**Ass. No: 4. Title: Secure Hash Algorithm 512 (SHA 512)**

**Exp. No: 5**

**Date: 08/03/2025**

**Aim:**

To find the padding bits and total number of blocks required to process the original message using SHA 512 in “JAVA”

Concept:   
  
SHA-512 processes messages in 1024-bit blocks. Since messages are often not a perfect multiple of 1024 bits, padding bits are added to ensure proper processing. The message is also appended with a 128-bit length field, representing the original message length.

Key Points:

* The original message length is in bits.
* Padding bits are added to make the message length + 128 bits a multiple of 1024.
* The number of blocks required is determined based on the total size after padding.
* The padding scheme follows SHA-512 standards, ensuring secure hashing.

**Algorithm:**

1. Input the original message length (in bits).
2. Calculate the padding bits required:

* The total message size should be (message length + 128 bits).
* Padding is added to make this value a multiple of 1024 bits.

1. Compute the total number of bits after padding.
2. Determine the total number of blocks by dividing the total number of bits by 1024.
3. Display the results:

* Original message length
* Padding bits required
* Total blocks needed

**Program:**

import *java.util.Scanner*;

*public* *class* SHA512Padding {

*public* *static* void main(String[] args) {

Scanner scanner = new Scanner(System.in);

*// Input message size in bits*

System.out.print("Enter the message size in bits: ");

long messageSize = scanner.nextLong();

*// SHA-512 block size is 1024 bits*

long blockSize = 1024;

*// Padding calculation*

long paddingBits = (blockSize - (messageSize + 128) % blockSize) % blockSize;

long totalBlocks = (messageSize + paddingBits + 128) / blockSize;

System.out.println("Padding bits required: " + paddingBits);

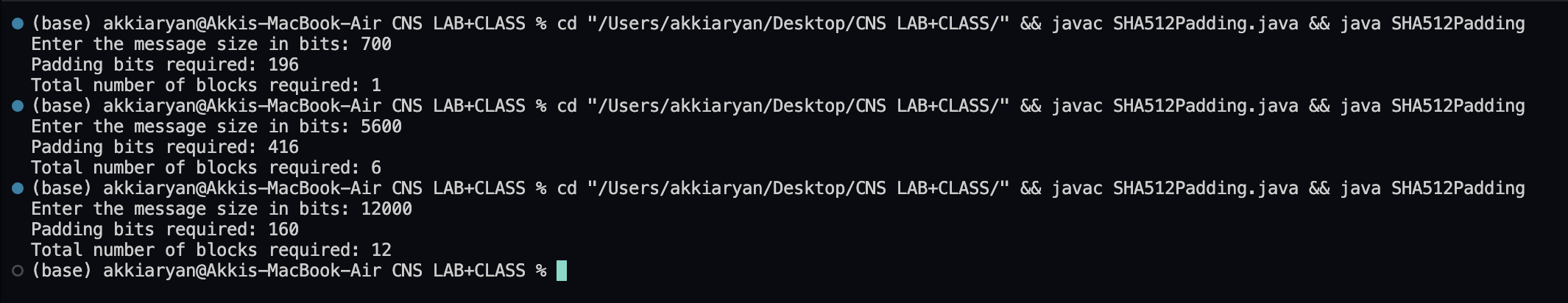
System.out.println("Total number of blocks required: " + totalBlocks);

scanner.close();

}

}

**Output:**



**Ex1. Enter the message size in bits: 700**

**Padding bits required: 196**

**Total number of blocks required: 1**

**Ex2. Enter the message size in bits: 5600**

**Padding bits required: 416**

**Total number of blocks required: 6**

**Ex3. Enter the message size in bits: 12000**

**Padding bits required: 160**

**Total number of blocks required: 12**

**Ass. No: 4 Title: Digital Signature Standard (DSS)**

**Exp. No: 6**

**Aim:** To create the signature in the sender side and verify the signature in the receiver side using DSS in “JAVA”

**Concept:**

Digital Signature Standard (DSS) is a cryptographic technique used to ensure the authenticity and integrity of a message. It involves two main processes: Signature Generation (Sender Side) and Signature Verification (Receiver Side).

**Key Concepts:**

* A private key is used to generate the signature at the sender's side.
* A public key is used to verify the signature at the receiver's side.
* A hash function is applied to the message to generate a fixed-length digest.
* A random integer (k) is used to introduce randomness in the signing process.
* The receiver uses verification formulas to validate the signature.

**Steps in Digital Signature Process:**

**Sender Side (Signature Creation):**

* Choose two large prime numbers, p and q.
* Generate a private key and compute the public key.
* Select a random integer k.
* Compute the hash h of the original message.
* Generate the signature using r and s values.
* Send the message and signature to the receiver.

**Receiver Side (Signature Verification):**

* Compute the hash (h) of the received message.
* Calculate the values u1 and u2 using the verification formula.
* Compute v and compare it with r.
* If v == r, the signature is valid; otherwise, it is invalid.

**Algorithm:**

**Sender Side (Signature Generation)**

1. Input: Take two prime numbers p and q from the user.
2. Generate Keys:

* Choose a private key x such that x < q.
* Compute the public key y = g^x mod p (where g is a generator).

1. Input Random Integer: Take a random integer k from the user.
2. Hash Calculation:

* Compute the hash h of the original message using SHA-256.

1. Signature Computation:

* Compute r = (g^k mod p) mod q.
* Compute s = (h + x \* r) \* k^(-1) mod q.

1. Send the Signature:

* Transmit (r, s) and the original message to the receiver.

**Receiver Side (Signature Verification)**

1. Input: Receive r, s, and the original message.
2. Hash Calculation: Compute the hash h of the received message.
3. Verification Calculation:

* Compute w = s^(-1) mod q.
* Compute u1 = (h \* w) mod q.
* Compute u2 = (r \* w) mod q.
* Compute v = ((g^u1 \* y^u2) mod p) mod q.

1. Signature Validation:

* If v == r, the signature is valid.
* Otherwise, the signature is invalid.

**Program:**

import *java.math.BigInteger*;

import *java.security.\**;

import *java.util.Scanner*;

*public* *class* DSS\_Signature {

*public* *static* void main(String[] args) *throws* Exception {

Scanner sc = new Scanner(System.in);

*// Step 1: Get two prime numbers from user*

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

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

System.out.print("Enter second prime number (q): ");

BigInteger q = new BigInteger(sc.nextLine());

*// Step 2: Generate the public key*

System.out.print("Enter a private key (should be less than q): ");

BigInteger privateKey = new BigInteger(sc.nextLine());

System.out.print("Enter a random integer (k): ");

BigInteger k = new BigInteger(sc.nextLine());

System.out.print("Enter the original message: ");

String message = sc.nextLine();

*// Compute Hash (h) of the message*

MessageDigest md = MessageDigest.getInstance("SHA-256");

byte[] messageHash = md.digest(message.getBytes());

BigInteger h = new BigInteger(1, messageHash);

*// Step 3: Compute r = (g^k mod p) mod q*

BigInteger g = new BigInteger("2"); *// Example generator g*

BigInteger r = g.modPow(k, p).mod(q);

*// Step 4: Compute s = (h + privateKey \* r) \* k^-1 mod q*

BigInteger kInv = k.modInverse(q);

BigInteger s = (h.add(privateKey.multiply(r))).multiply(kInv).mod(q);

*// Public Key computation: y = g^x mod p*

BigInteger publicKey = g.modPow(privateKey, p);

*// Sender Side Output*

System.out.println("\nSender Side Output:");

System.out.println("r: " + r);

System.out.println("Public Key (y): " + publicKey);

System.out.println("a (random integer k): " + k);

System.out.println("b (message hash h): " + h);

*// Receiver Side: Verification*

BigInteger w = s.modInverse(q);

BigInteger u1 = (h.multiply(w)).mod(q);

BigInteger u2 = (r.multiply(w)).mod(q);

BigInteger v = ((g.modPow(u1, p).multiply(publicKey.modPow(u2, p))).mod(p)).mod(q);

*// Receiver Side Output*

System.out.println("\nReceiver Side Output:");

System.out.println("z (Hash of received message h): " + h);

System.out.println("u1: " + u1);

System.out.println("u2: " + u2);

System.out.println("Verification Formula (v): " + v);

if (v.equals(r)) {

System.out.println("Signature Verified: ✅ Valid");

} else {

System.out.println("Signature Verification Failed: ❌ Invalid");

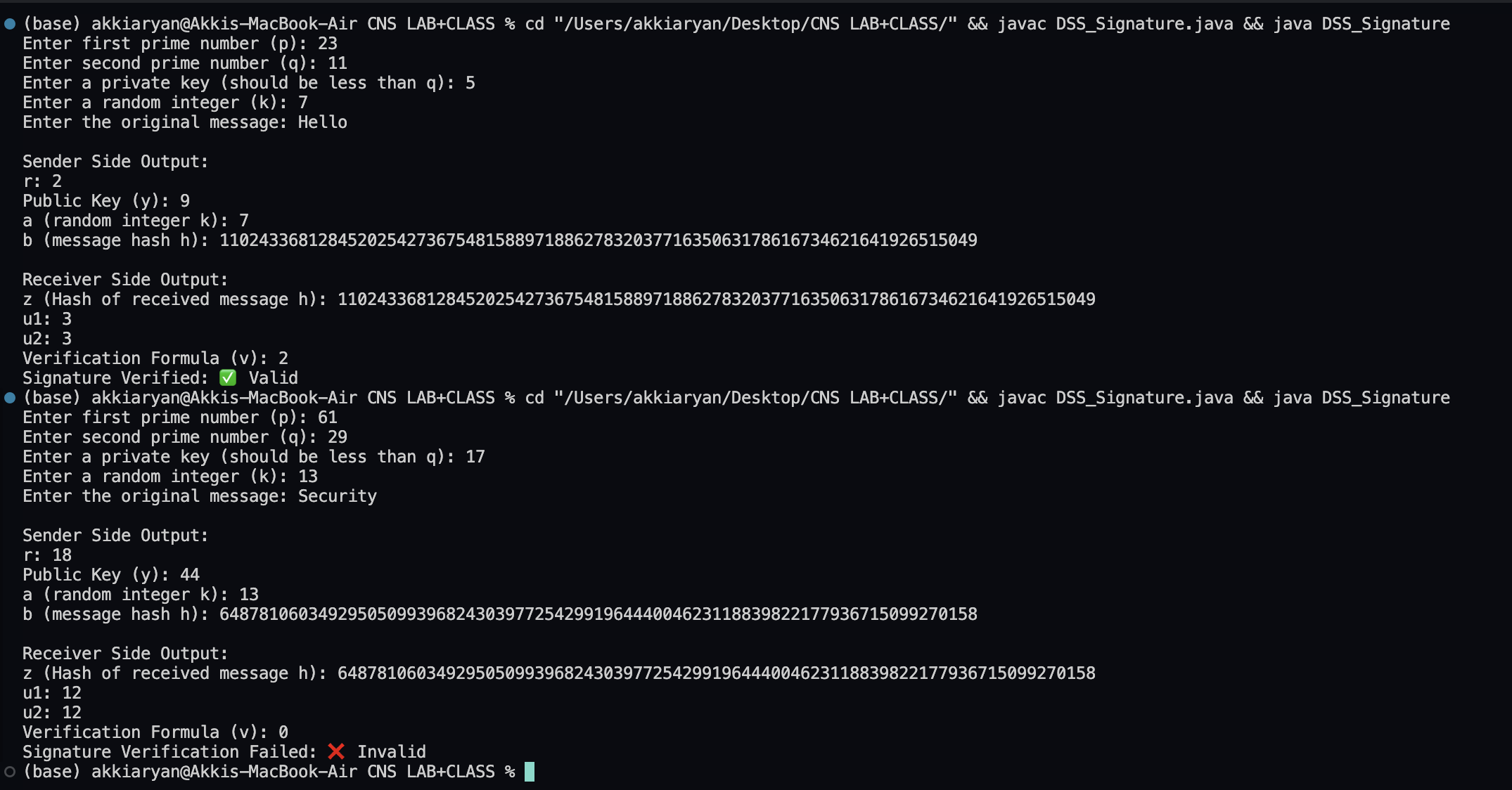
}

sc.close();

}

}

**Output:**

****

**Ass. No: 4 Title: Message Digest 5 (MD5)**

**Exp. No: 7**

**Aim:**

1. To find the 1. Original message in binary bits, 2. Padding binary bits & 3. Length of the message in binary bits using MD 5 “JAVA”
2. Find the output of MD buffers A, B, C & D of steps 1 & 2 of Round 1 MD5.

**Concept:**

**i) Finding:**

* The original message is converted to binary representation.
* MD5 requires the message to be padded to ensure its length is congruent to 448 modulo 512.
* Padding involves adding a 1 bit followed by 0s until the length reaches 448 bits, then appending a 64-bit representation of the original length.

**ii) MD Buffers Calculation:**

* MD5 initializes four buffers: A, B, C, and D.
* It divides the message into 512-bit chunks and processes them using bitwise operations.
* Round 1 consists of 16 operations where the buffers undergo bitwise transformations using predefined constants and functions.

**Algorithm:**

**i) Finding the original message, padding, and length in binary:**

1. Get the original message from the user.
2. Convert the message to binary.
3. Calculate the length of the message in bits.
4. Compute the required padding bits.
5. Append the padding bits and length to form a 512-bit block.
6. Display the results.

**ii) Finding MD Buffers A, B, C, & D in Step 1 & 2 of Round 1:**

1. Initialize MD5 buffers (A, B, C, D) with predefined values.
2. Get the inputs M0 and K1 for step 1.
3. Apply the first transformation function to update A, B, C, and D.
4. Display the new values of A, B, C, and D after step 1.
5. Use the updated buffer values as input for step 2.
6. Get inputs M1 and K2.
7. Apply the transformation for step 2.
8. Display the updated values of A, B, C, and D after step 2.

**Program:**

For Aim 1:

import *java.util.Scanner*;

*public* *class* MD5Padding {

*public* *static* void main(String[] args) {

Scanner scanner = new Scanner(System.in);

*// Get the original message*

System.out.print("Enter the original message: ");

String message = scanner.nextLine();

*// Convert message to binary*

StringBuilder binaryMessage = new StringBuilder();

for (char c : message.toCharArray()) {

binaryMessage.append(String.format("%8s", Integer.toBinaryString(c)).replace(' ', '0'));

}

int originalLength = binaryMessage.length();

*// Padding process (1 bit followed by 0s)*

StringBuilder padding = new StringBuilder("1");

while ((binaryMessage.length() + padding.length() + 64) % 512 != 0) {

padding.append("0");

}

*// Convert original length to binary (64-bit representation)*

String lengthBinary = String.format("%64s", Integer.toBinaryString(originalLength)).replace(' ', '0');

*// Display results*

System.out.println("\nOriginal Message in Binary: " + binaryMessage);

System.out.println("Total Padding Bits: " + padding.length());

System.out.println("Padding Bits in Binary: " + padding);

System.out.println("Original Message Length: " + originalLength + " bits");

System.out.println("Length in Binary (64-bit representation): " + lengthBinary);

scanner.close();

}

}

**Program:**

For Aim 2:

import *java.util.Scanner*;

*public* *class* MD5Round1 {

*public* *static* void main(String[] args) {

Scanner scanner = new Scanner(System.in);

*// Getting user input for MD buffers*

System.out.print("Enter initial MD buffers (A, B, C, D) in hex format:\n");

System.out.print("A: "); int A = Integer.parseUnsignedInt(scanner.next(), 16);

System.out.print("B: "); int B = Integer.parseUnsignedInt(scanner.next(), 16);

System.out.print("C: "); int C = Integer.parseUnsignedInt(scanner.next(), 16);

System.out.print("D: "); int D = Integer.parseUnsignedInt(scanner.next(), 16);

*// Getting user input for M0 and K1*

System.out.print("Enter M0 (in hex): "); int M0 = Integer.parseUnsignedInt(scanner.next(), 16);

System.out.print("Enter K1 (in hex): "); int K1 = Integer.parseUnsignedInt(scanner.next(), 16);

*// Step 1 Transformation*

int[] buffers = {A, B, C, D};

md5Transform(buffers, M0, K1);

System.out.println("\nAfter Step 1 - Round 1:");

printBuffers(buffers);

*// Using the output of Step 1 as input for Step 2*

System.out.print("\nEnter M1 (in hex): "); int M1 = Integer.parseUnsignedInt(scanner.next(), 16);

System.out.print("Enter K2 (in hex): "); int K2 = Integer.parseUnsignedInt(scanner.next(), 16);

*// Step 2 Transformation*

md5Transform(buffers, M1, K2);

System.out.println("\nAfter Step 2 - Round 1:");

printBuffers(buffers);

scanner.close();

}

*// Simulated MD5 transformation step*

*private* *static* void md5Transform(int[] buffers, int M, int K) {

int F = (buffers[1] & buffers[2]) | (~buffers[1] & buffers[3]); *// Nonlinear function*

int temp = buffers[0] + F + M + K;

buffers[0] = buffers[3];

buffers[3] = buffers[2];

buffers[2] = buffers[1];

buffers[1] = buffers[1] + Integer.rotateLeft(temp, 7);

}

*// Print the buffer values in hexadecimal*

*private* *static* void printBuffers(int[] buffers) {

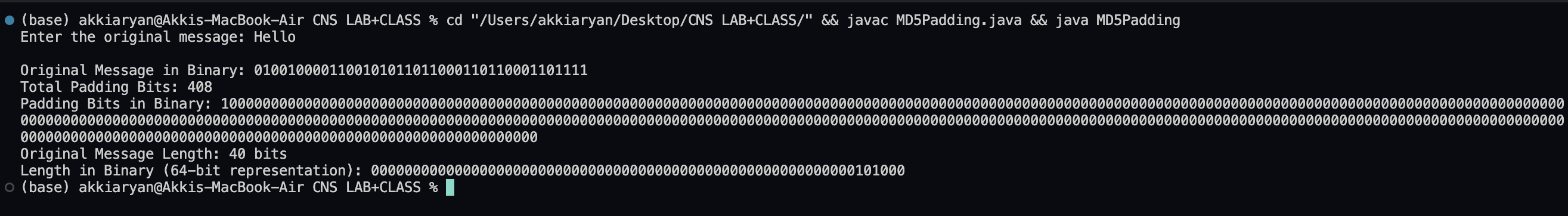
System.out.printf("A: %08X, B: %08X, C: %08X, D: %08X%n", buffers[0], buffers[1], buffers[2], buffers[3]);

}

}

**Output:**

**For Aim 1:**

****  
Original Message in Binary: 0100100001100101011011000110110001101111

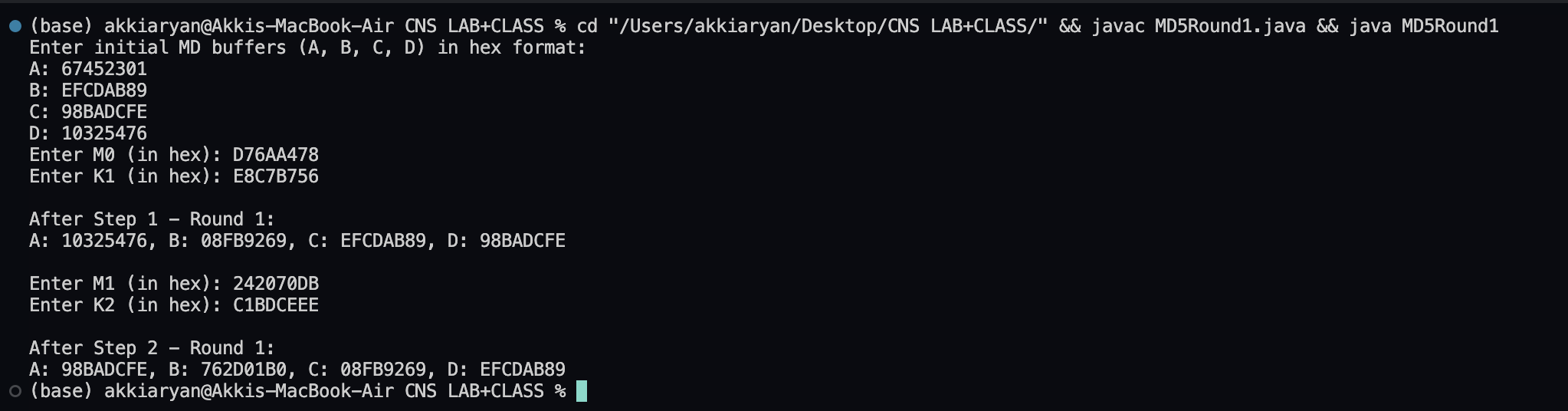
Total Padding Bits: 408

Padding Bits in Binary: 100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

Original Message Length: 40 bits

Length in Binary (64-bit representation): 0000000000000000000000000000000000000000000000000000000000101000  
  
  
  
**Output:**

**For Aim 2:**

  
  
Enter initial MD buffers (A, B, C, D) in hex format:

A: 67452301

B: EFCDAB89

C: 98BADCFE

D: 10325476

Enter M0 (in hex): D76AA478

Enter K1 (in hex): E8C7B756

After Step 1 - Round 1:

A: 10325476, B: 08FB9269, C: EFCDAB89, D: 98BADCFE

Enter M1 (in hex): 242070DB

Enter K2 (in hex): C1BDCEEE

After Step 2 - Round 1:

A: 98BADCFE, B: 762D01B0, C: 08FB9269, D: EFCDAB89