**Practical 1**

**Aim:** Write a program to implement following:

A)Chinese Reminder Theorem

B)Fermat’s Little Theorem

**Theory:**

**A) Chinese Remainder Theorem:**

The Chinese Remainder Theorem is a theorem in number theory that describes the solution to a system of simultaneous congruences. It states that if you have a system of congruences with pairwise relatively prime moduli, then there exists a unique solution modulo the product of those moduli.

Example:

Let's say we want to find the smallest non-negative integer x that satisfies:

x ≡ (mod 3)

x ≡ (mod 5)

Using CRT, we can solve this system of congruences:

- We find that x ≡ (mod 15) satisfies both congruences.

**Source Code:**

import java.math.BigInteger;

public class ChineseRemainderTheorem {

public static BigInteger chineseRemainderTheorem(int[] residues, int[] moduli) {

if (residues.length != moduli.length) {

throw new IllegalArgumentException("Number of residues must equal number of moduli");

}

int n = residues.length;

// Compute the product of all moduli

BigInteger N = BigInteger.ONE;

for (int i = 0; i < n; i++) {

N = N.multiply(BigInteger.valueOf(moduli[i]));

}

// Compute the sum of the residues

BigInteger sum = BigInteger.ZERO;

for (int i = 0; i < n; i++) {

BigInteger Ni = N.divide(BigInteger.valueOf(moduli[i]));

BigInteger Mi = Ni.modInverse(BigInteger.valueOf(moduli[i]));

sum = sum.add(BigInteger.valueOf(residues[i]).multiply(Ni).multiply(Mi));

}

// Return the solution modulo the product of all moduli

return sum.mod(N);

}

public static void main(String[] args) {

int[] residues = {2, 3, 2};

int[] moduli = {3, 5, 7};

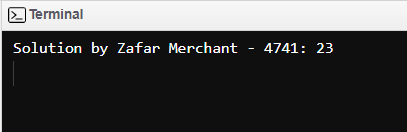
BigInteger solution = chineseRemainderTheorem(residues, moduli);

System.out.println("Solution by Zafar Merchant - 4741: " + solution);

}

}

**Output:**

****

1. **Fermat’s Little Theorem:**

Fermat's Little Theorem states that if p is a prime number and a is an integer not divisible by p, then a^(p-1) ≡ 1(mod p).

Example:

Let's take p = 7 and a = 3.

According to Fermat's Little Theorem,3^6 ≡ 1(mod 7).

And indeed, 3^6 = 729, which is congruent to 1 modulo 7.

**Source Code:**

// Java program to find modular

// inverse of a under modulo m

// using Fermat's little theorem.

// This program works only if m is prime.

class GFG {

static int \_\_gcd(int a, int b)

{

if (b == 0) {

return a;

}

else {

return \_\_gcd(b, a % b);

}

}

// To compute x^y under modulo m

static int power(int x, int y, int m)

{

if (y == 0)

return 1;

int p = power(x, y / 2, m) % m;

p = (p \* p) % m;

return (y % 2 == 0) ? p : (x \* p) % m;

}

// Function to find modular

// inverse of a under modulo m

// Assumption: m is prime

static void modInverse(int a, int m)

{

if (\_\_gcd(a, m) != 1)

System.out.print("Inverse doesn't exist");

else {

// If a and m are relatively prime, then

// modulo inverse is a^(m-2) mode m

System.out.print(

"4741-Zafar Merchant -- Fermats Theorem -- ");

System.out.print(

"Modular multiplicative inverse is "

+ power(a, m - 2, m));

}

}

// Driver code

public static void main(String[] args)

{

int a = 3, m = 11;

modInverse(a, m);

}

}

**Output:**

****

**Conclusion:** Hence Chinese Remainder theorem and Fermat’s Little theorem were implemented successfully.

**Practical 2**

**Aim:** Write a program to implement the (i) Affine Cipher (ii) Rail Fence Technique (iii)

Simple Columnar Technique (iv) Vermin Cipher (v) Hill Cipher to perform

encryption and decryption.

**Theory:**

(i) Affine Cipher:

The Affine Cipher is a type of substitution cipher where each letter in an alphabet is mapped to its numeric equivalent, encrypted using a simple mathematical function, and converted back to a letter. It uses a pair of numbers (a and b) as the key. The encryption function for a letter x is (ax + b) mod m, where m is the size of the alphabet.

Example:

Let's encrypt the message "HELLO" using the key (a = 5, b = 8) in a standard English alphabet with 26 letters.

* H -> 7, E -> 4, L -> 11, L -> 11, O -> 14

Encrypting: (75 + 8) mod 26 = 45 mod 26 = 19 -> T, (45 + 8) mod 26 = 28 mod 26 = 2 -> C, (11\*5 + 8) mod 26 = 63 mod 26 = 11 -> L, and so on.

* So, "HELLO" would be encrypted as "TCXXA".

(ii) Rail Fence Technique:

The Rail Fence Technique is a transposition cipher where the plaintext is written diagonally and read off horizontally in a zigzag pattern. The number of "rails" or lines used is the key for this cipher.

Example:

Encrypting the message "HELLO" with 3 rails:

H . . . O . . . // First rail

. E . L . . . // Second rail

. . L . . . . . // Third rail

Reading off horizontally: "HOELLL"

(iii) Simple Columnar Technique:

This is a transposition cipher where the plaintext is written out in rows of a fixed length, and then read out again column by column according to a permutation of the columns.

Example:

Encrypting the message "HELLO" with a keyword "KEY" (where alphabetical order of keyword letters determines column order):

K E Y // Columns determined by the keyword

---------

H E L // Rows of plaintext

L O . // Filled with a null character for uneven length

Reading off column by column: "HLELO".

(iv) Vermin Cipher:

The Vermin Cipher is a substitution cipher similar to the Caesar Cipher but with a varying shift that depends on the position of the letter within the plaintext.

Example:

Encrypting the message "HELLO" using a starting shift of 3:

* H -> (3+1) mod 26 = 4 -> E
* E -> (3+2) mod 26 = 5 -> F
* L -> (3+3) mod 26 = 6 -> G
* L -> (3+4) mod 26 = 7 -> H

O -> (3+5) mod 26 = 8 -> I

* So, "HELLO" would be encrypted as "EFHIG".

(v) Hill Cipher:

The Hill Cipher is a polygraphic substitution cipher based on linear algebra. It encrypts blocks of plaintext letters at a time and uses matrix multiplication. The key for this cipher is a matrix.

Example:

Encrypting the message "HELLO" with a 2x2 key matrix:

Key matrix: [[1, 2], [3, 1]]

Plaintext matrix: [H, E; L, L; O, null]

Multiplying the key matrix with the plaintext matrix:

[[1, 2], [3, 1]] \* [[7, 4], [11, 11]] = [[7+22, 4+8], [33+11, 11+3]] = [[29, 12], [44, 14]]

Taking mod 26 for each element:

[[29 mod 26, 12 mod 26], [44 mod 26, 14 mod 26]] = [[3, 12], [18, 14]]

Converting back to letters: "3" -> C, "12" -> L, "18" -> S, "14" -> O

So, "HELLO" would be encrypted as "CLOS".

**Source Code:**

import java.util.Scanner;

public class CiphersDemo {

public static void main(String[] args) {

System.out.println("4741-Zafar Merchant");

System.out.println("\*\*\*\*Ciphers\*\*\*\*");

Scanner scanner = new Scanner(System.in);

System.out.println("Enter plaintext: ");

String plaintext = scanner.nextLine().toUpperCase();

// Affine Cipher

int a = 5;

int b = 8;

String affineCipher = affineEncrypt(plaintext, a, b);

System.out.println("Affine Cipher: " + affineCipher);

// Rail Fence Cipher

int rails = 3;

String railFenceCipher = railFenceEncrypt(plaintext, rails);

System.out.println("Rail Fence Cipher: " + railFenceCipher);

// Simple Columnar Cipher

String key = "231";

String simpleColumnarCipher = simpleColumnarEncrypt(plaintext, key);

System.out.println("Simple Columnar Cipher: " + simpleColumnarCipher);

// Vermin Cipher

int shift = 3;

String verminCipher = verminEncrypt(plaintext, shift);

System.out.println("Vermin Cipher: " + verminCipher);

// Hill Cipher

int[][] keyMatrix = {{6, 24, 1}, {13, 16, 10}, {20, 17, 15}};

String hillCipher = hillEncrypt(plaintext, keyMatrix);

System.out.println("Hill Cipher: " + hillCipher);

scanner.close();

}

// Affine Cipher

public static String affineEncrypt(String text, int a, int b) {

StringBuilder result = new StringBuilder();

for (char ch : text.toCharArray()) {

if (Character.isLetter(ch)) {

int x = (int) ch - 'A';

x = (a \* x + b) % 26;

result.append((char) (x + 'A'));

} else {

result.append(ch);

}

}

return result.toString();

}

// Rail Fence Cipher

public static String railFenceEncrypt(String text, int rails) {

StringBuilder result = new StringBuilder();

char[][] grid = new char[rails][text.length()];

int row = 0;

boolean down = false;

for (int i = 0; i < text.length(); i++) {

grid[row][i] = text.charAt(i);

if (row == 0 || row == rails - 1) {

down = !down;

}

row += down ? 1 : -1;

}

for (int i = 0; i < rails; i++) {

for (int j = 0; j < text.length(); j++) {

if (grid[i][j] != 0) {

result.append(grid[i][j]);

}

}

}

return result.toString();

}

// Simple Columnar Cipher

public static String simpleColumnarEncrypt(String text, String key) {

int[] indexes = new int[key.length()];

for (int i = 0; i < key.length(); i++) {

indexes[i] = key.indexOf(i + '1');

}

StringBuilder result = new StringBuilder();

for (int i = 0; i < key.length(); i++) {

for (int j = indexes[i]; j < text.length(); j += key.length()) {

result.append(text.charAt(j));

}

}

return result.toString();

}

// Vermin Cipher

public static String verminEncrypt(String text, int shift) {

StringBuilder result = new StringBuilder();

for (char ch : text.toCharArray()) {

if (Character.isLetter(ch)) {

int x = (int) ch + shift;

if (Character.isUpperCase(ch) && x > 'Z') {

x -= 26;

} else if (Character.isLowerCase(ch) && x > 'z') {

x -= 26;

}

result.append((char) x);

} else {

result.append(ch);

}

}

return result.toString();

}

// Hill Cipher

public static String hillEncrypt(String text, int[][] key) {

StringBuilder result = new StringBuilder();

int n = key.length;

int[] vec = new int[n];

for (int i = 0; i < text.length(); i += n) {

for (int j = 0; j < n; j++) {

vec[j] = (int) text.charAt(i + j) - 'A';

}

for (int j = 0; j < n; j++) {

int sum = 0;

for (int k = 0; k < n; k++) {

sum += key[j][k] \* vec[k];

}

result.append((char) ((sum % 26) + 'A'));

}

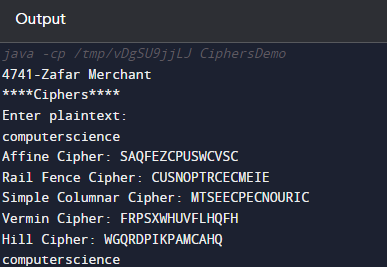
}

return result.toString();

}

}

**Output:**

****

**Conclusion:** Hence All the ciphers were implemented successfully encrypting and decrypting plaintext.

**Practical 3**

**Aim:** Write a program to implement the RSA Algorithm to perform encryption and

decryption.

**Theory:**

RSA algorithm is an asymmetric cryptography algorithm. Asymmetric actually means that it works on two different keys i.e. Public Key and Private Key. As the name describes that the Public Key is given to everyone, and the Private key is kept private.

An example of asymmetric cryptography:

* A client (for example browser) sends its public key to the server and requests some data.
* The server encrypts the data using the client’s public key and sends the encrypted data.
* The client receives this data and decrypts it.

Since this is asymmetric, nobody else except the browser can decrypt the data even if a third party has the public key of the browser.

The idea! The idea of RSA is based on the fact that it is difficult to factorize a large integer. The public key consists of two numbers where one number is a multiplication of two large prime numbers. And private key is also derived from the same two prime numbers. So if somebody can factorize the large number, the private key is compromised. Therefore encryption strength totally lies on the key size and if we double or triple the key size, the strength of encryption increases exponentially. RSA keys can be typically 1024 or 2048 bits long, but experts believe that 1024-bit keys could be broken in the near future. But till now it seems to be an infeasible task.

Advantages:

* Security: RSA algorithm is considered to be very secure and is widely used for secure data transmission.
* Public-key cryptography: RSA algorithm is a public-key cryptography algorithm, which means that it uses two different keys for encryption and decryption. The public key is used to encrypt the data, while the private key is used to decrypt the data.
* Key exchange: RSA algorithm can be used for secure key exchange, which means that two parties can exchange a secret key without actually sending the key over the network.

Disadvantages:

* Slow processing speed: RSA algorithm is slower than other encryption algorithms, especially when dealing with large amounts of data.
* Large key size: RSA algorithm requires large key sizes to be secure, which means that it requires more computational resources and storage space.
* Vulnerability to side-channel attacks: RSA algorithm is vulnerable to side-channel attacks, which means an attacker can use information leaked through side channels such as power consumption, electromagnetic radiation, and timing analysis to extract the private key.

**Source Code:**

/\*package whatever //do not write package name here \*/

import java.io.\*;

import java.math.\*;

import java.util.\*;

/\*

\* Java program for RSA asymmetric cryptographic algorithm.

\* For demonstration, values are

\* relatively small compared to practical application

\*/

public class RSA {

public static double gcd(double a, double h)

{

/\*

\* This function returns the gcd or greatest common

\* divisor

\*/

double temp;

while (true) {

temp = a % h;

if (temp == 0)

return h;

a = h;

h = temp;

}

}

public static void main(String[] args)

{

System.out.println("4741-Zafar Merchant");

System.out.println("\*\*\*\*RSA Algorithm\*\*\*\*");

double p = 3;

double q = 7;

// Stores the first part of public key:

double n = p \* q;

// Finding the other part of public key.

// double e stands for encrypt

double e = 2;

double phi = (p - 1) \* (q - 1);

while (e < phi) {

/\*

\* e must be co-prime to phi and

\* smaller than phi.

\*/

if (gcd(e, phi) == 1)

break;

else

e++;

}

int k = 2; // A constant value

double d = (1 + (k \* phi)) / e;

// Message to be encrypted

double msg = 12;

System.out.println("Message data = " + msg);

// Encryption c = (msg ^ e) % n

double c = Math.pow(msg, e);

c = c % n;

System.out.println("Encrypted data = " + c);

// Decryption m = (c ^ d) % n

double m = Math.pow(c, d);

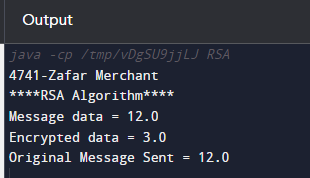
m = m % n;

System.out.println("Original Message Sent = " + m);

}

}

**Output:**

****

**Conclusion:** Hence RSA algorithm was implemented successfully passing message data encrypting and printing original message sent with public and private keys.

**Practical 4**

**Aim:** Write a program to implement the (i) Miller-Rabin Algorithm (ii) pollard p-1

Algorithm to perform encryption and decryption.

**Theory:**

(i) Miller-Rabin Algorithm:

The Miller-Rabin Algorithm is a probabilistic primality test used to determine if a given number is likely to be prime or composite. It repeatedly applies a test based on modular exponentiation and checks if the number passes the test with a certain probability.

Example:

Let's test if the number 561 is prime using the Miller-Rabin Algorithm.

We'll choose a random base a (let's say 2) and apply the test:

* Calculate b = a^d mod n, where d = 561 - 1 = 560 and n = 561.
* Since b = 2^560 mod 561 ≡ 1 (mod 561), move to the next step.
* Check if there's a value of r (with 0 <= r < s) such that b^(2^r) mod n is congruent to n - 1 (560 in this case). If yes, then the number might be prime.

Since 561 doesn't pass the Miller-Rabin test for base 2, it's composite.

(ii) Pollard p-1 Algorithm:

Pollard p-1 Algorithm is a method for factoring large composite numbers. It's based on Fermat's Little Theorem and uses the idea that if p is a prime factor of n and a is coprime to n, then a^(p-1) ≡ 1 (mod p). It iteratively tries different values of a to find a common factor of n.

Example:

Let's factor the number 91 using the Pollard p-1 Algorithm.

Choose a random value for a (let's say 2) and a bound B (let's say 10).

* Compute gcd(a^B - 1, n), where gcd is the greatest common divisor function.
* If the result is not 1 or n, then it's a non-trivial factor of n.
* Otherwise, increment a and repeat the process until a suitable factor is found or the bound is exceeded.

Using a = 2 and B = 10, we compute gcd(2^10 - 1, 91) = gcd(1023, 91) = 13, which is a non-trivial factor of 91.

So, the Pollard p-1 Algorithm found one factor of 91 as 13. Further iterations or different choices of a and B could potentially find more factors.

**Source Code:**

import java.util.Random;

public class PrimalityTestAndFactorization {

public static void main(String[] args) {

System.out.println("4741-Zafar Merchant");

System.out.println("\*\*\*\*Miller-Rabin Pollard p-1\*\*\*\*");

long n = 561; // Example number for demonstration

// Miller-Rabin Test

int k = 5; // Number of iterations for accuracy

boolean isPrime = millerRabinTest(n, k);

System.out.println("Miller-Rabin Test Result:");

System.out.println(n + " is " + (isPrime ? "probably prime." : "composite."));

// Pollard p-1 Factorization

long factor = pollardPMinus1Factorization(n);

System.out.println("\nPollard p-1 Factorization Result:");

System.out.println("One factor of " + n + " is: " + factor);

}

// Miller-Rabin Primality Test

public static boolean millerRabinTest(long n, int k) {

if (n <= 1 || n == 4)

return false;

if (n <= 3)

return true;

long d = n - 1;

while (d % 2 == 0)

d /= 2;

for (int i = 0; i < k; i++) {

long a = 2 + (long) (Math.random() % (n - 3));

long x = modularExponentiation(a, d, n);

if (x == 1 || x == n - 1)

continue;

boolean isProbablePrime = false;

while (d != n - 1) {

x = (x \* x) % n;

d \*= 2;

if (x == 1)

return false;

if (x == n - 1) {

isProbablePrime = true;

break;

}

}

if (!isProbablePrime)

return false;

}

return true;

}

// Pollard p-1 Factorization Algorithm

public static long pollardPMinus1Factorization(long n) {

long x = 2;

long y = 2;

long d = 1;

long c = 1;

Random random = new Random();

while (d == 1) {

x = (modularExponentiation(x, c, n) + n) % n;

y = (modularExponentiation(modularExponentiation(y, 2, n) + n, c, n) + n) % n;

d = gcd(Math.abs(x - y), n);

c++;

if (random.nextInt(100) == 0) { // Occasionally reset to minimize loop size

x = 2;

y = 2;

d = 1;

c = 1;

}

}

return d;

}

// Modular Exponentiation

public static long modularExponentiation(long base, long exponent, long mod) {

long result = 1;

base = base % mod;

while (exponent > 0) {

if (exponent % 2 == 1)

result = (result \* base) % mod;

exponent >>= 1;

base = (base \* base) % mod;

}

return result;

}

// Greatest Common Divisor

public static long gcd(long a, long b) {

if (b == 0)

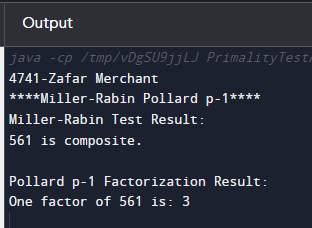
return a;

return gcd(b, a % b);

}

}

**Output:**

****

**Conclusion:** Hence Miller Rabin and Pollard p-1 algorithms were implemented successfully.

**Practical 5**

**Aim:** Write a program to implement the Diffie-Hellman Key Agreement algorithm to

generate symmetric keys.

**Theory:**

The Diffie-Hellman Key Exchange is a method for two parties to securely generate a shared secret key over an insecure channel. It allows two parties to agree on a shared secret without directly exchanging any secret information.

Here's a brief explanation with an example:

1. \*\*Initialization\*\*: Both parties agree on a public prime number \( p \) and a primitive root modulo \( p \), usually denoted as \( g \).

2. \*\*Key Generation\*\*:

- Each party chooses a private key:

- Party A selects a private key \( a \) and calculates \( A = g^a \mod p \).

- Party B selects a private key \( b \) and calculates \( B = g^b \mod p \).

3. \*\*Exchange\*\*:

- Party A sends \( A \) to Party B.

- Party B sends \( B \) to Party A.

4. \*\*Secret Key Calculation\*\*:

- Party A calculates the shared secret key using the received value \( B \):

\( K = B^a \mod p \).

- Party B calculates the shared secret key using the received value \( A \):

\( K = A^b \mod p \).

5. \*\*Result\*\*: Both parties now have the same shared secret key \( K \), which they can use for symmetric encryption.

Example:

Let's say Party A and Party B agree on \( p = 23 \) and \( g = 5 \).

Party A chooses a private key \( a = 6 \):

\( A = 5^6 \mod 23 = 8 \).

Party B chooses a private key \( b = 15 \):

\( B = 5^{15} \mod 23 = 19 \).

Party A sends \( A = 8 \) to Party B.

Party B sends \( B = 19 \) to Party A.

Party A calculates the shared secret key:

\( K = 19^6 \mod 23 = 2 \).

Party B calculates the shared secret key:

\( K = 8^{15} \mod 23 = 2 \).

Both parties now have the same shared secret key \( K = 2 \), which they can use for symmetric encryption.

**Source Code:**

// This program calculates the Key for two persons

// using the Diffie-Hellman Key exchange algorithm

class DiffieHillman {

// Power function to return value of a ^ b mod P

private static long power(long a, long b, long p)

{

if (b == 1)

return a;

else

return (((long)Math.pow(a, b)) % p);

}

// Driver code

public static void main(String[] args)

{

System.out.println("4741-Zafar Merchant");

System.out.println("\*\*\*\*Diffie Hillman Algorithm\*\*\*\*");

long P, G, x, a, y, b, ka, kb;

// Both the persons will be agreed upon the

// public keys G and P

// A prime number P is taken

P = 23;

System.out.println("The value of P:" + P);

// A primitive root for P, G is taken

G = 9;

System.out.println("The value of G:" + G);

// Alice will choose the private key a

// a is the chosen private key

a = 4;

System.out.println("The private key a for Alice:"

+ a);

// Gets the generated key

x = power(G, a, P);

// Bob will choose the private key b

// b is the chosen private key

b = 3;

System.out.println("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

System.out.println("Secret key for the Alice is:"

+ ka);

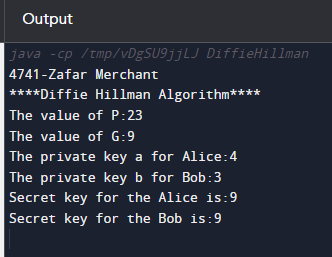
System.out.println("Secret key for the Bob is:"

+ kb);

}

}

**Output:**

****

**Conclusion:** Hence Diffie Hillman algorithm was implemented successfully.

**Practical 6**

**Aim:** Write a program to implement the MD5 algorithm compute the message digest.

**Theory:**

MD5 (Message Digest Algorithm 5) is a widely used cryptographic hash function that produces a 128-bit (16-byte) hash value, typically expressed as a 32-character hexadecimal number. It takes an input message of any length and produces a fixed-size output hash. Here's a brief explanation of how MD5 works:

1. \*\*Message Padding\*\*: The input message is padded to a multiple of 512 bits (64 bytes) to ensure it's a multiple of the block size used in MD5.

2. \*\*Initialization\*\*: MD5 uses four 32-bit registers (A, B, C, D) and 64 constant values derived from the sine function.

3. \*\*Processing Blocks\*\*: The message is processed in 512-bit (64-byte) blocks. Each block is divided into 16 32-bit words.

4. \*\*Four Rounds\*\*: Each block goes through four rounds of processing, consisting of 16 operations each.

5. \*\*Operations\*\*: MD5 primarily uses bitwise logical functions (AND, OR, XOR) and modular addition to manipulate the data in each round. These operations shuffle the bits and mix the input data in a non-linear fashion.

6. \*\*Updating the State\*\*: After processing each block, the MD5 hash state (the values in registers A, B, C, D) is updated based on the processed block.

7. \*\*Output\*\*: After processing all blocks, the final hash value is obtained by concatenating the values in the four registers (A, B, C, D) in hexadecimal format.

MD5 is vulnerable to collision attacks, where two different messages produce the same hash value, and it's no longer recommended for cryptographic purposes due to its vulnerabilities. However, it's still used in non-security contexts like checksums and checksum verification.

**Source Code:**

import java.math.BigInteger;

import java.security.MessageDigest;

import java.security.NoSuchAlgorithmException;

// Java program to calculate MD5 hash value

public class MD5 {

public static String getMd5(String input)

{

try {

// Static getInstance method is called with hashing MD5

MessageDigest md = MessageDigest.getInstance("MD5");

// digest() method is called to calculate message digest

// of an input digest() return array of byte

byte[] messageDigest = md.digest(input.getBytes());

// Convert byte array into signum representation

BigInteger no = new BigInteger(1, messageDigest);

// Convert message digest into hex value

String hashtext = no.toString(16);

while (hashtext.length() < 32) {

hashtext = "0" + hashtext;

}

return hashtext;

}

// For specifying wrong message digest algorithms

catch (NoSuchAlgorithmException e) {

throw new RuntimeException(e);

}

}

// Driver code

public static void main(String args[]) throws NoSuchAlgorithmException

{

String s = "GeeksForGeeks";

System.out.println("4741 - Zafar Merchant");

System.out.println("\*\*\*\*MD5 Hash\*\*\*\*");

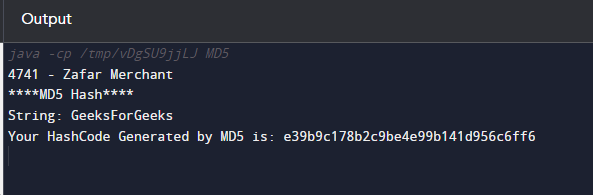
System.out.println("String: " + s);

System.out.println("Your HashCode Generated by MD5 is: " + getMd5(s));

}

}

**Output:**

****

**Conclusion:** Hence MD5 hash was implemented on String “GeeksForGeeks” successfully.

**Practical 7**

**Aim:** Write a program to implement HMAC signatures.

**Theory:**

HMAC (Hash-based Message Authentication Code) is a mechanism for generating a cryptographic signature for verifying the integrity and authenticity of a message. It uses a cryptographic hash function in combination with a secret key. Here's a brief explanation of how HMAC signatures work:

1. \*\*Initialization\*\*: Both the sender and receiver share a secret key \( K \).

2. \*\*Message Preprocessing\*\*: The message to be authenticated is preprocessed if necessary (e.g., padding).

3. \*\*Hashing\*\*: The message is hashed using a cryptographic hash function (such as SHA-256 or MD5) to generate a hash value.

4. \*\*Key Modification\*\*: The secret key \( K \) is modified or padded to match the block size of the hash function, if needed.

5. \*\*Inner and Outer Padding\*\*: The modified key is used to create two different padded keys: one with the XOR operation (inner padding) and another with the XOR operation followed by a bitwise NOT operation (outer padding).

6. \*\*Hashing with Keys\*\*: The hash function is applied twice:

- First, the inner padded key is concatenated with the hashed message and hashed together.

- Then, the outer padded key is concatenated with the result from the first step, and the combined data is hashed again.

7. \*\*Result\*\*: The output of the second hash operation is the HMAC signature, which is sent along with the message.

Example:

Let's say we want to generate an HMAC signature for the message "Hello, world!" using the SHA-256 hash function and a secret key "secretkey".

1. Secret Key: "secretkey"

2. Message: "Hello, world!"

3. Hash Function: SHA-256

First, we modify the key to match the block size of SHA-256.

Then, we compute the inner and outer padded keys.

Next, we hash the message with the inner padded key and then hash the result with the outer padded key.

The final output is the HMAC signature, which is a fixed-length hash value.

The HMAC signature is then sent along with the original message. The receiver can recompute the HMAC signature using the same secret key and verify if it matches the received signature. If the signatures match, the receiver can be confident that the message has not been tampered with and originates from a trusted source.

**Source Code:**

import javax.crypto.Mac;

import javax.crypto.spec.SecretKeySpec;

import java.security.InvalidKeyException;

import java.security.NoSuchAlgorithmException;

import java.util.Base64;

public class HMACSignature {

public static void main(String[] args) {

System.out.println("4741-Zafar Merchant");

System.out.println("\*\*\*\*HMAC Signatures\*\*\*\*");

try {

// Secret key

String secretKey = "MySecretKey123";

// Message to be signed

String message = "Hello, world!";

// Generate HMAC signature

String hmacSignature = generateHMAC(message, secretKey);

// Print HMAC signature

System.out.println("HMAC Signature: " + hmacSignature);

} catch (NoSuchAlgorithmException | InvalidKeyException e) {

e.printStackTrace();

}

}

// Function to generate HMAC signature

public static String generateHMAC(String message, String secretKey)

throws NoSuchAlgorithmException, InvalidKeyException {

// Get HMAC-SHA256 algorithm instance

Mac hmacSHA256 = Mac.getInstance("HmacSHA256");

// Initialize HMAC with secret key

SecretKeySpec secretKeySpec = new SecretKeySpec(secretKey.getBytes(), "HmacSHA256");

hmacSHA256.init(secretKeySpec);

// Compute HMAC signature

byte[] hmacBytes = hmacSHA256.doFinal(message.getBytes());

// Convert byte array to base64 string

return Base64.getEncoder().encodeToString(hmacBytes);

}

}

**Output:**

**A screenshot of a computer

Description automatically generated**

**Conclusion:** Hence HMAC signature was implemented successfully.