**PRN No: 2020BTECS00023**

**Name: Sumit Narake**

**Batch: B2**

**Assignment: 9**

**Title of assignment:** Implementation of Diffie Hellman Key Exchange Algorithm

1. **Aim:**

Implementation of Diffie Hellman Key Exchange Algorithm

1. **Theory:**

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, we will 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.

**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 mod 23) = (6561 mod 23) = 6

Bob: y = (9^3 mod 23) = (729 mod 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: ka = y^a mod p = 65536 mod 23 = 9

Bob: kb = x^b mod p = 216 mod 23 = 9

**Step 7:** 9 is the shared secret.

**Code:**

**Client:**

import socket

# Function to find mod: a^m mod n

def findExpoMod(a, m, n):

    # Decimal to binary conversion

    m\_bin = bin(m).replace("0b", "")

    # Convert it into list (individual characters)

    m\_bin\_lst = [int(i) for i in m\_bin]

    # Initialize the list

    a\_lst = [a]

    # Functions to perform operations

    # If next value = 0

    def oneOperation(num):

        return (num\*num) % n

    # If next value = 1

    def twoOperation(num):

        return (a \* oneOperation(num)) % n

    for j in range(len(m\_bin\_lst)):

        if j+1 == len(m\_bin\_lst):

            break

        if(m\_bin\_lst[j+1] == 0):

            a\_lst.append(oneOperation(a\_lst[j]))

        else:

            a\_lst.append(twoOperation(a\_lst[j]))

    return a\_lst[-1]

HOST = 'localhost'

PORT = 12345

client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_address = (HOST, PORT)  # Server address and port

client\_socket.connect(server\_address)

print(f"Connected to server at: {HOST}:{PORT}")

# Receive the server's public key, q, alpha

received\_Ya = client\_socket.recv(1024)

Ya = int(received\_Ya.decode())

print(f"Received server's public key as: {Ya}")

received\_q = client\_socket.recv(1024)

q = int(received\_q.decode())

print(f"Received large prime as: {q}")

received\_alpha = client\_socket.recv(1024)

alpha = int(received\_alpha.decode())

print(f"Received alpha as: {alpha}")

# Client's Private Key

Xb = int(input(f"Enter the private key for B (Xb) [less than {q}]:\n"))

if Xb >= q:

    print("Private key must be less than choosen prime!")

    exit()

# Client's Public Key

Yb = findExpoMod(alpha, Xb, q)

print(f"Client's Public key is: {Yb}")

# Send this to Server

print("Sending Client's Public Key to Server...")

send\_Yb = str(Yb).encode()

client\_socket.sendall(send\_Yb)

# Receive Server's Shared key

received\_Ks = client\_socket.recv(1024)

Ks = int(received\_Ks.decode())

print(f"Received Server's Shared key as: {Ks}")

# Compute shared key and send to server

Kc = findExpoMod(Ya, Xb, q)

print(f"Client's Shared key is: {Kc}")

print("Sending it to server...")

send\_Kc = str(Kc).encode()

client\_socket.sendall(send\_Kc)

if Kc == Ks:

    print("Both shared keys are equal\nKeys exchanged successfully!")

else:

    print("Both shared keys aren't equal.\nKey exchange failed!")

client\_socket.close()

Server:

from generate\_prime import is\_prime, generate\_prime\_no

import socket

# Function to find mod: a^m mod n

def findExpoMod(a, m, n):

    # Decimal to binary conversion

    m\_bin = bin(m).replace("0b", "")

    # Convert it into list (individual characters)

    m\_bin\_lst = [int(i) for i in m\_bin]

    # Initialize the list

    a\_lst = [a]

    # Functions to perform operations

    # If next value = 0

    def oneOperation(num):

        return (num\*num) % n

    # If next value = 1

    def twoOperation(num):

        return (a \* oneOperation(num)) % n

    for j in range(len(m\_bin\_lst)):

        if j+1 == len(m\_bin\_lst):

            break

        if(m\_bin\_lst[j+1] == 0):

            a\_lst.append(oneOperation(a\_lst[j]))

        else:

            a\_lst.append(twoOperation(a\_lst[j]))

    return a\_lst[-1]

def is\_primitive\_root(alpha, q):

    L = []

    for i in range(1, q):

        L.append(findExpoMod(alpha, i, q))

    for i in range(1, q):

        if L.count(i) > 1:

            L.clear()

            return False

        return True

# Initialize Socket

HOST = 'localhost'

PORT = 12345

server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)

server\_address = (HOST, PORT)  # Server address and port

server\_socket.bind(server\_address)

server\_socket.listen(1)

print(f"Server started at: {HOST}:{PORT}")

print("Waiting for a client to connect...")

client\_socket, client\_address = server\_socket.accept()

print("Client connected: ", client\_address)

# DH Key-exchange

# Choose prime no. 'q'

print("Choose a large integer prime number(q):")

gen\_r = input("Do you want to generate the prime number automatically ? [y/n]\n")

if gen\_r == 'y':

    dig\_p = int(input("Enter the number of digits in prime number: "))

    q = generate\_prime\_no(dig\_p)

    print(f"q = {q}")

elif gen\_r == 'n':

    q = int(input("Enter a large prime number:\n"))

    if not is\_prime(q):

        print(f"Entered number is not prime!")

        exit()

else:

    print("Invaild choice!")

    exit()

# Choose primitive root 'alpha'

print("Choose primitive root (alpha):")

gen\_pr = input("Do you want to find the primitive root automatically ? [y/n]\n")

if gen\_pr == 'y':

    for a in range(2, q):

        if is\_primitive\_root(a, q):

            alpha = a

            break

    print(f"Alpha = {alpha}")

elif gen\_pr == 'n':

    alpha = int(input(f"Enter the primiitive root of {q}:\n"))

    if not is\_primitive\_root(alpha, q):

        print(f"This is not the primitive root!")

        exit()

else:

    print("Invaild choice!")

    exit()

# Server's Private key

Xa = int(input(f"Enter the private key for A (Xa) [less than {q}]:\n"))

if Xa >= q:

    print("Private key must be less than choosen prime!")

    exit()

# Server's Public Key

Ya = findExpoMod(alpha, Xa, q)

# Send this data to client

print(f"Server's Public Key is: {Ya}")

print("Sending Public Key to client...")

send\_Ya = str(Ya).encode()

client\_socket.sendall(send\_Ya)

print("Sending choosen large prime to client...")

send\_q = str(q).encode()

client\_socket.sendall(send\_q)

print("Sending primitive root to client...")

send\_alpha = str(alpha).encode()

client\_socket.sendall(send\_alpha)

print("Waiting for Client's Public Key...")

# Receive Client's Public Key

received\_Yb = client\_socket.recv(1024)

Yb = int(received\_Yb.decode())

print(f"Received Public Key of Client: {Yb}")

# Compute shared key and send to client

Ks = findExpoMod(Yb, Xa, q)

print(f"Server's Shared Key is: {Ks}")

print("Sending it to client...")

send\_Ks = str(Ks).encode()

client\_socket.sendall(send\_Ks)

# Receive Client's Shared Key

print("Waiting for Client's Shared Key...")

received\_Kc = client\_socket.recv(1024)

Kc = int(received\_Kc.decode())

print(f"Received Client's Shared Key as: {Kc}")

if Ks == Kc:

    print("Both shared keys are equal\nKeys exchanged successfully!")

else:

    print("Both shared keys aren't equal.\nKey exchange failed!")

client\_socket.close()

server\_socket.close()

**Ouput:**

