

Lab Manual

CSC232 – Information Security



CUI

**Department of Computer Science
Islamabad Campus**

Lab Contents:

Topics include: Introduction to python, How to Implement Algorithms; Overview of Cryptography and Cryptanalysis, aspects of security, classical encryption techniques, and substitution and transposition ciphers. Block cipher and data encryption standard. Basic Concepts in Number Theory and Finite Fields. Key distribution problems, Public-Key Cryptography and RSA. Hash function and Digital Signature.

Student Outcomes (SO)

S.#	Description
2	Identify, formulate, research literature, and solve <i>complex</i> computing problems reaching substantiated conclusions using fundamental principles of mathematics, computing sciences, and relevant domain disciplines.
4	Create, select, adapt, and apply appropriate techniques, resources, and modern computing tools to <i>complex</i> computing activities, with an understanding of the limitations.

Intended Learning Outcomes

Sr.#	Description	Blooms Taxonomy Learning Level	SO
CLO-5	Implement a cryptographic algorithm to ensure information security	<i>Applying</i>	2,4
CLO-6	Build a small project that utilizes information security principles and cryptographic algorithms within a team environment.	<i>Creating</i>	

Lab Assessment Policy

The lab work done by the student is evaluated using **Psycho-motor rubrics** defined by the course instructor, viva-voce, project work/performance. Marks distribution

Assignments	Lab Mid Term Exam	Lab Terminal Exam	Total
25	25	50	100

Note: Midterm and Final term exams must be computer based.

Table of Contents

Lab #	Main Topic	Page #
Lab 01	Python Programming-Part I	1
Lab 02	Python Collections-Part II	14
Lab 03	Classical Ciphers	38
Lab 04	Stream Ciphers and PRG	49
Lab 05	Block Ciphers	65
Lab 06	Secure Hash Function	84
Lab 07	Blockchain Technology	96
Lab 08	Midterm-Exam	115
Lab 09	Key distribution problem	118
Lab 10	Cryptographic Math	126
Lab 11	ElGamal encryption	135
Lab 12	RSA	141
Lab 13	Elliptic Curve Cryptography	145
Lab 14	Digital signatures	149
Lab 15	Public-Key Certificates (PKC)	164

Lab 01

Python Introduction

Objective:

This lab is an introductory session on Python. The lab will equip students with necessary concepts needed to build algorithms in Python.

Activity Outcomes:

The lab will teach students to:

- Basic Operations on strings and numbers
- Basic use of conditionals
- Basic use of loops

Instructor Note:

As a pre-lab activity, read Chapters 3-5 from the book (Introduction to Programming Using Python - Y.Liang (Pearson, 2013)) to gain an insight about python programming and its fundamentals.

1) Useful Concepts

Python refers to the Python programming language (with syntax rules for writing what is considered valid Python code) and the Python interpreter software that reads source code (written in the Python language) and performs its instructions.

Python is a general-purpose programming language. That means you can use Python to write code for any programming task. Python is now used in the Google search engine, in mission-critical projects at NASA, and in transaction processing at the New York Stock Exchange.

Python's most obvious feature is that it uses indentation as a control structure. Indentation is used in Python to delimit blocks. **The number of spaces** is variable, but all statements within the same block must be indented the same amount. The header line for compound statements, such as `if`, `while`, `def`, and `class` should be terminated with a colon (`:`)

Example: Indentation as a control-structure

```
for i in range(20):
    if i%3 == 0:
        print i
    if i%5 == 0:
        print "Bingo!"
    print "---"
```

Variables

Python is dynamically typed. You do not need to declare variables! The declaration happens automatically when you assign a value to a variable. Variables can change type, simply by assigning them a new value of a different type. Python allows you to assign a single value to several variables simultaneously. You can also assign multiple objects to multiple variables.

Data types in Python

Every value in Python has a datatype. Since everything is an object in Python programming, data types are actually classes and variables are instance (object) of these classes.

There are various data types in Python. Some of the important types are listed below.

Integers

In Python 3, there is effectively no limit to how long an integer value can be. Of course, it is constrained by the amount of memory your system has, as are all things, but beyond that an integer can be as long as you need it to be:

```
>>> print(123123123123123123123123123123123123123123123123123123123 + 1)
123123123123123123123123123123123123123123123123123123124
Python interprets a sequence of decimal digits without any prefix to be a decimal number:
>>> print(10)
10
```

The following strings can be prepended to an integer value to indicate a base other than 10:

Prefi x	Interpretation	Base
0b (zero + lowercase letter 'b') 0B (zero + uppercase letter 'B')	Binary	2
0o (zero + lowercase letter 'o') 0O (zero + uppercase letter 'O')	Octal	8
0x (zero + lowercase letter 'x') 0X (zero + uppercase letter 'X')	Hexadecimal	16

For example:

```
>>> print(0o10)
8
>>> print(0x10)
16
>>> print(0b10)
2
```

The underlying type of a Python integer, irrespective of the base used to specify it, is called int:

```
>>> type(10)
<class 'int'>
>>> type(0o10)
<class 'int'>
>>> type(0x10)
<class 'int'>
```

Floating-Point Numbers

The float type in Python designates a floating-point number. float values are specified with a decimalpoint. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation:

```
>>> 4.2
4.2
>>> type(4.2)
<class 'float'>
>>> 4.
4.0
>>> .2
0.2

>>> .4e7
4000000.0
>>> type(.4e7)
<class 'float'>
>>> 4.2e-4
0.00042
```

Floating-Point Representation

The following is a bit more in-depth information on how Python represents floating-point numbers internally. You can readily use floating-point numbers in Python without understanding them to this level, so don't worry if this seems overly complicated. The information is presented here in case you are curious. Almost all platforms represent Python float values as 64-bit "double-precision" values, according to the [IEEE 754](#) standard. In that case, the maximum value a floating-point number can have is approximately 1.8×10^{308} . Python will indicate a number greater than that by the string `inf`:

```
>>> 1.79e308
1.79e+308
>>> 1.8e308
Inf
```

The closest a nonzero number can be to zero is approximately 5.0×10^{-324} . Anything closer to zero than that is effectively zero:

```
>>> 5e-324
5e-324
>>> 1e-325
0.0
```

Floating point numbers are represented internally as binary (base-2) fractions. Most decimal fractions cannot be represented exactly as binary fractions, so in most cases the internal representation of a floating-point number is an approximation of the actual value. In practice, the difference between the actual value and the represented value is very small and should not usually cause significant problems.

Complex Numbers :

Complex numbers are specified as `<real part>+<imaginary part>j`. For example:

```
>>> 2+3j
(2+3j)
>>> type(2+3j)
<class 'complex'>
```

Strings

Strings are sequences of character data. The string type in Python is called `str`.

String literals may be delimited using either single or double quotes. All the characters between the opening delimiter and matching closing delimiter are part of the string:

```
>>> print("I am a string.")
I am a string.
>>> type("I am a string.")
<class 'str'>
>>> print('I am too.')
I am too.
>>> type('I am too.')
<class 'str'>
```

A string in Python can contain as many characters as you wish. The only limit is your machine's memory resources. A string can also be empty:

What if you want to include a quote character as part of the string itself? Your first impulse might be to

```
>>> ""  
"
```

try something like this:

```
>>> print("This string contains a single quote (') character.")  
SyntaxError: invalid syntax
```

As you can see, that doesn't work so well. The string in this example opens with a single quote, so Python assumes the next single quote, the one in parentheses which was intended to be part of the string, is the closing delimiter. The final single quote is then a stray and causes the syntax error shown. If you want to include either type of quote character within the string, the simplest way is to delimit the string with the other type. If a string is to contain a single quote, delimit it with double quotes and vice versa:

```
>>> print("This string contains a single quote (') character.")  
This string contains a single quote (') character.  
  
>>> print('This string contains a double quote (") character.')  
This string contains a double quote (") character.
```

Escape Sequences in Strings

Sometimes, you want Python to interpret a character or sequence of characters within a string differently. This may occur in one of two ways:

- You may want to suppress the special interpretation that certain characters are usually given within a string.
- You may want to apply special interpretation to characters in a string which would normally be taken literally.

You can accomplish this using a backslash (\) character. A backslash character in a string indicates that one or more characters that follow it should be treated specially. (This is referred to as an escape sequence, because the backslash causes the subsequent character sequence to "escape" its usual meaning.)

Suppressing Special Character Meaning

You have already seen the problems you can come up against when you try to include quote characters in a string. If a string is delimited by single quotes, you can't directly specify a single quote character as part of the string because, for that string, the single quote has special meaning—it terminates the string:

```
>>> print("This string contains a single quote (') character.")  
SyntaxError: invalid syntax
```


Specifying a backslash in front of the quote character in a string “escapes” it and causes Python to suppress its usual special meaning. It is then interpreted simply as a literal single quote character:

```
>>> print("This string contains a single quote (\') character.")
This string contains a single quote (') character.
```

The same works in a string delimited by double quotes as well:

```
>>> print("This string contains a double quote (\") character.")
This string contains a double quote (") character.
```

The following is a table of escape sequences which cause Python to suppress the usual special interpretation of a character in a string:

Escape Sequence	Usual Interpretation of Character(s) After Backslash	“Escaped” Interpretation
\'	Terminates string with single quote opening delimiter	Literal single quote (') character
\"	Terminates string with double quote opening delimiter	Literal double quote (") character
\<newline>	Terminates input line	Newline is ignored
\\	Introduces escape sequence	Literal backslash (\) character

Ordinarily, a newline character terminates line input. So Enter in the middle of a string will pressing cause Python to think it is incomplete:

```
>>> print('a
SyntaxError: EOL while scanning string literal
```

To break up a string over more than one line, include a backslash before each newline, and the newlines will be ignored:

```
>>> print('a\
... b\
... c')
Abc
```

To include a literal backslash in a string, escape it with a backslash:

```
>>> print('foo\\bar')
foo\bar
```

Applying Special Meaning to Characters : Next, suppose you need to create a string that contains a tab character in it. Some text editors may allow you to insert a tab character directly into your code. But many programmers consider that poor practice, for several reasons:

- The computer can distinguish between a tab character and a sequence of space characters, but you can't. To a human reading the code, tab and space characters are visually indistinguishable.
- Some text editors are configured to automatically eliminate tab characters by expanding them to the appropriate number of spaces.
- Some Python REPL environments will not insert tabs into code.

In Python (and almost all other common computer languages), a tab character can be specified by the escape sequence `\t`:

```
>>> print('foo\tbar')
foo  bar
```

The escape sequence `\t` causes the `t` character to lose its usual meaning, that of a literal `t`. Instead, the combination is interpreted as a tab character.

Here is a list of escape sequences that cause Python to apply special meaning instead of interpreting literally:

Escape Sequence	“Escaped” Interpretation
<code>\a</code>	ASCII Bell (BEL) character
<code>\b</code>	ASCII Backspace (BS) character
<code>\f</code>	ASCII Formfeed (FF) character
<code>\n</code>	ASCII Linefeed (LF) character
<code>\N{<name>}</code>	Character from Unicode database with given <name>
<code>\r</code>	ASCII Carriage Return (CR) character
<code>\t</code>	ASCII Horizontal Tab (TAB) character
<code>\uxxxx</code>	Unicode character with 16-bit hex value <code>xxxx</code>
<code>\Uxxxxxxxx</code>	Unicode character with 32-bit hex value <code>xxxxxxxx</code>
<code>\v</code>	ASCII Vertical Tab (VT) character
<code>\ooo</code>	Character with octal value <code>ooo</code>
<code>\xhh</code>	Character with hex value <code>hh</code>

Examples:

```
>>> print("a\tb")
a  b
>>> print("a\141\x61")
aaa
>>> print("a\nb")
a
b
>>> print("\u2192 \N{rightwards arrow}")
→ →
```

This type of escape sequence is typically used to insert characters that are not readily generated from the keyboard or are not easily readable or printable.

Raw Strings

A raw string literal is preceded by r or R, which specifies that escape sequences in the associated string are not translated. The backslash character is left in the string:

```
>>> print('foo\nbar')
```

```
foo
bar
>>> print(r'foo\nbar')
foo\nbar

>>> print('foo\\bar')
foo\bar
>>> print(R'foo\\bar')
foo\\bar
```

Triple-Quoted Strings

There is yet another way of delimiting strings in Python. Triple-quoted strings are delimited by matching groups of three single quotes or three double quotes. Escape sequences still work in triple-quoted strings, but single quotes, double quotes, and newlines can be included without escaping them. This provides a convenient way to create a string with both single and double quotes in it:

```
>>> print("""This string has a single (') and a double (") quote.""")
This string has a single (') and a double (") quote.
Because newlines can be included without escaping them, this also allows for multiline strings:
>>> print("""This is a
string that spans
across several lines""")
This is a
string that spans
across several lines
```

You will see in the upcoming tutorial on Python Program Structure how triple-quoted strings can be used to add an explanatory comment to Python code.

Boolean Type, Boolean Context, and “Truthiness”

Python 3 provides a [Boolean data type](#). Objects of Boolean type may have one of two values, True or False:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

As you will see in upcoming tutorials, expressions in Python are often evaluated in Boolean context, meaning they are interpreted to represent truth or falsehood. A value that is true in Boolean context is sometimes said to be “truthy,” and one that is false in Boolean context is said to be “falsy.” (You may also see “falsy” spelled “falsey.”)

The “truthiness” of an object of Boolean type is self-evident: Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false). But non-Boolean objects can be evaluated in Boolean context as well and determined to be true or false.

2) Solved Lab Activities

<i>Sr.No</i>	<i>Allocated Time</i>	<i>Level of Complexity</i>	<i>CLO Mapping</i>
<i>1</i>	<i>5</i>	<i>Low</i>	<i>CLO-6</i>
<i>2</i>	<i>5</i>	<i>Low</i>	<i>CLO-6</i>
<i>3</i>	<i>10</i>	<i>Medium</i>	<i>CLO-6</i>
<i>4</i>	<i>5</i>	<i>Low</i>	<i>CLO-6</i>
<i>5</i>	<i>5</i>	<i>Low</i>	<i>CLO-6</i>
<i>6</i>	<i>5</i>	<i>Low</i>	<i>CLO-6</i>
<i>7</i>	<i>10</i>	<i>Medium</i>	<i>CLO-6</i>

Activity 1:

Let us take an integer from user as input and check whether the given value is even or not. If the given value is not even then it means that it will be odd. So here we need to use if-else statement as demonstrated below

A. Create a new Python file from Python Shell and type the following code.

B. Run the code by pressing F5.

```
n=input("Enter a number ")
if int(n)%2==0:
    print("The given number is an even number")
else:
    print("The given number is an odd number")
```

Output

```
Enter a number 11
The given number is an odd number
>>>
```

Activity 2:

Write a Python code to keep accepting integer values from user until 0 is entered. Display sum of the given values.

Solution:

```
sum=0
s=input("Enter an integer value...")
n=int(s)
while n!=0:
    sum=sum+n
    s=input("Enter an integer value...")
    n=int(s)
print("Sum of given values is ",sum)
```

Output

```
Enter an integer value...10
Enter an integer value...521
Enter an integer value...5
Enter an integer value...22
Enter an integer value...0
Sum of given values is 558
>>>
```

Activity 3:

Write a Python code to accept an integer value from user and check that whether the given value is prime number or not.

Solution:

```
isPrime = True
i=2
n=int(input("enter a number"))
while i<n:
    remainder=n%i
    if remainder==0:
        isPrime=False
        break
    else:
        i=i+1

if isPrime:
    print("Number is Prime")
else:
    print("Number is not Prime")
```

Activity 4:

Accept 5 integer values from user and display their sum. Draw flowchart before coding in python.

Solution:

Create a new Python file from Python Shell and type the following code. Run the code by pressing F5.

```
summ = 0
i=0
while i<=4:
    s=input("enter a number")
    n=int(s)
    summ=summ+n
    i=i+1

print("sum is ",summ)
```

You will get the following output.

```
enter a number1
enter a number2
enter a number3
enter a number4
enter a number5
sum is 15
>>>
```

Activity 5:

Calculate the sum of all the values between 0-10 using while loop.

Solution:

Create a new Python file from Python Shell and type the following code.

Run the code by pressing F5.

```
summation = 0
i=1
while i<=10:
    summation=summation+i
    i=i+1

print("sum is ",summation)
```

You will get the following output.

```
sum is 55
>>>
```

Activity 6:

Take input from the keyboard and use it in your program.

Solution:

In Python and many other programming languages you can get user input. In Python the input() function will ask keyboard input from the user. The input function prompts text if a parameter is given. The function reads input from the keyboard, converts it to a string and removes the newline (Enter). Type and experiment with the script below.

```
#!/usr/bin/env python3

name = input('What is your name? ')
print('Hello ' + name)

job = input('What is your job? ')
print('Your job is ' + job)

num = input('Give me a number? ')
print('You said: ' + str(num))
```

Activity 7:

Generate a random number between 1 and 9 (including 1 and 9). Ask the user to guess the number, then tell them whether they guessed too low, too high, or exactly right. (Hint: remember to use the user input lessons from the very first exercise)

Extras:

Keep the game going until the user types “exit”

Keep track of how many guesses the user has taken, and when the game ends, print this out.

Solution:

```
import random
# Awroken

MINIMUM = 1
MAXIMUM = 9
NUMBER = random.randint(MINIMUM, MAXIMUM)
GUESS = None
ANOTHER = None
TRY = 0
RUNNING = True

print "Alright..."

while RUNNING:
    GUESS = raw_input("What is your lucky number? ")
    if int(GUESS) < NUMBER:
        print "Wrong, too low."
    elif int(GUESS) > NUMBER:
        print "Wrong, too high."
    elif GUESS.lower() == "exit":
        print "Better luck next time."
    elif int(GUESS) == NUMBER:
        print "Yes, that's the one, %s." % str(NUMBER)
        if TRY < 2:
            print "Impressive, only %s tries." % str(TRY)
        elif TRY > 2 and TRY < 10:
            print "Pretty good, %s tries." % str(TRY)
        else:
            print "Bad, %s tries." % str(TRY)
        RUNNING = False
    TRY += 1
```

3) Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab.

Lab Task 1:

Write a program that prompts the user to input an integer and then outputs the number with the digits reversed. For example, if the input is 12345, the output should be 54321.

Lab Task 2:

Write a program that reads a set of integers, and then prints the sum of the even and odd integers.

Lab Task 3:

Fibonacci series is that when you add the previous two numbers the next number is formed. You have to start from 0 and 1.

E.g. $0+1=1 \rightarrow 1+1=2 \rightarrow 1+2=3 \rightarrow 2+3=5 \rightarrow 3+5=8 \rightarrow 5+8=13$

So the series becomes

0 1 1 2 3 5 8 13 21 34 55

Steps: You have to take an input number that shows how many terms to be displayed. Then use loops for displaying the Fibonacci series up to that term e.g. input no is =6 the output should be

0 1 1 2 3 5

Lab Task 4:

Write a Python code to accept marks of a student from 1-100 and display the grade according to the following formula.

Grade F if marks are less than 50

Grade E if marks are between 50 to

60 Grade D if marks are between 61

to 70 Grade C if marks are between

71 to 80 Grade B if marks are

between 81 to 90 Grade A if marks

are between 91 to 100

Lab Task 5:

Write a program that takes a number from user and calculate the factorial of that number.

Lab 02

Python Lists and Dictionaries

Objective:

This lab will give you practical implementation of different types of **sequences** including **lists, tuples, sets and dictionaries**. We will use lists alongside loops in order to know about indexing individual items of these containers. This lab will also allow students to write their own **functions**.

Activity Outcomes:

This lab teaches you the following topics:

- How to use lists, tuples, sets and dictionaries
- How to use loops with lists
- How to write customized functions

Instructor Note:

As a pre-lab activity, read Chapters 6, 10 and 14 from the book (*Introduction to Programming Using Python - Y. Liang (Pearson, 2013)*) to gain an insight about python programming and its fundamentals.

1) Useful Concepts

Python provides different types of data structures as sequences. In a sequence, there are more than one values and each value has its own index. The first value will have an index 0 in python, the second value will have index 1 and so on. These indices are used to access a particular value in the sequence.

Python Lists:

Lists are just like dynamically sized arrays, declared in other languages (vector in C++ and ArrayList in Java). Lists need not be homogeneous always which makes it the most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

List in Python are ordered and have a definite count. The elements in a list are indexed according to a definite sequence and the indexing of a list is done with 0 being the first index. Each element in the list has its definite place in the list, which allows duplicating of elements in the list, with each element having its own distinct place and credibility.

Creating a List

Lists in Python can be created by just placing the sequence inside the square brackets []. Unlike Sets, a list doesn't need a built-in function for the creation of a list.

```
# Python program to demonstrate
# Creation of List

# Creating a List
List = []
print("Blank List: ")
print(List)

# Creating a List of numbers
List = [10, 20, 14]
print("\nList of numbers: ")
print(List)

# Creating a List of strings and accessing
# using index
List = ["Geeks", "For", "Geeks"]
print("\nList Items: ")
print(List[0])
print(List[2])

# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List = [['Geeks', 'For'], ['Geeks']]
print("\nMulti-Dimensional List: ")
print(List)
```

Output:

Blank List:

```
[]
```

List of numbers:

```
[10, 20, 14]
```

List Items

```
Geeks
```

```
Geeks
```

Multi-Dimensional List:

```
[['Geeks', 'For'], ['Geeks']]
```

Creating a list with multiple distinct or duplicate elements

A list may contain duplicate values with their distinct positions and hence, multiple distinct or duplicate values can be passed as a sequence at the time of list creation.

```
# Creating a List with
# the use of Numbers
# (Having duplicate values)
List = [1, 2, 4, 4, 3, 3, 3, 6, 5]
print("\nList with the use of Numbers: ")
print(List)

# Creating a List with
# mixed type of values
# (Having numbers and strings)
List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']
print("\nList with the use of Mixed Values: ")
print(List)
```

Output:

List with the use of Numbers:

```
[1, 2, 4, 4, 3, 3, 3, 6, 5]
```

List with the use of Mixed Values:

```
[1, 2, 'Geeks', 4, 'For', 6, 'Geeks']
```

Knowing the size of List

```
# Creating a List
List1 = []
```

```
print(len(List1))
```

```
# Creating a List of numbers
```

```
List2 = [10, 20, 14]
```

```
print(len(List2))
```

Output:

```
0
```

```
3
```

Adding Elements to a List Using `append()` method

Elements can be added to the List by using the built-in [`append\(\)`](#) function. Only one element at a time can be added to the list by using the `append()` method, for the addition of multiple elements with the `append()` method, loops are used. Tuples can also be added to the list with the use of the `append` method because tuples are immutable. Unlike Sets, Lists can also be added to the existing list with the use of the `append()` method.

```
# Python program to demonstrate
```

```
# Addition of elements in a List
```

```
# Creating a List
```

```
List = []
```

```
print("Initial blank List: ")
```

```
print(List)
```

```
# Addition of Elements
```

```
# in the List
```

```
List.append(1)
```

```
List.append(2)
```

```
List.append(4)
```

```
print("\nList after Addition of Three elements: ")
```

```
print(List)
```

```
# Adding elements to the List
```

```
# using Iterator
```

```
for i in range(1, 4):
```

```
    List.append(i)
```

```
print("\nList after Addition of elements from 1-3: ")
```

```
print(List)
```

```
# Adding Tuples to the List
```

```
List.append((5, 6))
```

```
print("\nList after Addition of a Tuple: ")
```

```
print(List)

# Addition of List to a List
List2 = ['For', 'Geeks']
List.append(List2)
print("\nList after Addition of a List: ")
print(List)
```

Output:

Initial blank List:

```
[]
```

List after Addition of Three elements:

```
[1, 2, 4]
```

List after Addition of elements from 1-3:

```
[1, 2, 4, 1, 2, 3]
```

List after Addition of a Tuple:

```
[1, 2, 4, 1, 2, 3, (5, 6)]
```

List after Addition of a List:

```
[1, 2, 4, 1, 2, 3, (5, 6), ['For', 'Geeks']]
```

Using insert() method

append() method only works for the addition of elements at the end of the List, for the addition of elements at the desired position, insert() method is used. Unlike append() which takes only one argument, the insert() method requires two arguments(position, value).

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = [1,2,3,4]
print("Initial List: ")
print(List)

# Addition of Element at
# specific Position
# (using Insert Method)
List.insert(3, 12)
List.insert(0, 'Geeks')
print("\nList after performing Insert Operation: ")
print(List)
```

Output:

Initial List:

[1, 2, 3, 4]

List after performing Insert Operation:

['Geeks', 1, 2, 3, 12, 4]

Using extend() method

Other than append() and insert() methods, there's one more method for the Addition of elements, [extend\(\)](#), this method is used to add multiple elements at the same time at the end of the list.

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = [1, 2, 3, 4]
print("Initial List: ")
print(List)

# Addition of multiple elements
# to the List at the end
# (using Extend Method)
List.extend([8, 'Geeks', 'Always'])
print("\nList after performing Extend Operation: ")
print(List)
```

Output:

Initial List:

[1, 2, 3, 4]

List after performing Extend Operation:

[1, 2, 3, 4, 8, 'Geeks', 'Always']

Accessing elements from the List

In order to access the list items refer to the index number. Use the index operator [] to access an item in a list. The index must be an integer. Nested lists are accessed using nested indexing.

```
# Python program to demonstrate
# accessing of element from list

# Creating a List with
# the use of multiple values
List = ["Geeks", "For", "Geeks"]
```

Negative indexing

In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in `List[len(List)-3]`, it is enough to just write `List[-3]`. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

```
List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']

# accessing an element using negative indexing
print("Accessing element using negative indexing")

# print the last element of list
print(List[-1])

# print the third last element of list
print(List[-3])

Output:
Accessing element using negative indexing
Geeks
For
```

Removing Elements from the List Using `remove()` method

Elements can be removed from the List by using the built-in [`remove\(\)`](#) function but an Error arises if the element doesn't exist in the list. [`Remove\(\)`](#) method only removes one element at a time, to remove a range of elements, the iterator is used. The `remove()` method removes the specified item.


```

# Python program to demonstrate
# Removal of elements in a List

# Creating a List
List = [1, 2, 3, 4, 5, 6,
        7, 8, 9, 10, 11, 12]
print("Initial List: ")
print(List)

# Removing elements from List
# using Remove() method
List.remove(5)
List.remove(6)
print("\nList after Removal of two elements: ")
print(List)

# Removing elements from List
# using iterator method
for i in range(1, 5):
    List.remove(i)
print("\nList after Removing a range of elements: ")
print(List)

```

Output:

Initial List:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

List after Removal of two elements:
[1, 2, 3, 4, 7, 8, 9, 10, 11, 12]

List after Removing a range of elements:
[7, 8, 9, 10, 11, 12]

Using pop() method

[Pop\(\)](#) function can also be used to remove and return an element from the list, but by default it removes only the last element of the list, to remove an element from a specific position of the List, the index of the element is passed as an argument to the pop() method.

```

List = [1,2,3,4,5]
# Removing element from the

```

```
# Set using the pop() method
List.pop()
print("\nList after popping an element: ")
print(List)

# Removing element at a
# specific location from the
# Set using the pop() method
List.pop(2)
print("\nList after popping a specific element: ")
print(List)
```

Output:

List after popping an element:
[1, 2, 3, 4]

List after popping a specific element:
[1, 2, 4]

Slicing of a List

In Python List, there are multiple ways to print the whole List with all the elements, but to print a specific range of elements from the list, we use the [Slice operation](#). Slice operation is performed on Lists with the use of a colon(:). To print elements from beginning to a range use [: Index], to print elements from end-use[:-Index], to print elements from specific Index till the end use [Index:], to print elements within a range, use [Start Index:End Index] and to print the whole List with the use of slicing operation, use [:]. Further, to print the whole List in reverse order, use[::-1].

Note – To print elements of List from rear-end, use Negative Indexes.

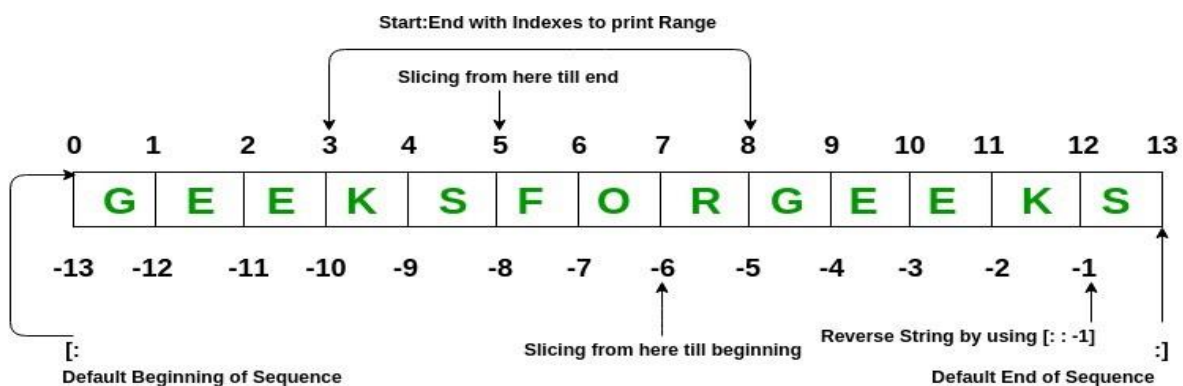


Figure 1 - List

```
# Python program to demonstrate  
# Removal of elements in a List
```

```
# Creating a List
```

```
List = ['G', 'E', 'E', 'K', 'S', 'F',  
        'O', 'R', 'G', 'E', 'E', 'K', 'S']
```

```

print("Initial List: ")
print(List)

# Print elements of a range
# using Slice operation
Sliced_List = List[3:8]
print("\nSlicing elements in a range 3-8: ")
print(Sliced_List)

# Print elements from a
# pre-defined point to end
Sliced_List = List[5:]
print("\nElements sliced from 5th "
      "element till the end: ")
print(Sliced_List)

# Printing elements from
# beginning till end
Sliced_List = List[:]
print("\nPrinting all elements using slice operation: ")
print(Sliced_List)

```

Output:

Initial List:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Slicing elements in a range 3-8:
['K', 'S', 'F', 'O', 'R']

Elements sliced from 5th element till the end:
['F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Printing all elements using slice operation:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Negative index List slicing

```

# Creating a List
List = ['G', 'E', 'E', 'K', 'S', 'F',
        'O', 'R', 'G', 'E', 'E', 'K', 'S']
print("Initial List: ")
print(List)

# Print elements from beginning
# to a pre-defined point using Slice

```

```

Sliced_List = List[:-6]
print("\nElements sliced till 6th element from last: ")
print(Sliced_List)

# Print elements of a range
# using negative index List slicing
Sliced_List = List[-6:-1]
print("\nElements sliced from index -6 to -1")
print(Sliced_List)

# Printing elements in reverse
# using Slice operation
Sliced_List = List[::-1]
print("\nPrinting List in reverse: ")
print(Sliced_List)

```

Output:

Initial List:

```
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']
```

Elements sliced till 6th element from last:

```
['G', 'E', 'E', 'K', 'S', 'F', 'O']
```

Elements sliced from index -6 to -1

```
['R', 'G', 'E', 'E', 'K']
```

Printing List in reverse:

```
['S', 'K', 'E', 'E', 'G', 'R', 'O', 'F', 'S', 'K', 'E', 'E', 'G']
```

List Comprehension

List comprehensions are used for creating new lists from other iterables like tuples, strings, arrays, lists,

etc

.

A list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element.

Syntax:

newList = [expression(element) for element in oldList if condition]

Example:

```
# Python program to demonstrate list comprehension in Python  
# below list contains square of all odd numbers from range 1 to 10
```

```
odd_square = [x ** 2 for x in range(1, 11) if x % 2 == 1]  
print(odd_square)
```

Output:

```
[1, 9, 25, 49, 81]
```

For better understanding, the above code is similar to –

```
# for understanding, above generation is same as,
odd_square = []

for x in range(1, 11):
    if x % 2 == 1:
        odd_square.append(x**2)

print(odd_square)
Output:
[1, 9, 25, 49, 81]
```

Dictionary in Python is an unordered collection of data values, used to store data values like a map, which, unlike other Data Types that hold only a single value as an element, Dictionary holds **key:value** pair. Key-value is provided in the dictionary to make it more optimized.

Note – Keys in a dictionary don't allow Polymorphism.

Disclaimer: *It is important to note that Dictionaries have been modified to maintain insertion order with the release of Python 3.7, so they are now ordered collection of data values.*

Creating a Dictionary

In Python, a Dictionary can be created by placing a sequence of elements within curly {} braces, separated by 'comma'. Dictionary holds pairs of values, one being the Key and the other corresponding pair element being its **Key:value**. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be *immutable*.

```
# Creating a Dictionary
# with Integer Keys
Dict = {1: 'Geeks', 2: 'For', 3: 'Geeks'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

# Creating a Dictionary
# with Mixed keys
Dict = {'Name': 'Geeks', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
```

Output:

Dictionary with the use of Integer Keys:

```
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
```

Dictionary with the use of Mixed Keys:

```
{1: [1, 2, 3, 4], 'Name': 'Geeks'}
```

Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing curly braces {}.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Geeks', 2: 'For', 3: 'Geeks'})
print("\nDictionary with the use of dict(): ")
print(Dict)

# Creating a Dictionary
# with each item as a Pair
Dict = dict([(1, 'Geeks'), (2, 'For')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Output:

Empty Dictionary:

```
{}
```

Dictionary with the use of dict():

```
{1: 'Geeks', 2: 'For', 3: 'Geeks'}
```

Dictionary with each item as a pair:

```
{1: 'Geeks', 2: 'For'}
```

Nested Dictionary:

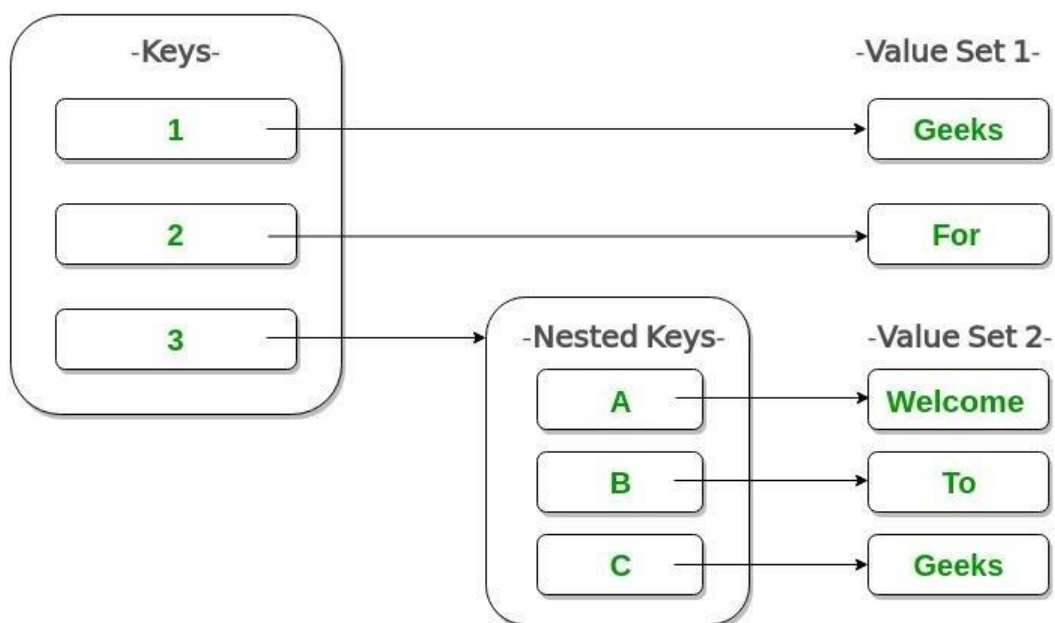


Figure 2 - Dictionary


```
# Creating a Nested Dictionary
# as shown in the below image
Dict = {1: 'Geeks', 2: 'For',
        3: {'A': 'Welcome', 'B': 'To', 'C': 'Geeks'}}

print(Dict)
Output:
{1: 'Geeks', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'Geeks'}}
```

Adding elements to a Dictionary

In Python Dictionary, the Addition of elements can be done in multiple ways. One value at a time can be added to a Dictionary by defining value along with the key e.g. Dict[Key] = 'Value'. Updating an existing value in a Dictionary can be done by using the built-in **update()** method. Nested key values can also be added to an existing Dictionary.

Note- While adding a value, if the key-value already exists, the value gets updated otherwise a new Key with the value is added to the Dictionary.

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)

# Adding elements one at a time
Dict[0] = 'Geeks'
Dict[2] = 'For'
Dict[3] = 1
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Adding set of values
# to a single Key
Dict['Value_set'] = 2, 3, 4
print("\nDictionary after adding 3 elements: ")
print(Dict)

# Updating existing Key's Value
Dict[2] = 'Welcome'
print("\nUpdated key value: ")
print(Dict)

# Adding Nested Key value to Dictionary
Dict[5] = {'Nested': {'1': 'Life', '2': 'Geeks'}}
```

```
print("\nAdding a Nested Key: ")
print(Dict)
```

Output:

Empty Dictionary:
{}

Dictionary after adding 3 elements:
{0: 'Geeks', 2: 'For', 3: 1}

Dictionary after adding 3 elements:
{0: 'Geeks', 2: 'For', 3: 1, 'Value_set': (2, 3, 4)}

Updated key value:
{0: 'Geeks', 2: 'Welcome', 3: 1, 'Value_set': (2, 3, 4)}

Adding a Nested Key:
{0: 'Geeks', 2: 'Welcome', 3: 1, 5: {'Nested': {'1': 'Life', '2': 'Geeks'}}, 'Value_set': (2, 3, 4)}

Accessing elements from a Dictionary

In order to access the items of a dictionary refer to its key name. Key can be used inside squarebrackets.

```
# Python program to demonstrate
# accessing a element from a Dictionary

# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# accessing a element using key
print("Accessing a element using key:")
print(Dict['name'])

# accessing a element using key
print("Accessing a element using key:")
print(Dict[1])
```

Output:

Accessing a element using key:
For
Accessing a element using key:
Geeks

There is also a method called [get\(\)](#) that will also help in accessing the element from a dictionary.

```
# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# accessing a element using get()
# method
print("Accessing a element using get:")
print(Dict.get(3))
Output:
Accessing a element using get:
Geeks
```

Accessing an element of a nested dictionary

In order to access the value of any key in the nested dictionary, use indexing [] syntax.

```
# Creating a Dictionary
Dict = {'Dict1': {1: 'Geeks'},
        'Dict2': {'Name': 'For'}}

# Accessing element using key
print(Dict['Dict1'])
print(Dict['Dict1'][1])
print(Dict['Dict2']['Name'])

Output:
{1: 'Geeks'}
Geeks
For
```

Removing Elements from Dictionary Using del keyword

In Python Dictionary, deletion of keys can be done by using the **del** keyword. Using the **del** keyword, specific values from a dictionary as well as the whole dictionary can be deleted. Items in a Nested dictionary can also be deleted by using the **del** keyword and providing a specific nested key and particular key to be deleted from that nested Dictionary.

```
# Initial Dictionary
Dict = { 5: 'Welcome', 6: 'To', 7: 'Geeks',
        'A': {1: 'Geeks', 2: 'For', 3: 'Geeks'},
        'B': {1: 'Geeks', 2: 'Life'}}
print("Initial Dictionary: ")
print(Dict)

# Deleting a Key value
del Dict[6]
```

```
print("\nDeleting a specific key: ")
print(Dict)

# Deleting a Key from
# Nested Dictionary
del Dict['A'][2]
print("\nDeleting a key from Nested Dictionary: ")
print(Dict)
```

Output:

Initial Dictionary:

```
{'A': {1: 'Geeks', 2: 'For', 3: 'Geeks'}, 'B': {1: 'Geeks', 2: 'Life'}, 5: 'Welcome', 6: 'To', 7: 'Geeks'}
```

Deleting a specific key:

```
{'A': {1: 'Geeks', 2: 'For', 3: 'Geeks'}, 'B': {1: 'Geeks', 2: 'Life'}, 5: 'Welcome', 7: 'Geeks'}
```

Deleting a key from Nested Dictionary:

```
{'A': {1: 'Geeks', 3: 'Geeks'}, 'B': {1: 'Geeks', 2: 'Life'}, 5: 'Welcome', 7: 'Geeks'}
```

Using pop() method

[Pop\(\)](#) method is used to return and delete the value of the key specified.

```
# Creating a Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}

# Deleting a key
# using pop() method
pop_ele = Dict.pop(1)
print("\nDictionary after deletion: " + str(Dict))
print("Value associated to popped key is: " + str(pop_ele))
```

Output:

Dictionary after deletion: {3: 'Geeks', 'name': 'For'}

Value associated to popped key is: Geeks

Using popitem() method

The [popitem\(\)](#) returns and removes an arbitrary element (key, value) pair from the dictionary.

```
# Creating Dictionary
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}
# Deleting an arbitrary key
# using popitem() function
pop_ele = Dict.popitem()
print("\nDictionary after deletion: " + str(Dict))
```

```
print("The arbitrary pair returned is: " + str(pop_ele))
```

Output:

Dictionary after deletion: {3: 'Geeks', 'name': 'For'}

The arbitrary pair returned is: (1, 'Geeks')

Using clear() method

All the items from a dictionary can be deleted at once by using **clear()** method.

```
# Creating a Dictionary
```

```
Dict = {1: 'Geeks', 'name': 'For', 3: 'Geeks'}
```

```
# Deleting entire Dictionary
```

```
Dict.clear()
```

```
print("\nDeleting Entire Dictionary: ")
```

```
print(Dict)
```

Output:

Deleting Entire Dictionary:

```
{}
```

Dictionary Methods

Methods	Description
copy()	They copy() method returns a shallow copy of the dictionary.
clear()	The clear() method removes all items from the dictionary.
pop()	Removes and returns an element from a dictionary having the given key.
popitem()	Removes the arbitrary key-value pair from the dictionary and returns it as tuple.
get()	It is a conventional method to access a value for a key.
dictionary_name.values()	returns a list of all the values available in a given dictionary.
str()	Produces a printable string representation of a dictionary.
update()	Adds dictionary dict2's key-values pairs to dict
setdefault()	Set dict[key]=default if key is not already in dict
keys()	Returns list of dictionary dict's keys
items()	Returns a list of dict's (key, value) tuple pairs
has_key()	Returns true if key in dictionary dict, false otherwise
fromkeys()	Create a new dictionary with keys from seq and values set to value.

type() Returns the type of the passed variable.
cmp() Compares elements of both dict.

Solved Lab Activities:

<i>Sr.No</i>	<i>Allocated Time</i>	<i>Level of Complexity</i>	<i>CLO Mapping</i>
<i>1</i>	<i>10</i>	<i>Low</i>	<i>CLO-4</i>
<i>2</i>	<i>10</i>	<i>Low</i>	<i>CLO-4</i>
<i>3</i>	<i>10</i>	<i>Low</i>	<i>CLO-4</i>
<i>4</i>	<i>15</i>	<i>Medium</i>	<i>CLO-4</i>
<i>5</i>	<i>15</i>	<i>Medium</i>	<i>CLO-4</i>
<i>6</i>	<i>15</i>	<i>Medium</i>	<i>CLO-4</i>

Activity 1

Accept two lists from user and display their join.

Solution:

```
myList1=[]
print("Enter objects of first list...")
for i in range(5):
    val=input("Enter a value:")
    n=int(val)
    myList1.append(n)

myList2=[]
print("Enter objects of second list...")
for i in range(5):
    val=input("Enter a value:")
    n=int(val)
    myList2.append(n)

list3=myList1+myList2;
print(list3)
```

You will get the following output.

```
Enter objects of first list...
Enter a value:1
Enter a value:2
Enter a value:3
Enter a value:4
Enter a value:5
Enter objects of second list...
Enter a value:6
Enter a value:7
Enter a value:8
Enter a value:9
Enter a value:0
[1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
>>>
```

Activity 2:

A palindrome is a string which is same read forward or backwards.

For example: "dad" is the same in forward or reverse direction. Another example is "aibohphobia" which literally means, an irritable fear of palindromes.

Write a function in python that receives a string and returns True if that string is a palindrome and False otherwise. Remember that difference between upper and lower case characters are ignored during this determination.

Solution:

```
def isPalindrome(word):
    temp=word[::-1]
    if temp.capitalize()==word.capitalize():
        return True
    else:
        return False

print(isPalindrome("deed"))
```

Activity 3:

Imagine two matrices given in the form of 2D lists as under; $a = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]$
 $b = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]$

Write a python code that finds another matrix/2D list that is a product of a and b , i.e., $C=a*b$

Solution:

```
for indrow in range (3):
    c.append ([])
    for indcol in range(3):
        c[indrow].append (0)
        for indaux in range (3):
            c[indrow][indcol] += a[indrow][indaux] * b[indcol][indaux]

print (c)
```

Activity 4:

A closed polygon with N sides can be represented as a list of tuples of N connected coordinates, i.e., $[(x_1, y_1), (x_2, y_2), (x_3, y_3), \dots, (x_N, y_N)]$. A sample polygon with 6 sides ($N=6$) is shown below.

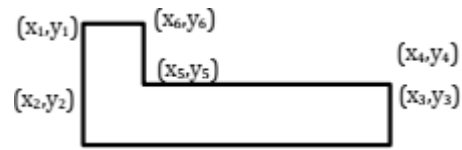


Figure 3 - Polygon

Write a python function that takes a list of N tuples as input and returns the perimeter of the polygon. Remember that your code should work for any value of N.

Hint: A perimeter is the sum of all sides of a polygon.

Solution:

```
def perimeter(listing):
    leng=len(listing)
    perimeter=0;
    for i in range(0,leng-1):
        dist = (((listing[i][0]-listing[i+1][0])**2)+
                ((listing[i][1]-listing[i+1][1])**2))**0.5
        perimeter = perimeter + dist
    perimeter = perimeter + (((listing[0][0]-listing[leng-1][0])**2)
                              +((listing[0][1]-listing[leng-1][1])**2))**0.5
    return perimeter

L = [(1,3), (2,7), (3,9), (-1,8)]
print(perimeter(L))
```

Activity 5:

Imagine two sets A and B containing numbers. Without using built-in set functionalities, write your own function that receives two such sets and returns another set C which is a symmetric difference of the two input sets. (A symmetric difference between A and B will return a set C which contains only those items that appear in one of A or B. Any items that appear in both sets are not included in C). Now compare the output of your function with the following built-in functions/operators.

- ✓ `A.symmetric_difference(B)`
- ✓ `B.symmetric_difference(A)`
- ✓ `A ^ B`
- ✓ `B ^ A`

Solution:

```
#Function defined
def symmDiff(a,b):
    e=set() #empty set
    for i in a: #for loop used to access in a
        if i not in b:
            e.add(i)
    for i in b: #for loop used to access in b
        if i not in a:
            e.add(i)
    return e

set1={0,1,2,4,5}
set2={4,5,7,8,9}
print(symmDiff(set1,set2))

#verification using inbuilt function
print(set1.symmetric_difference(set2))
print(set2.symmetric_difference(set1))
print(set1^set2)
print(set2^set1)
```

Activity 6:

Create a Python program that contains a dictionary of names and phone numbers. Use a tuple of separate first and last name values for the key field. Initialize the dictionary with at least three names and numbers. Ask the user to search for a phone number by entering a first and last name. Display the matching number if found, or a message if not found.

Solution:

```
sample={("sohaib","ali"):"0246585468445", ("aib","li"):"02465854645",
        ("sib","ai"):"0246585468445",}
firstName = input("enter first name")
lastName = input("enter last name")

searchTuple = (firstName, lastName)
if searchTuple in sample:
    print(sample[searchTuple])
else:
    print("name not found")
```

Graded Lab Tasks

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the solved lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab Task 1:

Create two lists based on the user values. Merge both the lists and display in sorted order.

Lab Task 2:

Repeat the above activity to find the smallest and largest element of the list. (Suppose all the elements are integer values)

Lab Task 3:

For this exercise, you will keep track of when our friend's birthdays are, and be able to find that information based on their name. Create a dictionary (in your file) of names and birthdays. When you run your program it should ask the user to enter a name, and return the birthday of that person back to them. The interaction should look something like this:

```
>>> Welcome to the birthday dictionary. We know the
birthdays of:Albert Einstein
Benjamin
FranklinAda
Lovelace
```

```
>>> Who's birthday do you want to  
look up? Benjamin Franklin  
>>> Benjamin Franklin's birthday is 01/17/1706.
```

Lab Task 4:

Create a dictionary by extracting the keys from a given dictionary

Write a Python program to create a new dictionary by extracting the mentioned keys from the below dictionary.

Given

dictionary:

```
sample_dict = {  
    "name": "Kelly",  
    "age": 25,  
    "salary": 8000,  
    "city": "New york"}
```

Keys to extract

```
keys = ["name", "salary"]
```

Expected output:

```
{'name': 'Kelly', 'salary': 8000}
```

Lab 03

Classical Ciphers

L

In this lab students will get an understanding of some common classical cryptography schemes, and learn the Python code that will bring all the topics together. Specifically, students will gain cryptographic knowledge about Substitution Ciphers, Caesar Cipher, Vigenère Cipher, Playfair, Column Transposition, Affine Cipher.

Lab Outcomes:

- Explore the basics of encryption schemes
- Explore the use of historical ciphers and their cryptanalysis
- Gain an understanding of why it is critical to use well-established encryption algorithms

Instructor Note:

Perform an understanding practice of given different ciphers codes and after execution write line wise code description

1) Solved Lab Activities

<i>Sr.No</i>	<i>Alloc ated Time</i>	<i>Level of Complexity</i>	<i>CLO Mapping</i>
<i>1</i>	<i>35</i>	<i>Medium</i>	<i>CLO-4</i>
<i>2</i>	<i>35</i>	<i>Medium</i>	<i>CLO-4</i>
<i>3</i>	<i>35</i>	<i>Medium</i>	<i>CLO-4</i>
<i>4</i>	<i>35</i>	<i>Medium</i>	<i>CLO-4</i>
<i>5</i>	<i>35</i>	<i>Medium</i>	<i>CLO-4</i>

Practice -Lab Activity 1: Substitution Ciphers

The substitution cipher simply substitutes one letter in the alphabet for another based upon a cryptovariable. The substitution involves shifting positions in the alphabet. This includes the Caesar cipher and ROT-13, which will be covered shortly. Examine the following example:

Plaintext: WE HOLD THESE TRUTHS TO BE SELF-EVIDENT, THAT ALL MEN ARE CREATED EQUAL.

Ciphertext: ZH KROG WKHVH WUXWKV WR EH VHOI-HYLGHQW, WKDW DOO PHQ DUH FUHDWHG HTXDO.

The Python syntax to both encrypt and decrypt a substitution cipher is presented next. This example shows the use of

```
key = 'abcdefghijklmnopqrstuvwxyz'

def enc_substitution(n, plaintext):
    result = ''
    for l in plaintext.lower():
        try:
            i = (key.index(l) + n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result

def dec_substitution(n, ciphertext):
    result = ''
    for l in ciphertext:
        try:
            i = (key.index(l) - n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result

origtext = 'We hold these truths to be self-evident, that all men are
created equal.'
ciphertext = enc_substitution(13, origtext)
plaintext = dec_substitution(13, ciphertext)
print("Original Text:", origtext)
print("Ciphertext:", ciphertext)
print("Decrypted Text:", plaintext)
```

CODE OUTPUT:

```
Original Text: We hold these truths to be self-evident, that all men are created equal.
Ciphertext: jr ubey gurer gehguf gb or fry-svirqrag, gung nyy zra ner perngrq rdhn y.
Decrypted Text: we hold these truths to be self-evident, that all men are created equal.
```

Useful link to understand Try Except in python.

https://www.w3schools.com/python/python_try_except.asp

```
try:
    print(x)
except:
    print("An exception occurred")
```

```
An exception occurred
```

Practice -Lab Activity 2: Transposition Ciphers:

A simple example for a transposition cipher is columnar transposition cipher where each character in the plain text is written horizontally with specified alphabet width. The cipher is written vertically, which creates an entirely different cipher text. Consider the plain text hello world, and let us apply the simple columnar transposition technique as shown below:

h	e	l	l
o	w	o	r
l	d		

The plain text characters are placed horizontally and the cipher text is created with vertical format as: holewldlo lr. Now, the receiver has to use the same table to decrypt the cipher text to plain text.

Code

The following program code demonstrates the basic implementation of columnar transposition technique:

```
def split_len(seq, length):
    return [seq[i:i + length] for i in range(0, len(seq),
length)]

def encode(key, plaintext):
    order = {
        int(val): num for num, val in enumerate(key)
```

```

    }
    ciphertext = ''
    for index in sorted(order.keys()):
        for part in split_len(plaintext, len(key)):
            try:
                ciphertext += part[order[index]]
            except IndexError:
                pass
    return ciphertext

print(encode('3214', 'HELLO'))

```

Code Output

```
LEHOL
```

Explanation

- Using the function `split_len()`, we can split the plain text characters, which can be placed in columnar or row format.
- `encode` method helps to create cipher text with key specifying the number of columns and prints the cipher text by reading characters through each column.
-

Practice -Lab Activity 3: Caesar Cipher

The Caesar cipher is one of the oldest recorded ciphers. De Vita Caesarum, Divus Iulis (“The Lives of the Caesars, the Deified Julius), commonly known as The Twelve Caesars, was written in approximately 121 CE. In The Twelve Caesars, it states that if someone has a message that they want to keep private, they can do so by changing the order of the letters so that the original word cannot be determined. When the recipient of the message receives it, the reader must substitute the letters so that they shift by four positions.

Simply put, the cipher shifted letters of the alphabet three places forward so that the letter A was replaced with the letter D, the letter B was replaced with E, and so on. Once the end of the alphabet was reached, the letters would start over: \

The Caesar cipher is an example of a mono-alphabet substitution. This type of substitution substitutes one character of the ciphertext from a character in plaintext. Other examples that include this type of substitution are Atbash, Affine, and the ROT-13 cipher. There are many flaws with this type of cipher, the most obvious of which is that the encryption and decryption methods are fixed and require no shared key. This would allow anyone who knew this method to read Caesar’s encrypted messages with ease. Over the years, there have been several variations that include ROT-13, which shifts the letters 13

places instead of 3. We will explore how to encrypt and decrypt Caesar cipher and

ROT-13 codes using Python.

For example, given that x is the current letter of the alphabet, the Caesar cipher function adds three for encryption and subtracts three for decryption. While this could be a variable shift, let's start with the original shift of 3: $\text{Enc}(x) = (x + 3) \% 26$ $\text{Dec}(x) = (x - 3) \% 26$

These functions are the first use of modular arithmetic; there are other ways to get the same result, but this is the cleanest and fastest method. The encryption formula adds 3 to the numeric value of the number. If the value exceeds 26, which is the final position of Z, then the modular arithmetic wraps the value back to the beginning of the alphabet. While it is possible to get the ordinal (`ord`) of a number and convert it back to ASCII, the use of the key simplifies the alphabet indexing. You will learn how to use the `ord()` function when we explore the Vigenère cipher in the next section. In the following Python recipe, the `enc_caesar` function will access a variable index to encrypt the plaintext that is passed in.

```
key = 'abcdefghijklmnopqrstuvwxyz'
def enc_caesar(n, plaintext):
    result = ''
    for l in plaintext.lower():
        try:
            i = (key.index(l) + n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result

plaintext = 'We hold these truths to be self-evident, that all men are
created equal.'
ciphertext = enc_caesar(3, plaintext)
print(ciphertext)
```

The output of this should result in the following:

```
zh krog wkhvh wuxwkv wr eh vhoi-hylghqw, wkdw doo phq duh fuhdwhg htxdo.
```

Decryption

The reverse in this case is straightforward. Instead of adding, we subtract. The decryption would look like the following:

```
key = 'abcdefghijklmnopqrstuvwxyz'
def dec_caesar(n, ciphertext):
    result = ''
    for l in ciphertext:
        try:
            i = (key.index(l) - n) % 26
            result += key[i]
```

```
we hold these truths to be self-evident, that all men are created equal.
```

```
ciphertext = 'zh krog wkhvh wuxwkv wr eh vhoi-hylghqw, wkdw doo phq duh
fuhdwhg htxdo.'
plaintext = dec_caesar(3, ciphertext)
```

Practice -Lab Activity 4: ROT-13

Now that you understand the Caesar cipher, take a look at the ROT-13 cipher. The unique construction of the ROT-13 cipher allows you to encrypt and decrypt using the same method. The reason for this is that since ROT-13 moves the letter of the alphabet exactly halfway, when you run the process again, the letter goes back to its original value.

To see the code behind the cipher, take a look at the following:

```
key = 'abcdefghijklmnopqrstuvwxyz'
def enc_dec_ROT13(n, plaintext):
    result = ""
    for l in plaintext.lower():
        try:
            i = (key.index(l) + n) % 26
            result += key[i]
        except ValueError:
            result += l
    return result
plaintext = 'We hold these truths to be self-evident, that all men are created equal.'
ciphertext = enc_dec_ROT13(13, plaintext)
print(ciphertext)
# Decrypt the ciphertext by running the same function with the same shift of 13
plaintext = enc_dec_ROT13(13, ciphertext)
print(plaintext)
```

Code output:

```
jr ubyq gurfr gehguf gb or frys-rivrag, gung nyy zra ner perngrq rdhny.
we hold these truths to be self-evident, that all men are created equal.
```

Whether we use a Caesar cipher or the ROT-13 variation, brute-forcing an attack would take at most 25 tries, and we could easily decipher the plaintext results when we see a language we understand. This will get more complex as we explore the other historical ciphers; the cryptanalysis requires frequency analysis and language detectors. We will focus on these concepts in upcoming chapters.

Practice -Lab Activity 5: Vigenère Cipher

The Vigenère cipher consists of using several Caesar ciphers in sequence with different shift values. To encipher, a table of alphabets can be used, termed a tabula recta, Vigenère square, or Vigenère table. It consists of the alphabet written out 26 times in

different rows, each alphabet shifted cyclically to the left compared to the previous alphabet, corresponding to the 26 possible Caesar ciphers. At different points in the encryption process, the cipher uses a different alphabet from one of the rows. The alphabet used at each point depends on a repeating keyword.

Here's an example:

Keyword: DECLARATION

D	E	C	L	A	R	A	T	I	O	N
3	4	2	11	0	17	0	19	8	14	13

Plaintext: We hold these truths to be self-evident, that all men are created equal.

Ciphertext: zi jzlu tamgr wwwehj th js fhph-pvzdxvh, gkev llc mxv oeh gtpakew mehdp.

To create a numeric key such as the one shown, use the following syntax. You should see the output

[3, 4, 2, 11, 0, 17, 0, 19, 8, 14, 13]:

```
def key_vigenere(key):
    keyArray = []
    for i in range(0, len(key)):
        keyElement = ord(key[i].upper()) - 65 # Ensure uppercase
        keyArray.append(keyElement)
    return keyArray
secretKey = 'DECLARATION'
key = key_vigenere(secretKey)
print(key)
```

[3, 4, 2, 11, 0, 17, 0, 19, 8, 14, 13]:

```
def shiftEnc(c, n):
    if c.isalpha():
        return chr(((ord(c) - ord('A') + n) % 26) + ord('A'))
    else:
        return c # Keep non-alphabet characters unchanged

def enc_vigenere(plaintext, key):
    secret = ""
    for i in range(len(plaintext)):
        secret += shiftEnc(plaintext[i], key[i % len(key)])
    return secret

def key_vigenere(key):
    keyArray = []
    for i in range(0, len(key)):
        keyElement = ord(key[i].upper()) - 65 # Ensure uppercase
        keyArray.append(keyElement)
```

```

    return keyArray
secretKey = 'DECLARATION'
key = key_vigenere(secretKey)
plaintext = 'ALL MEN ARE CREATED EQUAL'
ciphertext = enc_vigenere(plaintext, key)
print(ciphertext)

```

Code Output:

```
DPN MVN IFR GTPAKEW SDXEN
```

When you know the key, such as in this case, you can decrypt the Vigenère cipher with the following:

```

def shiftDec(c, n):
    if c.isalpha():
        c = c.upper()
        return chr(((ord(c) - ord('A') - n) % 26) + ord('A'))
    else:
        return c # Return non-alphabetic characters unchanged

def dec_vigenere(ciphertext, key):
    plain = ""
    for i in range(len(ciphertext)):
        plain += shiftDec(ciphertext[i], key[i % len(key)])
    return plain

def key_vigenere(key):
    keyArray = []
    for i in range(0, len(key)):
        keyElement = ord(key[i].upper()) - 65 # Ensure uppercase
        keyArray.append(keyElement)
    return keyArray

secretKey = 'DECLARATION'
key = key_vigenere(secretKey)

ciphertext = 'DNP RYR OIU OIKORO VZZIV' # You may replace this with the actual ciphertext
decoded = dec_vigenere(ciphertext, key)

print(decoded)

```

Code Output:

```
AJN RHR GUH KGZAO NLMFR
```

We will perform cryptanalysis by creating a random key that will use the same encryption function, and then we will use frequency analysis to help find the appropriate key. For now, it is more important to understand how the Python code works with this cryptography scheme.

Lab Activity 5: One-Time Pad Function

Now that you have seen how XOR works, it will be easier to understand the one-time pad. OTP takes a random sequence of 0s and 1s as the secret key and will then XOR the key with your plaintext message to produce the ciphertext:

GEN: choose a random key uniformly from $\{0,1\}^\ell$ (the set of binary strings of length ℓ)
ENC: given $k \in \{0,1\}^\ell$ and $m \in \{0,1\}^\ell$ then output is $c := k \oplus m$
DEC: given $k \in \{0,1\}^\ell$ and $c \in \{0,1\}^\ell$, the output message is $m := k \oplus c$

The output given by the OTP satisfies Claude Shannon's notion of perfect secrecy (see "Shannon's Theorem"). Imagine all possible messages, all possible keys, and all possible ciphertexts. For every message and ciphertext pair, there is one key that causes that message to encrypt to that ciphertext. This is really saying that each key gives you a one-to-one mapping from messages to ciphertexts, and changing the key shuffles the mapping without ever repeating a pair.

The OTP remains unbreakable as long as the key meets the following criteria:

- The key is truly random.
- The key the same length as the encrypted message.
- The key is used only once!

When the key is the same length as the encrypted message, each plaintext letter's subkey is unique, meaning that each plaintext letter could be encrypted to any ciphertext letter with equal probability. This removes the ability to use frequency analysis against the encrypted text to learn anything about the cipher. Brute-forcing the OTP would take an incredible amount of time and would be computationally unfeasible, as the number of keys would equal 26 raised to the power of the total number of letters in the message. In Python 3.6 and later, you will have the option to use the `secrets` module, which will allow you to generate random numbers. The function `secrets.randbelow()` will return random numbers between zero and the argument passed to it:

```
>>> import secrets
>>> secrets.randbelow(10)
3
>>> secrets.randbelow(10)
1
>>> secrets.randbelow(10)
7
```

```
*cipher text.py - C:/Users/PAKISTAN/Desktop/cipher text.py (3.11.5)*
File Edit Format Run Options Window Help
import secrets
|

===== RESTART: C:/Users/PAKISTAN/Desktop/cipher text.py =====
Traceback (most recent call last):
  File "C:/Users/PAKISTAN/Desktop/cipher text.py", line 1, in <module>
    secrets.randbelow(10)
NameError: name 'secrets' is not defined
>>>
===== RESTART: C:/Users/PAKISTAN/Desktop/cipher text.py =====
>>>
>>> secrets.randbelow(10)
>>> 2
>>> |
```

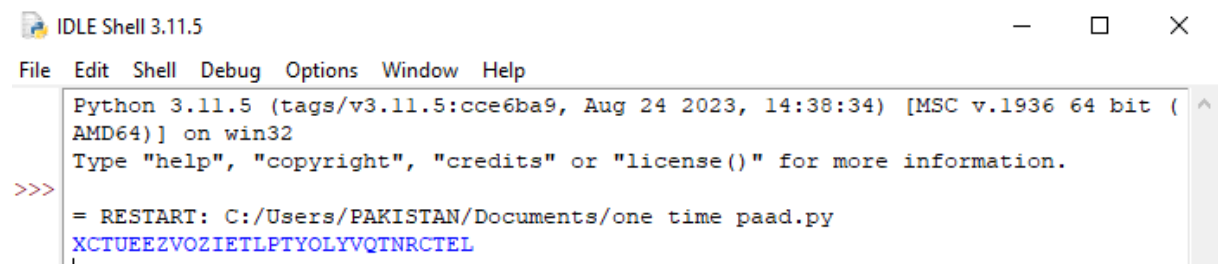
```
IDLE Shell 3.11.5
File Edit Shell Debug Options Window Help
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/PAKISTAN/Desktop/cipher text.py
DPN MVN IFR GTPAKEW SDXEN
>>>
===== RESTART: C:/Users/PAKISTAN/Desktop/cipher text.py =====
AJN RHR GUH KGZAO NLMFR
>>>
===== RESTART: C:/Users/PAKISTAN/Desktop/cipher text.py =====
Traceback (most recent call last):
  File "C:/Users/PAKISTAN/Desktop/cipher text.py", line 1, in <module>
    secrets.randbelow(10)
NameError: name 'secrets' is not defined
>>>
===== RESTART: C:/Users/PAKISTAN/Desktop/cipher text.py =====
>>>
>>> secrets.randbelow(10)
>>> 2
>>> secrets.randbelow(10)
>>> 5
>>> secrets.randbelow(10)
>>> 6
>>> .
```

You can generate a key equal to the length of the message using the following in the Python code.

```
import secrets

msg = "helloworldthisistheonetimepad"
key = ""
for i in range(len(msg)):
    key += secrets.choice('ABCDEFGHIJKLMNOPQRSTUVWXYZ') # Use
uppercase letters
print(key)
```

Code Output:



```
IDLE Shell 3.11.5
File Edit Shell Debug Options Window Help
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/PAKISTAN/Documents/one time paad.py
XCTUEEZVOZIELPTLYOLYVQTNRCTEL
```

TWPIRWVUSCGQPYTAHDNDGPIDTKCTB

One time pad code in python

```
import secrets

def key_generation(length):
    """
    Generates a random binary key of a given length.
    """
    return "".join(secrets.choice('01') for _ in range(length))

def xor_operation(binary_str1, binary_str2):
    """
    Performs bitwise XOR between two binary strings.
    """
    return "".join(str(int(a) ^ int(b)) for a, b in zip(binary_str1, binary_str2))

def encrypt(key, message):
    """
    Encrypts the message using the one-time pad encryption (XOR operation).
    """
    return xor_operation(key, message)

def decrypt(key, ciphertext):
    """
    Decrypts the ciphertext using the one-time pad decryption (XOR operation).
    """
    return xor_operation(key, ciphertext)

# Test the one-time pad algorithm
l = 10 # Length of the binary string (can be set to any desired length)
message = "".join(secrets.choice('01') for _ in range(l)) # Generate a random binary message
```

```

# Key generation
key = key_generation(1)

# Encryption
ciphertext = encrypt(key, message)

# Decryption
decrypted_message = decrypt(key, ciphertext)

# Display the results
print("Message:      ", message)
print("Key:          ", key)
print("Ciphertext:    ", ciphertext)
print("Decrypted Text: ", decrypted_message)

# Ensure the decrypted message matches the original
assert decrypted_message == message, "Decryption failed! The original message and decrypted message don't match."

```

Code Output:

```

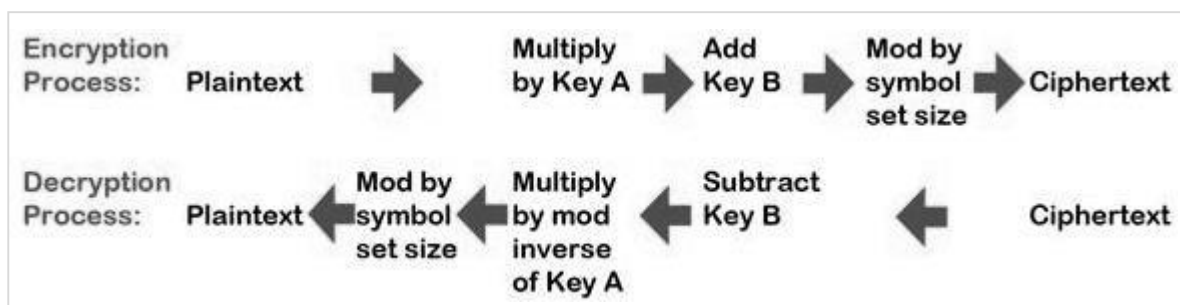
Message:      0111101111
Key:          0010010001
Ciphertext:   0101111110
Decrypted Text: 0111101111

```

1) Home- Graded Lab Tasks

Task 1

Affine Cipher is the combination of Multiplicative Cipher and Caesar Cipher algorithm. The basic implementation of affine cipher is as shown in the image below: Write python code for above mentioned Affine Cipher with code output.



Task 2

While using Caesar cipher technique, encrypting and decrypting symbols involves converting the values into numbers with a simple basic procedure of addition or subtraction. If multiplication is used to convert to cipher text, it is called a wrap-around situation. Consider the letters and the associated numbers to be used as shown below:

0	1	2	3	4	5	6	7	8	9	10	11	12
A	B	C	D	E	F	G	H	I	J	K	L	M
13	14	15	16	17	18	19	20	21	22	23	24	25
N	O	P	Q	R	S	T	U	V	W	X	Y	Z

The numbers will be used for multiplication procedure and the associated key is 7.
The basic formula to be used in such a scenario to generate a multiplicative cipher is

$$(\text{Alphabet Number} * \text{key}) \bmod (\text{total number of alphabets})$$

as follows:

Write Python code for multiplicative cipher and also provide how it can be hacked.
Your programs should include code output mechanism in any form.

Task 3

CrypTool: CrypTool is one of the most comprehensive open-source cryptography tools. It includes tutorials, simulations, and visualizations of classical and modern cryptographic algorithms. You can analyze ciphers, break them using various techniques (e.g., frequency analysis, brute force), and experiment with encryption and decryption. Practice encrypting plaintext using different ciphers and then try decrypting it.

- Select a cipher like Vigenère from the toolbox.
- Enter the plaintext and the key.
- Run the tool to encrypt the message.
- Try decrypting the ciphertext by breaking it using key guessing or cryptanalysis techniques like frequency analysis.

Perform Cryptanalysis: Perform / Explore CrypTool for cryptanalysis of Lab practice activities of ciphers.

- **Frequency Analysis:** Useful for classical ciphers like Caesar or Vigenère.
- **Known-Plaintext Attack:** Some tools allow you to input known plaintext to derive keys.

Task 4

Perform following tasks for following Transposition Cipher code.

```
def split_len(seq, length):
    return [seq[i:i + length] for i in range(0, len(seq),
length)]
```

```
def encode(key, plaintext):
    order = {
        int(val): num for num, val in enumerate(key)
    }
    ciphertext = ''
    for index in sorted(order.keys()):
        for part in split_len(plaintext, len(key)):
            try:
                ciphertext += part[order[index]]
            except IndexError:
                pass
    return ciphertext

print(encode('3214', 'HELLO'))
```

1) Handle Different Key Sizes

Modify the encode function to handle cases where the length of the key is not equal to the length of the plaintext. **Task:** Add padding to the plaintext when it is shorter than the key.

2) Decode Function

Create a decode function that reverses the encode process. **Task:** Write a function `decode(key, ciphertext)` that decipheres the encrypted message and returns the original plaintext.

3) Support for Uppercase and Lowercase Letters

Modify the code to preserve the original case (uppercase and lowercase letters) in the plaintext.

- **Task:** Adjust the encode function to handle both uppercase and lowercase letters, so it doesn't always convert to lowercase.
- ### 4) Encrypt Full Sentences with Spaces
- Modify the encode function to handle spaces and punctuation without removing them.
 - **Task:** Ensure that spaces and punctuation are preserved and not encrypted when encoding full sentences.
- ### 5) Dynamic Key Generation
- Automatically generate a random key if the user does not provide one. **Task:** Write a function that generates a random key based on the length of the plaintext.

6) Add a Menu Interface

Create a simple command-line interface where the user can choose to encode or decode a message. **Task:** Write a menu system where the user can input a choice to either encode, decode, or exit.

Note: Please note that from learning perspective try to work on you own implementation so that in Midterm assessments you will be able to solve challenging task yourself.

Note: The instructor can design graded lab activities according to the level of difficult and complexity of the practice lab activities. The lab tasks assigned by the instructor should be evaluated in the same lab

Lab 04

Streams Ciphers and Pseudo Random Generator (PRG)

This lab will introduce students to Streams Ciphers and Pseudo Random Generator (PRG). We will first learn the working principles of stream ciphers, including how they use a key and a pseudo-random generator to produce a keystream. Implement a basic stream cipher to see how XOR-based encryption works. In this lab students will also learn how PRGs generate keystreams for stream ciphers.

Activity Outcomes:

This lab teaches you the following topics:

- Understand the Basics of Stream Ciphers
- Implement a Simple Stream Cipher (e.g., Vernam/RC4)
- Understand Pseudo-Random Generator (PRG) and Its Role in Stream Ciphers

Practice -Lab Activity 1: ARC4

The RC4 stream cipher was created by Ron Rivest in 1987. RC4 was classified as a trade secret by RSA Security but was eventually leaked to a message board in 1994. RC4 was originally trademarked by RSA Security so it is often referred to as ARCFOUR or ARC4 to avoid trademark issues. ARC4 would later become commonly used in a number of encryption protocols and standards such as SSL, TLS, WEP, and WPA. In 2015, it was prohibited for all versions of TLS by RFC 7465. ARC4 has been used in many hardware and software implementations. One of the main advantages of ARC4 is its speed and simplicity, which you will notice in the following code:

"""

Implement the ARC4 stream cipher.

"""

```
def arc4crypt(data, key):
    x = 0
    box = list(range(256)) # Ensure box is a list, not range object
    # Key-scheduling algorithm (KSA)
    for i in range(256):
        x = (x + box[i] + ord(key[i % len(key)])) % 256
        box[i], box[x] = box[x], box[i] # Swap values

    x = 0
    y = 0
    out = []
    # Pseudo-random generation algorithm (PRGA)
    for char in data:
        x = (x + 1) % 256
        y = (y + box[x]) % 256
        box[x], box[y] = box[y], box[x] # Swap values
        out.append(chr(ord(char) ^ box[(box[x] + box[y]) % 256])) # XOR with
        keystream

    return ''.join(out)

# Testing the ARC4 encryption and decryption
key = 'SuperSecretKey!!'
origtext = 'Dive Dive Dive'
ciphertext = arc4crypt(origtext, key)
plaintext = arc4crypt(ciphertext, key)

print("The original text is: {}".format(origtext))
print("The ciphertext is: {}".format(ciphertext))
print("The plaintext is: {}".format(plaintext))
```

OUTPUT

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
PS C:\Users\PAKISTAN> & C:/Python311/python.exe c:/Users/PAKISTAN/Documents/RC4.py
The original text is: Dive Dive Dive
The ciphertext is: 'WU' »~«3â~!W
The plaintext is: Dive Dive Dive
PS C:\Users\PAKISTAN> 
```

Practice -Lab Activity 2: Vernam Cipher

The Vernam cipher was developed by Gilbert Vernam in 1917. It is a type of onetime pad for data streams and is considered to be unbreakable. The algorithm is symmetrical, and the plaintext is combined with a random stream of data of the same length using the Boolean XOR function; the Boolean XOR function is also known as the Boolean exclusive OR function. Claude Shannon would later mathematically prove that it is unbreakable. The characteristics of the Vernam cipher include:

- The plaintext is written as a binary sequence of 0s and 1s.
- The secret key is a completely random binary sequence and is the same length as the plaintext.
- The ciphertext is produced by adding the secret key bitwise modulo 2 to the plaintext.
- XOR Operation:
 - XOR is a bitwise operation, and it's symmetric. This means:
 - $a \oplus b$ will encrypt a with b .
 - Applying XOR again with b reverses the operation: $(a \oplus b) \oplus b = a$.
 - This is why using the same function for both encryption and decryption works in the Vernam cipher.

One of the disadvantages of using an OTP is that the keys must be as long as the message it is trying to conceal; therefore, for long messages, you will need a long key:

CODE

```
def VernamEncDec(text, key):
    result = ""
    ptr = 0
    for char in text:
        result += chr(ord(char) ^ ord(key[ptr])) # XOR each character with the key
        ptr += 1
        if ptr == len(key): # Reset the key pointer if it reaches the end of the key
            ptr = 0
    return result

# Key for Vernam Cipher
key = "thisismykey12345"

while True:
```

```

input_text = input("\nEnter Text To Encrypt:\t")

# Encrypt the input text
ciphertext = VernamEncDec(input_text, key)
print("\nEncrypted Vernam Cipher Text:\t" + ciphertext)

# Decrypt the ciphertext
plaintext = VernamEncDec(ciphertext, key)
print("\nDecrypted Vernam Cipher Text:\t" + plaintext)

```

OUTPUT

```

===== RESTART: C:/Users/PAKISTAN/Documents/Vernam.py =====

Enter Text To Encrypt: Cryptographic protocols

Encrypted Vernam Cipher Text:  7
[]
[]xQ[]dg[]

Decrypted Vernam Cipher Text: Cryptographic protocols

```

Home-Practice - Activity 3: Salsa20 Cipher

The Salsa20 cipher was developed in 2005 by Daniel Bernstein, and submitted to eSTREAM. The Salsa20/20 (Salsa20 with 20 rounds) is built on a pseudorandom function that is based on add-rotate-xor (ARX) operations. ARX algorithms are designed to have their round function support modular addition, fixed rotation, and XOR. These ARX operations are popular because they are relatively fast and cheap in hardware and software, and because they run in constant time, and are therefore immune to timing attacks. The rotational cryptanalysis technique attempts to attack such round functions.

The core function of Salsa20 maps a 128-bit or 256-bit key, a 64-bit nonce/IV, and a 64-bit counter to a 512-bit block of the keystream. Salsa20 provides speeds of around 4–14 cycles per byte on modern x86 processors and is considered acceptable hardware performance. The numeric indicator in the Salsa name specifies the number of encryption rounds. Salsa20 has 8, 12, and 20 variants. One of the biggest benefits of Salsa20 is that Bernstein has written several implementations that have been released to the public domain, and the cipher is not patented.

Salsa20 is composed of sixteen 32-bit words that are arranged in a 4×4 matrix. The initial state is made up of eight words of key, two words of the stream position, two words for the nonce/IV, and four fixed words or constants. The initial state would look like the following:

Constant	Key	Key	Key
Key	Constant	Nonce	Nonce
Stream	Stream	Constant	Key
Key	Key	Key	Constant

The Salsa20 core operation is the quarter-round that takes a four-word input and produces a four-word output. The quarter-round is denoted by the following function: $QR(a, b, c, d)$. The odd-numbered rounds apply $QR(a, b, c, d)$ to each of the four columns in the preceding 4×4 matrix; the even-numbered rounds apply the rounding to each of the four rows. Two consecutive rounds (one for a column and one for a row) operate together and are known as a double-round. To help understand how the rounds work, let us first examine a 4×4 matrix with labels from 0 to 15:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

The first double-round starts with a quarter-round on column 1 and row 1. The first QR round examines column 1, which contains 0, 4, 8, and 12. The second QR round examines row 1, which contains 0, 1, 2, 3. The second double-round picks up starting at the second column and second row position (5, 9, 13, 1). The even round picks up 5, 5, 7, 4. Notice that the starting cell is the same for each double-round:

DOUBLE-ROUND 1	DOUBLE-ROUND 3
QR(0, 4, 8, 12)	QR(10, 14, 2, 6)
QR(0, 1, 2, 3)	QR(10, 11, 8, 9)
DOUBLE-ROUND 2	DOUBLE-ROUND 4
QR(5, 9, 13, 1)	QR(15, 3, 7, 11)
QR(5, 6, 7, 4)	QR(15, 12, 13, 14)

A couple libraries are available that will help simplify the Salsa20 encryption scheme. You can access the salsa20 library by doing a `pip install salsa20`.

Once you have the library installed, you can use the `XSalsa20_keystream` to generate a keystream of the desired length, or you can pass any message plaintext or ciphertext) to have it XOR'd with the keystream. All values must be binary strings that include `str` for Python 2 or the `byte` for Python 3. Here, you will see a [Python implementation of the salsa20 library](#):

```
from salsa20 import XSalsa20_xor
from os import urandom
IV = urandom(24)
```

```

KEY = b'*secret**secret**secret**secret*'
ciphertext = XSalsa20_xor(b"IT'S A YELLOW SUBMARINE", IV, KEY)
print(XSalsa20_xor(ciphertext, IV, KEY).decode())

from nacl.secret import SecretBox
from nacl.utils import random

# The key must be 32 bytes for XSalsa20
key = b'*secret**secret**secret**secret*'

# Create a SecretBox, which uses XSalsa20 internally
box = SecretBox(key)

# The nonce must be 24 bytes for XSalsa20
nonce = random(24)

# Encrypting the message
message = b"IT'S A YELLOW SUBMARINE"
ciphertext = box.encrypt(message, nonce)

# Decrypting the message
decrypted = box.decrypt(ciphertext)

print(decrypted.decode()) # Should output: IT'S A YELLOW
SUBMARINE

```

Code Output

```
IT'S A YELLOW SUBMARINE
```

One of the reasons why you should be familiar with Salsa20 is that it is consistently faster than AES. It is recommended to use Salsa20 for encryption in typical cryptographic applications.

Home-Task - Activity 4: Salsa20 Cipher

1. Write python code for your designed stream cipher approach for encryption decryption, you can use approach from more than one already developed ciphers as given in lab practice exercises.
2. Design and implement an adversarial attack approach for your proposed stream cipher approach.

Lab 05

Block Ciphers

Objective:

Stream ciphers work by generating pseudorandom bits and XORing them with your message. Block ciphers take in a fixed-length message, a private key, and they produce a ciphertext that is the same length as the fixed-length plaintext message. In this Lab students will understand working of DES, Triple DES AES.

Activity Outcomes:

- How to learn coding of cryptographic block ciphers like DES, and AES.
- How to code and analyze attack scenarios for complex block ciphers.

<i>Sr.No</i>	<i>Allocated Time</i>	<i>Level of Complexity</i>	<i>CLO Mapping</i>
<i>1</i>	<i>80</i>	<i>High</i>	<i>CLO-4</i>
<i>2</i>	<i>40</i>	<i>Medium</i>	
<i>3</i>	<i>50</i>	<i>High</i>	

Practice -Lab Activity 1: DES

AES and Triple DES are the most common block ciphers in use today. From the student's point of view, DES is still interesting to study, but due to its small 56-bit key size, it is considered insecure. In 1999, two partners, Electronic Frontier Foundation and distributed.net collaborated to publicly break a DES key in 22 hours and 15 minutes. Here, we will use the PyCrypto library to demonstrate how to use DES to encrypt a message. The following code is using the ECB block mode; you will learn about the various modes later in this chapter. To execute the following recipe, perform a pip install PyCrypto:

```
from Crypto.Cipher import DES
key = b'shhhhhh!'
origText = b'The US Navy has submarines in Kingsbay!!'
des = DES.new(key, DES.MODE_ECB)
ciphertext = des.encrypt(origText)
plaintext = des.decrypt(ciphertext)
print('The original text is {}'.format(origText))
print('The ciphertext is {}'.format(ciphertext))
print('The plaintext is {}'.format(plaintext))
print()
```

This should produce the following output:

```
The original text is b'The US Navy has submarines in Kingsbay!!'
The ciphertext is b'\xf6\x0bb\xf9L\x15I\xf9\x0f\xe2\xee_\^xdaQX\xe1y\
xe5\xea\xd3Z\xc8y\xee\xd3\x86H\xf0Nn\x83\x93\nOd@6H\xd4'
The plaintext is b'The US Navy has submarines in Kingsbay!!'
Press any key to continue . . .
```

The key was 'shhhhhh!' and the message was 'The US Navy has submarines in Kingsbay!!'. The ciphertext was 40 bytes long; $40 \bmod 8 = 0$, so there is no need to pad this example. If you were to implement a block cipher in reality, you should use a padding function that ensures the block length.

What is DES?

Data Encryption Standard (DES) is a block cipher with a 56-bit key length that has played a significant role in data security. Data encryption standard (DES) has been found vulnerable to very powerful attacks therefore, the popularity of DES has been found slightly on the decline. DES is a block cipher and encrypts data in blocks of size of **64 bits** each, which means 64 bits of plain text go as the input to DES, which produces 64 bits of ciphertext. The same algorithm and key are used for encryption and [decryption](#), with minor differences. The key length is **56 bits**.

The basic idea is shown below:

We have mentioned that DES uses a 56-bit key. Actually, The initial key consists of 64 bits. However, before the DES process even starts, every 8th bit of the key is discarded to produce a 56-bit key. That is bit positions 8, 16, 24, 32, 40, 48, 56, and 64 are discarded.

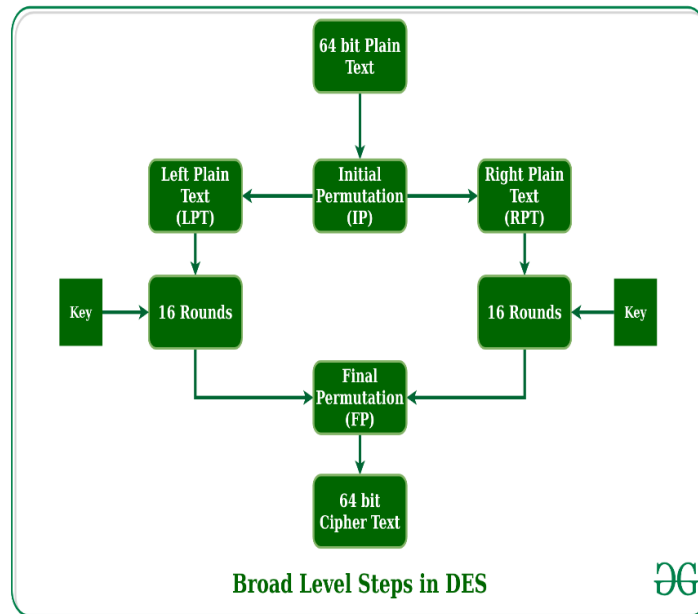
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64

Figure - discarding of every 8th bit of original key

Thus, the discarding of every 8th bit of the key produces a **56-bit key** from the original **64-bit key**.

DES is based on the two fundamental attributes of [cryptography](#): substitution (also called confusion) and transposition (also called diffusion). DES consists of 16 steps, each of which is called a round. Each round performs the steps of substitution and transposition. Let us now discuss the broad-level steps in DES.

- In the first step, the 64-bit plain text block is handed over to an initial [Permutation](#) (IP) function.
- The initial permutation is performed on plain text.
- Next, the initial permutation (IP) produces two halves of the permuted block; saying Left Plain Text (LPT) and Right Plain Text (RPT).
- Now each LPT and RPT go through 16 rounds of the encryption process.
- In the end, LPT and RPT are rejoined and a Final Permutation (FP) is performed on the combined block
- The result of this process produces 64-bit ciphertext.



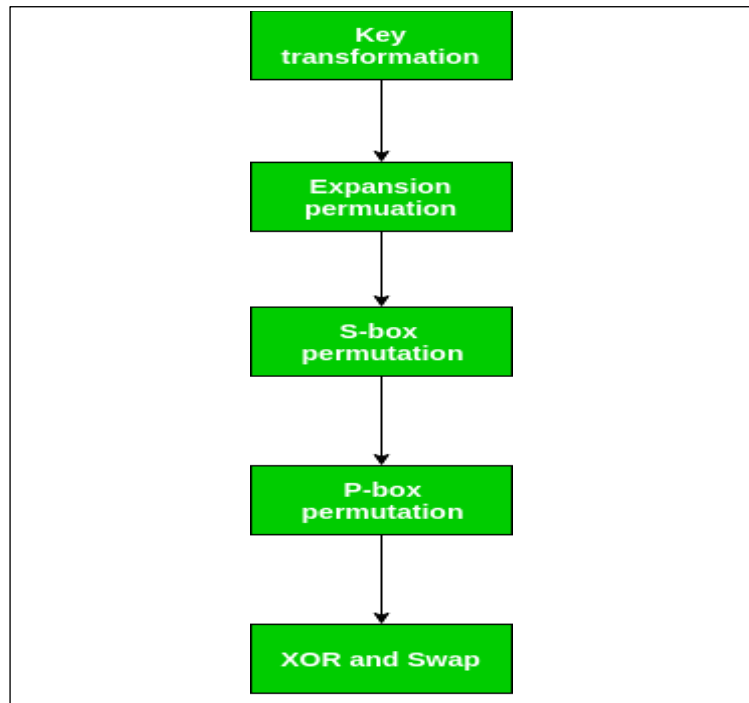
Initial Permutation (IP)

As we have noted, the initial permutation (IP) happens only once and it happens before the first round. It suggests how the transposition in IP should proceed, as shown in the figure. For example, it says that the IP replaces the first bit of the original plain text block with the 58th bit of the original plain text, the second bit with the 50th bit of the original plain text block, and so on. This is nothing but jugglery of bit positions of the original plain text block. The same rule applies to all the other bit positions shown in the figure.

58	50	42	34	26	18	10	2	60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6	64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1	59	51	43	35	27	19	11	3
61	33	45	37	29	21	13	5	63	55	47	39	31	23	15	7

Figure - Initial permutation table

As we have noted after IP is done, the resulting 64-bit permuted text block is divided into two half blocks. Each half-block consists of 32 bits, and each of the 16 rounds, in turn, consists of the broad-level steps outlined in the figure.



Step 1: Key transformation

We have noted initial 64-bit key is transformed into a 56-bit key by discarding every 8th bit of the initial key. Thus, for each a 56-bit key is available. From this 56-bit key, a different 48-bit Sub Key is generated during each round using a process called key transformation. For this, the 56-bit key is divided into two halves, each of 28 bits. These halves are circularly shifted left by one or two positions, depending on the round.

For example: if the round numbers 1, 2, 9, or 16 the shift is done by only one position for other rounds, the circular shift is done by two positions. The number of key bits shifted per round is shown in the figure.

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
#key bits shifted	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Figure - number of key bits shifted per round

After an appropriate shift, 48 of the 56 bits are selected. From the 48 we might obtain 64 or 56 bits based on requirement which helps us to recognize that this model is very versatile and can handle any range of requirements needed or provided. for selecting 48 of the 56 bits the table is shown in the figure given below. For instance, after the shift, bit number 14 moves to the first position, bit number 17 moves to the second position, and so on. If we observe the table, we will realize that it contains only 48-bit positions. Bit number 18 is discarded (we will not find it in the table), like 7 others, to reduce a 56-bit key to a 48-bit key. Since the key transformation process involves permutation as well as a selection of a 48-bit subset of the original 56-bit key it is called Compression Permutation.

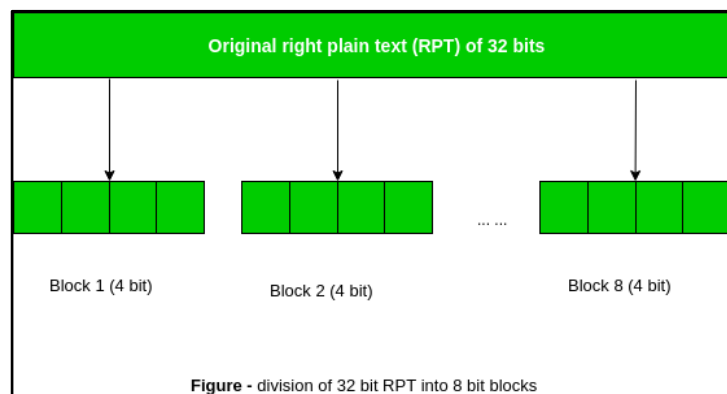
14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

Figure - compression permutation

Because of this compression permutation technique, a different subset of key bits is used in each round. That makes DES not easy to crack.

Step 2: Expansion Permutation

Recall that after the initial permutation, we had two 32-bit plain text areas called Left Plain Text(LPT) and Right Plain Text(RPT). During the expansion permutation, the RPT is expanded from 32 bits to 48 bits. Bits are permuted as well hence called expansion permutation. This happens as the 32-bit RPT is divided into 8 blocks, with each block consisting of 4 bits. Then, each 4-bit block of the previous step is then expanded to a corresponding 6-bit block, i.e., per 4-bit block, 2 more bits are added.



This process results in expansion as well as a permutation of the input bit while creating output. The key transformation process compresses the 56-bit key to 48 bits. Then the expansion permutation process expands the **32-bit RPT** to **48-bits**. Now the 48-bit key is XOR with 48-bit RPT and the resulting output is given to the next step, which is the **S-Box substitution**.

Python3 code for the above approach

```
# Hexadecimal to binary conversion
```

```
def hex2bin(s):
    mp = {'0': "0000",
          '1': "0001",
          '2': "0010",
          '3': "0011",
```

```

        '4': "0100",
        '5': "0101",
        '6': "0110",
        '7': "0111",
        '8': "1000",
        '9': "1001",
        'A': "1010",
        'B': "1011",
        'C': "1100",
        'D': "1101",
        'E': "1110",
        'F': "1111"}
    bin = ""
    for i in range(len(s)):
        bin = bin + mp[s[i]]
    return bin

```

Binary to hexadecimal conversion

```

def bin2hex(s):
    mp = {"0000": '0',
          "0001": '1',
          "0010": '2',
          "0011": '3',
          "0100": '4',
          "0101": '5',
          "0110": '6',
          "0111": '7',
          "1000": '8',
          "1001": '9',
          "1010": 'A',
          "1011": 'B',
          "1100": 'C',
          "1101": 'D',
          "1110": 'E',
          "1111": 'F'}
    hex = ""
    for i in range(0, len(s), 4):
        ch = ""
        ch = ch + s[i]
        ch = ch + s[i + 1]
        ch = ch + s[i + 2]
        ch = ch + s[i + 3]
        hex = hex + mp[ch]

    return hex

```

Binary to decimal conversion

```

def bin2dec(binary):

    binary1 = binary
    decimal, i, n = 0, 0, 0
    while(binary != 0):
        dec = binary % 10
        decimal = decimal + dec * pow(2, i)
        binary = binary//10
        i += 1
    return decimal

# Decimal to binary conversion

def dec2bin(num):
    res = bin(num).replace("0b", "")
    if(len(res) % 4 != 0):
        div = len(res) / 4
        div = int(div)
        counter = (4 * (div + 1)) - len(res)
        for i in range(0, counter):
            res = '0' + res
    return res

# Permute function to rearrange the bits

def permute(k, arr, n):
    permutation = ""
    for i in range(0, n):
        permutation = permutation + k[arr[i] - 1]
    return permutation

# shifting the bits towards left by nth shifts

def shift_left(k, nth_shifts):
    s = ""
    for i in range(nth_shifts):
        for j in range(1, len(k)):
            s = s + k[j]
        s = s + k[0]
        k = s
        s = ""
    return k

# calculating xow of two strings of binary number a and b

```



```

def xor(a, b):
    ans = ""
    for i in range(len(a)):
        if a[i] == b[i]:
            ans = ans + "0"
        else:
            ans = ans + "1"
    return ans

# Table of Position of 64 bits at initial level: Initial Permutation Table
initial_perm = [58, 50, 42, 34, 26, 18, 10, 2,
                60, 52, 44, 36, 28, 20, 12, 4,
                62, 54, 46, 38, 30, 22, 14, 6,
                64, 56, 48, 40, 32, 24, 16, 8,
                57, 49, 41, 33, 25, 17, 9, 1,
                59, 51, 43, 35, 27, 19, 11, 3,
                61, 53, 45, 37, 29, 21, 13, 5,
                63, 55, 47, 39, 31, 23, 15, 7]

# Expansion D-box Table
exp_d = [32, 1, 2, 3, 4, 5, 4, 5,
         6, 7, 8, 9, 8, 9, 10, 11,
         12, 13, 12, 13, 14, 15, 16, 17,
         16, 17, 18, 19, 20, 21, 20, 21,
         22, 23, 24, 25, 24, 25, 26, 27,
         28, 29, 28, 29, 30, 31, 32, 1]

# Straight Permutation Table
per = [16, 7, 20, 21,
       29, 12, 28, 17,
       1, 15, 23, 26,
       5, 18, 31, 10,
       2, 8, 24, 14,
       32, 27, 3, 9,
       19, 13, 30, 6,
       22, 11, 4, 25]

# S-box Table
sbox = [[[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
         [0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
         [4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
         [15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13]],

        [[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
         [3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
         [0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
         [13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9]],

        [[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],

```

```

[13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
[13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
[1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12]],

[[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
[13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
[10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
[3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14]],

[[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
[14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
[4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
[11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3]],

[[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],
[10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],
[9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],
[4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13]],

[[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
[13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
[1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
[6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12]],

[[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
[1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
[7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
[2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11]]]

```

```
# Final Permutation Table
```

```
final_perm = [40, 8, 48, 16, 56, 24, 64, 32,
              39, 7, 47, 15, 55, 23, 63, 31,
              38, 6, 46, 14, 54, 22, 62, 30,
              37, 5, 45, 13, 53, 21, 61, 29,
              36, 4, 44, 12, 52, 20, 60, 28,
              35, 3, 43, 11, 51, 19, 59, 27,
              34, 2, 42, 10, 50, 18, 58, 26,
              33, 1, 41, 9, 49, 17, 57, 25]
```

```
def encrypt(pt, rkb, rk):
```

```
    pt = hex2bin(pt)
```

```
    # Initial Permutation
```

```
    pt = permute(pt, initial_perm, 64)
```

```
    print("After initial permutation", bin2hex(pt))
```

```
    # Splitting
```

```
    left = pt[0:32]
```

```
    right = pt[32:64]
```

```

    for i in range(0, 16):
        # Expansion D-box: Expanding the 32 bits data into 48 bits
        right_expanded = permute(right, exp_d, 48)

        # XOR RoundKey[i] and right_expanded
        xor_x = xor(right_expanded, rkb[i])

        # S-boxes: substituting the value from s-box table by calculating row
and column
        sbox_str = ""
        for j in range(0, 8):
            row = bin2dec(int(xor_x[j * 6] + xor_x[j * 6 + 5]))
            col = bin2dec(
                int(xor_x[j * 6 + 1] + xor_x[j * 6 + 2] + xor_x[j * 6 + 3]
+ xor_x[j * 6 + 4]))
            val = sbox[j][row][col]
            sbox_str = sbox_str + dec2bin(val)

        # Straight D-box: After substituting rearranging the bits
        sbox_str = permute(sbox_str, per, 32)

        # XOR left and sbox_str
        result = xor(left, sbox_str)
        left = result

        # Swapper
        if(i != 15):
            left, right = right, left
        print("Round ", i + 1, " ", bin2hex(left),
            " ", bin2hex(right), " ", rk[i])

    # Combination
    combine = left + right

    # Final permutation: final rearranging of bits to get cipher text
    cipher_text = permute(combine, final_perm, 64)
    return cipher_text

pt = "123456ABCD132536"
key = "AABB09182736CCDD"

# Key generation
# --hex to binary
key = hex2bin(key)

# --parity bit drop table
keyp = [57, 49, 41, 33, 25, 17, 9,
        1, 58, 50, 42, 34, 26, 18,
        10, 2, 59, 51, 43, 35, 27,

```

```

        19, 11, 3, 60, 52, 44, 36,
        63, 55, 47, 39, 31, 23, 15,
        7, 62, 54, 46, 38, 30, 22,
        14, 6, 61, 53, 45, 37, 29,
        21, 13, 5, 28, 20, 12, 4]

# getting 56 bit key from 64 bit using the parity bits
key = permute(key, keyp, 56)

# Number of bit shifts
shift_table = [1, 1, 2, 2,
               2, 2, 2, 2,
               1, 2, 2, 2,
               2, 2, 2, 1]

# Key- Compression Table : Compression of key from 56 bits to 48 bits
key_comp = [14, 17, 11, 24, 1, 5,
            3, 28, 15, 6, 21, 10,
            23, 19, 12, 4, 26, 8,
            16, 7, 27, 20, 13, 2,
            41, 52, 31, 37, 47, 55,
            30, 40, 51, 45, 33, 48,
            44, 49, 39, 56, 34, 53,
            46, 42, 50, 36, 29, 32]

# Splitting
left = key[0:28] # rkb for RoundKeys in binary
right = key[28:56] # rk for RoundKeys in hexadecimal

rkb = []
rk = []
for i in range(0, 16):
    # Shifting the bits by nth shifts by checking from shift table
    left = shift_left(left, shift_table[i])
    right = shift_left(right, shift_table[i])

    # Combination of left and right string
    combine_str = left + right

    # Compression of key from 56 to 48 bits
    round_key = permute(combine_str, key_comp, 48)

    rkb.append(round_key)
    rk.append(bin2hex(round_key))

print("Encryption")
cipher_text = bin2hex(encrypt(pt, rkb, rk))
print("Cipher Text : ", cipher_text)

print("Decryption")

```

```
rkb_rev = rkb[::-1]
rk_rev = rk[::-1]
text = bin2hex(encrypt(cipher_text, rkb_rev, rk_rev))
print("Plain Text : ", text)
```

Output:

...60AF7CA5

Round 12 FF3C485F 22A5963B C2C1E96A4BF3

Round 13 22A5963B 387CCDAA 99C31397C91F

Round 14 387CCDAA BD2DD2AB 251B8BC717D0

Round 15 BD2DD2AB CF26B472 3330C5D9A36D

Round 16 19BA9212 CF26B472 181C5D75C66D

Cipher Text: C0B7A8D05F3A829C

Decryption

After initial permutation: 19BA9212CF26B472

After splitting: L0=19BA9212 R0=CF26B472

Round 1 CF26B472 BD2DD2AB 181C5D75C66D

Round 2 BD2DD2AB 387CCDAA 3330C5D9A36D

Round 3 387CCDAA 22A5963B 251B8BC717D0

Round 4 22A5963B FF3C485F 99C31397C91F

Round 5 FF3C485F 6CA6CB20 C2C1E96A4BF3

Round 6 6CA6CB20 10AF9D37 6D5560AF7CA5

Round 7 10AF9D37 308BEE97 02765708B5BF

Round 8 308BEE97 A9FC20A3 84BB4473DCCC

Round 9 A9FC20A3 2E8F9C65 34F822F0C66D

Round 10 2E8F9C65 A15A4B87 708AD2DDB3C0

Round 11 A15A4B87 236779C2 C1948E87475E

Round 12 236779C2 B8089591 69A629FEC913

Round 13 B8089591 4A1210F6 DA2D032B6EE3

Round 14 4A1210F6 5A78E394 06EDA4ACF5B5
Round 15 5A78E394 18CA18AD 4568581ABCCE
Round 16 14A7D678 18CA18AD 194CD072DE8C

Plain Text: 123456ABCD132536

Output:

Encryption:

After initial permutation: 14A7D67818CA18AD

After splitting: L0=14A7D678 R0=18CA18AD

Round 1 18CA18AD 5A78E394 194CD072DE8C
Round 2 5A78E394 4A1210F6 4568581ABCCE
Round 3 4A1210F6 B8089591 06EDA4ACF5B5
Round 4 B8089591 236779C2 DA2D032B6EE3
Round 5 236779C2 A15A4B87 69A629FEC913
Round 6 A15A4B87 2E8F9C65 C1948E87475E
Round 7 2E8F9C65 A9FC20A3 708AD2DDB3C0
Round 8 A9FC20A3 308BEE97 34F822F0C66D
Round 9 308BEE97 10AF9D37 84BB4473DCCC
Round 10 10AF9D37 6CA6CB20 02765708B5BF
Round 11 6CA6CB20 FF3C485F 6D5560AF7CA5
Round 12 FF3C485F 22A5963B C2C1E96A4BF3
Round 13 22A5963B 387CCDAA 99C31397C91F
Round 14 387CCDAA BD2DD2AB 251B8BC717D0
Round 15 BD2DD2AB CF26B472 3330C5D9A36D
Round 16 19BA9212 CF26B472 181C5D75C66D

Cipher Text: C0B7A8D05F3A829C

Decryption

After initial permutation: 19BA9212CF26B472

After splitting: L0=19BA9212 R0=CF26B472

Round 1 CF26B472 BD2DD2AB 181C5D75C66D
Round 2 BD2DD2AB 387CCDAA 3330C5D9A36D
Round 3 387CCDAA 22A5963B 251B8BC717D0
Round 4 22A5963B FF3C485F 99C31397C91F
Round 5 FF3C485F 6CA6CB20 C2C1E96A4BF3
Round 6 6CA6CB20 10AF9D37 6D5560AF7CA5
Round 7 10AF9D37 308BEE97 02765708B5BF
Round 8 308BEE97 A9FC20A3 84BB4473DCCC
Round 9 A9FC20A3 2E8F9C65 34F822F0C66D
Round 10 2E8F9C65 A15A4B87 708AD2DDB3C0
Round 11 A15A4B87 236779C2 C1948E87475E
Round 12 236779C2 B8089591 69A629FEC913
Round 13 B8089591 4A1210F6 DA2D032B6EE3
Round 14 4A1210F6 5A78E394 06EDA4ACF5B5
Round 15 5A78E394 18CA18AD 4568581ABCCE

Round 16 14A7D678 18CA18AD 194CD072DE8C
Plain Text: 123456ABCD132536

Graded Task 1

You have implemented DES there is built in implemented DES in python in crypto cipher module use it for encryption/decryption and provide output sample example

```
from Crypto.Cipher import DES
from Crypto.Random import get_random_bytes
from Crypto.Util.Padding import pad, unpad

# DES key must be exactly 8 bytes long
key = get_random_bytes(8)

def des_encrypt(data, key):
    cipher = DES.new(key, DES.MODE_ECB)
    padded_data = pad(data, DES.block_size)
    encrypted_data = cipher.encrypt(padded_data)
    return encrypted_data

def des_decrypt(encrypted_data, key):
    cipher = DES.new(key, DES.MODE_ECB)
    decrypted_data = unpad(cipher.decrypt(encrypted_data), DES.block_size)
    return decrypted_data

# Example usage
if __name__ == "__main__":
    # Input data (must be bytes)
    data = b"Secret123"

    print(f"Original Data: {data}")

    # Encrypt the data
    encrypted_data = des_encrypt(data, key)
    print(f"Encrypted Data: {encrypted_data}")

    # Decrypt the data
    decrypted_data = des_decrypt(encrypted_data, key)
    print(f"Decrypted Data: {decrypted_data}")
```

Graded Task 2

Visualization of MITM Attack Flow:

Attacker intercepts plaintext (P) and ciphertext (C)

1. Guess K1
P --[Encrypt with K1]--> Intermediate Value 1 (I1)
2. Guess K2
C --[Decrypt with K2]--> Intermediate Value 2 (I2)
3. If I1 == I2, then K1 and K2 are likely correct.
Found the DES key: $K = K1 + K2$

Assumptions:

- The attacker knows or can guess some **plaintext-ciphertext pairs** (this is called a **known-plaintext attack**).
- The DES encryption and decryption processes can be divided into independent stages, allowing the attacker to perform partial encryption and decryption.

Steps in Meet-in-the-Middle Attack:

1. **Intercept the Ciphertext:** The attacker intercepts a **known plaintext** (i.e., a piece of the original message that is known or can be guessed) and the corresponding **ciphertext** (i.e., the encrypted version of that message).
2. **Divide the DES Algorithm into Two Stages:**
 - DES operates in multiple rounds of encryption, but the MITM attack divides this into two stages:
 - 1. **First encryption stage** (Encrypt a known plaintext).
 - 2. **Second decryption stage** (Decrypt a known ciphertext).

The attack leverages the fact that the encryption can be split into these stages, and the intermediate value after one encryption round should match with the decrypted intermediate value after one decryption round.

3. **Guess the First Half of the Key:** The attacker guesses the first half of the key (let's call it K1). The attacker encrypts the known plaintext using K1 and stores the result (intermediate encryption value).
4. **Guess the Second Half of the Key:** The attacker guesses the second half of the key (let's call it K2). The attacker decrypts the intercepted ciphertext using K2 and stores the result (intermediate decryption value).
5. **Matching Intermediate Values:**
 - The attacker compares the intermediate values from the first stage (encrypting with K1) and the second stage (decrypting with K2).

- If the intermediate values match, the attacker has likely found the correct combination of the two keys (K_1 and K_2), which together form the full DES key.
- Since DES uses a single 56-bit key, this is broken down into halves for this type of attack.

Home –Task: Advanced Encryption Standard (AES)

AES stands for Advanced Encryption Standard, and it is the only public encryption scheme that the NSA approves for confidential information. We focus on its use as our main block cipher from now on. AES is the current de facto block cipher, and it works on 16 bytes at a time. It has three possible key lengths: 16-byte, 24-byte, or 32-byte. We know that a block cipher is effectively a deterministic permutation on binary strings, like a fixed-length reversible hash. Given a proper-length key and a 16-byte input we should always get the same 16-byte output. Note that there are typically three ways to work with bytes: plain ASCII, hex digest, and base64 (we haven't played with this yet but we will). A good chunk of your bugs come from transferring between hex and raw. You explore AES in the next chapter as you manipulate images.

Using AES with Python Earlier in this chapter, you were introduced to PyCrypto as a Python module that enables block ciphers using DES; it also has methods for encrypting AES. The PyCrypto module is similar to the Java Cryptography Extension (JCE) that is used in Java. The first step we will take in our AES encryption is to generate a strong key. As you know, the stronger the key, the stronger the encryption. The key we use for our encryption is oftentimes the weakest link in our encryption chain. The key we select should not be guessable and should provide sufficient entropy, which simply means that the key should lack order or predictability. The following Python code will create a random key that is 16 bytes:

```
import os
import binascii
key = binascii.hexlify(os.urandom(16))
print('key', [x for x in key] )
key [97, 53, 99, 97, 102, 99, 102, 102, 50, 101, 98, 57, 97, 51, 50, 50,
51, 52, 102, 49, 101, 51, 102, 52, 100, 49, 48, 51, 51, 49, 56, 51]
```

Now that you have generated a key, you will need an initialization vector. The IV should be generated for each message to ensure a different encrypted text each time the message is encrypted. The IV adds significant protection in case the message is intercepted; it should mitigate the use of cryptanalysis to infer message or key data. The IV is required to be transmitted to the message receiver to ensure proper decryption, but unlike the message key, the IV does not need to be kept secret. You can add the IV to process the encrypted text. The message receiver will need to know where the IV is located inside the message. You can create a random IV by using the following snippet; note the use of `random.randint`. This method of generating random numbers is less effective and it has a lower entropy, but in this case we are using it to create the IV that will be used in the encryption process so there is less concern with the use of `randint` here:

```
iv = "".join([chr(random.randint(0, 0xFF)) for i in range(16)])
```

The next step in the process is to create the ciphertext. In this example, we will use the CBC mode; this links the current block to the previous block in the stream. See the previous section to review the various AES block modes.

Remember that for this implementation of AES using PyCrypto, you will need to ensure that you pad the block to guarantee you have enough data in the block:

```
aes = AES.new(key, AES.MODE_CBC, iv)
data = 'Playing with AES' # <- 16 bytes
encd = aes.encrypt(data)
```

To decrypt the ciphertext, you will need the key that was used for the encryption. Transporting the key, inside itself, can be a challenge. You will learn about key exchange in a later chapter. In addition to the key, you will also need the IV. The IV can be transmitted over any line of communication as there are no requirements to encrypt it. You can safely send the IV along with the encrypted file and embed it in plaintext, as shown here:

```
from base64 import b64encode
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import binascii, os
import random
data = b"secret"
key = binascii.hexlify(os.urandom(16))
iv = "".join(chr(random.randint(0, 0xFF)) for i in range(16))
#print ('key: ', [x for x in key] )
#print()
cipher = AES.new(key, AES.MODE_CBC)
ct_bytes = cipher.encrypt(pad(data, AES.block_size))
ct = b64encode(ct_bytes).decode('utf-8')
print('iv: {}'.format(iv))
print()
print('ciphertext: {}'.format(ct))
print()
iv: öhLÒδ™[2q]>°mâ
ciphertext: BDE+z8ME6r0QgkraNXLuuQ==
```

File Encryption Using AES

Next, you can encrypt a file using AES by implementing the following Python recipe. The primary difference is opening the file and passing the packs into blocks:

```
aes = AES.new(key, AES.MODE_CBC, iv)
filesize = os.path.getsize(infile)
with open (encrypted, 'w') as fout:
    fout.write(struct.pack('<Q', filesize))
fout.write(iv)
```

File Decryption Using AES

To decrypt the previous example, use the following code to reverse the process:

with open(verfile, 'w') as fout:

```
while True:
```

```
    data = fin.read(sz)
```

```
    n = len(data)
```

```
    if n == 0:
```

```
        break
```

```
    decd = aes.decrypt(data)
```

```
    n = len(decd)
```

```
    if fsz > n:
```

```
        fout.write(decd)
```

```
    else:
```

```
        fout.write(decd[:fsz]) # <- remove padding on last block
```

```
    fsz -= n
```

Lab 06

Secure Hash Function

Objective

This lab will introduce students to hashing mechanism. In particular we will implementa HMAC, MD5 and learn difference in SHA versions.

Activity Outcomes:

This lab teaches you the following topics:

- How to create: HMAC MD5 digest in Python
- How to apply SHA different versions to message.
- How to Simulating a Birthday Attack

1) Useful concepts:

For a one-way hash to be used in cryptographic systems, the algorithm must provide preimage resistance, secondary resistance, and collision resistance:

■ *Preimage resistance means that an attempt to find the original message that produces a hash is computationally unrealistic or for a given h in the output space of the hash function, it is hard to find any message x with $H(x) = h$.*

■ *Secondary resistance means that an attempt to find a second message that produces the same hash is computationally unrealistic or for a given message x_1 with $H(x_1) = H(x_2)$.*

■ *Collision resistance means that finding any two messages that will produce the same hash is computationally unrealistic for the message pair or $x_1 \neq x_2$ with $H(x_1) = H(x_2)$.*

In examining the rules, while the secondary resistance and collision resistance may appear very similar, they are slightly different. From a (second) preimage attack we also get a collision attack. The other direction doesn't work as easily, though some collision attacks on broken hash functions seem to be extensible to be almost as useful as second preimage attacks (i.e., we find collisions where most parts of the message can be arbitrarily fixed by the attacker).

The strength of the hash function does not equal the hash length. The strength of the hash is about half the length of the hash due to the probability produced by the **Birthday Attack**. *The birthday attack exploits the mathematics behind the birthday problem in probability theory.*

Consider the scenario in which a teacher with a class of 30 students ($n = 30$) asks for everybody's birthday to determine whether any two students have the same birthday. The birthday attack treats our birthdays as uniformly distributed values out of 365 days. The general intuition is that it takes \sqrt{N} samples from a space of size N to have 50% chance of collision. Imagine selecting some value (k) at random from N . Then out of the k values you picked there are $k(k - 1)/2$ pairs. For any given pair there is a $1/N$ chance of collision. This gives $k(k - 1)/2N$ chance of collision. Therefore, $k \sim \sqrt{N}$ will lead to around 50% chance of collision.

The *birthday attack* relies on any match coming from within a set and not a specific match to a specific value. That intuition should guide us as we approach Message Authentication Codes (MACs). This birthday attack gives us a generic approach for finding two messages that hash to the same value in far less time than brute force. The size that matters is the output size of the hash function, too.

HMAC can use a variety of hashing algorithms, like MD5, SHA1, SHA256, etc. The HMAC function is not process intensive, so it has been widely accepted, and it is easy to implement in mobile and embedded devices while maintaining decent security. The following code example shows how to generate an HMAC MD5 digest with Python:

Practice -Lab Activity 1: HMAC MD5 digest generation in Python:

```
import hmac # Import the hmac module to use the HMAC algorithm
from hashlib import md5 # Import the md5 function from hashlib to use as the
hashing algorithm

# Define the secret key as a byte string (required for HMAC).
# HMAC works with bytes, so we prefix the string with 'b' to convert it to bytes.
key = b'DECLARATION'

# Create a new HMAC object using the secret key and MD5 as the hashing algorithm.
# The second argument is an optional initial message (here we provide an empty byte
string b'').
h = hmac.new(key, b'', md5)

# Add the message to be hashed to the HMAC object.
# 'h.update()' adds the content (also in bytes) to the HMAC instance.
# Since HMAC works with bytes, we need to prefix the string with 'b' to convert it to
bytes.
h.update(b'We hold these truths to be self-evident, that all men are created equal')

# Compute the HMAC digest and print it as a hexadecimal string.
# 'h.hexdigest()' returns the digest (hash value) in hexadecimal form.
print(h.hexdigest())
```

Practice -Lab Activity 2: MD5 Hash

This hash function accepts sequence of bytes and returns 128 bit hash value, usually used to check data integrity but has security issues. Functions associated:

- `encode()`: Converts the string into bytes to be acceptable by hash function.
- `digest()`: Returns the encoded data in byte format.
- `hexdigest()`: Returns the encoded data in hexadecimal format.

Note:

The md5 library was a Python library that provided a simple interface for generating MD5 hashes. This library has been deprecated in favor of the hashlib library, which provides a more flexible and secure interface for generating hashes.

The below code demonstrates the working of MD5 hash accepting bytes and output as bytes.

```
# Python 3 code to demonstrate the
# working of MD5 (byte - byte)

import hashlib # Import hashlib to use hash functions

# Encoding the string 'GeeksforGeeks' as bytes and hashing it using the MD5 hash
function
# MD5 requires a byte input, so we prefix the string with 'b' to convert it to bytes.
result = hashlib.md5(b'GeeksforGeeks')

# Printing the equivalent byte value of the MD5 hash.
# The 'digest()' function returns the hash value as bytes.
print("The byte equivalent of hash is: ", end="")
print(result.digest())
```

Code Output:

```
The byte equivalent of hash is : b'\xf1\xe0ix~\xcetS\x1d\x11%Y\x94\hq'
```

Practice -Lab Activity 3:

Explanation: The above code takes byte and can be accepted by the hash function. The md5 hash function encodes it and then using digest (), byte equivalent encoded string is printed.

Below code demonstrated how to take string as input and output hexadecimal equivalent of the encoded value.

Solution

```
# Python 3 code to demonstrate the
# working of MD5 (string - hexadecimal)

import hashlib # Import hashlib to use hash functions

# Initializing string to hash
str2hash = "GeeksforGeeks"

# Encoding the string using encode() to convert it to bytes
# Then sending it to the md5() function to compute the MD5 hash
result = hashlib.md5(str2hash.encode())
# Printing the equivalent hexadecimal value of the MD5 hash
# The 'hexdigest()' function returns the hash value in hexadecimal format
print("The hexadecimal equivalent of hash is: ", end="")
print(result.hexdigest())
```

Output:

The hexadecimal equivalent of hash is: f1e069787ece74531d112559945c6871

Practice -Lab Activity 4: SHA, (Secure Hash Algorithms)

SHA, (Secure Hash Algorithms) are set of cryptographic hash functions defined by the language to be used for various applications such as password security etc. Some variants of it are supported by Python in the “**hashlib**” library. These can be found using “algorithms guaranteed” function of hashlib.

```
# Python 3 code to check
# available algorithms
import hashlib
# prints all available algorithms
print ("The available algorithms are : ", end = "")
print (hashlib.algorithms_guaranteed)
```

Output:

The available algorithms are: {'sha256', 'sha384', 'sha224', 'sha512', 'sha1', 'md5'}

To proceed with, lets first discuss the functions going to be used in this article.

Functions associated:

- **encode()** : Converts the string into bytes to be acceptable by hash function.
- **hexdigest()** : Returns the encoded data in hexadecimal format.

SHA Hash

The different SHA hash functions are explained below.

- **SHA256:** This hash function belong to hash class SHA-2, the internal block size of it is 32 bits.
- **SHA384:** This hash function belong to hash class SHA-2, the internal block size of it is 32 bits. This is one of the truncated version.
- **SHA224:** This hash function belong to hash class SHA-2, the internal block size of it is 32 bits. This is one of the truncated version.
- **SHA512:** This hash function belong to hash class SHA-2, the internal block size of it is 64 bits.
- **SHA1:** The 160 bit hash function that resembles MD5 hash in working and was

discontinued to be used seeing its security vulnerabilities.

Below code implements these hash functions.

```
# Python 3 code to demonstrate
# SHA hash algorithms.

import hashlib

# initializing string
str = "GeeksforGeeks"

# encoding GeeksforGeeks using encode()
# then sending to SHA256()
result = hashlib.sha256(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA256 is : ")
print(result.hexdigest())

print ("\r")

# initializing string
str = "GeeksforGeeks"

# encoding GeeksforGeeks using encode()
# then sending to SHA384()
result = hashlib.sha384(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA384 is : ")
print(result.hexdigest())

print ("\r")

# initializing string
str = "GeeksforGeeks"

# encoding GeeksforGeeks using encode()
# then sending to SHA224()
result = hashlib.sha224(str.encode())

# printing the equivalent hexadecimal value.
```

```

print("The hexadecimal equivalent of SHA224 is : ")
print(result.hexdigest())

print ("\r")

# initializing string
str = "GeeksforGeeks"

# encoding GeeksforGeeks using encode()
# then sending to SHA512()
result = hashlib.sha512(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA512 is : ")
print(result.hexdigest())

print ("\r")

# initializing string
str = "GeeksforGeeks"

# encoding GeeksforGeeks using encode()
# then sending to SHA1()
result = hashlib.sha1(str.encode())

# printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of SHA1 is : ")
print(result.hexdigest())

```

Output:

```

The hexadecimal equivalent of SHA256 is :
f6071725e7ddeb434fb6b32b8ec4a2b14dd7db0d785347b2fb48f9975126178f

The hexadecimal equivalent of SHA384 is :
d1e67b8819b009ec7929933b6fc1928dd64b5df31bcde6381b9d3f90488d25324049046
0c0a5a1a873da8236c12ef9b3

The hexadecimal equivalent of SHA224 is :
173994f309f727ca939bb185086cd7b36e66141c9e52ba0bdcfd145d

The hexadecimal equivalent of SHA512 is :
0d8fb9370a5bf7b892be4865cdf8b658a82209624e33ed71cae353b0df254a75db63d1b

```

```
aa35ad99f26f1b399c31f3c666a7fc67ecef3bdcdb7d60e8ada90b722
```

The hexadecimal equivalent of SHA1 is :

```
4175a37afd561152fb60c305d4fa6026b7e79856
```

Explanation: The above code takes string and converts it into the byte equivalent using `encode()` so that it can be accepted by the hash function. The SHA hash functions encode it and then using `hexdigest()`, hexadecimal equivalent encoded string is printed.

Graded Task 1: Simulating a Birthday Attack

Objective: Create a program that simulates a birthday attack on a simplified hash function.

- **Step 1:** Use a reduced hash size (e.g., truncate the output of SHA-256 to 16 bits).
- **Step 2:** Generate random strings and compute truncated hashes.
- **Step 3:** Find two different strings that produce the same truncated hash (collision).

```
python
Copy code
import hashlib

def truncate_hash(hash_string, bits=16):
    # Truncate the hash to the desired number of bits
    return hash_string[:bits // 4] # Each hex character
represents 4 bits

def birthday_attack(iterations=10000):
    hashes = {}
    for _ in range(iterations):
        random_string = generate_random_string()
        sha256_hash =
hashlib.sha256(random_string.encode()).hexdigest()
        truncated_hash = truncate_hash(sha256_hash)

        if truncated_hash in hashes:
            print(f"Collision found: {random_string} and
{hashes[truncated_hash]} have the same truncated hash!")
            return
        hashes[truncated_hash] = random_string

    print("No collision found.")

birthday_attack()
```

Graded Task 2: Exploring Hash Stretching (PBKDF2)

Objective: Implement hash stretching using PBKDF2 to demonstrate its defense against brute-force attacks.

- **Step 1:** Implement PBKDF2 to hash a password with multiple iterations.
- **Step 2:** Compare how different iteration counts affect the computation time.

```
python
Copy code
import hashlib
import time

def pbkdf2_hash(password, iterations):
    start_time = time.time()
    hash_obj = hashlib.pbkdf2_hmac('sha256',
password.encode(), b'salt', iterations)
    end_time = time.time()
    return hash_obj.hex(), end_time - start_time

password = input("Enter a password: ")

for iterations in [1000, 10000, 100000]:
    hash_value, time_taken = pbkdf2_hash(password,
iterations)
    print(f"Iterations: {iterations}, Hash: {hash_value},
Time taken: {time_taken:.5f} seconds")
```

Graded Task 3: Collision Detection

Objective: Demonstrate the difficulty of finding a collision in SHA-256.

- **Step 1:** Write a Python program to generate two different random strings.
- **Step 2:** Compute their SHA-256 hash and check if the hashes are the same (collision).
- **Step 3:** Loop through many iterations to see how unlikely a collision is with a strong hash function like SHA-256.

```
python
Copy code
import hashlib
import random
import string

def generate_random_string(length=10):
    return ''.join(random.choice(string.ascii_letters +
string.digits) for _ in range(length))

def find_collision(iterations=100000):
```

```

hashes = {}
for _ in range(iterations):
    random_string = generate_random_string()
    sha256_hash =
hashlib.sha256(random_string.encode()).hexdigest()

    if sha256_hash in hashes:
        print(f"Collision found: {random_string} and
{hashes[sha256_hash]} have the same hash!")
        return
    hashes[sha256_hash] = random_string

print("No collision found after many attempts.")

find_collision()

```

Graded Task 4: Password Hashing with Salt

Objective: Implement password hashing with a salt to demonstrate how salt improves security.

- **Step 1:** Generate a random salt for each password.
- **Step 2:** Hash the concatenation of the password and the salt.
- **Step 3:** Store both the salt and the hash for future verification.

```

python
Copy code
import hashlib
import os

def hash_password_with_salt(password):
    # Generate a random 16-byte salt
    salt = os.urandom(16)
    hash_obj = hashlib.pbkdf2_hmac('sha256',
password.encode(), salt, 100000)
    return salt, hash_obj

def verify_password(password, salt, stored_hash):
    # Verify the password by re-hashing it with the
stored salt
    new_hash = hashlib.pbkdf2_hmac('sha256',
password.encode(), salt, 100000)
    return new_hash == stored_hash

password = input("Enter a password: ")
salt, password_hash = hash_password_with_salt(password)
print(f"Salt: {salt.hex()}")
print(f>Password hash: {password_hash.hex()}")

```

```
# Simulate password verification
password_check = input("Re-enter the password to verify:
")
if verify_password(password_check, salt, password_hash):
    print("Password verified!")
else:
    print("Password verification failed.")
```

Graded Task 5: Hash Stretching

11. Task 11: Key Derivation and Hash Stretching

- Implement a password-based key derivation function (e.g., PBKDF2) using a hash function.
- Demonstrate how increasing the number of iterations affects the time taken to compute the hash.
- Discuss the role of hash stretching in defending against brute-force attacks.

Lab 07

Blockchain Technology

This lab will introduce students to **Blockchain** concept which is a time-stamped decentralized series of fixed records that contains data of any size is controlled by a large network of computers that are scattered around the globe and not owned by a single organization. Every block is secured and connected with each other using hashing technology which protects it from being tampered by an unauthorized person.

At the end of lab session students will be able to understand and implement Blockchain at basic level. Students will be given challenging real world problem to solve as end term projects.

<https://www.techtarget.com/searchcio/definition/blockchain>

Activity Outcomes:

This lab teaches you the following topics:

- How to implement blockchain concept.
- How to find a solution of real world problems using Blockchain Technology.

<i>Sr.No</i>	<i>Allocated Time</i>	<i>Level of Complexity</i>	<i>CLO Mapping</i>
<i>1</i>	<i>30</i>	<i>Medium</i>	<i>CLO-4</i>
<i>2</i>	<i>30</i>	<i>Medium</i>	<i>CLO-4</i>
<i>3</i>	<i>30</i>	<i>Medium</i>	<i>CLO-4</i>
<i>4</i>	<i>30</i>	<i>Medium</i>	<i>CLO-4</i>
<i>5</i>	<i>30</i>	<i>High</i>	<i>CLO-4</i>
<i>6</i>		<i>High</i>	<i>CLO-4</i>
<i>7</i>		<i>High</i>	<i>CLO-4</i>

What is blockchain?

Blockchain is a record-keeping technology designed to make it impossible to hack the system or forge the data stored on the blockchain, thereby making it secure and immutable. It's a type of distributed ledger technology ([DLT](#)), a digital record-keeping system for recording transactions and related data in multiple places at the same time. Each computer in a blockchain network maintains a copy of the ledger where transactions are recorded to prevent a [single point of failure](#). Also, all copies are updated and validated simultaneously.

Blockchain is also considered a type of database, but it differs substantially from conventional databases in how it stores and manages information. Instead of storing data in rows, columns, tables and files as traditional databases do, [blockchain stores data](#) in blocks that are digitally chained together. In addition, a blockchain is a decentralized database managed by computers belonging to a [peer-to-peer](#) network instead of a central computer like in traditional databases.

[Bitcoin](#), launched in 2009 on the Bitcoin blockchain, was the first [cryptocurrency](#) and popular application to successfully use blockchain. As a result, blockchain has been most often associated with Bitcoin and alternatives such as Dogecoin and Bitcoin Cash, which both use public ledgers.

How blockchain and distributed ledger technology work

Blockchain uses a multistep process that includes these five steps:

An authorized participant inputs a transaction, which must be authenticated by the technology.

That action creates a block that represents that specific transaction or data.

The block is sent to every computer node in the network.

Authorized nodes validate transactions and add the block to the existing blockchain.

The update is distributed across the network, which finalizes the transaction.

These steps take place in near real time and involve a range of elements. Nodes in public blockchain networks are referred to as miners; they're typically paid for this task -- often in processes called proof of work or proof of stake -- usually in the form of cryptocurrency.

A blockchain ledger consists of two types of records, individual transactions and blocks. The first block has a header and data that pertain to transactions taking place within a set time period. The block's timestamp is used to help create an alphanumeric string called a hash. After the first block has been created, each subsequent block in the ledger uses the previous block's hash to calculate its own hash.

Before a new block can be added to the chain, its authenticity must be verified by a computational process called validation or consensus. At this point in the blockchain process, a majority of nodes in the network must agree the new block's hash has been calculated correctly. Consensus ensures that all copies of the blockchain distributed ledger share the same state.

Once a block has been added, it can be referenced in subsequent blocks, but it can't be changed. If someone attempts to swap out a block, the hashes for previous and subsequent blocks will also change and disrupt the ledger's shared state.

When consensus is no longer possible, other computers in the network are aware that a problem has occurred, and no new blocks will be added to the chain until the problem is solved. Typically, the block causing the error will be discarded and the consensus process will be repeated. This eliminates a single point of failure.

Key features of blockchain technology

Blockchain technology is built on a foundation of unique characteristics that differentiate it from traditional databases. The following are its most important and defining characteristics:

Decentralization. Blockchain decentralization is one of the fundamental aspects of the technology. Unlike centralized databases where a central authority, such as a bank, controls and verifies transactions, blockchain operates on a distributed ledger. This means multiple transparent participants, known as nodes, maintain, verify and update the ledger. Each node is spread across a network and contains a copy of the whole blockchain.

Immutability and security. Cryptographic algorithms are used in blockchain to provide strong security, recording transactions and making tampering nearly impossible. Information is stored in blocks that are linked together using cryptographic hashes. If someone tries to tamper or modify a block, it would require the alteration of every subsequent block, making tampering computationally infeasible. This inherent blockchain security feature ensures immutability of information and makes blockchain an ideal platform for storing sensitive data and conducting secure transactions.

Transparency and traceability. The inherent transparency of blockchain technology ensures every network participant has access to identical information. For instance, every transaction becomes part of a public ledger, visible to all participants. This transparency ensures trust and network accountability, because any inconsistency can be promptly recognized and resolved. Additionally, the blockchain's capacity to track the origin and trajectory of assets facilitates audits and decreases the likelihood of fraudulent activities.

Smart contracts. These contracts are automated agreements encoded in software that execute the stipulations of a contract automatically. Smart contract codes are stored on the blockchain and carry out their functions once predetermined conditions are met. These contracts eliminate the need for intermediaries, streamline transactions, save money and speed up close times. They're used in a range of diverse sectors, including supply chain management, insurance and finance

Blockchain and smart contracts

Smart contracts are one of the most important features of blockchain technology. These are self-executing digital contracts written in code. They operate automatically according to predefined rules and conditions. Smart contracts are designed to facilitate, verify and enforce the negotiation or performance of an agreement without the need for intermediaries, such as lawyers, banks or other third parties. Once the specified conditions are met, the smart contract automatically executes the agreed-upon actions or transactions, ensuring that all parties involved adhere to the terms of the contract.

Smart contracts are typically deployed on blockchain platforms, which provide the necessary security and transparency for their execution. Ethereum is a popular blockchain platform for smart contracts. It's used for a range of applications such as financial transactions, supply chain management, real estate deals and digital identity verification.

Smart contracts have several benefits. By eliminating intermediaries, smart contract technology reduces the costs. It also cuts out complications and interference intermediaries can cause, speeding processes while also enhancing security.

How Blockchain Works

1 TRANSACTION 2 BLOCK 3 VERIFICATION 4 HASH 5 EXECUTION

1

Transaction

Two parties, A and B, decide to exchange a unit of value (digital currency or a digital representation of some other asset, such as land title, birth certificate or educational degree) and initiate the transaction.



2

Block

The transaction is packaged with other pending transactions thereby creating a "block." The block is sent to the blockchain system's network of participating computers.

3

Verification

The participating computers (called "miners" in the Bitcoin blockchain) evaluate the transactions and through mathematical calculations determine whether they are valid, based on agreed-upon rules. When "consensus" has been achieved, typically among 51% of participating computers, the transactions are considered verified.



5

Execution

The unit of value moves from the account of party A to the account of party B.



4

Hash

Each verified block of transactions is time-stamped with a cryptographic hash. Each block also contains a reference to the previous block's hash, thus creating a "chain" of records that cannot be falsified except by convincing participating computers that the tampered data in one block and in all prior blocks is true. Such a feat is considered impossible.

Blockchain technology works in five basic steps, sometimes referred to as mining, in which transactions and data are executed and verified.

Here are several **blockchain-related programming tasks** that will introduce fundamental concepts like hashing, proof of work, and block validation, helping students understand how blockchains operate at a technical level.

Example

Create a class named Person, use the `__init__()` function to assign values for name and age:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Note: The `__init__()` function is called automatically every time the class is being used to create a new object.

Practice -Lab Activity 1: Block Structure Implementation

Objective: Create a basic block structure for a blockchain.

- **Step 1:** Define a Block class that includes properties like index, timestamp, data, previous_hash, and hash.
- **Step 2:** Compute the block's hash by combining the block's properties (excluding hash) and hashing them using SHA-256.

Example:

```
import hashlib # Importing the hashlib library to use the SHA-256 hashing function
import time    # Importing the time module to get the current timestamp for each block

# Block class definition
class Block:
    def __init__(self, index, data, previous_hash):
        """
        Constructor to initialize a block in the blockchain.
        Parameters:
        - index: The position of the block in the blockchain (starting with 0 for the genesis block).
        - data: The transaction data or information to be stored in the block.
        - previous_hash: The hash of the previous block in the chain, ensuring continuity and security.
        """
```

```

        self.index = index                                # Index or
position of the block in the chain
        self.timestamp = time.time()                    # Timestamp of
block creation
        self.data = data                                # Data stored in
the block (e.g., transactions)
        self.previous_hash = previous_hash              # Hash of the
previous block, linking to it
        self.hash = self.compute_hash()                 # Hash of the
current block, generated using compute_hash method

    def compute_hash(self):
        """
        Method to calculate the SHA-256 hash of the block's
        contents.
        The hash is generated using the block's index,
        timestamp, data, and the previous block's hash.
        """
        # Combine the block's properties into a single string
        block_string =
f"{self.index}{self.timestamp}{self.data}{self.previous_hash}"
        # Compute and return the SHA-256 hash of the block
        string
        return
        hashlib.sha256(block_string.encode()).hexdigest()

# Creating the genesis block (the first block in the
blockchain)
genesis_block = Block(0, "Genesis Block", "0")
# Output the hash of the genesis block
print(f"Genesis Block Hash: {genesis_block.hash}")

```

The screenshot shows a terminal window titled 'IDLE Shell 3.11.5'. The prompt is 'Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)] on win32'. The user has entered a command to run a script, and the output shows the Genesis Block Hash: ebec68ab0e41e78561a49d6ad381553c0deff225d54ecbaf3c77d43d5e5a4af3. The status bar at the bottom indicates 'Ln: 6 Col: 0'.

Practice -Lab Activity 2: Blockchain Construction

Objective: Create a blockchain by linking blocks together.

- **Step 1:** Create a `Blockchain` class that stores a list of blocks.
- **Step 2:** Implement a method to add new blocks to the chain.

- **Step 3:** Ensure that each block correctly references the previous block's hash.

```
# Blockchain class definition
class Blockchain:
    def __init__(self):
        """
        Constructor to initialize the blockchain.
        It starts with a list containing only the genesis
block.
        """
        # The blockchain starts with the genesis block
        self.chain = [self.create_genesis_block()]

    def create_genesis_block(self):
        """
        Creates the first block in the blockchain (the genesis
block).
        The genesis block has an index of 0, default data
"Genesis Block",
        and a previous hash of "0" since it's the first block.
        """
        return Block(0, "Genesis Block", "0")

    def add_block(self, data):
        """
        Adds a new block to the blockchain.
        Parameters:
        - data: The data to be stored in the new block.
        The new block links to the previous block by including
its hash.
        """
        # Get the last block in the current chain (previous
block)
        last_block = self.chain[-1]
        # Create a new block with the next index, the provided
data, and the hash of the last block
        new_block = Block(len(self.chain), data,
last_block.hash)
        # Append the new block to the blockchain
        self.chain.append(new_block)

    def print_blockchain(self):
        """
        Prints out each block in the blockchain.
        Displays the block's index, data, hash, and the hash of
the previous block.
        """
        # Iterate over all blocks in the chain and print their
details
        for block in self.chain:
            print(f"Index: {block.index}, Data: {block.data},
Hash: {block.hash}, Previous Hash: {block.previous_hash}")

# Example usage of the Blockchain class
```

```

blockchain = Blockchain()           # Create a new
blockchain with a genesis block
blockchain.add_block("Block 1 Data") # Add a block with
data "Block 1 Data"
blockchain.add_block("Block 2 Data") # Add a block with
data "Block 2 Data"
blockchain.print_blockchain()       # Print the details of
the blockchain

```

```

IDLE Shell 3.11.5
File Edit Shell Debug Options Window Help
>>>
===== RESTART: C:/Users/PAKISTAN/Documents/BCK.py =====
Traceback (most recent call last):
  File "C:/Users/PAKISTAN/Documents/BCK.py", line 16, in <module>
    blockchain = Blockchain()
  File "C:/Users/PAKISTAN/Documents/BCK.py", line 3, in __init__
    self.chain = [self.create_genesis_block()]
  File "C:/Users/PAKISTAN/Documents/BCK.py", line 6, in create_genesis_block
    return Block(0, "Genesis Block", "0")
NameError: name 'Block' is not defined
>>>
===== RESTART: C:/Users/PAKISTAN/Documents/BCK.py =====
Genesis Block Hash: b9668bf86e681033849530b0ed316eddfbc4c154d71594e4c115aee3db96
b24e
Index: 0, Data: Genesis Block, Hash: 0fa04a6c37d9c744ab3e9bda01b5482949305e59509
a4be59aa6145858fcl7d3, Previous Hash: 0
Index: 1, Data: Block 1 Data, Hash: b6548ccbbe36968821a562712df03a498b4921ebe5ba
01d11b8d7c52c990371a, Previous Hash: 0fa04a6c37d9c744ab3e9bda01b5482949305e59509
a4be59aa6145858fcl7d3
Index: 2, Data: Block 2 Data, Hash: 61b12d9ae053af8bb856bc84e2b4c2368fed078a4ca5
87927c304c792a743a91, Previous Hash: b6548ccbbe36968821a562712df03a498b4921ebe5b
a01d11b8d7c52c990371a
>>>

```

Practice -Lab Activity 3: Proof of Work

Purpose of PoW

The purpose of a consensus mechanism is to bring all the nodes in agreement, that is, trust one another, in an environment where the nodes don't trust each other.

- All the transactions in the new block are then validated and the new block is then added to the blockchain.
- The block will get added to the chain which has the longest block height (see [blockchain forks](#) to understand how multiple chains can exist at a point in time).
- Miners(special computers on the network) perform computation work in solving a complex mathematical problem to add the block to the network, hence named, Proof-of-Work.
- With time, the mathematical problem becomes more complex.

Objective: Implement a basic Proof of Work (PoW) algorithm.

- **Step 1:** Modify the Block class to include a nonce and a difficulty target.

- **Step 2:** Implement the PoW mechanism that requires the block's hash to start with a certain number of zeros.

```
import hashlib # Importing hashlib to use the SHA-256 hashing
function
import time    # Importing time to capture the current
timestamp for each block

# Block class definition
class Block:
    def __init__(self, index, data, previous_hash,
difficulty=2):
        """
        Constructor to initialize a block in the blockchain.
        Parameters:
        - index: The position of the block in the blockchain.
        - data: The data or information stored in the block
(e.g., transactions).
        - previous_hash: The hash of the previous block in the
chain, ensuring blockchain continuity.
        - difficulty: The difficulty level for the Proof of
Work (PoW), default is 2.
        """
        self.index = index # The block's
position in the chain
        self.timestamp = time.time() # Timestamp of
block creation
        self.data = data # Data stored in
the block
        self.previous_hash = previous_hash # Hash of the
previous block
        self.nonce = 0 # Nonce (number
used in PoW to find valid hash)
        self.hash = self.compute_proof_of_work(difficulty) #
Find the valid hash using Proof of Work

    def compute_hash(self):
        """
        Compute the SHA-256 hash of the block's contents.
        The hash includes the block's index, timestamp, data,
previous hash, and nonce.
        """
        # Combine the block's attributes into a single string
        block_string =
f"{self.index}{self.timestamp}{self.data}{self.previous_hash}{s
elf.nonce}"
        # Compute and return the SHA-256 hash of the block
string
        return
hashlib.sha256(block_string.encode()).hexdigest()

    def compute_proof_of_work(self, difficulty):
        """
        Implements the Proof of Work (PoW) algorithm.
```

```

    PoW requires finding a hash that starts with a certain
    number of leading zeros,
    defined by the difficulty parameter.
    """
    # The required hash prefix is a string of '0' repeated
    difficulty times (e.g., "00" for difficulty=2)
    prefix = '0' * difficulty
    # Loop until we find a hash that starts with the
    required number of leading zeros
    while True:
        self.hash = self.compute_hash() # Compute the
    block's hash
        if self.hash.startswith(prefix): # Check if the
    hash satisfies the difficulty requirement
            return self.hash # Return the valid
    hash if it meets the condition
        self.nonce += 1 # Increment the
    nonce and try again (to find a new hash)

# Example usage with Proof of Work
block = Block(1, "Some Data", "0", difficulty=4) # Create a
new block with difficulty level 4
# Print the block's hash and the nonce value found through
Proof of Work
print(f"Block Hash with Proof of Work: {block.hash}, Nonce:
{block.nonce}")

```

The screenshot shows an IDLE Shell 3.11.5 window. The title bar says "IDLE Shell 3.11.5". The menu bar includes "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The shell area shows a prompt ">>>" followed by a line of text: "===== RESTART: C:/Users/PAKISTAN/Documents/BCK.py =====". Below this, the output of the script is displayed: "Block Hash with Proof of Work: 00004c7867b5f6aa2ce85a9add749f50856b6bad3c639f7bf43bbff2ec0b233e, Nonce: 159254". The prompt ">>>" is shown again on the next line. The status bar at the bottom right indicates "Ln: 82 Col: 0".

Practice -Lab Activity 4: Blockchain Validation

Objective: Implement a method to validate the integrity of the blockchain.

- **Step 1:** Ensure that each block's `previous_hash` matches the hash of the previous block.
- **Step 2:** Recompute the hash of each block and verify it matches the stored hash.

```

import hashlib # Import hashlib to use the SHA-256 hashing
algorithm for block creation
import time    # Import time to capture the timestamp for each
block

# Block class definition
class Block:
    def __init__(self, index, data, previous_hash,
difficulty=2):

```

```

    """
    Constructor to initialize a block in the blockchain.
    Parameters:
    - index: The position of the block in the blockchain.
    - data: The information or transactions stored in the
block.
    - previous_hash: The hash of the previous block in the
blockchain, ensuring continuity.
    - difficulty: The difficulty level for Proof of Work
(PoW) that defines how hard it is to mine the block.
    """
    self.index = index                    # Block's
position in the blockchain
    self.timestamp = time.time()         # Current
timestamp for block creation
    self.data = data                     # Data stored in
the block
    self.previous_hash = previous_hash   # Hash of the
previous block in the chain
    self.nonce = 0                       # Nonce used for
Proof of Work, starts at 0
    self.hash = self.compute_proof_of_work(difficulty) #
Compute the block's hash using PoW with the given difficulty

    def compute_hash(self):
        """
        Compute the SHA-256 hash of the block.
        The hash is generated from the block's index,
timestamp, data, previous hash, and nonce.
        """
        # Concatenate the block's attributes into a string
        block_string =
f"{self.index}{self.timestamp}{self.data}{self.previous_hash}{s
elf.nonce}"
        # Return the SHA-256 hash of the concatenated string
        return
hashlib.sha256(block_string.encode()).hexdigest()

    def compute_proof_of_work(self, difficulty):
        """
        Implements Proof of Work (PoW).
        PoW requires finding a hash that starts with a certain
number of leading zeros (determined by the difficulty).
        """
        prefix = '0' * difficulty # String of '0's, the length
of which is the difficulty level
        while True:
            self.hash = self.compute_hash() # Compute the
block's hash
            if self.hash.startswith(prefix): # Check if the
hash satisfies the difficulty condition (starts with the
required number of leading zeros)
                return self.hash # Return the valid
hash if it meets the condition
            self.nonce += 1 # Increment the
nonce and try again to find a valid hash

```

```

# Blockchain class definition
class Blockchain:
    def __init__(self):
        """
        Constructor to initialize the blockchain.
        The blockchain starts with the genesis block (the first
        block).
        """
        self.chain = [self.create_genesis_block()] #
        Initialize the chain with the genesis block

    def create_genesis_block(self):
        """
        Creates the genesis block (the first block in the
        chain).
        The genesis block has an index of 0, default data
        "Genesis Block", and a previous hash of "0".
        """
        return Block(0, "Genesis Block", "0") # Return the
        first block with index 0

    def add_block(self, data, difficulty=2):
        """
        Adds a new block to the blockchain.
        Parameters:
        - data: The data or transactions to be stored in the
        block.
        - difficulty: The difficulty level for Proof of Work
        (PoW) for the new block.
        """
        last_block = self.chain[-1] # Get the last block in
        the chain
        new_block = Block(len(self.chain), data,
        last_block.hash, difficulty) # Create a new block with the
        current index, data, and previous block's hash
        self.chain.append(new_block) # Append the newly
        created block to the blockchain

    def is_chain_valid(self):
        """
        Validates the entire blockchain by ensuring the hashes
        and block linking are correct.
        """
        # Loop through the blockchain from the second block
        (index 1) onward
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]
            previous_block = self.chain[i-1]

            # Check if the current block's hash is correct
            (recompute and compare)
            if current_block.hash !=
            current_block.compute_hash():
                print(f"Invalid block at index {i}") # Print
                error if the current block's hash is invalid

```

```

        return False # Return False if the blockchain
is invalid

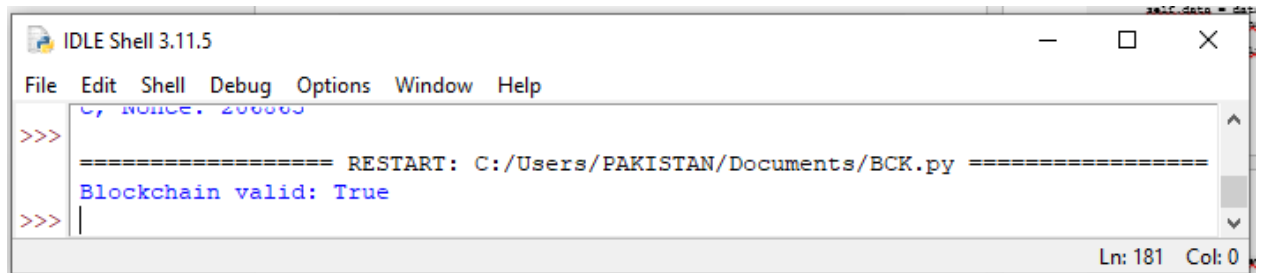
        # Check if the current block's previous hash
matches the previous block's hash
        if current_block.previous_hash !=
previous_block.hash:
            print(f"Invalid chain at index {i}") # Print
error if the chain is broken
            return False # Return False if the chain
linking is invalid

        return True # Return True if the entire blockchain is
valid

# Example usage of Blockchain class
blockchain = Blockchain() # Create a new
blockchain with the genesis block
blockchain.add_block("Block 1 Data") # Add a block
with data "Block 1 Data"
blockchain.add_block("Block 2 Data", difficulty=4) # Add
another block with difficulty level 4 for PoW

# Validate the blockchain and print whether it's valid
print("Blockchain valid:", blockchain.is_chain_valid()) #
Output whether the blockchain is valid

```



The screenshot shows an IDLE Shell 3.11.5 window. The menu bar includes File, Edit, Shell, Debug, Options, Window, and Help. The command prompt shows a restart of the script 'C:/Users/PAKISTAN/Documents/BCK.py'. The output of the script is 'Blockchain valid: True'. The status bar at the bottom right indicates 'Ln: 181 Col: 0'.

Practice -Lab Activity 5: Simulate a 51% Attack

A 51% attack refers to a scenario in which a single entity or group of attackers gains control over more than 50% of the computational power (hashing power) in a blockchain network, particularly in proof-of-work (PoW) blockchains like Bitcoin. This gives the attacker the ability to manipulate the blockchain by performing actions that compromise its integrity. Some of the risks associated with a 51% attack include:

Objective: Simulate a scenario where the blockchain is compromised by a malicious actor who controls over 51% of the mining power.

- **Step 1:** Create two versions of a blockchain: one valid and one modified by an attacker.

- **Step 2:** The attacker rewrites history by modifying the data in an earlier block and recalculating all subsequent blocks' hashes.

```
import hashlib # Import hashlib to use the SHA-256 hashing
algorithm
import time    # Import time to generate timestamps for each
block

# Block class definition
class Block:
    def __init__(self, index, data, previous_hash,
difficulty=2):
        """
        Constructor to initialize a block in the blockchain.
        Parameters:
        - index: The position of the block in the blockchain.
        - data: The information or transactions stored in the
block.
        - previous_hash: The hash of the previous block in the
blockchain.
        - difficulty: The difficulty level for the Proof of
Work (PoW) algorithm.
        """
        self.index = index # The index of
the block in the chain
        self.timestamp = time.time() # Timestamp when
the block is created
        self.data = data # Data stored in
the block
        self.previous_hash = previous_hash # Hash of the
previous block
        self.nonce = 0 # Nonce used for
the PoW algorithm, starts at 0
        self.hash = self.compute_proof_of_work(difficulty) #
Compute the block's hash using PoW

    def compute_hash(self):
        """
        Compute the SHA-256 hash of the block.
        This method concatenates the block's attributes and
returns the hash of the concatenated string.
        """
        # Combine block properties into a string and hash it
        block_string =
f"{self.index}{self.timestamp}{self.data}{self.previous_hash}{s
elf.nonce}"
        return
hashlib.sha256(block_string.encode()).hexdigest()

    def compute_proof_of_work(self, difficulty):
        """
        Implements Proof of Work (PoW) by finding a hash that
starts with a certain number of leading zeros.
        The number of zeros is determined by the 'difficulty'
parameter.
```

```

        """
        prefix = '0' * difficulty # The target hash must start
with a specific number of leading zeros
        while True:
            self.hash = self.compute_hash() # Compute the hash
            if self.hash.startswith(prefix): # Check if the
hash meets the difficulty requirement
                return self.hash # If valid, return
the hash
            self.nonce += 1 # If not,
increment the nonce and try again

# Blockchain class definition
class Blockchain:
    def __init__(self):
        """
        Constructor to initialize the blockchain with the
genesis block (the first block in the chain).
        """
        self.chain = [self.create_genesis_block()] # Start the
chain with the genesis block

    def create_genesis_block(self):
        """
        Creates the genesis block, the first block in the
blockchain.
        The genesis block has an index of 0, fixed data, and no
previous block (previous hash = "0").
        """
        return Block(0, "Genesis Block", "0") # The first
block with index 0 and default previous hash

    def add_block(self, data, difficulty=2):
        """
        Adds a new block to the blockchain.
        Parameters:
        - data: The data to be stored in the new block.
        - difficulty: The difficulty level for the Proof of
Work (PoW) algorithm.
        """
        last_block = self.chain[-1] # Get the last block in
the blockchain
        new_block = Block(len(self.chain), data,
last_block.hash, difficulty) # Create a new block
        self.chain.append(new_block) # Append the new block to
the chain

    def is_chain_valid(self):
        """
        Validates the integrity of the blockchain by checking
each block's hash and previous hash linkage.
        """
        # Loop through the chain starting from the second block
(index 1)
        for i in range(1, len(self.chain)):
            current_block = self.chain[i]

```

```

        previous_block = self.chain[i-1]

        # Check if the current block's hash is correct (by
recomputing the hash)
        if current_block.hash !=
current_block.compute_hash():
            print(f"Invalid block at index {i}") # If the
hash is incorrect, print an error message
            return False # Return False as the chain is
invalid

        # Check if the current block's previous hash
matches the hash of the previous block
        if current_block.previous_hash !=
previous_block.hash:
            print(f"Invalid chain at index {i}") # If the
chain linkage is broken, print an error
            return False # Return False as the chain is
invalid

        return True # If all checks pass, return True (the
chain is valid)

# Simulating a 51% attack
# Example usage of the blockchain
blockchain = Blockchain() # Create a new blockchain with the
genesis block
blockchain.add_block("Block 1 Data") # Add a valid block with
data "Block 1 Data"
blockchain.add_block("Block 2 Data") # Add a second block
blockchain.add_block("Block 3 Data") # Add a third block

# Attack: An attacker modifies the data of Block 1 and tries to
manipulate the chain
# This simulates a 51% attack where a majority of miners
(malicious actors) try to modify a block
blockchain.chain[1].data = "Malicious Block 1 Data" #
Modify the data of Block 1
blockchain.chain[1].hash =
blockchain.chain[1].compute_hash() # Recompute the hash
for Block 1

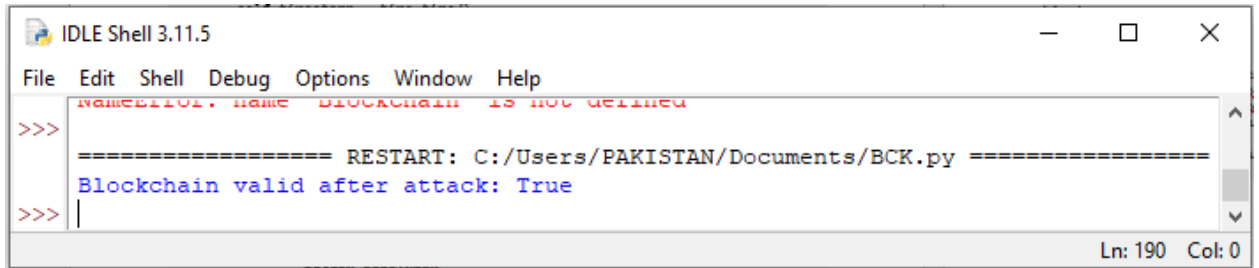
# The attacker then recalculates all subsequent block
hashes to make the chain appear valid
for i in range(2, len(blockchain.chain)): # Loop through
the remaining blocks
    blockchain.chain[i].previous_hash =
blockchain.chain[i-1].hash # Set the previous hash of
the current block
    blockchain.chain[i].hash =
blockchain.chain[i].compute_hash() # Recompute the
current block's hash

# Validate the blockchain after the attack

```



```
# This demonstrates that if enough blocks are controlled (as in
a 51% attack), an invalid chain could be made valid
print("Blockchain valid after attack:",
blockchain.is_chain_valid()) # Output whether the blockchain
is still valid
```



```
IDLE Shell 3.11.5
File Edit Shell Debug Options Window Help
NameError: name 'Blockchain' is not defined
>>>
===== RESTART: C:/Users/PAKISTAN/Documents/BCK.py =====
Blockchain valid after attack: True
>>>
Ln: 190 Col: 0
```

Home Task 6: Simplified Blockchain Mining Reward System

Objective: Implement a simple reward system where a miner gets rewarded for successfully mining a block.

- **Step 1:** Define a reward that gets added to the block data.
- **Step 2:** Create a miner function that simulates mining and adds the reward to the miner's wallet.

```
class Blockchain:
    def __init__(self):
        self.chain = [self.create_genesis_block()]
        self.mining_reward = 50
        self.pending_rewards = {}

    def create_genesis_block(self):
        return Block(0, "Genesis Block", "0")

    def mine_block(self, miner_address, difficulty=2):
        # Add reward to miner's wallet for mining the block
        new_block = Block(len(self.chain), f"Reward to
{miner_address}: {self.mining_reward} coins", self.chain[-
1].hash, difficulty)
        self.chain.append(new_block)
        self.pending_rewards[miner_address] =
self.pending_rewards.get(miner_address, 0) + self.mining_reward

    def print_rewards(self):
        for miner, reward in self.pending_rewards.items():
            print(f"Miner: {miner}, Reward: {reward} coins")

# Example usage
blockchain = Blockchain()
blockchain.mine_block("Miner1")
blockchain.mine_block("Miner2", difficulty=4)

blockchain.print_rewards()
```

Home Task 7: Basic Blockchain Peer-to-Peer (P2P) Network Simulation

Objective: Simulate a simplified blockchain peer-to-peer network where multiple nodes exchange blocks.

- **Step 1:** Define multiple nodes that maintain their own copy of the blockchain.
- **Step 2:** Implement a method to synchronize the blockchain among the nodes.

```
class Node:
    def __init__(self, name):
        self.name = name
        self.blockchain = Blockchain()

    def sync_with_node(self, other_node):
        if len(self.blockchain.chain) <
len(other_node.blockchain.chain):
            self.blockchain.chain = other_node.blockchain.chain
            print(f"{self.name} synchronized with
{other_node.name}'s blockchain.")

# Example usage
node1 = Node("Node1")
node2 = Node("Node2")

# Node1 mines a block
node1.blockchain.mine_block("Miner1")
# Node2 synchronizes with Node1
node2.sync_with_node(node1)
node2.blockchain.print_blockchain()
```

These tasks introduce various blockchain concepts like block creation, proof of work, blockchain integrity validation, mining rewards, and P2P synchronization. Students can further build on these tasks by adding more complex features like transaction handling, consensus algorithms, or smart contracts.

Lab 08

Midterm-Exam



COMSATS UNIVERSITY ISLAMABAD (CUI)

**DEPARTMENT OF COMPUTER SCIENCE
LAB-MIDTERM EXAMINATION FALL - 2024
BS (CYS)– III SEMESTER**

Course: CSC232-Information Security.
Maximum Marks: 25
Minutes

Dated: 31-10-2024
Time Allowed: 120

Course Instructor:

Dr. Tehsin Kanwal.

- *Lab manual, any code generating online site (chatgpt) are strictly prohibited during examination.*
- *Mobile /WhatsApp is not allowed during examination.*
- *Attempt all questions.*

CLO-4 Implement a cryptographic algorithm to ensure information security.

<i>Q.No</i>	<i>Allocated Time</i>	<i>Marks</i>	<i>CLO Mapping</i>
1	20	5	CLO-4
2-a	10	2.5	CLO-4
2-b	20	2.5	CLO-4
3	20	5	CLO-4
4	20	5	CLO-4
5-a	20	3	CLO-4
5-b	10	2	CLO-4

Question 1. [5 Marks]

Write a Python program that demonstrates how Caesar Cipher encryption and decryption work using modular arithmetic.

Steps:

1. Ask the user for the plaintext and shift value.
2. Encrypt using the formula: $C = (P + \text{shift}) \% 26$
3. Decrypt using the formula: $P = (C - \text{shift}) \% 26$
4. Print the results.

Question 2 [2.5+ 2.5 = 5 Marks]

Task 2-a: Correct following Vigenère cipher code also provide explanation of mistake you have corrected.

```
def key_vigenere(key):
    keyArray = []
    for i in range(0, len(key))
        keyElement = ord(key[i].upper) - 65
        keyArray.append(keyElemnt)
    return keyArray
secretKey = 'DECLARATION'
key = key_vigenere(secretKey)
print(keys)
```

Lab Task 2b: Write a program to simulate a **brute-force attack** on DES using a small key space (for learning purposes, use a smaller key size like 8 bits). This will demonstrate how easily DES can be broken with modern computing power when the key space is limited.

Steps:

1. Ask the user for a plaintext message and a small key size (e.g., 4-bit or 8-bit keys for the simulation).
2. Encrypt the message using DES.
3. Write a brute-force algorithm that tries all possible keys within the small key space.
4. Print the correct key when found and the decrypted message.

Question 3 [5 Marks]

Implement password hashing with a salt to demonstrate how salt improves security.

Step 1: Generate a random salt for each password.

Step 2: Hash the concatenation of the password and the salt.

Step 3: Store both the salt and the hash for future verification.

Question 4. [5 Marks]

Write code to apply an HMAC digest to a signed message. Using HMAC to Sign Message The file that we are creating the message digest for is a simple text file that contains only Hello. When run, the code reads a data file and computes an HMAC signature for it

Steps:

- Define **Secret Key**: Set a secret key to use in the HMAC process.

- Initialize **HMAC Object**: Create an HMAC object with the specified key and hash function.
- Open **the Target File**: Open the file you want to generate an HMAC for, usually in binary mode for accurate data reading.
- Read **the File in Chunks**: Use a loop to read the file in manageable chunks (e.g., 1024 bytes) for memory efficiency.
- Process **Each Chunk**: For each chunk, update the HMAC object with the current data block.
- Finalize and **Retrieve Digest**: After reading all chunks, compute the final HMAC digest, typically in hexadecimal format.
- Output **the Digest**: Display or return the resulting digest, representing the HMAC of the entire file.

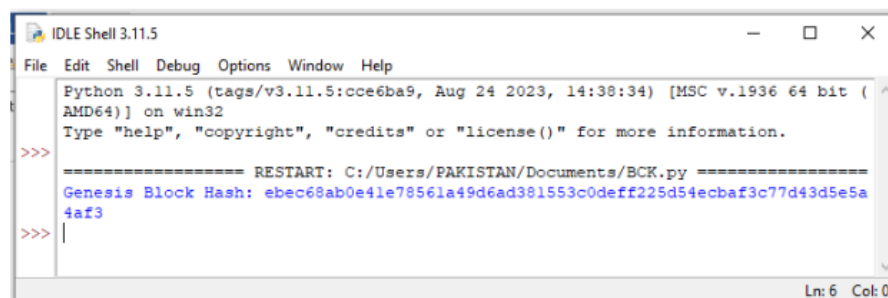
Question 5. [3 +2 = 5 Marks]

Lab Task 5-a: Create a basic block structure for a blockchain.

Step 1: Define a Block class that includes properties like index, timestamp, data, previous hash, and hash.

Step 2: Compute the block's hash by combining the block's properties (excluding hash) and hashing them using SHA-256.

Your code output should resemble given python shell sample output.



```

Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>>
===== RESTART: C:/Users/PAKISTAN/Documents/BCK.py =====
Genesis Block Hash: ebec68ab0e41e78561a49d6ad381553c0deff225d54ecbaf3c77d43d5e5a4af3
>>>
  
```

Lab Task 5-b: Write a Python program that simulates a secure messaging system using the Vernam Cipher. The key is shared securely between two users and is only used once (One-Time Pad).

Steps:

1. Generate a random key.
2. User 1 encrypts a message with the key.
3. User 2 receives the encrypted message and decrypts it using the same key.
4. Ensure that the key is destroyed after use.

-----Best of Luck-----

Lab 09

Key distribution problem

Objective:

This lab will introduce students into Key distribution problem. The key distribution problem was solved by Diffie and Hellman as that in which they introduced public key cryptography. Their protocol for key distribution, called Diffie–Hellman Key Exchange, allows two parties to agree a secret key over an insecure channel.

Activity Outcomes:

This lab teaches you the following topics:

- How to program key distribution problem.
- Diffie-Hellman algorithm coding and attacks description.

Lab Activity 1: Diffie-Hellman protocol:

The Diffie-Hellman algorithm is being used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

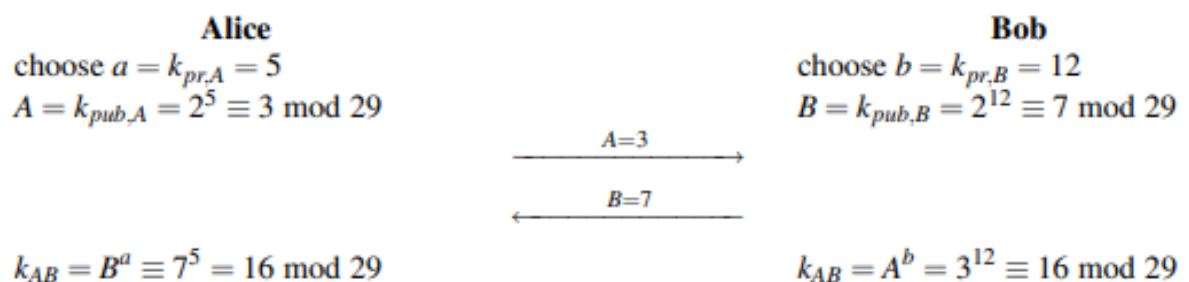
- For the sake of simplicity and practical implementation of the algorithm, consider only 4 variables, one prime P and G (a primitive root of P) and two private values a and b .
- P and G are both publicly available numbers. Users (say Alice and Bob) pick private values a and b and they generate a key and exchange it publicly. The opposite person receives the key and that generates a secret key, after which they have the same secret key to encrypt.

Step-by-Step explanation is as follows:

Alice	Bob
Public Keys available = P, G	Public Keys available = P, G
Private Key Selected = a	Private Key Selected = b
Key generated = $x = G_a \bmod P, x = G_a \bmod P$	Key generated = $y = G_b \bmod P, y = G_b \bmod P$
Exchange of generated keys takes place	

Alice	Bob
Key received = y	key received = x
Generated Secret Key = $k_a = y^a \bmod P$	Generated Secret Key = $k_b = x^b \bmod P$
Algebraically, it can be shown that $k_a = k_b$	

Example:



Step 1: Alice and Bob get public numbers $P = 23$, $G = 9$
 Step 2: Alice selected a private key $a = 4$ and
 Bob selected a private key $b = 3$
 Step 3: Alice and Bob compute public values
 Alice: $x = (9^4 \bmod 23) = (6561 \bmod 23) = 6$
 Bob: $y = (9^3 \bmod 23) = (729 \bmod 23) = 16$
 Step 4: Alice and Bob exchange public numbers
 Step 5: Alice receives public key $y = 16$ and
 Bob receives public key $x = 6$
 Step 6: Alice and Bob compute symmetric keys
 Alice: $k_a = y^a \bmod p = 65536 \bmod 23 = 9$
 Bob: $k_b = x^b \bmod p = 216 \bmod 23 = 9$
 Step 7: 9 is the shared secret.

Implementation:

```

# Diffie-Hellman Code

# Power function to return value of a^b mod P
def power(a, b, p):
    if b == 1:
        return a
    else:
        return pow(a, b) % p
  
```

```

# Main function
def main():
    # Both persons agree upon the public keys G and P
    # A prime number P is taken
    P = 23
    print("The value of P:", P)

    # A primitive root for P, G is taken
    G = 9
    print("The value of G:", G)

    # Alice chooses the private key a
    # a is the chosen private key
    a = 4
    print("The private key a for Alice:", a)

    # Gets the generated key
    x = power(G, a, P)

    # Bob chooses the private key b
    # b is the chosen private key
    b = 3
    print("The private key b for Bob:", b)

    # Gets the generated key
    y = power(G, b, P)

    # Generating the secret key after the exchange of keys
    ka = power(y, a, P) # Secret key for Alice
    kb = power(x, b, P) # Secret key for Bob

    print("Secret key for Alice is:", ka)
    print("Secret key for Bob is:", kb)

if __name__ == "__main__":
    main()

```

Output

```

The value of P : 23
The value of G : 9
The private key a for Alice : 4
The private key b for Bob : 3
Secret key for the Alice is : 9

```


Secret key for the Bob is : 9

Practice –Home Task 1: *Suggest 3 programing changes in above Diffie-Hellman Code, your updated code should generate code output.*

Practice -Lab Activity 2: Security Attack Threat of the Diffie-Hellman

Let's assume that the eavesdropper EVE knows the public values p and g like everyone else, and from her eavesdropping, she learns the values exchanged by Alice and Bob, $g^a \bmod p$ and $g^b \bmod p$, as well. With all her knowledge, she still can't compute the secret key S , as it turns out, if p and g are properly chosen, it's very, very hard for her to do.

For instance, you could brute force it and try all the options, but The calculations ($\bmod p$) make the discrete log calculation super slow when the numbers are large. If p and g have thousands of bits, then the best-known algorithms to compute discrete logs, although faster than plain brute force, will still take millions of years to compute.

Even with its immunity to brute force, it's vulnerable to MITM (man in the middle position).

SOLVED LAB ACTIVITY-

Lab Activity 2: Security Attack Threat of the Diffie-Hellman

Man in the Middle (MITM) against Diffie-Hellman:

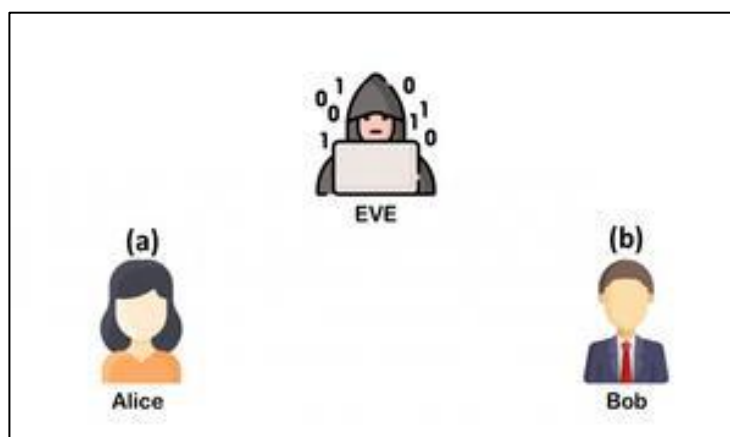
A malicious Malory, that has a MitM (man in the middle) position, can manipulate the communications between Alice and Bob, and break the security of the key exchange.

Step by Step explanation of this process:

Step 1: Selected public numbers p and g , p is a prime number, called the “modulus” and g is called the base.

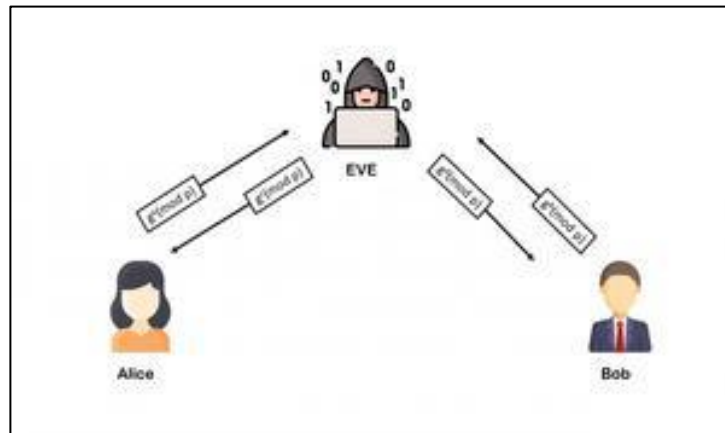
Step 2: Selecting private numbers.

Let Alice pick a private random number a and let Bob pick a private random number b , Malory picks 2 random numbers c and d .



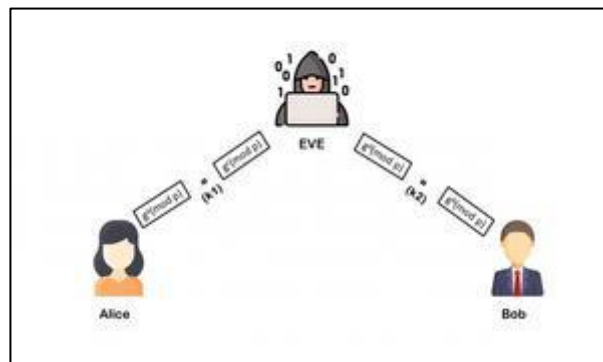
Step 3: Intercepting public values,

Malory intercepts Alice's public value ($g^a(\text{mod } p)$), block it from reaching Bob, and instead sends Bob her own public value ($g^c(\text{mod } p)$) and Malory intercepts Bob's public value ($g^b(\text{mod } p)$), block it from reaching Alice, and instead sends Alice her own public value ($g^d(\text{mod } p)$)



Step 4: Computing secret key

Alice will compute a key $S_1 = g^{da}(\text{mod } p)$, and Bob will compute a different key, $S_2 = g^{cb}(\text{mod } p)$.



Step 5: If Alice uses S_1 as a key to encrypt a later message to Bob, Malory can decrypt it, re-encrypt it using S_2 , and send it to Bob. Bob and Alice won't notice any problem and may assume their communication is encrypted, but in reality, Malory can decrypt, read, modify, and then re-encrypt all their conversations.

Implementation Code:

```
import random

# public keys are taken
# p is a prime number
# g is a primitive root of p
p = int(input('Enter a prime number : '))
g = int(input('Enter a number : '))
```

```

class A:
    def __init__(self):
        # Generating a random private number selected by alice
        self.n = random.randint(1, p)

    def publish(self):
        # generating public values
        return (g**self.n)%p

    def compute_secret(self, gb):
        # computing secret key
        return (gb**self.n)%p

class B:
    def __init__(self):
        # Generating a random private number selected for alice
        self.a = random.randint(1, p)
        # Generating a random private number selected for bob
        self.b = random.randint(1, p)
        self.arr = [self.a, self.b]

    def publish(self, i):
        # generating public values
        return (g**self.arr[i])%p

    def compute_secret(self, ga, i):
        # computing secret key
        return (ga**self.arr[i])%p

alice = A()
bob = A()
eve = B()

# Printing out the private selected number by Alice and Bob
print(f'Alice selected (a) : {alice.n}')
print(f'Bob selected (b) : {bob.n}')
print(f'Eve selected private number for Alice (c) : {eve.a}')
print(f'Eve selected private number for Bob (d) : {eve.b}')

# Generating public values
ga = alice.publish()
gb = bob.publish()
gea = eve.publish(0)
geb = eve.publish(1)
print(f'Alice published (ga): {ga}')

```

```
print(f'Bob published (gb): {gb}')
print(f'Eve published value for Alice (gc): {gea}')
print(f'Eve published value for Bob (gd): {geb}')

# Computing the secret key
sa = alice.compute_secret(gea)
sea = eve.compute_secret(ga,0)
sb = bob.compute_secret(geb)
seb = eve.compute_secret(gb,1)
print(f'Alice computed (S1) : {sa}')
print(f'Eve computed key for Alice (S1) : {sea}')
print(f'Bob computed (S2) : {sb}')
print(f'Eve computed key for Bob (S2) : {seb}')
```

Output:

```
Enter a prime number (p) : 227
Enter a number (g) : 14

Alice selected (a) : 227
Bob selected (b) : 170

Eve selected private number for Alice (c) : 65
Eve selected private number for Bob (d) : 175

Alice published (ga): 14
Bob published (gb): 101

Eve published value for Alice (gc): 41
Eve published value for Bob (gd): 32

Alice computed (S1) : 41
Eve computed key for Alice (S1) : 41

Bob computed (S2) : 167
Eve computed key for Bob (S2) : 167
```

Code output

```

===== RESTART: C:/Users/PAKISTAN/Documents/examp.py =====
Enter a prime number : 13
Enter a number : 3
Alice selected (a) : 6
Bob selected (b) : 11
Eve selected private number for Alice (c) : 7
Eve selected private number for Bob (d) : 13
Alice published (ga): 1
Bob published (gb): 9
Eve published value for Alice (gc): 3
Eve published value for Bob (gd): 3
Alice computed (S1) : 1
Eve computed key for Alice (S1) : 1
Bob computed (S2) : 9
Eve computed key for Bob (S2) : 9
|

```

Graded Home Task

Search about latest attack on key exchange algorithms and provide its implementation in python.

Lab 10

Cryptographic Math

This lab introduces basic number theory, Fermat's, Euclidean algorithm, Euler's and the Chinese remainder theorem, and gives a more in-depth look at modular arithmetic.

Activity Outcomes:

In this lab, students will gain following mathematical cryptographic knowledge:

- Gain an understanding of modular arithmetic
- Understand the importance of the greatest common divisor (GCD)
- Gain an understanding of Fermat's Theorem Euclidean algorithm and Euler's Theorem
- Gain an understanding of Chinese remainder Theorem.

<i>Sr.No</i>	<i>Allocated Time</i>	<i>Level of Complexity</i>	<i>CLO Mapping</i>
<i>1</i>	<i>20</i>	<i>Low</i>	<i>CLO-4</i>
<i>2</i>	<i>20</i>	<i>Medium</i>	<i>CLO-4</i>
<i>3</i>	<i>30</i>	<i>Medium</i>	<i>CLO-4</i>
<i>4</i>	<i>40</i>	<i>Medium</i>	<i>CLO-4</i>
<i>5</i>	<i>40</i>	<i>High</i>	<i>CLO-4</i>
<i>6</i>	<i>Home Task</i>	<i>High</i>	<i>CLO-4</i>

1) Practice -Lab Activity 1:

Modular Arithmetic and the Greatest Common Divisor

The logic of modular arithmetic began with the *quotient-remainder theorem*. The Quotient-remainder theorem states that for every integer A and positive B there exist different integers Q and R such that: $A = B * Q + R$, $0 < r < b$. When $a = 95$ and $b = 10$, what is the unique value of q (quotient) and r (remainder)? You find that the quotient equals 9 and the remainder equals 5. Once you understand the quotient-remainder theorem, it is easier to understand our first bit of cryptographic math: modular arithmetic. Here is an example: $23 \equiv 2 \pmod{7}$, which reads as "23 is equivalent to 2 mod 7." You can also type it into a search engine as 23 mod 7 to see the answer. You can further examine the modulo by stating that $a \equiv b \pmod{q}$ when a minus b is a multiple of q. Another way to state it numerically would be: $123 \equiv 13 \pmod{11}$ because $123 - 13 = 110 = 11 * 10$. An alternative way to think about it is to examine 53 (mod

13), which would be to say that 53 is equivalent to $53 - 13 = 40$, which is equivalent to 27, which is equivalent to 14, which is equivalent to 1, which is equivalent to -12 , which is equivalent to -25 , and so on. In fact, $53 \equiv \{53 + k \cdot 13 | \forall k \in \mathbb{Z}\}$ (you can read that as “is equivalent to the set of all numbers of form 53 plus an integer multiple of 13”). the modulus is represented by the % sign. To illustrate this example in Python, type the following:

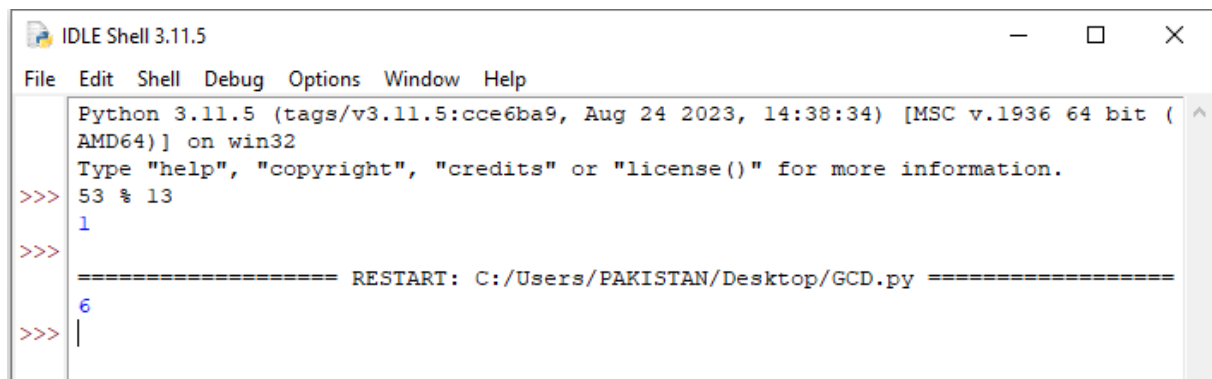
```
>>> 53 % 13
1
>>> 40 % 13
1
>>> 27 % 13
1
>>> 14 % 13
1
>>> -12 % 13
1
```

Now that you understand modular arithmetic, we turn our attention to the greatest common divisor (GCD). *The GCD is the largest number that perfectly divides two integers: a and b .* For example, the GCD of 12 and 18 is 6. This is an excellent opportunity to introduce **Euclid’s algorithm, which is a technique for finding the GCD of two integers with negligible effort.** To find the GCD of two integers A and B, use the following rules:

```
If A = 0 then GCD(A, B) = B
If B = 0 then GCD(A, B) = A
A = B * Q + R and B ≠ 0 then GCD (A, B) = GCD (B,R)
Therefore, write A using the quotient remainder form: A = B * Q + R
Find GCD(B, R)
```

Euclid’s algorithm works by continuously dividing one number into another and calculating the quotient and remainder at each step. Each phase produces a decreasing sequence of remainders, which terminate at zero, and the last non-zero remainder in the sequence is the GCD. You will revisit Euclid’s algorithm shortly when you examine the modular inverses; for now, you can use the Algorithm to write a GCD function in Python:

```
def gcd(a,b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)
print(gcd(12,18))
```



```
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 53 % 13
1
>>>
===== RESTART: C:/Users/PAKISTAN/Desktop/GCD.py =====
6
>>> |
```

Now that you know how to create your own GCD function, note that it is very inefficient due to its use of recursion. Prefer using **Python's built-in GCD function**, which is part of the standard Python math library.

Visit site to explore in detail: <https://www.geeksforgeeks.org/gcd-in-python/>

2) Practice -Lab Activity 2:

Prime Numbers

Prime numbers in cryptography are vital to the security of our encryption schemes. Prime factorization, also known as integer factorization, is a mathematical problem that is used to *secure public-key encryption schemes*. This is achieved by using extremely large semiprime numbers that are a result of the multiplication of two prime numbers. As you may remember, *a prime number is any number that is only divisible by 1 and itself*. The first prime number is 2. Additional prime numbers include 3, 5, 7, 11, 13, 17, 19, 23, and so on. An infinite number of prime numbers exist, and all numbers have one prime factorization. A semiprime number, also known as biprime, 2-almost prime, or a pq number, is a natural number that is the product of two prime numbers. The semiprimes less than 50 are 4, 6, 9, 10, 14, 15, 21, 22, 25, 26, 33, 34, 35, 38, 39, 46, and 49. Prime numbers are significant in cryptography. Here is a simple Python script that tests if an integer value is prime:

```
def isprime(x):
    x = abs(int(x)) # Ensure x is a non-negative integer
    if x < 2:
        return False # Numbers less than 2 are not prime
    elif x == 2:
        return True # 2 is the only even prime number
    elif x % 2 == 0:
        return False # Other even numbers are not prime
    else:
        for n in range(3, int(x ** 0.5) + 1, 2):
            if x % n == 0:
                return False # Found a factor, thus not prime
```



```

    return True # No factors found, thus prime

# Example usage
print(isprime(100000007)) # It will return True or False

```

3) Practice -Lab Activity 3:

Modular Arithmetic and the Greatest Common Divisor: Fermat's Little Theorem

Fermat's Little Theorem Fermat's little theorem is used in number theory to compute the powers of integers modulo prime numbers. The theorem is a special case of Euler's theorem. Fermat's little theorem states let p be a prime number, and a be any integer. If n is a prime number, then for every a , $1 < a < p - 1$, $a^{p-1} \equiv 1 \pmod{p}$ or $a^{p-1} \% p = 1$. To ensure this makes sense, let's look at an example: $p =$ prime integer number $a =$ integer which is not a multiple of p . According to Fermat's little theorem, $2^{(17-1)} \equiv 1 \pmod{17}$ $65,536 \% 17 \equiv 1$. This means $(65,536 - 1)$ is a multiple of 17. This is proven by multiplying $17 * 3,855$, which equals $(65,536 - 1)$ or 65,535. If you know the modulo m is prime, then you can also use Fermat's little theorem to find the inverse. Here is a quick and easy function that will return whether an integer is prime or not:

```

def CheckIfProbablyPrime(x):
    if x < 2:
        return False
    return pow(2, x-1, x) == 1

```

Example Usage

```

>>> CheckIfProbablyPrime(19)
True
>>> CheckIfProbablyPrime(31)
True
>>> CheckIfProbablyPrime(589)
False

```

4) Practice -Lab Activity 4: Euclidean algorithm

The Euclidean algorithm is a way to find the greatest common divisor of two positive integers. GCD of two numbers is the largest number that divides both of them. A simple way to find GCD is to factorize both numbers and multiply common prime factors.

$$\begin{aligned} 36 &= 2 \times 2 \times 3 \times 3 \\ 60 &= 2 \times 2 \times 3 \times 5 \end{aligned}$$

$$\begin{aligned} \text{GCD} &= \text{Multiplication of common factors} \\ &= 2 \times 2 \times 3 \\ &= 12 \end{aligned}$$

Basic Euclidean Algorithm for GCD:

The algorithm is based on the below facts.

- If we subtract a smaller number from a larger one (we reduce a larger number), GCD doesn't change. So if we keep subtracting repeatedly the larger of two, we end up with GCD.
- Now instead of subtraction, if we divide the larger number, the algorithm stops when we find the remainder 0.

Python3 program to demonstrate Basic Euclidean Algorithm

```
# Function to return gcd of a and b
def gcd(a, b):
    if a == 0:
        return b

    return gcd(b % a, a)

# Driver code
if __name__ == "__main__":
    a = 10
    b = 15
    print("gcd(", a, ",", b, ") = ", gcd(a, b))

    a = 35
    b = 10
    print("gcd(", a, ",", b, ") = ", gcd(a, b))

    a = 31
    b = 2
    print("gcd(", a, ",", b, ") = ", gcd(a, b))
```

Output

```
GCD(10, 15) = 5
GCD(35, 10) = 5
GCD(31, 2) = 1
```

5) Practice -Lab Activity5: Euler's Theorem

Earlier in this lab, you were introduced to Fermat's little theorem. We will now examine

a generalization of Fermat's theorem known as Euler's theorem. Both Fermat's and Euler's theorems play an important role in public-key cryptography, which will be explored in greater detail in next lab. In number theory, Euler's theorem, also known as *Euler's totient theorem* or the Fermat–Euler theorem, states that *if n and a are coprime positive integers, then $a\varphi(n) \equiv 1 \pmod n$ where $\varphi(n)$ is Euler's totient function*. In 1736, Leonhard Euler published his proof of Fermat's little theorem, which Fermat had presented without proof. Subsequently, Euler presented other proofs of the theorem, culminating with “Euler's theorem” in his paper of 1763, in which he attempted to find the smallest exponent for which Fermat's little theorem was always true.

Euler investigated the properties of numbers; he specifically studied the distribution of prime numbers. One crucial function he defined is named the PHI function; the PHI function measures the breakability of a number. Assume you have the number n ; the function calculates the number of integers that are less than or equal to n and do not share any common factor with n ; you see it in the following notation: $\phi[n]$.

For example, if you wanted to examine $\phi[8]$, you would examine all values from 1 to 8 and count all integers with which 8 does not share a factor greater than 1; the numbers are 1, 3, 5, 7. The function produces 4. As it turns out, calculating the PHI of a prime number P is simple. $\phi[P] = P - 1$. To calculate $\phi[7]$, you count all integers except 7 since none of the integers share a factor with 7; therefore, $\phi[7] = 6$. Assume a larger prime such as 21,377. $\phi[21,377] = 21,376$. The equation looks like the following:

$$a^{p-1} \equiv 1 \pmod p$$

Euler totient functions offer benefits to speed up modular inverse computations. Euler's totient function $\Phi(n)$ for an input n is the count of numbers in the format of $\{1, 2, 3, 4, 5, n\}$ that are relatively prime to n , i.e., the numbers whose GCD with n is 1. Examine the following six examples, which calculate the Euler's totient function $\Phi(n)$ in respect to the inputs 1 through 6. The output will be the number of positive integers that do not exceed n and also have no common divisors with n other than the common divisor 1: $\Phi(1) = 1$ gcd (1, 1) is 1 $\Phi(2) = 1$ gcd (1, 2) is 1, but gcd (2, 2) is 2 $\Phi(3) = 2$ gcd (1, 3) is 1 and gcd (2, 3) is 1 $\Phi(4) = 2$ gcd (1, 4) is 1 and gcd (3, 4) is 1 $\Phi(5) = 4$ gcd (1, 5) is 1, gcd (2, 5) is 1, gcd (3, 5) is 1, and gcd (4, 5) is 1 $\Phi(6) = 2$ gcd (1, 6) is 1 and gcd (5, 6) is 1

Use the following Python to test Euler's totient function on integers 1 through 20. The output should resemble Figure 4.3.

```
import math
def phi(n):
    amount = 0
    for k in range(1, n + 1):
        if math.gcd(n, k) == 1:
            amount += 1
    return amount
for n in range(1,20) :
    print("Φ(",n,") = ",phi(n))
```

in above Python program, you can use the matplotlib library and create a graph, as shown here:

```

import math
import numpy as np
from matplotlib import pyplot as plt
def phi(n):
    amount = 0
    for k in range(1, n + 1):
        if math.gcd(n, k) == 1:
            amount += 1
    return amount
for i in range(500):
    phi_n = phi(i)
    #print(i, phi_n)
    plt.plot(i, phi_n, 'o')
plt.xlabel("Value of x")
plt.ylabel("Value of y")
plt.title("Euler's Theorem")
plt.show()

```

Figure 4.4 shows the output of using Euler's theorem and the MatPlot library to create a graph. Figure 4.3: Euler.py tes

```

Φ( 1 ) = 1
Φ( 2 ) = 1
Φ( 3 ) = 2
Φ( 4 ) = 2
Φ( 5 ) = 4
Φ( 6 ) = 2
Φ( 7 ) = 6
Φ( 8 ) = 4
Φ( 9 ) = 6
Φ( 10 ) = 4
Φ( 11 ) = 10
Φ( 12 ) = 4
Φ( 13 ) = 12
Φ( 14 ) = 6
Φ( 15 ) = 8
Φ( 16 ) = 8
Φ( 17 ) = 16
Φ( 18 ) = 6
Φ( 19 ) = 18
Φ( 20 ) = 8

Press any key to continue . . .

```

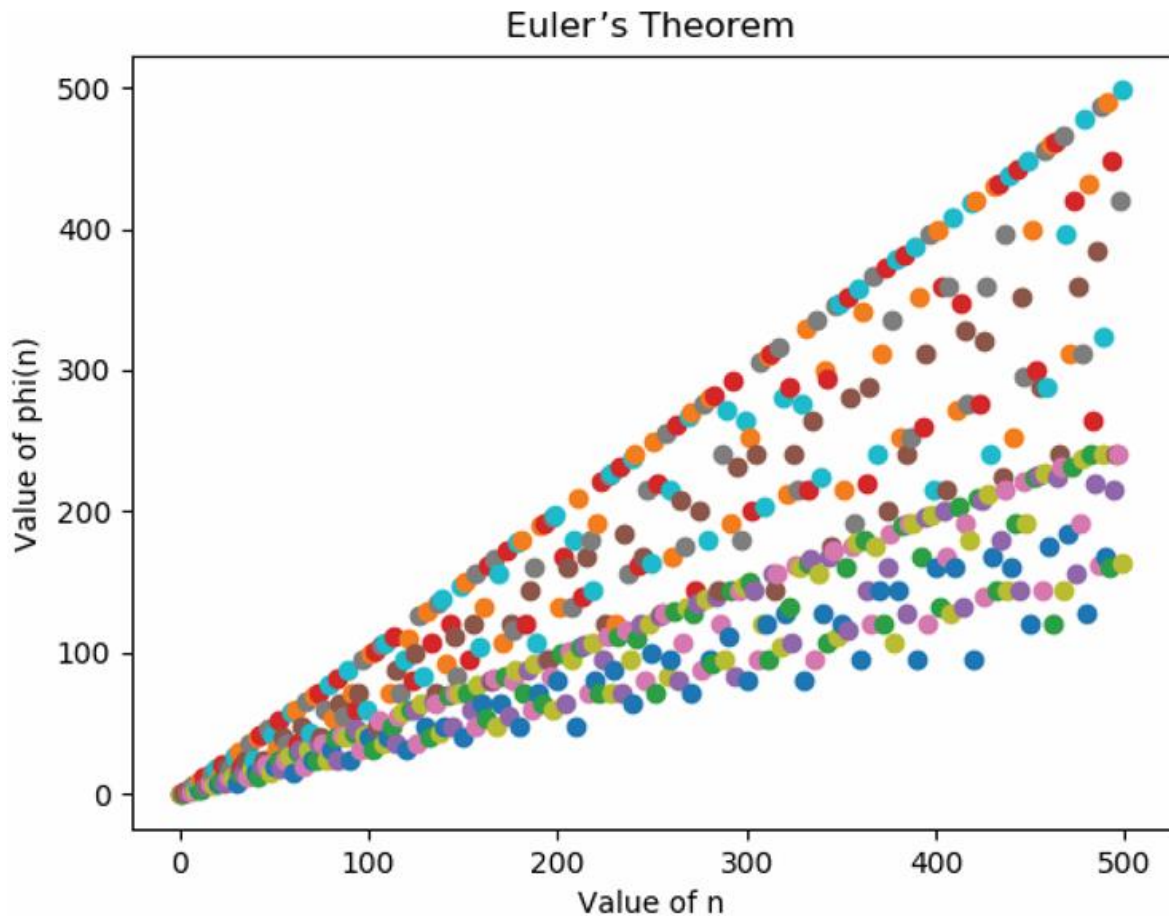


Figure 4.4: EulerPlot.py tes

6) Practice –Home Task 5: Chinese Remainder Theorem:

Chinese Remainder Theorem:

The Chinese Remainder Theorem states that for positive integers **num**[0], **num**[1], ..., **num**[**k**-1] that are pairwise coprime, and any given sequence of integers **rem**[0], **rem**[1], ..., **rem**[**k**-1], there exists an integer **x** that solves the system of simultaneous congruences as described earlier.

Chinese Remainder Theorem in Python Using Naive Approach:

*Fetching the mystery number **x** is fairly easy if we start with 1 and keep on seeing if numbers being divided by the elements in **num**[] equals the corresponding remainders in **rem**[]. Upon locating a variable like **x**, the algorithm stops, and the solution is given. Yet, it can lead to inefficiency in cases with higher than average **num**[].*

Steps-by-step approach for Chinese Remainder Theorem in Python:

- Initialize **x** to 1 as the starting point for the search.
- Iterate through each **congruence** and check if **x** satisfies the congruence by checking if **x mod nums[j]=rems[j]**.
- If **x** satisfies all **congruences**, then we have found the solution **x** and **return** it.
- If **x** does not satisfy all congruences, increment **x** by 1 and **repeat** the process.

```

def find_min_x(nums, rems):
    # Initialize result
    x = 1

    while True:
        # Check if remainder of x % nums[j] is rem[j] for all j from 0 to k-1
        for j in range(len(nums)):
            if x % nums[j] != rems[j]:
                break

        # If all remainders matched, we found x
        if j == len(nums) - 1:
            return x

        # Else, try the next number
        x += 1

    return x

# Example Usage
nums = [3, 4, 5]
rems = [2, 3, 1]
print(find_min_x(nums, rems))

```

Output

11

Lab 11

ElGamal encryption

This lab will introduce students to ElGamal encryption (1984), T. ElGamal announced a public-key scheme based on discrete logarithms, closely related to the Diffie–Hellman technique [ELGA84, ELGA85]. The ElGamal cryptosystem is used in some form in a number of standards including the digital signature standard (DSS), which is covered in Chapter 13, and the S/MIME email standard (Chapter 21).

Activity Outcomes:

This lab teaches you the following topics:

- Present an overview of the ElGamal cryptographic system implementaton

<i>Sr.No</i>	<i>Allocated Time</i>	<i>Level of Complexity</i>	<i>CLO Mapping</i>
<i>1</i>	<i>60</i>	<i>High</i>	<i>CLO-4</i>

.Instructor Note:

As pre-lab activity, read Chapter 10 from the book (Learn cryptography and Network Security Principles and Practice by William Stallings.,) for concept understanding.

1) Idea of ElGamal Cryptosystem:

El-Gamal encryption is a public-key cryptosystem. El-Gamal uses asymmetric key encryption for communicating between two parties and encrypting the message. This cryptosystem is based on the difficulty of finding a discrete logarithm in a cyclic group. That is, even if we know g^a and g^k , it is extremely difficult to compute g^{ak} . In this section, we will examine the basic idea of the cryptosystem with an example using cryptography's: Alice and Bob.

Components of the ElGamal Algorithm

1. Key Generation:

- Public Parameters: Select a large prime number p and a generator g of the multiplicative group \mathbb{Z}^*_p .
- Private Key: Select a private key x such that $1 \leq x \leq p-2$.
- Public Key: Compute $h = g^x \bmod p$. The public key is (p, g, h) and the private key is x .

2. Encryption:

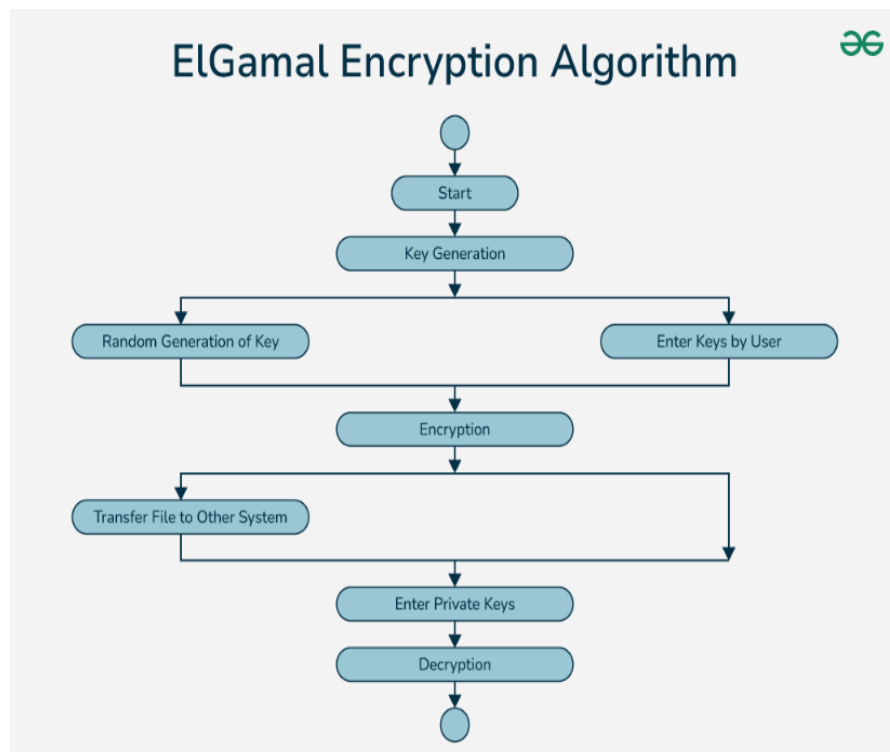
- To encrypt a message M :
 - Choose a random integer k such that $1 \leq k \leq p-2$.
 - Compute $C1 = g^k \bmod p$.
 - Compute $C2 = M \cdot h^k \bmod p$.
 - The ciphertext is $(c1, c2)$.

3. Decryption:

- To decrypt the ciphertext $(c1, c2)$ using the private key x :
 - Compute the shared secret $s = C1^x \bmod p$.
 - Compute $s^{-1} \bmod p$ (the modular inverse of s).
 - Compute the original message $M = C2 \cdot s^{-1} \bmod p$.
 -

Suppose Alice wants to communicate with Bob.

1. Bob generates public and private keys:
 - Bob chooses a very large number q and a cyclic group F_q .
 - From the cyclic group F_q , he choose any element g and an element a such that $\gcd(a, q) = 1$.
 - Then he computes $h = g^a$.
 - Bob publishes F , $h = g^a$, q , and g as his public key and retains a as a private key.
2. Alice encrypts data using Bob's public key :
 - Alice selects an element k from cyclic group F such that $\gcd(k, q) = 1$.
 - Then she computes $p = g^k$ and $s = h^k = g^{ak}$.
 - She multiples s with M .
 - Then she sends $(p, M*s) = (g^k, M*s)$.
3. Bob decrypts the message :
 - Bob calculates $s' = p^a = g^{ak}$.
 - He divides $M*s$ by s' to obtain M as $s = s'$.



The following Python code will help you gain an understanding of these steps.

```

import random
from math import pow, gcd

# Generating large random numbers
def gen_key(q):
    key = random.randint(pow(10, 20), q)
    while gcd(q, key) != 1:
        key = random.randint(pow(10, 20), q)
    return key

# Compute the power
def power(a, b, c):
    x = 1
    y = a
    while b > 0:
        if b % 2 == 0:
            x = (x * y) % c
            y = (y * y) % c
            b //= 2 # Use integer division
    return x % c

# Encrypt the message
def encrypt(msg, q, h, g):
    en_msg = []
    k = gen_key(q) # Private key for sender
    s = power(h, k, q)
    p = power(g, k, q)

    print("g^k used : ", p)
    print("g^ak used : ", s)

    for char in msg:
        en_msg.append(s * ord(char))
    return en_msg, p

# Decrypt the message
def decrypt(en_msg, p, key, q):
    dr_msg = []
    h = power(p, key, q)

    for value in en_msg:
        dr_msg.append(chr(int(value / h)))
    return dr_msg

def main():
    msg = 'Please do not let the enemy know our position.'
    print("Original Message :", msg)
    print()

```

```

q = random.randint(pow(10, 20), pow(10, 50))
g = random.randint(2, q)
key = gen_key(q) # Private key for receiver
h = power(g, key, q)

print("g used : ", g)
print("g^a used : ", h)

en_msg, p = encrypt(msg, q, h, g)
dr_msg = decrypt(en_msg, p, key, q)
dmsg = ".join(dr_msg)

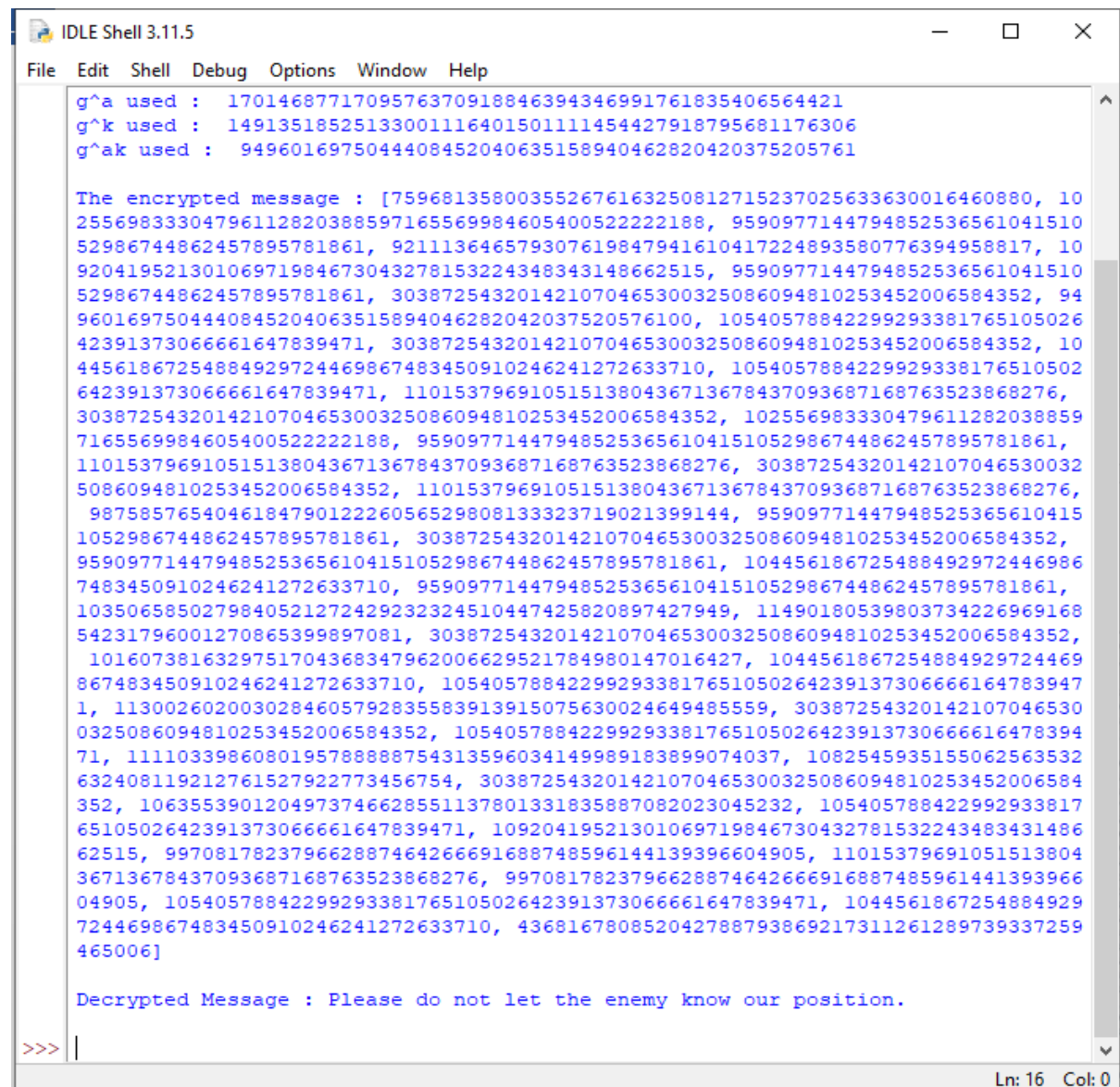
print()
print("The encrypted message :", en_msg)
print()
print("Decrypted Message :", dmsg)
print()

if __name__ == '__main__':
    main()

```

The preceding code should take the message “Please do not let the enemy know our position.” and generate an ElGamal key that is used to encrypt and decrypt the message. Examine Figure. The program displays the g^a and g^k that is produced along with the g^{ak} .

Code Output



```
IDLE Shell 3.11.5
File Edit Shell Debug Options Window Help

g^a used : 1701468771709576370918846394346991761835406564421
g^k used : 14913518525133001116401501111454427918795681176306
g^ak used : 9496016975044408452040635158940462820420375205761

The encrypted message : [759681358003552676163250812715237025633630016460880, 10
25569833304796112820388597165569984605400522222188, 9590977144794852536561041510
52986744862457895781861, 921113646579307619847941610417224893580776394958817, 10
92041952130106971984673043278153224348343148662515, 9590977144794852536561041510
52986744862457895781861, 303872543201421070465300325086094810253452006584352, 94
9601697504440845204063515894046282042037520576100, 10540578842299293381765105026
42391373066661647839471, 303872543201421070465300325086094810253452006584352, 10
44561867254884929724469867483450910246241272633710, 1054057884229929338176510502
642391373066661647839471, 1101537969105151380436713678437093687168763523868276,
303872543201421070465300325086094810253452006584352, 102556983330479611282038859
7165569984605400522222188, 959097714479485253656104151052986744862457895781861,
1101537969105151380436713678437093687168763523868276, 30387254320142107046530032
5086094810253452006584352, 1101537969105151380436713678437093687168763523868276,
987585765404618479012226056529808133323719021399144, 95909771447948525365610415
1052986744862457895781861, 303872543201421070465300325086094810253452006584352,
959097714479485253656104151052986744862457895781861, 104456186725488492972446986
7483450910246241272633710, 959097714479485253656104151052986744862457895781861,
1035065850279840521272429232324510447425820897427949, 11490180539803734226969168
54231796001270865399897081, 303872543201421070465300325086094810253452006584352,
1016073816329751704368347962006629521784980147016427, 1044561867254884929724469
867483450910246241272633710, 105405788422992933817651050264239137306666164783947
1, 1130026020030284605792835583913915075630024649485559, 30387254320142107046530
0325086094810253452006584352, 10540578842299293381765105026423913730666616478394
71, 1111033986080195788888754313596034149989183899074037, 1082545935155062563532
632408119212761527922773456754, 303872543201421070465300325086094810253452006584
352, 1063553901204973746628551137801331835887082023045232, 105405788422992933817
6510502642391373066661647839471, 10920419521301069719846730432781532243483431486
62515, 997081782379662887464266691688748596144139396604905, 11015379691051513804
36713678437093687168763523868276, 9970817823796628874642666916887485961441393966
04905, 1054057884229929338176510502642391373066661647839471, 1044561867254884929
724469867483450910246241272633710, 436816780852042788793869217311261289739337259
465006]

Decrypted Message : Please do not let the enemy know our position.

>>> |
```

Ln: 16 Col: 0

Figure: ElGamal key

Lab 12

RSA

The pioneering paper by Diffie and Hellman [DIFF76b] introduced a new approach to cryptography and, in effect, challenged cryptologists to come up with a cryptographic algorithm that met the requirements for public-key systems. One of the first successful responses to the challenge was developed in 1977 by Ron Rivest, Adi Shamir, and Len Adleman at MIT and first published in 1978 [RIVE78]. The Rivest-Shamir-Adleman (RSA) scheme has since that time reigned supreme as the most widely accepted and implemented general-purpose approach to public-key encryption.

Activity Outcomes:

This lab teaches you the following topics:

- Gain an understanding of the importance of PKI
- Learn how to implement a PKI solution in Python
- Gain an understanding of RSA

Instructor Note:

As pre-lab activity, read Chapter 9, page 297-300 from the book (Cryptography and Network Security Principles and Practice Eighth Edition Global Edition William Stallings) to know the basics of RSA.

1) Solved lab Activity:

This example provide simple implementation of the RSA algorithm that can encrypt and decrypt a message. Note that the keys here are far too small to be of practical use since they are still relatively easy to factor:

```
import random

# Euclid's algorithm to find the greatest common divisor (GCD) of two numbers
def gcd(a, b):
    while b != 0:
        a, b = b, a % b
    return a

# Extended Euclid's algorithm to find the multiplicative inverse
def multiplicative_inverse(e, phi):
    d = 0
    x1 = 0
    x2 = 1
    y1 = 1
    temp_phi = phi
    while e > 0:
        temp1 = temp_phi // e
        temp2 = temp_phi - temp1 * e
        temp_phi = e
        e = temp2
        x = x2 - temp1 * x1
        y = d - temp1 * y1
        x2 = x1
        x1 = x
        d = y1
        y1 = y
    if temp_phi == 1:
        return d + phi

# Verify if the given number is prime
def is_prime(num):
    if num == 2:
        return True
    if num < 2 or num % 2 == 0:
        return False
    for n in range(3, int(num**0.5) + 1, 2):
        if num % n == 0:
```

```

        return False
    return True

# Generate RSA keypair
def generate_keypair(p, q):
    if not (is_prime(p) and is_prime(q)):
        raise ValueError('Both numbers must be prime.')
    elif p == q:
        raise ValueError('p and q cannot be equal')
    n = p * q
    phi = (p - 1) * (q - 1)
    e = random.randrange(1, phi)
    g = gcd(e, phi)
    while g != 1:
        e = random.randrange(1, phi)
        g = gcd(e, phi)
    d = multiplicative_inverse(e, phi)
    return ((e, n), (d, n))

# Encrypt the plaintext using the public key
def encrypt(pk, plaintext):
    key, n = pk
    cipher = [(ord(char) ** key) % n for char in plaintext]
    return cipher

# Decrypt the ciphertext using the private key
def decrypt(pk, ciphertext):
    key, n = pk
    plain = [chr((char ** key) % n) for char in ciphertext]
    return ''.join(plain)

if __name__ == '__main__':
    print("Chapter 8 - Understanding RSA")

    while True:
        try:
            p = int(input("Enter a prime number: "))
            q = int(input("Enter a second distinct prime number: "))
            break # Exit the loop if both inputs are valid
        except ValueError:
            print("Invalid input. Please enter valid integers.")

    print("\nGenerating your public/private keypairs . . .")
    public, private = generate_keypair(p, q)

```

```

print("\nYour public key is {} and your private key is {}\n".format(public, private))

message = input("Enter a message to encrypt with your public key: ")
encrypted_msg = encrypt(public, message)
print("Your encrypted message is: {}".format(' '.join(map(str, encrypted_msg))))

print("\nDecrypting message with your personal private key {} . .
.".format(private))
print("\nYour message is: {}\n".format(decrypt(private, encrypted_msg)))

```

Code Output

```

Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:/Users/PAKISTAN/Documents/RSA.py =====
Chapter 8 - Understanding RSA
Enter a prime number: 3
Enter a second distinct prime number: 11

Generating your public/private keypairs . . .

Your public key is (13, 33) and your private key is (17, 33)

Enter a message to encrypt with your public key: |

```

To produce the output in Figure 8.1, enter 11 as your first prime and 17 as your second.

```

Select C:\Program Files (x86)\Microsoft Visual Studio\Shared\Python37_64\python.exe
Chapter 8 - Understanding RSA
Enter a prime number: 11
Enter a second distinct prime number: 17
Generating your public/private keypairs . . .
Your public key is (33, 187) and your private key is (97, 187)
Enter a message to encrypt with your public key: Hello World!
Your encrypted message is: 18611891911113287111979110033
Decrypting message with your personal private key (97, 187) . . .
Your message is: Hello World!
Press any key to continue . . .

```


Lab 13

Elliptic Curve Cryptography

This lab will introduce students to Elliptic Curve Cryptography in Python. Students will also get in depth of ECC mechanism with the process of key generation.

Activity Outcomes:

This lab teaches you the following topics:

- Basic concept understanding of ECC.
- ECC key generation process.
- Implementing Basic Operations for ECC

Instructor Note:

As pre-lab activity, read Chapter 10 from the book (Learn cryptography and Network Security Principles and Practice by William Stallings,) for concept understanding.

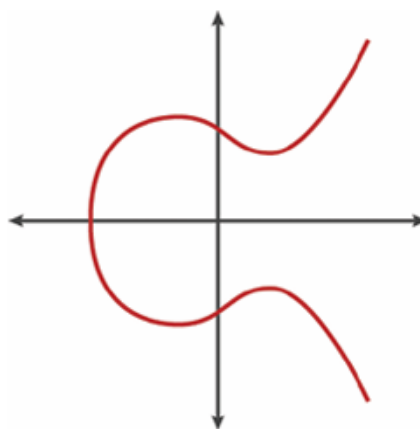
1) Elliptic Curve Cryptography: Concept

Now that you have a better understanding of how the more traditional algorithms work using Python, we will examine an alternative approach that is considered a more efficient type of public-key cryptography: elliptic curve cryptography, or as it is more simply known, ECC. The security of the cryptosystem lies within the difficulty of solving discrete logarithms on the field defined by specific equations computed over a curve; the group of cryptographic algorithms were introduced in 1985 and were based on the esoteric branch of mathematics called elliptic curves. Although the system was introduced in the mid '80s, it took another twenty years for the cryptosystem to gain wide acceptance. Several factors are contributing to its increasing popularity. First, the security of 1024-bit RSA encryption is degrading due to faster computing and a better understanding and analysis of encryption methods. While brute force is still unlikely to crack 1024-bit RSA keys, other approaches, including highly intensive parallel computing in distributed computing arrays, are resulting in more sophisticated attacks. These attacks have reduced the effectiveness of this level of security. Even 2,048-bit encryption is estimated by the RSA Security to be effective only until 2030. A second factor that is contributing to the adoption of ECC is that many government entities have started to accept ECC as an encryption method. Third, the authentication speed of ECC is faster than RSA in terms of server authentication. Finally, certificate authorities have started embedding ECC algorithms into their SSL certificates.

ECC was independently suggested by Neal Koblitz (University of Washington) and Victor S. Miller (IBM) in 1985. After the introduction of Diffie-Hellman and RSA, cryptographers started exploring other mathematics-based cryptographic solutions looking for other algorithms that would offer easy one-way calculations that were hard to find an inverse for; these types of functions are referred to as trapdoors. A trapdoor function is a function that is easy to perform one way but has a secret that is required to perform the inverse calculation efficiently. That is, if f is a trapdoor function, then $y = f(x)$ is easy to compute, but $x = f^{-1}(y)$ is hard to compute without some special knowledge k . Unless you have a mathematical background, elliptic curves may be new to you; so what exactly is an elliptic curve and how does the elliptic curve trapdoor function work?

An elliptic curve is the set of points that satisfy a specific mathematical equation. The equation for an elliptic curve looks something like this:

$$y^2 = x^3 + ax + b$$



That graphs to something that looks like Figure.

The most important takeaway for this section is that you understand that ECC produces encryption keys based on using points on a curve to define the public and private keys. An ECC key is very helpful for the current generation as more people are moving to the smartphone. As the utilization of smartphones continues to grow, there is an emerging need for a more flexible encryption for business to meet with increasing security requirements.

The elliptic curve cryptography certificates allow key size to remain small while providing a higher level of security. The ECC certificate key creation method is entirely different from previous algorithms, while relying on the use of a public key for encryption and a private key for decryption. By starting small and with a slow growth potential, ECC has a longer potential life span. Elliptic curves are likely to be the next generation of cryptographic algorithms, and we are seeing the beginning of their use now.

When you compare ECC with other algorithms like RSA, you will find the ECC key is significantly smaller yet offers the same level of security. One notable instance is that a 3,072-bit RSA key takes 768 bytes, whereas the equally strong NIST P-256 private key only takes 32 bytes (that is, 256 bits). PyCryptodome offers us an ECC module that provides mechanisms for generating new ECC keys, exporting and importing them using widely supported formats like PEM or DER.

To install PyCryptodome, execute the following pip command:

```
pip install pycryptodome
```

if you're worried about ensuring the highest level of security while maintaining performance, it makes sense to adopt ECC.

Practice -Lab Activity :

Generating ECC Keys

ECC private keys are integers that represent the curve's field size; the typical size is 256 bits. A 256-bit private key would look like the following:

```
0x51897b64e85c3f714bba707e867914295a1377a7463a9dae8ea6a8b914246319
```

Generating an ECC key requires generating a random integer within a specified range.

The public keys in the ECC are EC points—pairs of integer coordinates {x, y}, lying on the curve. Due to their special properties, EC points can be compressed to just one coordinate + 1 bit (odd or even). Thus the compressed public key, corresponding to a 256-bit ECC private key, is a 257-bit integer. An example of an ECC public key (corresponding to the preceding private key, encoded in the Ethereum format, as hex with prefix 02 or 03) is 0x02f54ba86dc1ccb5bed0224d23f01ed87e4a443c47fc690d7797a13d41d2340e1.

In this format, the public key takes 33 bytes (66 hex digits), which can be optimized to

exactly 257 bits.

The following example demonstrates how to generate a new ECC key, export it, and reload it back into your program. The code uses the NIST P-256 algorithm, which is the most-used elliptic curve, and there are no reasons to believe it's Insecure:

```
from Crypto.PublicKey import ECC
key = ECC.generate(curve='P-256')
f = open('myprivatekey.pem','wt')
f.write(key.export_key(format='PEM'))
f.close()
f = open('myprivatekey.pem','rt')
key = ECC.import_key(f.read())
print (key)
```

The key generated will look similar to the following:

```
EccKey(curve='NIST P-256',
point_x=85511317925193091591538005554467283386311991513737248626228047
21 0045098335773,
point_y=62834027958545080347454491553206133116570260879291164814224928
59 9382610602892,
d=2063641786698337143130043788498991558397573514836970341582277689437
768 1606808)
```

Practice -Lab Task:

Implementing Basic Operations

- **Task:** Implement basic elliptic curve operations:
 - Point addition
 - Point doubling
 - Scalar multiplication (using the double-and-add method)
- **Deliverable:** Code implementations in Python or another programming language of your choice, along with test cases.

Practice -Lab Task:

Digital Signatures

- **Task:** Implement the Elliptic Curve Digital Signature Algorithm (ECDSA):
 - Create functions for signing a message and verifying a signature.
- **Deliverable:** A working implementation with test cases demonstrating the signing and verification process.

Lab 14

Digital signatures

Objective:

This lab will enable students to understand Digital signature generation process, along with different types cryptographic mechanisms used in Digital signatures.

Activity Outcomes:

This lab teaches you the following topics:

- *Gain an understanding of Digital signatures.*
- *Learn how to implement a Digital signatures solution in Python*
- *Gain an understanding of DSA*

Instructor Note:

As pre-lab activity, read Chapter 13, from the book (Cryptography and Network Security Principles and Practice Eighth Edition Global Edition William Stallings) to know the basics of Digital signatures

Practice -Lab Activity 1:

Digital Signature is a verification method. Digital signatures do not provide confidential communication. If you want to achieve confidentiality, both the message and the signature must be encrypted using either a secret key or a public key cryptosystem. This additional layer of security can be incorporated into a basic digital signature scheme.

Lamport One Time Signature

Lamport One Time Signature is a method for constructing a digital signature and typically involved the use of a cryptographic hash function. As it is a one-time signature scheme, it can only be used to securely sign one message. Suppose Alice wants to digitally sign her message to Bob, the process can be explained in 3 steps:

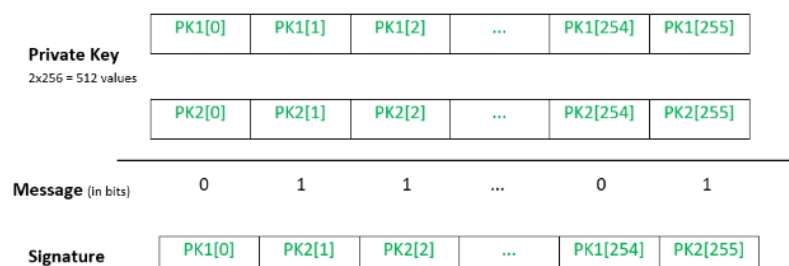
1. Key Generation
2. Signature Generation
3. Signature Verification.

1. Key Generation:

- Alice first needs to create a Lamport key pair, a private key and a corresponding public key.
- In order to create the private key, a secure random number generator is used to generate 256 pairs of random numbers. Each number consists of 256 bits. Alice will store this private key securely. Remember that the private key is not meant to be shared with anyone.
- In order to create the public key, Alice hashes each of the 512 numbers of her private key. This will produce another 512 numbers, each consisting of 256 bits. This is the public key that will be shared with anyone.

2. Signature Generation:

- Alice hashes her message using a 256-bit cryptographic hash function, eg SHA 256, to obtain a 256-bit digest.
- For each bit, depending on whether the bit value is 1 or 0, Alice will pick the corresponding number from the pair of numbers of her private key i.e. if the bit is 0, the first number is chosen, and if the bit is 1, the second number is chosen. This results in a sequence of 256 numbers which is her signature.



- Alice sends the message along with her signature to Bob.

3. Signature Verification:

- Bob hashes the message using the same 256-bit cryptographic hash function, to obtain a 256-bit digest.
- For each bit, depending on whether the bit value is 1 or 0, Bob will pick the corresponding number from Alice's public key i.e if the first bit of the message hash is 0, he picks the first hash in the first pair, and so on. This is done in the same manner as shown in the diagram above. This results in a sequence of 256 numbers.
- Bob hashes each of the numbers in Alice's signature to obtain a 256-bit digest. If this matches the sequence of 256 numbers that Bob had previously picked out, the signature is valid.

```
import hashlib
import secrets

# Creation of keys
def keygen():
    skey = [[0] * 255, [1] * 255]
    for i in range(len(skey)):
        for j in range(len(skey[i])):
            skey[i][j] = bin(secrets.randbits(255))[2:]
            skey[i][j] = '0' * (255 - len(skey[i][j])) + skey[i][j]

    pkey = [[0] * 255, [1] * 255]
    for i in range(len(pkey)):
        for j in range(len(pkey[i])):
            pkey[i][j] = hashlib.sha256(skey[i][j].encode()).hexdigest() # Store as hex
string

    keypair = [skey, pkey]
    return keypair

# Signature generation of the message
def signgen(message, skey):
    mhash = int(hashlib.sha256(message.encode()).hexdigest(), 16)
    signature = [0] * 255

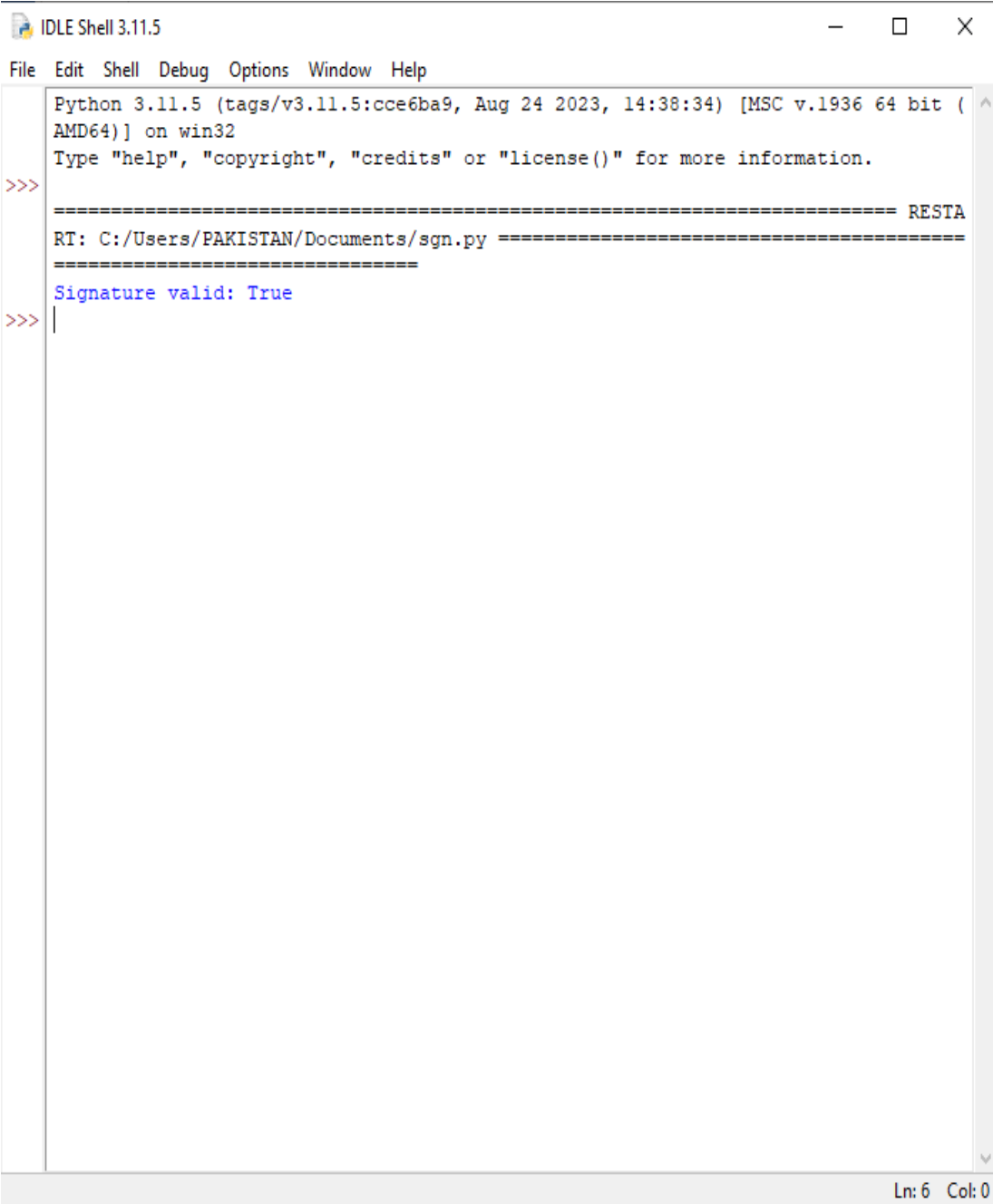
    for i in range(255):
        k = (mhash >> i) & 1 # Directly extract the bit
        signature[i] = skey[k][i]

    return signature

# Verification of signature
def verification(message, pkey, signature):
    mhash = int(hashlib.sha256(message.encode()).hexdigest(), 16)
    for i in range(255):
        k = (mhash >> i) & 1 # Directly extract the bit
```

```
    verify = hashlib.sha256(signature[i].encode()).hexdigest() # Get hash of the
signature
    if pkey[k][i] != verify:
        return False
    return True

# Example usage
keypair = keygen()
message = "I am god."
signature = signgen(message, keypair[0])
print("Signature valid:", verification(message, keypair[1], signature))
```



```
IDLE Shell 3.11.5
File Edit Shell Debug Options Window Help
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTA
RT: C:/Users/PAKISTAN/Documents/sgn.py =====
>>> Signature valid: True
>>> |
```

Ln: 6 Col: 0

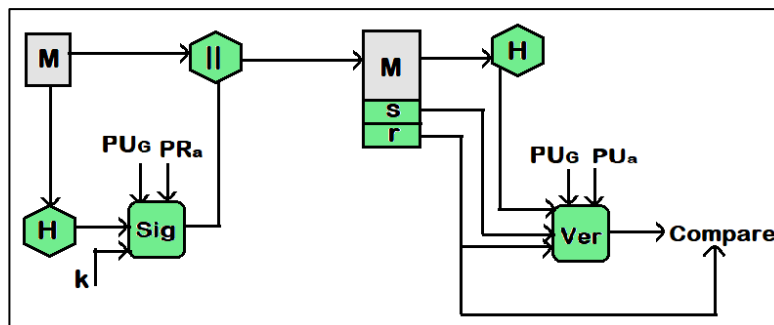
Practice -Lab Activity 2:

Digital Signature Algorithm (DSA)

The [DSA \(Digital Signature Algorithm\)](#) approach involves using of a hash function to create a hash code, same as RSA. This hash code is combined with a randomly generated number k as an input to a signature function. The signature function depends on the sender's private key (PR_a) as well as a set of parameters that are known to a group of communicating principals. This set can be considered as a global public key (PUG). The output of the signature function is a signature with two components, s and r . When an incoming message is received, a hash code is generated for the message. This hash code is then combined with the signature and input into a verification function. The verification function depends on the global public key as well as the sender's public key (PU_a) which is paired with the sender's private key. The output of the verification function returns a value equal to the signature's component r , if the signature is valid. The signature function is designed in such a way that only the sender, with knowledge of the private key, can produce a valid signature.

You can refer below diagram for DSA, where,

- M = Message or Plaintext
- H = Hash Function
- \parallel = bundle the plaintext and hash function (hash digest)
- E = Encryption Algorithm
- D = Decryption Algorithm
- PU_a = Public key of sender
- PR_a = Private key of sender
- Sig = Signature function
- Ver = Verification function
- PUG = Global public Key



DSA Approach

Primary Termologies

- **User's Private Key (PR):** This key is publicly known and can be shared with anyone. It's used to verify digital signatures created with a corresponding private key.
- **User's Public Key (PU):** A top-secret cryptographic key only possessed by the user is used in DSA algorithm's digital signature generation. As it is, the private key must be kept secret and secure because it proves that a given user is genuine.
- **Signing (Sig):** Signing involves creating a digital signature with the help of a user's private key. In case of DSA, this process requires mathematical operations to be performed on the message that should be signed using a given private key in order to generate a unique signature for that message.

- **Verifying (Ver):** Verifying is the process of verifying whether or not a digital signature has been forged using its corresponding public key. In DSA, this involves comparing the messages hash against the verification value through mathematical operations between two binary strings – one representing an encrypted data and another one representing plain-text original message.

Steps to Perform DSA

The Digital Signature Algorithm (DSA) is a [public-key technique](#) (i.e., asymmetric cryptography) and it is used to provide only the digital signature function, and it cannot be used for encryption or key exchange.

1. Global Public-Key Components

There are three parameters that are public and can be shared to a set of users.

- A prime number p is chosen with a length between 512 and 1024 bits such that q divides $(p - 1)$. So, p is prime number where $2^{L-1} < p < 2^L$ for $512 \leq L \leq 1024$ and L is a multiple of 64; i.e., bit length of between 512 and 1024 bits in increments of 64 bits.
- Next, an N -bit prime number q is selected. So, q is prime divisor of $(p - 1)$, where $2^{N-1} < q < 2^N$ i.e., bit length of N bits.
- Finally, g is selected to be of the form $h(p-1)/q \bmod p$, where h is an integer between 1 and $(p - 1)$ with the limitation that g must be greater than 1. So, $g = h(p - 1)/q \bmod p$, where h is any integer with $1 < h < (p - 1)$ such that $h(p-1)/q \bmod p > 1$.

If a user has these numbers, then it can select private key and generates a public key.

2. User's Private Key

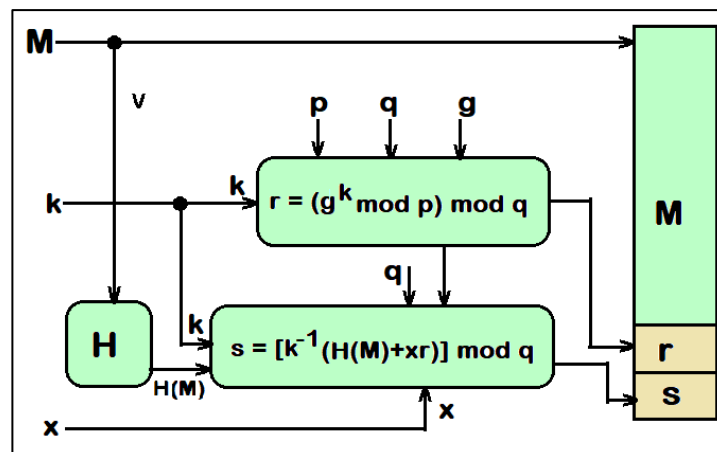
The private key x should be chosen randomly or pseudorandomly and it must be a number from 1 to $(q - 1)$, so x is random or pseudorandom integer with $0 < x < q$.

3. User's Public Key

The public key is computed from the private key as $y = gx \bmod p$. The computation of y given x is simple. But, given the public key y , it is believed to be computationally infeasible to choose x , which is the discrete logarithm of y to the base g , mod p .

4. Signing

If a user want to develop a signature, a user needs to calculate two quantities, r and s , that are functions of the public key components (p, q, g) , the hash code of the message $H(M)$, the user's private key (x) , and an integer k that must be generated randomly or pseudorandomly and be unique for each signing. k is generated randomly or pseudorandomly integer such that $0 < k < q$.



Signing

5. Verification

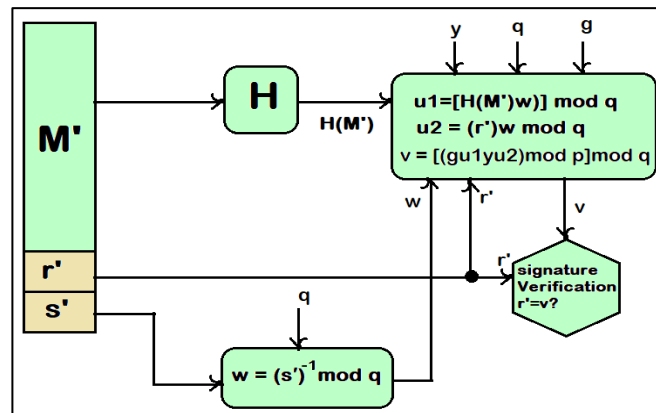
Let M , r' , and s' be the received versions of M , r , and s , respectively.

Verification is performed using the formulas shown in below:

- $w = (s')^{-1} \bmod q$
- $u1 = [H(M')w] \bmod q$
- $u2 = (r')w \bmod q$
- $v = [(gu1yu2) \bmod p] \bmod q$

The receiver needs to generate a quantity v that is a function of the public key components, the sender's public key, and the hash code of the message. If this value matches the r value of the signature, then the signature is considered as valid.

TEST: $v = r'$



Verification

Now, at the end it will test on the value r , and it does not depend on the message or plaintext as, r is the function of k and the three global public-key components as mentioned above. The multiplicative inverse of $k \pmod{q}$ when passed to the function that also has as inputs the message hash code and the user's private key. The structure of this function is such that the receiver can recover r using the incoming message and signature, the public key of the user, and the global public key.

It is given that there is difficulty in taking discrete logarithms, it is not feasible for an attacker to recover k from r or to recover x from s . The only computationally demanding task in signature generation is the exponential calculation $gk \bmod p$. Because this value does not depend on the message to be signed, it can be computed ahead of time. Indeed, a user could precalculate a number of values of r to be used to sign documents as needed. The only other somewhat demanding task is the determination of a multiplicative inverse, k^{-1} .

Key generation

```

from miller import *
from fractions import gcd

def loopIsPrime(number):
    #looping to reduce probability of rabin miller false +
    isNumberPrime = True
    for i in range(0,20):
        isNumberPrime*=isPrime(number)
        if(isNumberPrime == False):
            return isNumberPrime
    return isNumberPrime
def modexp( base, exp, modulus ):
    return pow(base, exp, modulus)

```

```

def squareAndMultiply(x,c,n):
    z=1
    #getting value of l by converting c into binary representation and getting its
length
    c="{0:b}".format(c)[::-1] #reversing the binary string

    l=len(c)
    for i in range(l-1,-1,-1):
        z=pow(z,2)
        z=z%n
        if(c[i] == '1'):
            z=(z*x)%n
    return z

def keyGeneration():

    print("Computing key values, please wait...")
    loop = True
    while loop:
        k=random.randrange(2**(415), 2**(416)) #416 bits
        q=generateLargePrime(160)
        p=(k*q)+1
        while not (isPrime(p)):
            k=random.randrange(2**(415), 2**(416)) #416 bits
            q=generateLargePrime(160)
            p=(k*q)+1
        L = p.bit_length()
        """
        g=t^(p-1)/q % p
        if(g^q % p = 1) we found g
        """

        t = random.randint(1,p-1)
        g = squareAndMultiply(t, (p-1)//q, p)

        if(L>=512 and L<=1024 and L%64 == 0 and (gcd(p-1,q)) > 1 and
squareAndMultiply(g,q,p) == 1):
            #if(L>=512 and L<=1024 and L%64 == 0):
                loop = False
                #print((p-1)%q)

                a = random.randint(2,q-1)
                h = squareAndMultiply(g,a,p)
                #print("p = ",p)
                #print("q = ",q)
                #print("g = ",g)
                #print("h = ",h)
                #print("a = ",a)

                file1 = open("key.txt","w")

```

```

        file1.write(str(p))
        file1.write("\n")
        file1.write(str(q))
        file1.write("\n")
        file1.write(str(g))
        file1.write("\n")
        file1.write(str(h))
        file1.close()
        file2 = open("secretkey.txt","w")
        file2.write(str(a))
        file2.close()

        print("Verification key stored at key.txt and secret key stored at
secretkey.txt")
keyGeneration()

```

Primality Testing with the Rabin-Miller Algorithm

import random

def rabinMiller(num):

 # Returns True if num is a prime number.

 s = num - 1

 t = 0

 while s % 2 == 0:

 # keep halving s while it is even (and use t

 # to count how many times we halve s)

 s = s // 2

 t += 1

 for trials in range(5): # try to falsify num's primality 5 times

 a = random.randrange(2, num - 1)

 v = pow(a, s, num)

 if v != 1: # this test does not apply if v is 1.

 i = 0

 while v != (num - 1):

 if i == t - 1:

 return False

 else:

 i = i + 1

 v = (v ** 2) % num

 return True

```

def isPrime(num):

    if (num < 2):
        return False # 0, 1, and negative numbers are not prime

    lowPrimes = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157,
163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251,
257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353,
359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457,
461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571,
577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673,
677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797,
809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911,
919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997]

    if num in lowPrimes:
        return True

    # See if any of the low prime numbers can divide num
    for prime in lowPrimes:
        if (num % prime == 0):
            return False

    # If all else fails, call rabinMiller() to determine if num is a prime.
    return rabinMiller(num)

def generateLargePrime(keysize):
    # Return a random prime number of keysized bits in size.
    while True:
        num = random.randrange(2**(keysize-1), 2**(keysize))
        if isPrime(num):
            return num

from miller import *
import sys
import hashlib
import math

#function to compute inverse
def computeInverse (in1,in2):
    aL = [in1]

```

```

bL = [in2]
tL = [0]
t = 1
sL = [1]
s = 0
q = math.floor((aL[0]/bL[0]))
r = (aL[0] - (q*bL[0]))

while r > 0 :
    temp = (tL[0] - (q*bL[0]))
    tL[0] = t
    t = temp
    temp = (sL[0] - (q*s))
    sL[0] = s
    s = temp
    aL[0] = bL[0]
    bL[0] = r
    q = math.floor(aL[0]/bL[0])
    r = (aL[0] - (q*bL[0]))

r = bL[0]

inverse = s % in2
return inverse

def squareAndMultiply(x,c,n):
    z=1
    #getting value of l by converting c into binary representation and getting its
length
    c="{0:b}".format(c)[::-1] #reversing the binary string

    l=len(c)
    for i in range(l-1,-1,-1):
        z=pow(z,2)
        z=z%n
        if(c[i] == '1'):
            z=(z*x)%n
    return z

def shaHash(fileName):
    BLOCKSIZE = 65536
    hasher = hashlib.sha1()
    with open(fileName, 'rb') as afile:
        buf = afile.read(BLOCKSIZE)
        while len(buf) > 0:
            hasher.update(buf)
            buf = afile.read(BLOCKSIZE)
    #print(hasher.hexdigest())
    hex = "0x"+hasher.hexdigest()

```

```

    #print(int(hex,0))
    return int(hex,0) #returns int value of hash

def sign():
    if(len(sys.argv) < 2):
        print("Format: python sign.py filename")
    elif(len(sys.argv) == 2):
        print("Signing the file...")
        fileName = sys.argv[1]

        file1 = open("key.txt","r")
        file2 = open("secretkey.txt","r")
        p=int(file1.readline().rstrip())
        q=int(file1.readline().rstrip())
        g=int(file1.readline().rstrip())
        h=int(file1.readline().rstrip())
        a=int(file2.readline().rstrip())

        loop = True
        while loop:
            r = random.randint(1,q-1)
            c1 = squareAndMultiply(g,r,p)
            c1 = c1%q
            c2 = shaHash(fileName) + (a*c1)
            Rinverse = computeInverse(r,q)
            c2 = (c2*Rinverse)%q

            if(c1 != 0 and c2 != 0):
                loop = False

        #print(shaHash(fileName))
        #print(c1)
        #print(c2)
        file = open("signature.txt","w")
        file.write(str(c1))
        file.write("\n")
        file.write(str(c2))
        print("cipher stored at signature.txt")

sign()

from miller import *
import sys
import hashlib
import math

#function to compute inverse
def computeInverse (in1,in2):
    aL = [in1]

```



```

bL = [in2]
tL = [0]
t = 1
sL = [1]
s = 0
q = math.floor((aL[0]/bL[0]))
r = (aL[0] - (q*bL[0]))

while r > 0 :
    temp = (tL[0] - (q*bL[0]))
    tL[0] = t
    t = temp
    temp = (sL[0] - (q*s))
    sL[0] = s
    s = temp
    aL[0] = bL[0]
    bL[0] = r
    q = math.floor(aL[0]/bL[0])
    r = (aL[0] - (q*bL[0]))

r = bL[0]

inverse = s % in2
return inverse

def squareAndMultiply(x,c,n):
    z=1
    #getting value of l by converting c into binary representation and getting its
length
    c="{0:b}".format(c)[::-1] #reversing the binary string

    l=len(c)
    for i in range(l-1,-1,-1):
        z=pow(z,2)
        z=z%n
        if(c[i] == '1'):
            z=(z*x)%n
    return z

def shaHash(fileName):
    BLOCKSIZE = 65536
    hasher = hashlib.sha1()
    with open(fileName, 'rb') as afile:
        buf = afile.read(BLOCKSIZE)
        while len(buf) > 0:
            hasher.update(buf)
            buf = afile.read(BLOCKSIZE)
    #print(hasher.hexdigest())
    hex = "0x"+hasher.hexdigest()

```

```

    #print(int(hex,0))
    return int(hex,0) #returns int value of hash

def verification():
    if(len(sys.argv) < 2):
        print("Format: python sign.py filename")
    elif(len(sys.argv) == 2):
        print("Checking the signature...")
        fileName = sys.argv[1]

        file1 = open("key.txt","r")
        file2 = open("signature.txt","r")
        p=int(file1.readline().rstrip())
        q=int(file1.readline().rstrip())
        g=int(file1.readline().rstrip())
        h=int(file1.readline().rstrip())

        c1=int(file2.readline().rstrip())
        c2=int(file2.readline().rstrip())
        #print(c1)
        #print(c2)

        t1=shaHash(fileName)
        #print(t1)
        inverseC2 = computeInverse(c2,q)
        t1 = (t1*inverseC2)%q

        t2 = computeInverse(c2,q)
        t2 = (t2*c1)%q

        valid1 = squareAndMultiply(g,t1,p)
        valid2 = squareAndMultiply(h,t2,p)
        valid = ((valid1*valid2)%p)%q
        #print(valid)
        if(valid == c1):
            print("Valid signature")
        else:
            print("Invalid signature")

verification()

```

Traceback (most recent call last):

```

File "C:/Users/PAKISTAN/Documents/sgnature.py", line 1, in <module>
    from miller import *
ModuleNotFoundError: No module named 'miller'

```

The error message you're seeing indicates that Python is unable to find a module

named miller that you're trying to import in your script (signature.py). Here are some steps you can follow to resolve this issue:

1. **Check for Typographical Errors:** Make sure that you have spelled the module name correctly in your import statement.
2. **Install the Module:** If miller is a third-party library, you may need to install it. You can install it using pip. Open your command prompt or terminal and run:

```
pip install miller
```

(Note: Replace miller with the actual package name if it's different.)

3. **Check Your Python Environment:** Ensure that you are using the correct Python environment where the module is installed. You can check your current environment by running:

```
python -m pip list
```

This command will show you a list of installed packages in your current environment.

4. **Check for Local Files:** If miller is a local module (a file named miller.py), make sure that the file is in the same directory as signature.py, or that it's in your Python path.
5. **Virtual Environment:** If you're using a virtual environment, ensure that it is activated before you run your script. To activate a virtual environment, you usually run:
 - On Windows:

```
.\venv\Scripts\activate
```

- On macOS/Linux:

```
source venv/bin/activate
```

6. **Updating Python Path:** If you have the miller.py file in a different directory, you can add that directory to your Python path in your script:

```
import sys
sys.path.append('path_to_directory')
from miller import *
```

7. **Check for Compatibility:** If miller is a package that requires a specific version of Python, make sure you're using a compatible version.

Lab 15

Public-Key Certificates (PKC)

Objective:

This lab will enable students to utilize their knowledge of **Public-Key Certificates** and will enable students to generate their own digital certificates in python.

Activity Outcomes:

This lab teaches you the following topics:

- Gain an understanding of the importance of PKI
- Learn how to implement a PKI solution in Python

-

Instructor Note:

As pre-lab activity, read Chapter 13, page 418-435 from the book (Cryptography and Network Security Principles and Practice Eighth Edition Global Edition William Stallings).

Practice -Lab Activity 1:

Public-Key Certificates

Public-key certificates essentially act as a passport that certifies that a public-key belongs to a specific name or organization. Certificates are issued by certificate authorities, more commonly known as CAs. One of the properties of using public-key certificates is that they allow all users to know without question that the public-key of the CA can be checked by each user. In addition, certificates do not require the online participation of a TTP. One thing that you must remember is that the security of the private key is crucial to the security of all users. The following represents the notation of a certificate binding a public key

+KA to user A issued by a certificate authority CA using its private key -CKCA:

$\text{Cert-CKCA}(+KA) = \text{CA}[V, SN, AI, CA, TCA, A, +KA]$ where:

V = version number SN = serial number

AI = algorithm identifier of signature algorithm used CA = name of certification authority

TCA = period of validity of this certificate

A = name to which the public key in this certificate is bound

+KA = public to be bound to a name

Certificate Chains and Certificate Hierarchy

Consider communication between our two users: Alice and Bob. Each user lives geographically apart. Each user may have public keys from different CAs. For simplicity, designate Alice's certificate authority as CAA and Bob's certificate authority as CAB. If Alice does not know or trust CAB, then Bob's certificate is useless to her; the same will hold true for Bob and his knowledge or trust of Alice's CAA. In order to provide a solution to this issue, you can construct a certificate chain. If CAA certifies CAB with a certificate $\text{CAA} \ll \text{CAB} \gg$ and CAB certifies CAA's public key with a certificate $\text{CAB} \ll \text{CAA} \gg$, then both Alice and Bob can check their certificates by checking a certificate chain. Assume Alice is presented with $\text{CAB} \ll \text{Bob} \gg$ and attempts to look up if there is a certificate $\text{CAA} \ll \text{CAB} \gg$. She checks the chain: $\text{CAA} \ll \text{CAB} \gg$, $\text{CAB} \ll \text{Bob} \gg$. Certificate chains are not limited to just the two certificates.

You can use Python to create X.509 certificates. The following code will generate two certificates: `rsakey.pem` and `csr.pem`. The `rsakey.pem` is a private key that is encrypted using the super-secret password `Ilik32Cod3` and will then use the private key to generate a public key. We then generate a certificate signing request (CSR) using a number of custom attributes.

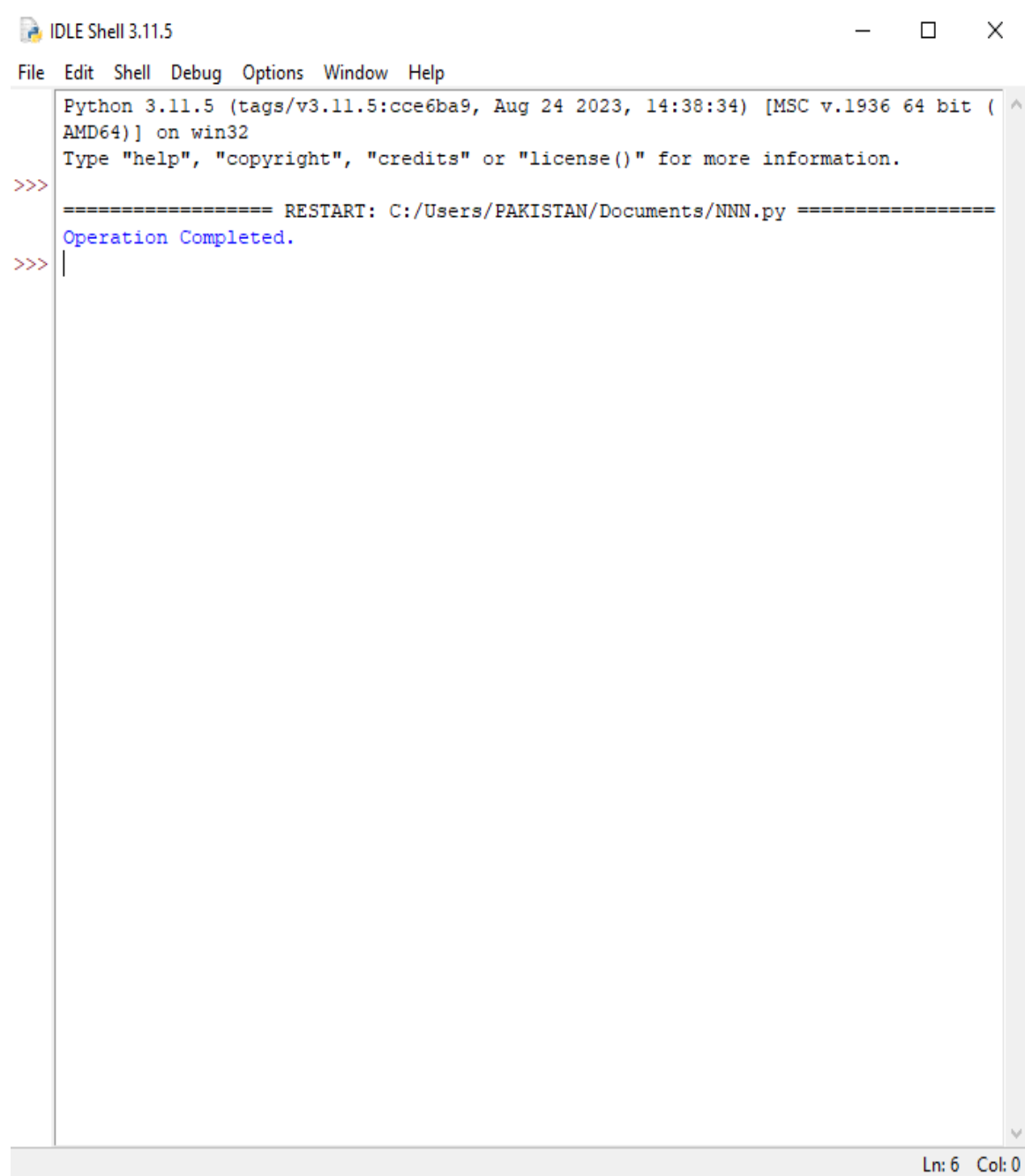
```
from cryptography.hazmat.backends import default_backend from
cryptography.hazmat.primitives import serialization from
cryptography.hazmat.primitives.asymmetric import rsa from cryptography import
x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.primitives import hashes #Generate Key (RSA,DSA,EC)
encryptedpass = b"Ilik32Cod3" key = rsa.generate_private_key(
public_exponent=65537, key_size=2048, backend=default_backend()
)
with open("rsakey.pem", "wb") as f: f.write(key.private_bytes(
encoding=serialization.Encoding.PEM,
```

```

format=serialization.PrivateFormat.TraditionalOpenSSL,
encryption_algorithm=serialization.BestAvailableEncryption(encrypted pass),
))
# Generate CSR
csr = x509.CertificateSigningRequestBuilder().subject_name(x509.Name([
x509.NameAttribute(NameOID.COUNTRY_NAME, u"US"),
x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"NC"),
x509.NameAttribute(NameOID.LOCALITY_NAME, u"Raleigh"),
x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"Python
Cryptography"), x509.NameAttribute(NameOID.COMMON_NAME,
u"shannonbray.us"),
])).add_extension( x509.SubjectAlternativeName([
x509.DNSName(u"shannonbray.us"),
]),
critical=False,
# Sign the CSR with our private key.
).sign(key, hashes.SHA256(), default_backend()) with open("csr.pem", "wb") as f:
f.write(csr.public_bytes(serialization.Encoding.PEM)) print('Operateion Completed.')

```

Code Output

A screenshot of the IDLE Shell 3.11.5 window. The window has a title bar with the text 'IDLE Shell 3.11.5' and standard window controls (minimize, maximize, close). Below the title bar is a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area shows the following output: 'Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)] on win32' followed by 'Type "help", "copyright", "credits" or "license()" for more information.' Then, after a prompt '>>>', it shows '==== RESTART: C:/Users/PAKISTAN/Documents/NNN.py =====' followed by 'Operation Completed.' and another prompt '>>>'. The status bar at the bottom right indicates 'Ln: 6 Col: 0'.

Practice -Lab Activity 2:

Certificate Revocation

If we continue to examine the situation presented, you can see that communications between Alice and Bob rely on the trust of the certificate authority and each party must keep their private key secure. Should one of their keys become compromised, the certificate needs to be nullified or revoked. If Alice's key was compromised in an attack, the attacker (Trent) can continue to impersonate Alice up to the end of the certificate's validity period. If Alice detects the compromise, she can ask for revocation of the

corresponding public-key certificate. Certificate revocation is performed by maintaining a list of compromised certificates; these lists are known as certificate revocation lists, or CRLs. CRLs are stored in the X.500 directory; when a user or process is checking a certificate, it must not only confirm that the certificate exists but also make sure the certificate is not on a CRL. The certificate revocation process is quite slow and can be costly and ineffective.

If you've used the previous example to generate a key, you will be able to load it using the following code. Examine the use of `load_pem_private_key()`, as shown here:

```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from cryptography import x509
from cryptography.x509.oid import NameOID
from cryptography.hazmat.primitives import hashes
from cryptography.hazmat.primitives.serialization import load_pem_private_key

# Define the encrypted password for the private key
encrypted_pass = b"Ilik32Cod3"

# Load the private key
with open('rsa_key.pem', 'rb') as key_file:
    private_key = load_pem_private_key(key_file.read(), password=encrypted_pass,
                                      backend=default_backend())

# Generate CSR
csr = x509.CertificateSigningRequestBuilder().subject_name(
    x509.Name([
        x509.NameAttribute(NameOID.COUNTRY_NAME, u"US"),
        x509.NameAttribute(NameOID.STATE_OR_PROVINCE_NAME, u"CA"),
        x509.NameAttribute(NameOID.LOCALITY_NAME, u"San Francisco"),
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, u"Python
    Cryptography"),
        x509.NameAttribute(NameOID.COMMON_NAME, u"8gwifi.org"),
    ])
).add_extension(
    x509.SubjectAlternativeName([
        x509.DNSName(u"mysite.com"),
    ]),
    critical=False
).sign(private_key, hashes.SHA256(), default_backend())

# Output the CSR
csr_pem = csr.public_bytes(serialization.Encoding.PEM)
print(csr_pem.decode('utf-8'))
```



```
IDLE Shell 3.11.5
File Edit Shell Debug Options Window Help

Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

>>>
===== RESTART: C:/Users/PAKISTAN/Documents/NNN.py =====
Operation Completed.
>>>
===== RESTART: C:/Users/PAKISTAN/Documents/hhh.py =====
-----BEGIN CERTIFICATE REQUEST-----
MIIC0jCCABoCAQAwZTELMAkGA1UEBhMCVVMxGzAABgNVBAGMAkNBMRywFAYDVQQH
DA1TYW4gRnJhbmNpc2NvMRwwGgYDVQQKDBNReXRob24gQ3J5cHRvZ3JhcGh5MRMw
EQYDVQDDAo4Z3dpZmkub3JnMIIIBjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKC
AQEAw0EYABtysh4vLNMCIPLuyqtcljx+rtUZGSiwTRJAoO4fIgtQ4l4iyaiDQcu
SjlGScOmAb7lz+vfd4Dts9z10lkIj1IPrpO8trlKWZ28D+AEHdoWVbhuBSBPtxt+
EZ4eJa15dW6c4L3/nCbsUqQoJ2BTvk4qyPflrxHUAZ9fqZh5CZPsrxleRnlQ16Y3
yl8dmTc5uoSr0yokFhiikE2lvfvVwusppqRr7Sg9v8Tf5ahnWRlnqOAcB9pizgOj
UeH048191Vwm5Q1DrUVKHhghP3c8rfd7Q+bvIbphEbvHNGGyTOoz4PpHHdw0RSsi
2vwG3doEotEEzsbNWAh7TOuKtwIDAQABOcgwJgYJKoZIhvcNAQkOMRkwFzAVBgNV
HREEDjAMggpteXNpdGUuY29tMA0GCSqGSIb3DQEBCwUAA4IBAQA5PY8wT0liaJr0
uOKuF90v86M8alo53nv41UetgbJfJPU025wpbGGXiosvo+5I2crlAuah7fJDH9G3
EpsOnjiHpgAQg2UMgZtcByRV51lgkvCr1sb//2KzRSEHp7ozcCiR9Wdki7ooeHA
UHCifFwJwVdF4KEqpYwJ4nfEkdnH9J9t3gH1ZCFxuQ3opy4E8B3au/nkUGQJK0
kCoubkyXAI3Y/SOMVJNs+18apoHafq5L/QxP6Zy9BnbBFOuhRJOxUk3xd6XIdtru
vDuuIuQ1NMzPRymT8Rxc6NCyRvK2++BwmzrS2rsRwwotVOREShvAfGQ8SDjYQh32
qkHAeWw4
-----END CERTIFICATE REQUEST-----
>>> |
```

Practice -Lab Task: Certificate Validation

Objective: Validate the self-signed certificate.

Instructions:

- Write a script to load the self-signed certificate and validate it.

Practice Lab Task: Revocation List

Objective: Create a simple Certificate Revocation List (CRL).

Instruction

Create a CRL that contains a list of revoked certificates.