5)]

def find_position(matrix, char):
    for i in range(5):
        for j in range(5):
            if matrix[i][j] == char:
                return i, j
    return None, None

def process_text(text):
    text = text.upper().replace("J", "I")
    i = 0
    processed = ""

    while i < len(text):
        a = text[i]
        b = text[i + 1] if i + 1 < len(text) else "X"
        if a == b:
            processed += a + "X"
            i += 1
        else:
            processed += a + b
            i += 2

    if len(processed) % 2 != 0:
        processed += "X"
    return processed

def encrypt_pair(a, b, matrix):
    row1, col1 = find_position(matrix, a)
    row2, col2 = find_position(matrix, b)

    if row1 == row2:
        return matrix[row1][(col1 + 1) % 5] + matrix[row2][(col2 + 1) % 5]
    elif col1 == col2:
        return matrix[(row1 + 1) % 5][col1] + matrix[(row2 + 1) % 5][col2]
    else:
        return matrix[row1][col2] + matrix[row2][col1]

def decrypt_pair(a, b, matrix):
    row1, col1 = find_position(matrix, a)
    row2, col2 = find_position(matrix, b)

    if row1 == row2:
        return matrix[row1][(col1 - 1) % 5] + matrix[row2][(col2 - 1) % 5]
    elif col1 == col2:
        return matrix[(row1 - 1) % 5][col1] + matrix[(row2 - 1) % 5][col2]
    else:
        return matrix[row1][col2] + matrix[row2][col1]

def playfair_cipher(text, matrix, mode='encrypt'):
    text = process_text(text)
    result = ""

    for i in range(0, len(text), 2):
        a, b = text[i], text[i + 1]
        if mode == 'encrypt':
            result += encrypt_pair(a, b, matrix)
        else:
            result += decrypt_pair(a, b, matrix)
    return result

def main():
    key = input("Enter the key: ")
    matrix = generate_key_matrix(key)
    mode = input("Choose mode (encrypt/decrypt): ").lower()
    text = input("Enter text: ")

    if mode == 'encrypt':
        result = playfair_cipher(text, matrix, 'encrypt')
    else:
        result = playfair_cipher(text, matrix, 'decrypt')

    print(f"Result: {result}")

main()
```

OUTPUT:



```
... def playfair_cipher(text, matrix, mode='encrypt'):
...     text = process_text(text)
...     result = ""
...
...     for i in range(0, len(text), 2):
...         a, b = text[i], text[i + 1]
...         if mode == 'encrypt':
...             result += encrypt_pair(a, b, matrix)
...         else:
...             result += decrypt_pair(a, b, matrix)
...     return result
...
... def main():
...     key = input("Enter the key: ").strip()
...     matrix = generate_key_matrix(key)
...     mode = input("Choose mode (encrypt/decrypt): ").strip().lower()
...     text = input("Enter plain text: ")
...
...     if mode == 'encrypt':
...         result = playfair_cipher(text, matrix, 'encrypt')
...     elif mode == 'decrypt':
...         result = playfair_cipher(text, matrix, 'decrypt')
...     else:
...         print("Invalid mode selected.")
...         return
...
...     print(f"Result: {result}")
...
... main()
...
Enter the key: PLAYFAIR
Choose mode (encrypt/decrypt): ENCRYPT
Enter plain text: HELLO HOW ARE YOU
Result: KGYVRVGQABIGLSVZ
>>> |
```

PLANETCALC.



PLANETCALC Online calculators

Professional > Computers

Playfair cipher

This online calculator encrypts and decrypts a message given Playfair cipher keyword. Non-alphabetic symbols (digits, whitespaces, etc.) are ignored.

The **Playfair cipher** or **Playfair square** or **Wheatstone-Playfair cipher** is a manual symmetric encryption technique and was the first literal digram substitution cipher. The scheme was invented in 1854 by Charles Wheatstone but bore the name of Lord Playfair for promoting its use.[1]

You can read about encoding and decoding rules at the Wikipedia link referred above. Below you can find the calculator for encryption and decryption to play with. It uses the most common rules for Playfair cipher:

1. 'J' is replaced with 'I' to fit 5x5 square
2. 'X' is used as a substitution in case you need to fill the second letter in the digram or split two identical letters
3. Playfair square is filled row-by-row, starting with the keyword.

Playfair cipher

Text: HELLO HOW ARE YOU

Playfair keyword: PLAYFAIR

Action: Encrypt

Playfair square

P	L	A	Y	F
I	R	B	C	D
E	G	H	K	M
N	O	Q	S	T
U	V	W	X	Z

Transformed text
KGYVRVGQABIGLSVZ

PRACTICAL – 4:

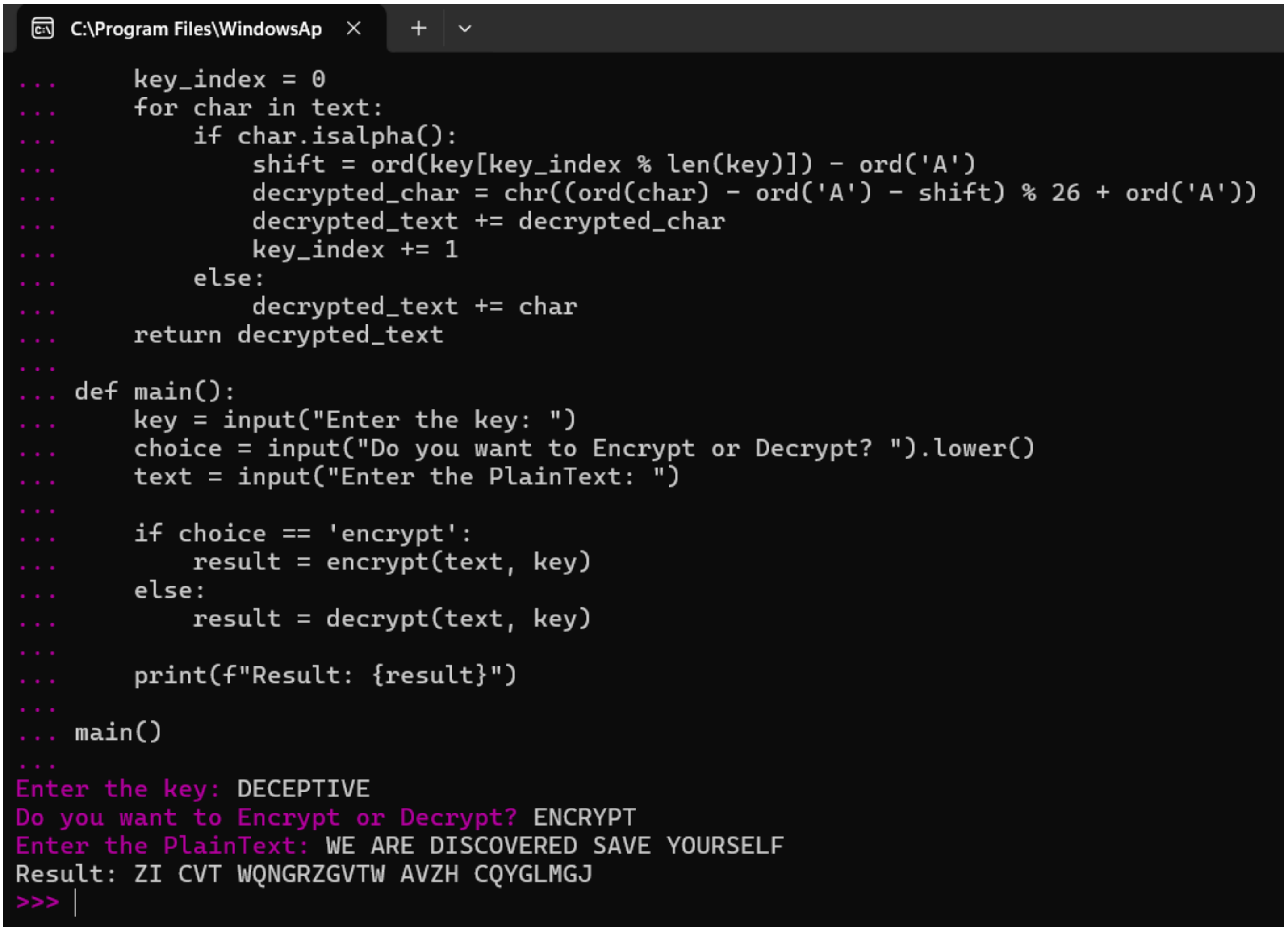
Aim: Implement Polyalphabetic cipher encryption-decryption (Vigenere).

Description: Polyalphabetic cipher uses multiple substitution alphabets to encrypt the data. The most famous example is the Vigenère cipher. This method uses a keyword where each letter in the keyword refers to a different Caesar cipher shift.

CODE:

```
def encrypt(text, key):
    encrypted_text = ""
    key = key.upper()
    text = text.upper()

    key_index = 0
    for char in text:
        if char.isalpha():
            shift = ord(key[key_index % len(key)]) - ord('A')
            encrypted_char = chr((ord(char) - ord('A') + shift) % 26 + ord('A'))
            encrypted_text += encrypted_char
            key_index += 1
        else:
            encrypted_text += char
    return encrypted_text

def decrypt(text, key):
    decrypted_text = ""
    key = key.upper()
    text = text.upper()

    key_index = 0
    for char in text:
        if char.isalpha():
            shift = ord(key[key_index % len(key)]) - ord('A')
            decrypted_char = chr((ord(char) - ord('A') - shift) % 26 + ord('A'))
            decrypted_text += decrypted_char
            key_index += 1
        else:
            decrypted_text += char
    return decrypted_text

def main():
    key = input("Enter the key: ")
    choice = input("Do you want to Encrypt or Decrypt? ").lower()
    text = input("Enter the message: ")
    if choice == 'encrypt':
        result = encrypt(text, key)
    else:
        result = decrypt(text, key)

    print(f"Result: {result}")

main()
```

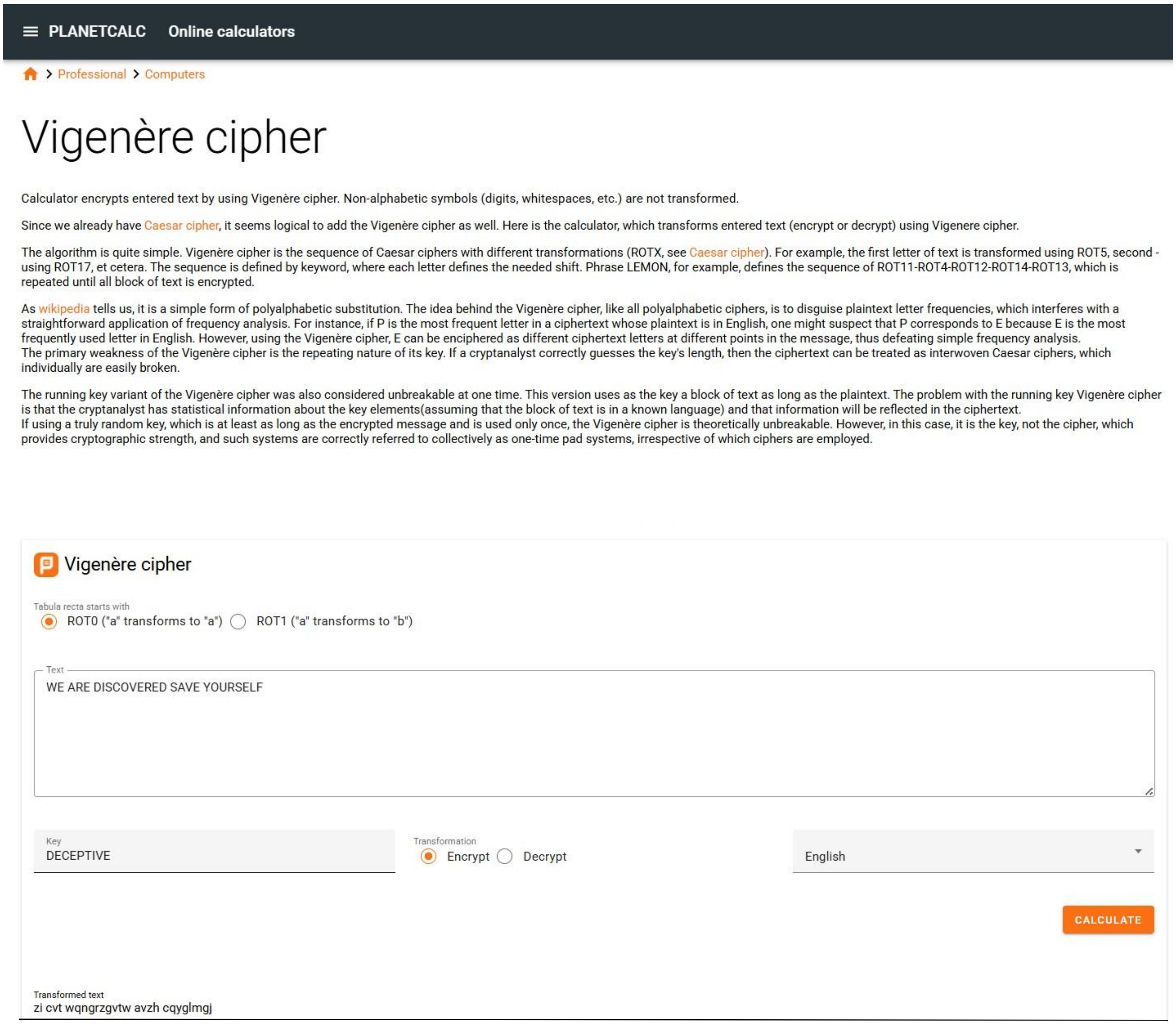
OUTPUT:



C:\Program Files\WindowsAp

```
...     key_index = 0
...     for char in text:
...         if char.isalpha():
...             shift = ord(key[key_index % len(key)]) - ord('A')
...             decrypted_char = chr((ord(char) - ord('A') - shift) % 26 + ord('A'))
...             decrypted_text += decrypted_char
...             key_index += 1
...         else:
...             decrypted_text += char
...     return decrypted_text
...
... def main():
...     key = input("Enter the key: ")
...     choice = input("Do you want to Encrypt or Decrypt? ").lower()
...     text = input("Enter the PlainText: ")
...
...     if choice == 'encrypt':
...         result = encrypt(text, key)
...     else:
...         result = decrypt(text, key)
...
...     print(f"Result: {result}")
...
... main()
...
Enter the key: DECEPTIVE
Do you want to Encrypt or Decrypt? ENCRYPT
Enter the PlainText: WE ARE DISCOVERED SAVE YOURSELF
Result: ZI CVT WQNGRZGVTW AVZH CQYGLMGJ
>>> |
```

PLANETCALC.



PLANETCALC Online calculators

Professional > Computers

Vigenère cipher

Calculator encrypts entered text by using Vigenère cipher. Non-alphabetic symbols (digits, whitespaces, etc.) are not transformed.

Since we already have Caesar cipher, it seems logical to add the Vigenère cipher as well. Here is the calculator, which transforms entered text (encrypt or decrypt) using Vigenere cipher.

The algorithm is quite simple. Vigenère cipher is the sequence of Caesar ciphers with different transformations (ROTX, see Caesar cipher). For example, the first letter of text is transformed using ROT5, second - using ROT17, et cetera. The sequence is defined by keyword, where each letter defines the needed shift. Phrase LEMON, for example, defines the sequence of ROT11-ROT4-ROT12-ROT14-ROT13, which is repeated until all block of text is encrypted.

As wikipedia tells us, it is a simple form of polyalphabetic substitution. The idea behind the Vigenère cipher, like all polyalphabetic ciphers, is to disguise plaintext letter frequencies, which interferes with a straightforward application of frequency analysis. For instance, if P is the most frequent letter in a ciphertext whose plaintext is in English, one might suspect that P corresponds to E because E is the most frequently used letter in English. However, using the Vigenère cipher, E can be enciphered as different ciphertext letters at different points in the message, thus defeating simple frequency analysis.
The primary weakness of the Vigenère cipher is the repeating nature of its key. If a cryptanalyst correctly guesses the key's length, then the ciphertext can be treated as interwoven Caesar ciphers, which individually are easily broken.

The running key variant of the Vigenère cipher was also considered unbreakable at one time. This version uses as the key a block of text as long as the plaintext. The problem with the running key Vigenère cipher is that the cryptanalyst has statistical information about the key elements(assuming that the block of text is in a known language) and that information will be reflected in the ciphertext.
If using a truly random key, which is at least as long as the encrypted message and is used only once, the Vigenère cipher is theoretically unbreakable. However, in this case, it is the key, not the cipher, which provides cryptographic strength, and such systems are correctly referred to collectively as one-time pad systems, irrespective of which ciphers are employed.

Vigenère cipher

Tabula recta starts with
[✓ ROT0 ("a" transforms to "a")] ROT1 ("a" transforms to "b")

Text
WE ARE DISCOVERED SAVE YOURSELF

Key
DECEPTIVE

Transformation
[✓ Encrypt] Decrypt

English

CALCULATE

Transformed text
zi cvt wqngrzgvtw avzh cqyglmgj

Practical5

Aim: Implement Hill cipher encryption-decryption.

Theory:

TheHillcipherisa**symmetrickeysubstitutioncipher**thatencryptsblocksofplaintextusing linear transformations based on matrix multiplication.

- **Block Cipher**: Operates on*n*-lengthblocksoftextinsteadofsinglecharacters.
- **Key**: A square matrix (size*nxn*) ofnumbersmodulo 26.
- **Encryption**:Usesmatrixmultiplicationoftheplaintextvectorwiththekeymatrix.
- **Decryption**:Requirestheinverseofthekeymatrixmodulo26.

EncryptionProcess

1. **KeyMatrix Setup**:

   o Choosean*n*×*n*matrixwith integersmodulo26 (A=0,B=1,..., Z=25). o Ensure the matrix is invertible modulo 26 (i.e., its determinant has a modular inverse mod 26).

2. **Plaintext Preparation**:

   o Converteachlettertoanumber(A=0to Z=25).

   o Groupplaintext intoblocksofsize *n*. Ifneeded, padwith filler characters.

3. **MatrixMultiplication**:

   o Multiplyeachblockasacolumnvectorbythekeymatrix.

   o Takemodulo26oftheresult.

4. **Conversionto Ciphertext**:

 Convertresultingnumericvaluesbacktoletters.

DecryptionProcess

1. **InverseKeyMatrix**:

o Computeinverseofthekeymatrixmodulo26.Thisrequiresmodulararithmetic and matrix operations.

2. **CiphertextProcessing**: o Convertciphertextintonumber blocks.

3. **DecryptBlocks**: o Multiplyeach blockby theinversekeymatrix modulo 26.
4. **Convertto Plaintext**:

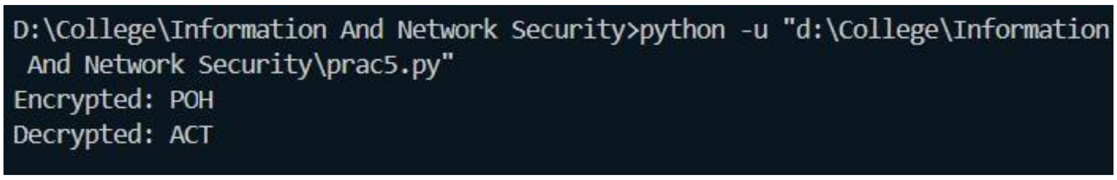
 Mapnumericresultsbackto letters.

Code:

```
importnumpy asnp fromsympyimport Matrix
fromnumpy.linalgimportLinAlgError
defletter_to_num(letter):
 returnord(letter.upper())-ord('A') def
num_to_letter(num):
 returnchr((num%26)+ord('A')) def
text_to_nums(text):
return[letter_to_num(c)forcintextifc.isalpha()]

defnums_to_text(nums):
return''.join([num_to_letter(n)for nin nums])

defpad_text(text,block_size): pad_len=block_size- len(text)%block_size
returntext+'X'*pad_len ifpad_len!=block_sizeelse text

defencrypt(text,key): size=key.shape[0]
text=pad_text(text.upper().replace("",""),size)
nums = text_to_nums(text)
cipher_nums=[]
```

```
for i in range(0, len(nums), size): block=np.array(nums[i:i+size])
enc_block=np.dot(key,block)%26 cipher_nums.extend(enc_block)
returnnums_to_text(cipher_nums)

def decrypt(cipher, key): size=key.shape[0] try:

sym_key=Matrix(key.tolist()) inv_key=np.array(sym_key.inv_mod(26)).astype(int)

except (ValueError, LinAlgError):

return"Keymatrixisnotinvertiblemod 26!"


nums=text_to_nums(cipher)

plain_nums = []


for i in range(0, len(nums), size): block=np.array(nums[i:i+size])
dec_block=np.dot(inv_key,block)%26 plain_nums.extend(dec_block)
returnnums_to_text(plain_nums)

if name______== "main":

key_3x3=np.array([[6,24, 1],

[13,16, 10],

[20,17, 15]])


message="ACT"

cipher=encrypt(message,key_3x3) print("Encrypted:", cipher)


decrypted=decrypt(cipher,key_3x3) print("Decrypted:", decrypted)
```

Output:



```
D:\College\Information And Network Security>python -u "d:\College\Information
 And Network Security\prac5.py"
Encrypted: POH
Decrypted: ACT
```

CryptanalysisofillCipherCryptanalysis:

1. **TheHillcipherisvulnerabletoa *known-plaintext attack*.**

   o **If $n \times n$ key size is used, acquiring n^2 plaintext–ciphertext character pairs can compromise the key.**

2. **Theattackmethodinvolveslinearalgebraovermodulararithmetic.**

   o **Constructmatricesofplaintextblocksandciphertextblocks.**

   o **Solveforthekeyusingmatrixinversionandmultiplicationmodulo 26.**

3. **KeyRecoveryEquation: K=C .P^-1(mod 26) Where:**

   o **K=Key matrix o C=Ciphertext matrix**

   o **P^{-1}=Inverseofplaintext matrixmodulo 26**

4. **Thecipherlacksresistancetostatistical analysis.**

   o **Becauseit'sdeterministic,patternsinciphertextcloselyreflectpatternsin plaintext blocks.**

5. **Poorkeymatrixchoicecancreatevulnerabilities.**

   o **Anon-invertiblekeymatrix(mod26)breaksthecipher's reversibility. o Ifdeterminantofkey≡0(mod26)orhasnomodularinverse→cipherbecomes unusable. 6. Hillcipherdoesn'tprovidediffusionandconfusion.**

   o **Changesininputdon'tripplefar;outputchangesarelocalizedwithinthebl ock.**

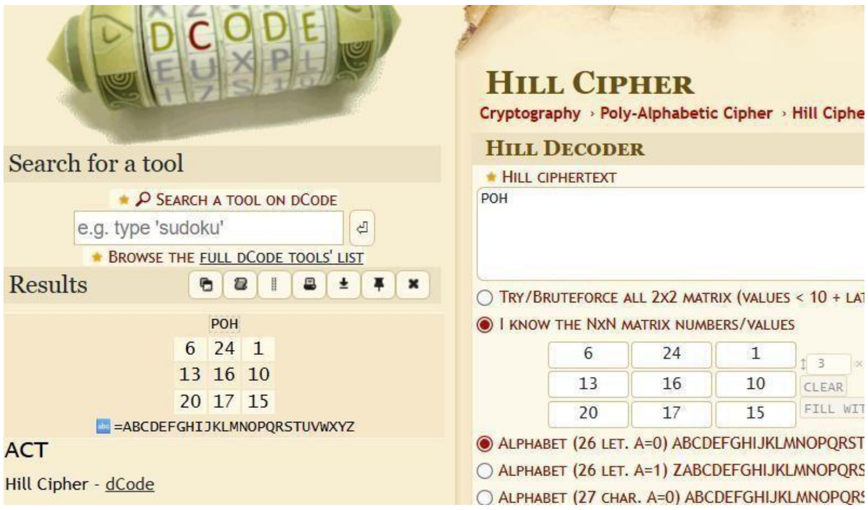
7. **Attackfeasibilityincreaseswithblocksizeandplaintextvolume.**

o Largerblocksneedmoreplaintext–ciphertextpairsbutmakestatisticalrecovery easier if data is available.

8. **Nopaddingstandardmakesplaintextrecoveryeveneasier.**

   Attackerscanguesspaddingschemesorinferprobabletextstructure.

Tool:



Results

POH

6	24	1
13	16	10
20	17	15

=ABCDEFGHIJKLMNOPQRSTUVWXYZ

ACT

Hill Cipher - dCode

HILL CIPHER

Cryptography › Poly-Alphabetic Cipher › Hill Ciphe

HILL DECODER

HILL CIPHERTEXT

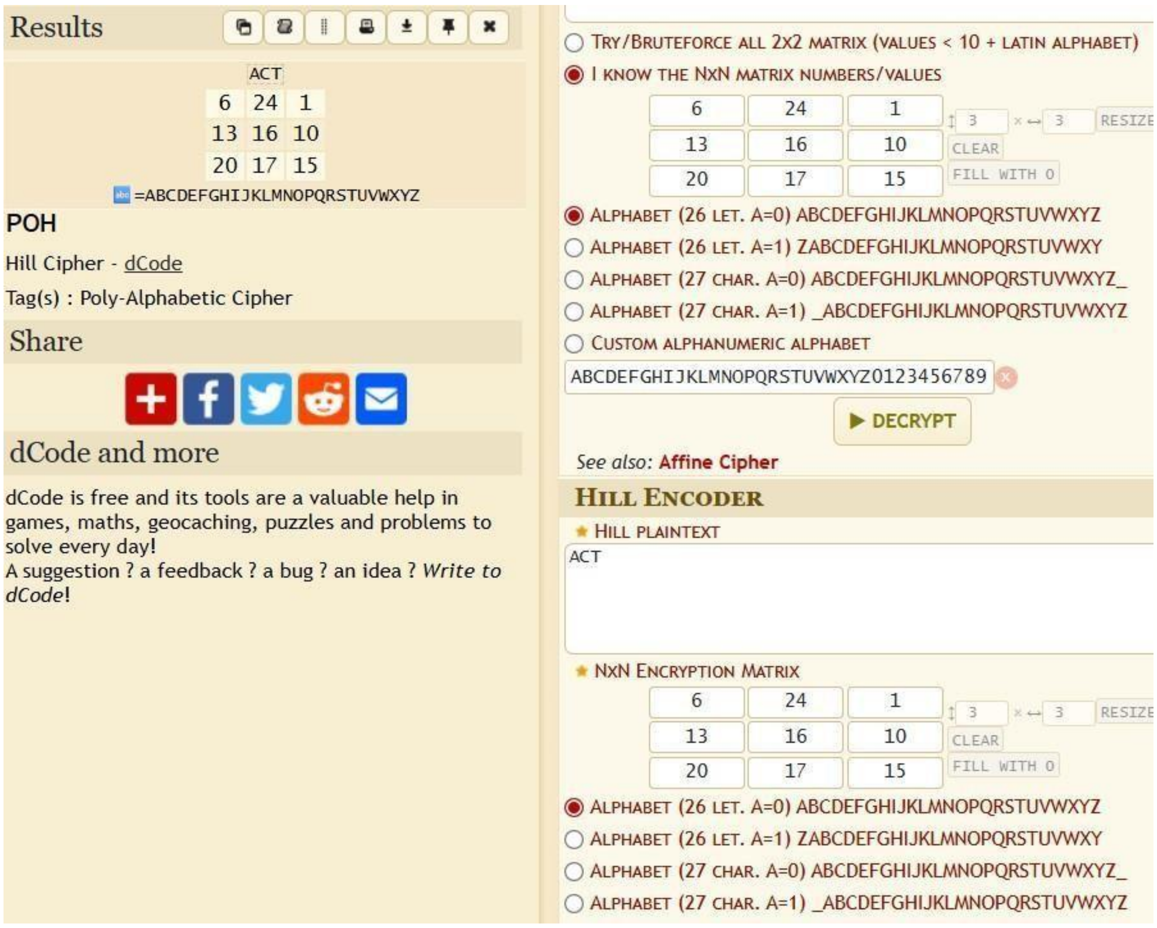
POH

○ TRY/BRUTEFORCE ALL 2X2 MATRIX (VALUES < 10 + LAT
[✓ I KNOW THE NXN MATRIX NUMBERS/VALUES]

6	24	1
13	16	10
20	17	15

3 × 3 CLEAR FILL WIT

[✓ ALPHABET (26 LET. A=0) ABCDEFGHIJKLMNOPQRST]
○ ALPHABET (26 LET. A=1) ZABCDEFGHIJKLMNOPQRS
○ ALPHABET (27 CHAR. A=0) ABCDEFGHIJKLMNOPQRS


Results

ACT

6	24	1
13	16	10
20	17	15

=ABCDEFGHIJKLMNOPQRSTUVWXYZ

POH

Hill Cipher - dCode

Tag(s) : Poly-Alphabetic Cipher

dCode and more

dCode is free and its tools are a valuable help in games, maths, geocaching, puzzles and problems to solve every day!
A suggestion ? a feedback ? a bug ? an idea ? *Write to dCode*!

○ TRY/BRUTEFORCE ALL 2X2 MATRIX (VALUES < 10 + LATIN ALPHABET)
[✓ I KNOW THE NXN MATRIX NUMBERS/VALUES]

6	24	1
13	16	10
20	17	15

3 × 3 RESIZE CLEAR FILL WITH 0

[✓ ALPHABET (26 LET. A=0) ABCDEFGHIJKLMNOPQRSTUVWXYZ]
○ ALPHABET (26 LET. A=1) ZABCDEFGHIJKLMNOPQRSTUVWXY
○ ALPHABET (27 CHAR. A=0) ABCDEFGHIJKLMNOPQRSTUVWXYZ_
○ ALPHABET (27 CHAR. A=1) _ABCDEFGHIJKLMNOPQRSTUVWXYZ
○ CUSTOM ALPHANUMERIC ALPHABET
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789

▶ DECRYPT

See also: Affine Cipher

HILL ENCODER

HILL PLAINTEXT

ACT

NXN ENCRYPTION MATRIX

6	24	1
13	16	10
20	17	15

3 × 3 RESIZE CLEAR FILL WITH 0

[✓ ALPHABET (26 LET. A=0) ABCDEFGHIJKLMNOPQRSTUVWXYZ]
○ ALPHABET (26 LET. A=1) ZABCDEFGHIJKLMNOPQRSTUVWXY
○ ALPHABET (27 CHAR. A=0) ABCDEFGHIJKLMNOPQRSTUVWXYZ_
○ ALPHABET (27 CHAR. A=1) _ABCDEFGHIJKLMNOPQRSTUVWXYZ

PRACTICAL: 06

AIM: Implement Simple Transposition encryption-decryption.

Theory:

The **Simple Transposition Cipher** is a classical cryptographic technique in which the positions of the characters in the plaintext are rearranged according to a specific rule or key to form the ciphertext. Unlike substitution ciphers, which replace letters with other letters or symbols, the transposition cipher preserves the original characters but changes their order, making the plaintext unreadable without the correct arrangement.

One common form of this cipher is the **Columnar Transposition Cipher**, where the plaintext is written row by row into a grid of fixed column size determined by the key, and then read column by column in a specific order. For example, if the key is the number of columns, the letters are written across rows and read down the columns to produce the ciphertext.

Decryption works by reversing the process — the ciphertext is written into the grid column by column according to the key, and then read row by row to reconstruct the original message. This method does not alter the actual characters, making it resistant to frequency analysis, but it can still be broken using anagramming techniques or brute force key searches if the key is small.

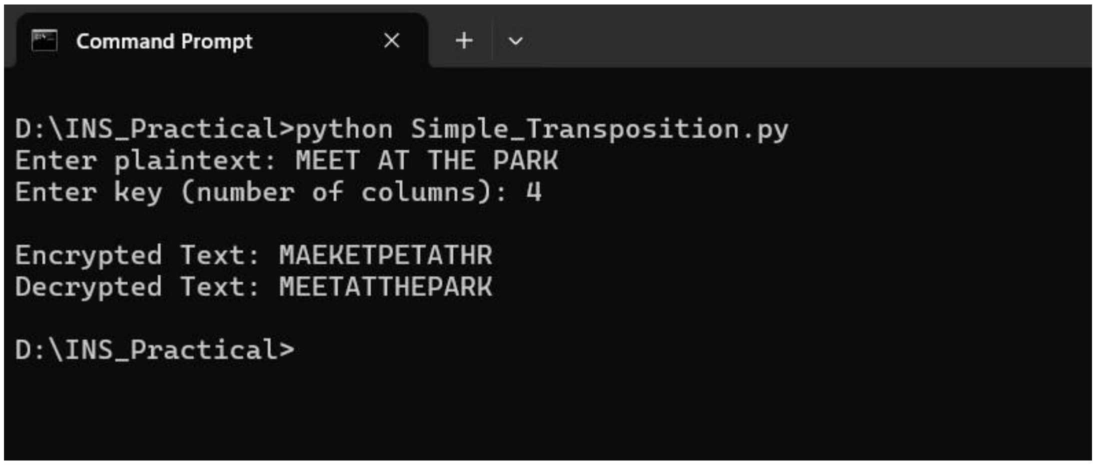
The simplicity and historical importance of the transposition cipher make it a good introductory example for understanding the principles of permutation in cryptography, as opposed to substitution. It also demonstrates the fundamental cryptographic principle that secrecy can be achieved through both **confusion** (changing symbols) and **permutation** (rearranging symbols).

Program:

```
import math

def encrypt_transposition(plaintext, key):
    plaintext = plaintext.replace(" ", "").upper()
    ciphertext = [''] * key

    for col in range(key):
        pointer = col
        while pointer < len(plaintext):
            ciphertext[col] += plaintext[pointer]
            pointer += key
    return ''.join(ciphertext)

def decrypt_transposition(ciphertext, key): num_of_rows
    = math.ceil(len(ciphertext) / key)
    num_of_shaded_boxes = (num_of_rows * key) - len(ciphertext)

    plaintext = [''] * num_of_rows
    col = 0
    row = 0

    for symbol in ciphertext:
        plaintext[row] += symbol
```

```
            row += 1

        if (row == num_of_rows) or (row == num_of_rows - 1 and col >= key -
num_of_shaded_boxes):

            row = 0

            col += 1


    return ''.join(plaintext)


if _name___== "_main___":

    plaintext = input("Enter plaintext: ")

    key = int(input("Enter key (number of columns): "))


    encrypted = encrypt_transposition(plaintext, key)

    decrypted = decrypt_transposition(encrypted, key)


    print("\nEncrypted Text:", encrypted)

    print("Decrypted Text:", decrypted)
```

OUTPUT:



```
D:\INS_Practical>python Simple_Transposition.py
Enter plaintext: MEET AT THE PARK
Enter key (number of columns): 4

Encrypted Text: MAEKETPETATHR
Decrypted Text: MEETATTHEPARK

D:\INS_Practical>
```

PRACTICAL: 07

AIM: Implement Diffie-Hellman Key exchange Method.

Theory:

The **Diffie–Hellman Key Exchange** is a cryptographic protocol that allows two parties to establish a shared secret key over an insecure communication channel without having to send the key itself. Proposed by Whitfield Diffie and Martin Hellman in 1976, it was the first published method for securely exchanging cryptographic keys and laid the foundation for modern public-key cryptography.

The method relies on the mathematical difficulty of the **Discrete Logarithm Problem**. Two large numbers are publicly agreed upon: a **prime modulus (p)** and a **primitive root modulo p (g)**, also called the generator. Each party selects a private key (a secret number) and computes a corresponding public value by raising the generator to the power of their private key modulo p. These public values are exchanged openly over the insecure channel.

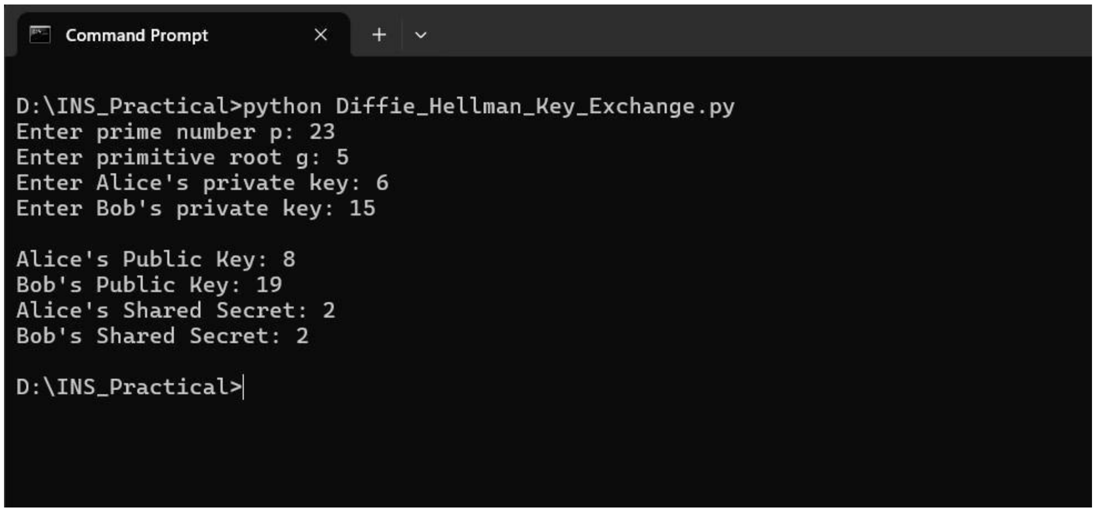
After receiving the other party's public value, each participant raises it to the power of their own private key modulo p. Due to the properties of modular exponentiation, both parties compute the same result, which becomes their shared secret key. This key can then be used for symmetric encryption.

While the Diffie–Hellman method is secure against passive eavesdropping if large enough parameters are chosen, it is vulnerable to **Man-in-the-Middle attacks** unless combined with authentication mechanisms such as digital signatures or certificates. Variants like Elliptic Curve Diffie–Hellman (ECDH) provide the same functionality with smaller key sizes and improved efficiency.

Program:

```
def diffie_hellman(p, g, private_key_a, private_key_b):

    public_key_a = pow(g, private_key_a, p)
    public_key_b = pow(g, private_key_b, p)
    shared_secret_a = pow(public_key_b, private_key_a, p)
    shared_secret_b = pow(public_key_a, private_key_b, p)

    return public_key_a, public_key_b, shared_secret_a, shared_secret_b

if __name___== "__main__":

    p = int(input("Enter prime number p: "))
    g = int(input("Enter primitive root g: "))
    private_key_a = int(input("Enter Alice's private key: "))
    private_key_b = int(input("Enter Bob's private key: "))

    public_a, public_b, secret_a, secret_b = diffie_hellman(p, g, private_key_a,
private_key_b)

    print("\nAlice's Public Key:", public_a)
    print("Bob's Public Key:", public_b)
    print("Alice's Shared Secret:", secret_a)
    print("Bob's Shared Secret:", secret_b)
```

OUTPUT:



Command Prompt

```
D:\INS_Practical>python Diffie_Hellman_Key_Exchange.py
Enter prime number p: 23
Enter primitive root g: 5
Enter Alice's private key: 6
Enter Bob's private key: 15

Alice's Public Key: 8
Bob's Public Key: 19
Alice's Shared Secret: 2
Bob's Shared Secret: 2

D:\INS_Practical>
```

PRACTICAL: 08

AIM: Implement One time pad encryption-decryption.

Theory:

The **One-Time Pad (OTP)** is a symmetric encryption technique that offers **perfect secrecy**, meaning the ciphertext reveals no information about the plaintext without the key. It was first described by Gilbert Vernam in 1917 and later mathematically proven secure by Claude Shannon. The method works by combining each character (or bit) of the plaintext with a corresponding character (or bit) from a **truly random key** that is the same length as the plaintext. The encryption is usually performed using the **XOR operation** in binary form or by modular addition in alphabetic form.

For encryption in alphabetic form, each letter of the plaintext and key is converted to a number (A=0, B=1, …, Z=25). The ciphertext is generated by adding the plaintext and key values modulo 26. Decryption is done by subtracting the key values from the ciphertext values modulo 26, recovering the original message.

The security of OTP comes from two conditions:

1. The key must be **truly random** (not generated by a predictable algorithm).
2. The key must **never be reused** for another message.

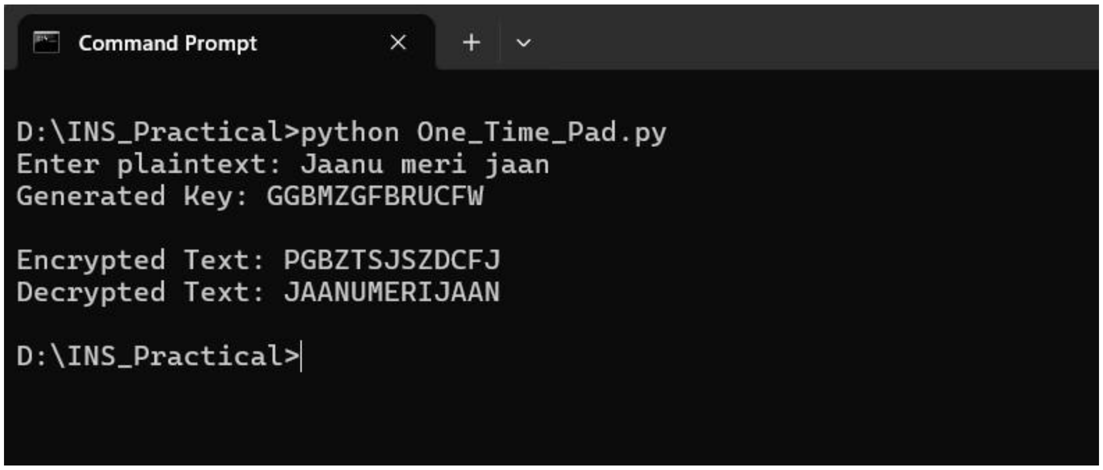
If either condition is violated, the cipher becomes vulnerable to cryptanalysis. In practice, OTP is rarely used for large-scale communication due to difficulties in generating, distributing, and securely storing long, truly random keys. However, it remains an important concept in cryptography because it demonstrates the upper limit of encryption security and influences the design of modern cryptographic systems.

Program:

```
import random
import string


def generate_key(length):
    return ''.join(random.choice(string.ascii_uppercase) for _ in range(length))


def encrypt_otp(plaintext, key):
    plaintext = plaintext.replace(" ", "").upper()
    key = key.upper()
    ciphertext = ""
    for p, k in zip(plaintext, key):
        encrypted_char = chr(((ord(p) - ord('A')) + (ord(k) - ord('A'))) % 26 +
ord('A'))
        ciphertext += encrypted_char
    return ciphertext


def decrypt_otp(ciphertext, key):
    ciphertext = ciphertext.upper()
    key = key.upper()
    plaintext = ""
    for c, k in zip(ciphertext, key):
        decrypted_char = chr(((ord(c) - ord('A')) - (ord(k) - ord('A'))) % 26 +
ord('A'))
```

```
        plaintext += decrypted_char
    return plaintext


if __name___== "__main__":
    plaintext = input("Enter plaintext: ").upper().replace(" ", "")
    key = generate_key(len(plaintext))
    print("Generated Key:", key)

    encrypted = encrypt_otp(plaintext, key)
    decrypted = decrypt_otp(encrypted, key)

    print("\nEncrypted Text:", encrypted)
    print("Decrypted Text:", decrypted)
```

OUTPUT:



Command Prompt

```
D:\INS_Practical>python One_Time_Pad.py
Enter plaintext: Jaanu meri jaan
Generated Key: GGBMZGFBRUCFW

Encrypted Text: PGBZTSJSZDCFJ
Decrypted Text: JAANUMERIJAAN

D:\INS_Practical>
```

PRACTICAL: 09

AIM: Implement RSA encryption-decryption algorithm.

Theory:

The RSA (Rivest–Shamir–Adleman) algorithm is one of the most widely used **public-key cryptographic systems**. It is an asymmetric encryption method, meaning it uses two keys: a **public key** for encryption and a **private key** for decryption. The algorithm is based on the mathematical difficulty of **prime factorization**. Specifically, while it is easy to multiply two large prime numbers, it is extremely difficult to factorize their product back into primes, which ensures security. RSA is extensively used in securing sensitive data, digital signatures, and establishing secure communication channels over the internet.

The RSA algorithm works in four main steps: **Key Generation, Key Distribution, Encryption, and Decryption**. In key generation, two large prime numbers are chosen, and their product forms the modulus n. The totient function φ(n) is computed, and then a public exponent e is chosen such that it is coprime with φ(n). The private key d is then calculated as the modular inverse of e modulo φ(n). Once the keys are generated, the **public key (e, n)** is shared openly, while the **private key (d, n)** is kept secret.

The encryption process involves converting plaintext into a numerical form and then applying the formula C = (P^e) mod n, where C is ciphertext and P is the plaintext integer. For decryption, the receiver uses the private key with the formula P = (C^d) mod n to retrieve the original plaintext. The correctness of RSA is guaranteed by Euler's theorem, which ensures that the decryption process reverses the encryption. This makes RSA reliable for both confidentiality and authenticity.

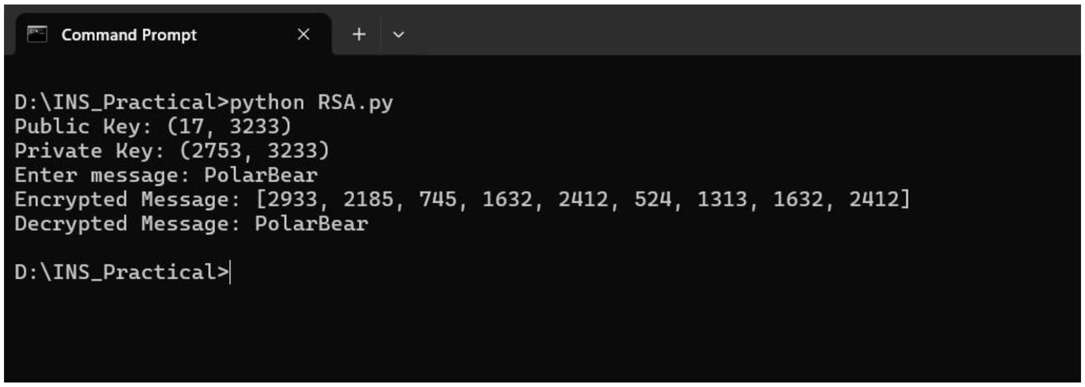
Another important aspect of RSA is its use in **digital signatures**. Instead of encrypting the message directly, the sender can sign the message using their private key, and the recipient can verify it using the sender's public key. This ensures **integrity and non- repudiation**, proving that the message has not been tampered with and verifying the sender's identity. Additionally, RSA can be combined with hash functions for efficient signature generation and verification, further improving performance.

Program:

```
import random

def gcd(a, b):
    while b != 0:
        a, b = b, a %
    b return a

def mod_inverse(e, phi):
    for d in range(3, phi):
        if (e * d) % phi == 1:
            return d
    return None

def generate_keys():
    p = 61
    q = 53
    n = p * q
    phi = (p - 1) * (q - 1)
    e = 17
    if gcd(e, phi) != 1:
        raise Exception("e and phi are not coprime")
    d = mod_inverse(e, phi)
    return ((e, n), (d, n))
```

```
def encrypt(public_key, plaintext):
    e, n = public_key
    ciphertext = [pow(ord(char), e, n) for char in plaintext]
    return ciphertext

def decrypt(private_key, ciphertext):
    d, n = private_key
    plaintext = ''.join([chr(pow(char, d, n)) for char in ciphertext])
    return plaintext

if __name___== "__main__":
    public_key, private_key = generate_keys()
    print("Public Key:", public_key)
    print("Private Key:", private_key)

    message = input("Enter message: ") encrypted_msg
    = encrypt(public_key, message) print("Encrypted
    Message:", encrypted_msg)

    decrypted_msg = decrypt(private_key, encrypted_msg)
    print("Decrypted Message:", decrypted_msg)
```

OUTPUT:



```
D:\INS_Practical>python RSA.py
Public Key: (17, 3233)
Private Key: (2753, 3233)
Enter message: PolarBear
Encrypted Message: [2933, 2185, 745, 1632, 2412, 524, 1313, 1632, 2412]
Decrypted Message: PolarBear

D:\INS_Practical>
```

PRACTICAL: 10

AIM: Demonstrate working of Digital Signature using Cryptool.

Theory:

A **Digital Signature** is a mathematical scheme that validates the authenticity and integrity of a digital message, document, or software. It is the digital equivalent of a handwritten signature or a stamped seal but much more secure. Digital signatures are widely used in cryptography to provide:

1. **Authentication** – Confirms the sender's identity.
2. **Integrity** – Ensures the message was not altered in transit.
3. **Non-repudiation** – The sender cannot deny sending the message later.

How Digital Signature Works:

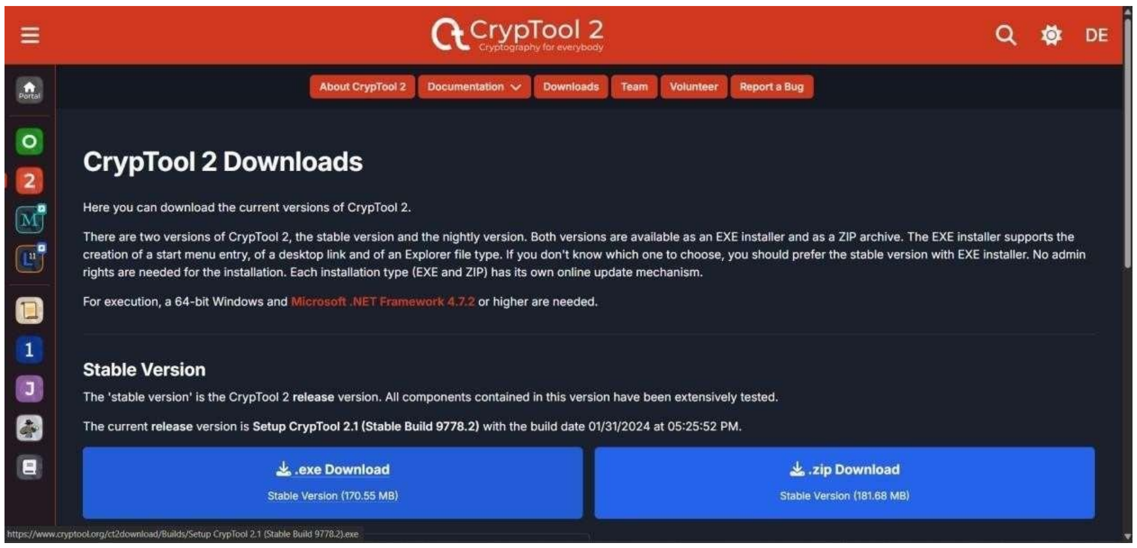
1. The sender generates a **hash** of the message using a hash function (e.g., SHA-256).
2. The hash value is **encrypted with the sender's private key** → this becomes the
digital signature.
3. The receiver decrypts the signature using the sender's **public key** to retrieve the hash value.
4. The receiver independently computes the hash of the received message.
5. If both hash values match → the message is authentic and untampered.

Digital signatures are commonly implemented with **RSA, DSA, or ECDSA algorithms**, along with hashing (SHA family).

Procedure:

Step 1: Download & Install CrypTool

1. Go to the official CrypTool website: https://www.cryptool.org/en/
2. Download **CrypTool 2** for Windows.



CrypTool 2 Downloads

Here you can download the current versions of CrypTool 2.

There are two versions of CrypTool 2, the stable version and the nightly version. Both versions are available as an EXE installer and as a ZIP archive. The EXE installer supports the creation of a start menu entry, of a desktop link and of an Explorer file type. If you don't know which one to choose, you should prefer the stable version with EXE installer. No admin rights are needed for the installation. Each installation type (EXE and ZIP) has its own online update mechanism.

For execution, a 64-bit Windows and Microsoft .NET Framework 4.7.2 or higher are needed.

Stable Version

The 'stable version' is the CrypTool 2 release version. All components contained in this version have been extensively tested.

The current release version is Setup CrypTool 2.1 (Stable Build 9778.2) with the build date 01/31/2024 at 05:25:52 PM.

.exe Download — Stable Version (170.55 MB)

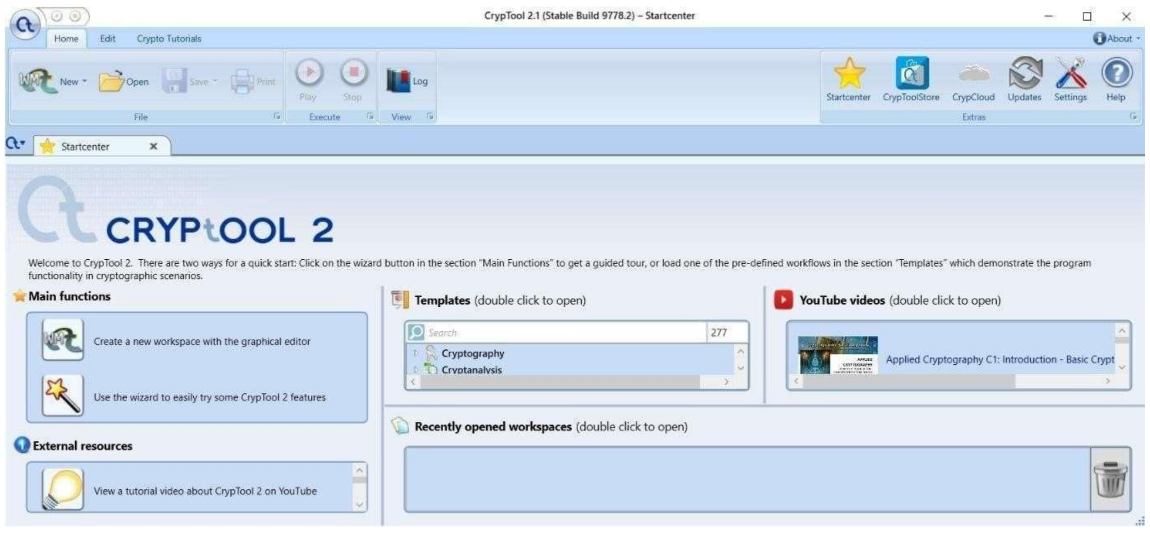
.zip Download — Stable Version (181.68 MB)

https://www.cryptool.org/ct2download/Builds/Setup CrypTool 2.1 (Stable Build 9778.2).exe

3. Install CrypTool by following the setup instructions.

Step 2: Launch CrypTool 2 Application

When you launch, you'll see the **Startcenter screen**.



Welcome to CrypTool 2. There are two ways for a quick start: Click on the wizard button in the section "Main Functions" to get a guided tour, or load one of the pre-defined workflows in the section "Templates" which demonstrate the program functionality in cryptographic scenarios.

Main functions
- Create a new workspace with the graphical editor
- Use the wizard to easily try some CrypTool 2 features

External resources
- View a tutorial video about CrypTool 2 on YouTube

Templates (double click to open)
Search 277
- Cryptography
- Cryptanalysis

YouTube videos (double click to open)
- Applied Cryptography C1: Introduction - Basic Crypt

Recently opened workspaces (double click to open)

There are two ways to proceed:

- **Option A (Quick and Easy):** Use a ready-made template
- **Option B (Manual):** Build the digital signature setup yourself

<u>Manually Build the Digital Signature Workflow</u>

Step 3: Create New Workspace

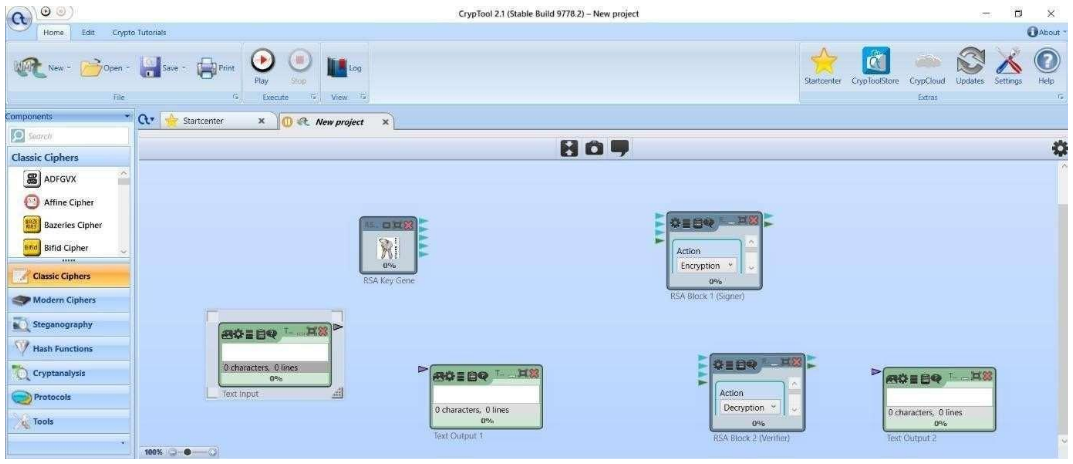
- In the Startcenter, click: **"Create a new workspace with the graphical editor".**

Step 4: Add Components to Workspace

- From the left panel (Components), search and add these components:

Component Name	**Purpose**
Text Input	To input the original message that needs to be digitally signed.
RSA Key Generator	To generate RSA key pairs (public and private keys).
RSA (Encryption/Decryption)	Used twice: one for signing with private key, another for verification with public key.
Text Output (×2)	To display: (1) original hash, (2) decrypted hash for manual comparison.

Components:



Step 5: Connect the Components

[Text Input] (text)

|

▼

RSA Block 1 (Signer)

Inputs:

1. n ← RSA Key Gen (1st output)

3. d ← RSA Key Gen (3rd output)

4. m ← Text Input

Output:

5. m → Text Output 1 (Signature)

RSA Block 2

(Verifier) Inputs:

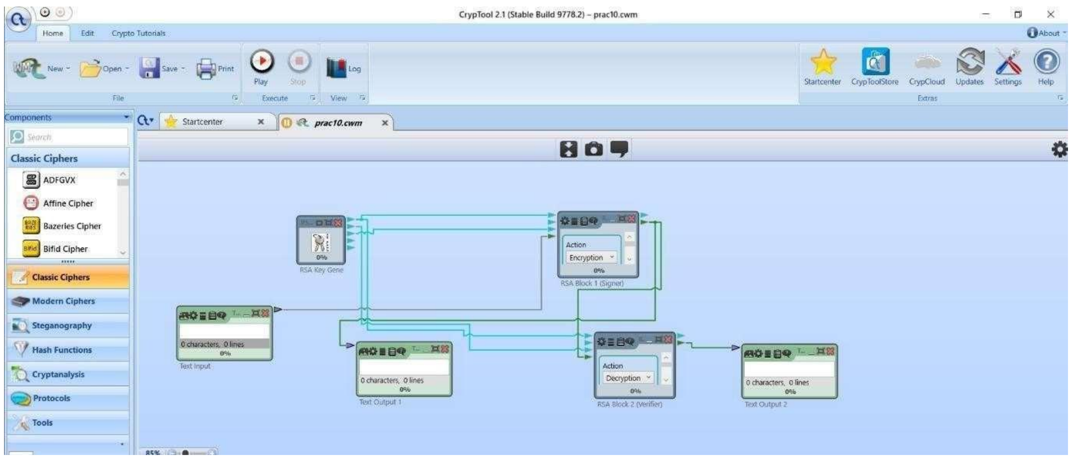
1. n ← RSA Key Gen (1st output)

3. e ← RSA Key Gen (2nd output)

4. m ← From RSA Block 1 (signature)

Output:

5. m → Text Output 2 (Verified Message)



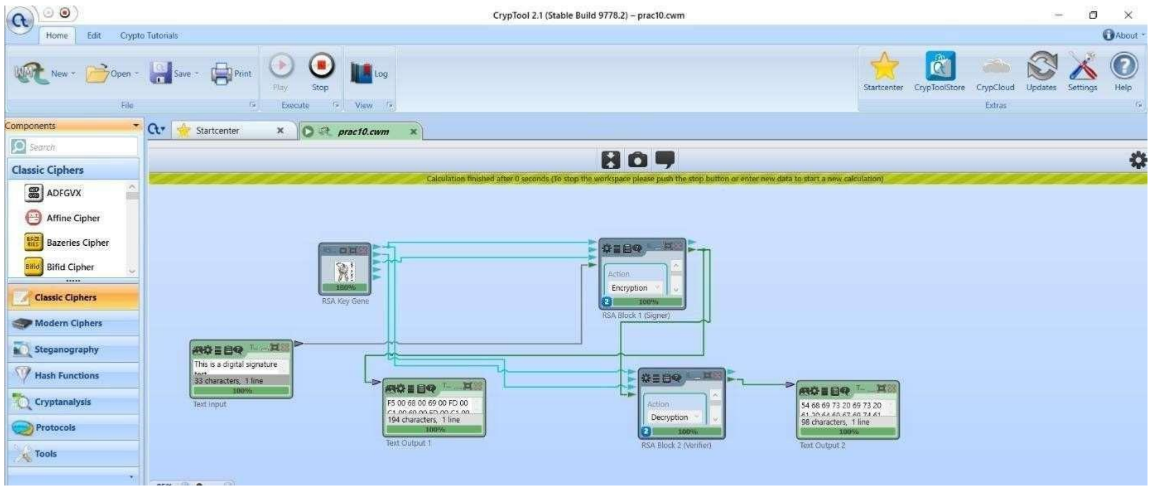
Action: Encryption

Action: Decryption

0 characters, 0 lines

Step 6: Enter message and run the setup

1. **Input the Message**:
   - Click the Text Input block.
   - Type a sample message, for example: "This is a digital signature test"
2. **Execute the Digital Signature Workflow**:
   - Click the Play button on the top toolbar to start the simulation.
   - CrypTool will process the flow in real-time.



Calculation finished after 0 seconds (To stop the workspace please push the stop button or enter new data to start a new calculation)

Text Input: This is a digital signature test — 33 characters, 1 line — 100%

RSA Block 1 (Signer): Action: Encryption — 100%

Text Output 1: F5 00 68 00 69 00 FD 00 C1 00 60 00 FD 00 C1 00 — 194 characters, 1 line — 100%

RSA Block 2 (Verifier): Action: Decryption — 100%

Text Output 2: 54 68 69 73 20 69 73 20 61 20 64 69 67 69 74 61 — 98 characters, 1 line — 100%

1. **Observe the Outputs:**
   - Text Output 1 will display the original SHA-256 hash of the message.
   - Text Output 2 will display the decrypted hash (retrieved using the public key).

4. **Manual Verification**:
   - Compare the values in Text Output 1 and Text Output 2.
   - If both hashes match exactly, the digital signature is valid, confirming:
     - Message authenticity (signed by private key)
     - Message integrity (not modified)
   - If they do not match, the message may have been altered or the signature is invalid.