**CNS LAB**

**NAME: Parshwa Herwade**

**PRN: 22510064**

**BATCH: B1**

**Experiment No. 08**

**Title –** Implement the Diffie-Hellman Key Exchange algorithm for a given problem.

**Objectives:**

To implement the **Diffie-Hellman Key Exchange Algorithm** to enable two parties to securely share a secret key over an insecure communication channel. This shared key can then be used for symmetric encryption or decryption in secure communication.

**Problem Statement:**

In secure communications, two parties (commonly referred to as Alice and Bob) need to agree on a secret key that can be used for encrypting and decrypting messages. However, they must do this over a public channel where an attacker (Eve) might be listening.

Implement the **Diffie-Hellman Key Exchange Algorithm**, which allows Alice and Bob to securely compute a shared secret key without directly transmitting it over the insecure channel. The algorithm should:

1. Accept a large prime number p and a primitive root modulo p, g.
2. Allow each party to select a private key (a for Alice, b for Bob).
3. Compute the corresponding public keys:
   * Alice computes A = g^a mod p
   * Bob computes B = g^b mod p
4. Exchange public keys between Alice and Bob.
5. Compute the shared secret key:
   * Alice computes K = B^a mod p
   * Bob computes K = A^b mod p
6. Validate that both computed secret keys are equal, i.e., K\_Alice == K\_Bob.

Additionally, demonstrate the correctness of the algorithm with an example and optionally simulate an attacker attempting to derive the secret key without access to the private keys.

**Equipment / Tools**

* Java Development Kit (JDK) 8 or later (for BigInteger and SecureRandom).
* A code editor (VS Code, IntelliJ, Notepad++) or terminal editor (vim, nano).
* Terminal / Command Prompt to compile and run Java.
* (Optional) Internet to look up recommended prime sizes (e.g., 2048-bit) or libraries for production usage.

**Theory (Detailed)**

**Basic idea**

Diffie–Hellman allows two parties to agree on a shared secret K using public values and their private secrets, without sending K over the channel.

Public parameters:

* p — a large prime
* g — a generator (primitive root) modulo p (i.e., g generates a large subgroup mod p)

Private keys:

* Alice chooses a secret a (private)
* Bob chooses a secret b (private)

Public keys:

* Alice computes A = g^a mod p and sends A to Bob.
* Bob computes B = g^b mod p and sends B to Alice.

Shared secret:

* Alice computes K\_A = B^a mod p = (g^b)^a mod p = g^(ab) mod p
* Bob computes K\_B = A^b mod p = (g^a)^b mod p = g^(ab) mod p

Thus K\_A == K\_B == g^(ab) mod p.

**Why it is secure (informal)**

* Given g, p, A=g^a mod p, recovering a is the **discrete logarithm problem (DLP)**, believed to be hard for appropriately large p and generator g.
* An eavesdropper who sees g, p, A, B would need to solve DLP to learn a or b and then compute g^(ab). For properly chosen parameters (e.g., 2048-bit p or elliptic curve variants), this is computationally infeasible.

**Practical considerations**

* **Prime choice**: Use safe primes or standardized parameters (RFCs). For production, use 2048-bit or larger prime groups or elliptic-curve Diffie–Hellman (ECDH).
* **Authentication**: DH by itself does not authenticate peers — vulnerable to man-in-the-middle (MitM). Use digital signatures, certificates, or authenticated variants (e.g., TLS) to prevent MitM.
* **Ephemeral DH**: Use ephemeral keys (generate fresh a, b per session) for forward secrecy.
* **Modular exponentiation**: Implemented efficiently via square-and-multiply; BigInteger.modPow in Java does that.

**Procedure (Step-by-step)**

1. Choose a large prime p and a generator g modulo p. (Publicly known.)
2. Alice chooses a private random integer a such that 1 <= a <= p-2.
3. Bob chooses a private random integer b such that 1 <= b <= p-2.
4. Alice computes public A = g^a mod p and sends A to Bob over the insecure channel.
5. Bob computes public B = g^b mod p and sends B to Alice.
6. Alice computes shared secret K\_A = B^a mod p.
7. Bob computes shared secret K\_B = A^b mod p.
8. Verify K\_A == K\_B. Use the resulting K (or a key derived from it via a key-derivation function) as the symmetric encryption key.
9. (Optional demonstrate attacker) Attempt to compute a from A by brute force for small prime p to show infeasibility for large p.

CODE:   
  
import java.math.BigInteger;

import java.security.SecureRandom;

import java.util.Scanner;

/\*\*

 \* Diffie-Hellman demonstration in Java using BigInteger.

 \*

 \* Features:

 \* - Interactive: accepts p, g, and private keys (or 'r' for random).

 \* - Uses BigInteger.modPow for modular exponentiation.

 \* - Optional brute-force discrete-log solver for small p (to simulate Eve).

 \*

 \* Note: For production use, use standardized primes and authenticated DH (e.g., within TLS).

 \*/

public class DiffieHellman {

    private static final SecureRandom random = new SecureRandom();

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Diffie-Hellman Key Exchange Demo (Java)");

        System.out.println("---------------------------------------------------");

        System.out.print("Enter prime p (decimal): ");

        BigInteger p = new BigInteger(sc.next());

        if (!p.isProbablePrime(20)) {

            System.out.println("Warning: p is not strongly prime according to quick test (probablePrime).");

        }

        System.out.print("Enter generator g (decimal): ");

        BigInteger g = new BigInteger(sc.next());

        BigInteger a = readPrivate("Alice", sc, p);

        BigInteger b = readPrivate("Bob", sc, p);

        // Compute public keys

        BigInteger A = g.modPow(a, p); // Alice's public

        BigInteger B = g.modPow(b, p); // Bob's public

        System.out.println("\nPublic values:");

        System.out.println("A (Alice's public key) = g^a mod p = " + A);

        System.out.println("B (Bob's public key)   = g^b mod p = " + B);

        // Compute shared secrets

        BigInteger K\_A = B.modPow(a, p);

        BigInteger K\_B = A.modPow(b, p);

        System.out.println("\nShared secret computed by Alice: " + K\_A);

        System.out.println("Shared secret computed by Bob:   " + K\_B);

        System.out.println("Shared keys equal? " + K\_A.equals(K\_B));

        // Optional: attempt brute force discrete log for small p

        if (p.bitLength() <= 20) { // small p -> feasible to brute force

            System.out.println("\nEve (attacker) simulation: trying to recover Alice's private key a by brute force...");

            BigInteger found = bruteForcePrivateKey(A, g, p);

            if (found != null) {

                System.out.println("Eve found a = " + found + " (verifies: g^a mod p = " + g.modPow(found, p) + ")");

                System.out.println("Eve can compute shared secret K = g^(ab) mod p = " + g.modPow(found.multiply(b), p).mod(p));

            } else {

                System.out.println("Eve failed to find a (unexpected).");

            }

        } else {

            System.out.println("\nEve simulation skipped: p is too large for brute-force in this demo.");

            System.out.println("For real-world p (e.g., 2048-bit), discrete log brute force is infeasible.");

        }

        sc.close();

    }

    private static BigInteger readPrivate(String who, Scanner sc, BigInteger p) {

        while (true) {

            System.out.print("Enter " + who + "'s private key (decimal) or 'r' for random: ");

            String s = sc.next();

            if (s.equalsIgnoreCase("r")) {

                // generate random 1 <= x <= p-2

                BigInteger max = p.subtract(BigInteger.valueOf(2));

                BigInteger x;

                do {

                    x = new BigInteger(max.bitLength(), random);

                } while (x.compareTo(BigInteger.ONE) < 0 || x.compareTo(max) > 0);

                System.out.println(who + " private key (randomly chosen): " + x);

                return x;

            } else {

                try {

                    BigInteger x = new BigInteger(s);

                    if (x.compareTo(BigInteger.ONE) < 0 || x.compareTo(p.subtract(BigInteger.ONE)) > 0) {

                        System.out.println("Private key must satisfy 1 <= key <= p-2. Try again.");

                        continue;

                    }

                    return x;

                } catch (NumberFormatException ex) {

                    System.out.println("Invalid number. Try again.");

                }

            }

        }

    }

    // Brute force discrete log: find x such that g^x mod p == publicKey.

    // Only feasible for small p; used for demonstration of insecurity with small primes.

    private static BigInteger bruteForcePrivateKey(BigInteger publicKey, BigInteger g, BigInteger p) {

        BigInteger x = BigInteger.ZERO;

        BigInteger limit = p; // search 0..p-1

        while (x.compareTo(limit) < 0) {

            if (g.modPow(x, p).equals(publicKey)) {

                return x;

            }

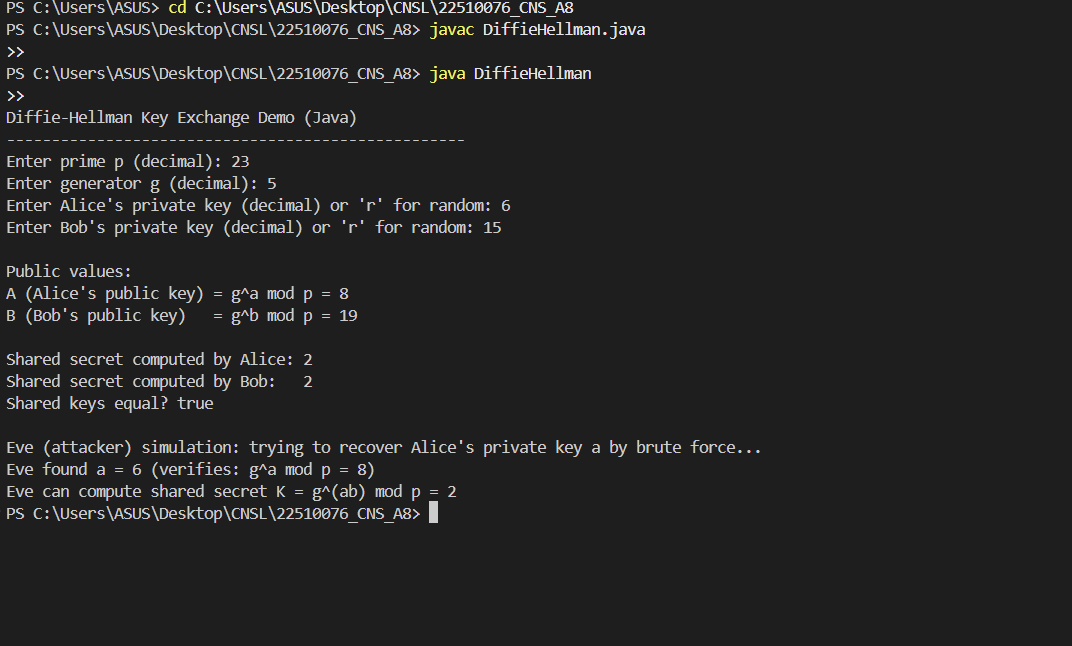
            x = x.add(BigInteger.ONE);

        }

        return null;

    }

}

RESULTS:  
  


**Steps (what you will show in assignment)**

1. State public parameters p and g.
2. Show chosen private keys a and b (or show they are randomly generated).
3. Show public keys A = g^a mod p and B = g^b mod p.
4. Show Alice’s computation K\_A = B^a mod p.
5. Show Bob’s computation K\_B = A^b mod p.
6. Verify K\_A == K\_B. Print the value of K.
7. Optionally, run Eve’s brute-force search (only for small p) and show she recovers a and reconstructs K.

**Observations and Conclusion**

**Observations**

* For the example (p=23, g=5, a=6, b=15) both parties computed the same shared secret K = 2 using the exchanged public keys — which demonstrates correctness.
* The computation uses modular exponentiation and relies on properties of exponents: (g^a)^b = g^(ab) = (g^b)^a.
* When p is small, a brute-force attacker can recover private keys quickly — demonstration shows this.
* For large primes (p with hundreds or thousands of bits) and good generators, discrete log algorithms become infeasible, so DH is secure in practice (assuming proper parameter choice).
* DH does **not** provide authentication: an active MitM attacker who can intercept and substitute public keys can perform two separate DH exchanges and decrypt communications unless authentication is added (e.g., signatures, certificates, pre-shared keys).

**Conclusion**

* The implemented Diffie–Hellman algorithm correctly allows two parties to compute a shared secret without sending the secret directly.
* Security depends entirely on parameter choice (large safe primes / standardized groups) and on protecting against active MitM attackers via authentication.
* For production systems, prefer using standardized groups, ephemeral keys for forward secrecy, and authenticated key exchange protocols (e.g., TLS/ECDHE).
* The brute-force demonstration shows why small primes are insecure — always use large primes or switch to elliptic-curve DH (ECDH) for better performance/security.