Cryptography and Network Security Lab
Digital Assignment 4
22BCE3939
Karan Sehgal

1. Without using library functions develop a menu-driven code to simulate the following Asymmetric algorithms.
i. SHA512 and MD5
ii. Digital Signature Standard

NOTE: The program should have sufficient test cases to perform data validation. The output should contain intermediate results [provide user-friendly I/O messages]

Source Code:

```
#include <iostream>
#include <string>
#include <vector>
#include <iomanip>
#include <sstream>
#include <cmath>
#include <cstring>

using namespace std;

// Typedefs for SHA-512
typedef unsigned long long uint64;
typedef unsigned int uint32;
typedef unsigned char uint8;

// SHA-512 Constants
const uint64 SHA512_K[80] = {
    0x428a2f98d728ae22, 0x7137449123ef65cd,
0xb5c0fbcfec4d3b2f, 0xe9b5dba58189dbbc,
    0x3956c25bf348b538, 0x59f111f1b605d019,
0x923f82a4af194f9b, 0xab1c5ed5da6d8118,
    0xd807aa98a3030242, 0x12835b0145706fbe,
0x243185be4ee4b28c, 0x550c7dc3d5ffb4e2,
    0x72be5d74f27b896f, 0x80deb1fe3b1696b1,
0x9bdc06a725c71235, 0xc19bf174cf692694,
    0xe49b69c19ef14ad2, 0xefbe4786384f25e3,
0x0fc19dc68b8cd5b5, 0x240ca1cc77ac9c65,
    0x2de92c6f592b0275, 0x4a7484aa6ea6e483,
0x5cb0a9dcbd41fbd4, 0x76f988da831153b5,
    0x983e5152ee66dfab, 0xa831c66d2db43210,
0xb00327c898fb213f, 0xbf597fc7beef0ee4,
    0xc6e00bf33da88fc2, 0xd5a79147930aa725,
0x06ca6351e003826f, 0x142929670a0e6e70,
    0x27b70a8546d22ffc, 0x2e1b21385c26c926,
0x4d2c6dfc5ac42aed, 0x53380d139d95b3df,
    0x650a73548baf63de, 0x766a0abb3c77b2a8,
0x81c2c92e47edaee6, 0x92722c851482353b,
    0xa2bfe8a14cf10364, 0xa81a664bbc423001,
0xc24b8b70d0f89791, 0xc76c51a30654be30,
    0xd192e819d6ef5218, 0xd69906245565a910,
0xf40e35855771202a, 0x106aa07032bbd1b8,
    0x19a4c116b8d2d0c8, 0x1e376c085141ab53,
0x2748774cdf8eeb99, 0x34b0bcb5e19b48a8,
```

```
    0x391c0cb3c5c95a63, 0x4ed8aa4ae3418acb,
0x5b9cca4f7763e373, 0x682e6ff3d6b2b8a3,
    0x748f82ee5defb2fc, 0x78a5636f43172f60,
0x84c87814a1f0ab72, 0x8cc702081a6439ec,
    0x90befffa23631e28, 0xa4506cebde82bde9,
0xbef9a3f7b2c67915, 0xc67178f2e372532b,
    0xca273eceea26619c, 0xd186b8c721c0c207,
0xeada7dd6cde0eb1e, 0xf57d4f7fee6ed178,
    0x06f067aa72176fba, 0x0a637dc5a2c898a6,
0x113f9804bef90dae, 0x1b710b35131c471b,
    0x28db77f523047d84, 0x32caab7b40c72493,
0x3c9ebe0a15c9bebc, 0x431d67c49c100d4c,
    0x4cc5d4becb3e42b6, 0x597f299cfc657e2a,
0x5fcb6fab3ad6faec, 0x6c44198c4a475817
};

// MD5 Constants
const uint32 MD5_K[64] = {
    0xd76aa478, 0xe8c7b756, 0x242070db, 0xc1bdceee, 0xf57c0faf,
0x4787c62a, 0xa8304613, 0xfd469501,
    0x698098d8, 0x8b44f7af, 0xffff5bb1, 0x895cd7be, 0x6b901122,
0xfd987193, 0xa679438e, 0x49b40821,
    0xf61e2562, 0xc040b340, 0x265e5a51, 0xe9b6c7aa,
0xd62f105d, 0x02441453, 0xd8a1e681, 0xe7d3fbc8,
    0x21e1cde6, 0xc33707d6, 0xf4d50d87, 0x455a14ed,
0xa9e3e905, 0xfcefa3f8, 0x676f02d9, 0x8d2a4c8a,
    0xfffa3942, 0x8771f681, 0x6d9d6122, 0xfde5380c, 0xa4beea44,
0x4bdecfa9, 0xf6bb4b60, 0xbebfbc70,
    0x289b7ec6, 0xeaa127fa, 0xd4ef3085, 0x04881d05,
0xd9d4d039, 0xe6db99e5, 0x1fa27cf8, 0xc4ac5665,
    0xf4292244, 0x432aff97, 0xab9423a7, 0xfc93a039, 0x655b59c3,
0x8f0ccc92, 0xffeff47d, 0x85845dd1,
    0x6fa87e4f, 0xfe2ce6e0, 0xa3014314, 0x4e0811a1, 0xf7537e82,
0xbd3af235, 0x2ad7d2bb, 0xeb86d391
};

// MD5 shift amounts
const uint32 MD5_S[64] = {
    7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22,
    5,  9, 14, 20, 5,  9, 14, 20, 5,  9, 14, 20, 5,  9, 14, 20,
    4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23,
    6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21
};
```

```
// Utility functions
// Right rotate for SHA-512
uint64 ROTR(uint64 x, int n) {
    return (x >> n) | (x << (64 - n));
}

// Right rotate for MD5
uint32 ROTL(uint32 x, int n) {
    return (x << n) | (x >> (32 - n));
}

// Convert string to hex
string toHex(const string& input) {
    stringstream ss;
    ss << hex << setfill('0');
    for (size_t i = 0; i < input.length(); i++) {
        ss << setw(2) << (int)(unsigned char)input[i];
    }
    return ss.str();
}

// Convert bytes to hex string
string bytesToHexString(const uint8* data, size_t length) {
    stringstream ss;
    ss << hex << setfill('0');
    for (size_t i = 0; i < length; i++) {
        ss << setw(2) << (int)data[i];
    }
    return ss.str();
}

// SHA-512 specific functions
uint64 Ch(uint64 x, uint64 y, uint64 z) {
    return (x & y) ^ (~x & z);
}

uint64 Maj(uint64 x, uint64 y, uint64 z) {
    return (x & y) ^ (x & z) ^ (y & z);
}

uint64 Sigma0(uint64 x) {
    return ROTR(x, 28) ^ ROTR(x, 34) ^ ROTR(x, 39);
}
```

```
uint64 Sigma1(uint64 x) {
    return ROTR(x, 14) ^ ROTR(x, 18) ^ ROTR(x, 41);
}

uint64 sigma0(uint64 x) {
    return ROTR(x, 1) ^ ROTR(x, 8) ^ (x >> 7);
}

uint64 sigma1(uint64 x) {
    return ROTR(x, 19) ^ ROTR(x, 61) ^ (x >> 6);
}

// MD5 specific functions
uint32 F(uint32 x, uint32 y, uint32 z) {
    return (x & y) | (~x & z);
}

uint32 G(uint32 x, uint32 y, uint32 z) {
    return (x & z) | (y & ~z);
}

uint32 H(uint32 x, uint32 y, uint32 z) {
    return x ^ y ^ z;
}

uint32 I(uint32 x, uint32 y, uint32 z) {
    return y ^ (x | ~z);
}

// SHA-512 Implementation
class SHA512 {
private:
    uint64 h[8]; // Hash values
    vector<uint8> message; // Message after padding
    vector<vector<uint64>> intermediateResults;

    void padMessage(const string& input) {
        // Convert input string to bytes
        for (char c : input) {
            message.push_back(static_cast<uint8>(c));
        }

        size_t originalLength = message.size();
        size_t originalLengthBits = originalLength * 8;
```

```
        // Append the bit '1' to the message
        message.push_back(0x80);

        // Append 0 ≤ k < 1024 bits '0', so that the resulting message
length (in bits)
        // is congruent to 896 (mod 1024)
        while ((message.size() * 8) % 1024 != 896) {
            message.push_back(0);
        }

        // Append the length of the original message (before padding)
as a 128-bit big-endian integer
        for (int i = 7; i >= 0; i--) {
            message.push_back(0); // Upper 64 bits are all zeros (we
don't support messages > 2^64 bits)
        }

        // Lower 64 bits
        for (int i = 7; i >= 0; i--) {
            message.push_back((originalLengthBits >> (i * 8)) & 0xFF);
        }
    }

    void processBlock(size_t block) {
        vector<uint64> stateValues;
        vector<uint64> w(80);
        uint64 a = h[0], b = h[1], c = h[2], d = h[3], e = h[4], f = h[5],
g = h[6], h_val = h[7];

        // Save initial state
        stateValues.push_back(a);
        stateValues.push_back(b);
        stateValues.push_back(c);
        stateValues.push_back(d);
        stateValues.push_back(e);
        stateValues.push_back(f);
        stateValues.push_back(g);
        stateValues.push_back(h_val);

        // Prepare the message schedule
        for (int t = 0; t < 16; t++) {
            w[t] = 0;
            for (int i = 0; i < 8; i++) {
```

```
            w[t] = (w[t] << 8) | message[block * 128 + t * 8 + i];
        }
    }

    for (int t = 16; t < 80; t++) {
        w[t] = sigma1(w[t-2]) + w[t-7] + sigma0(w[t-15]) + w[t-16];
    }

    // Save message schedule
    for (int t = 0; t < 80; t++) {
        stateValues.push_back(w[t]);
    }

    // Main loop
    for (int t = 0; t < 80; t++) {
        uint64 T1 = h_val + Sigma1(e) + Ch(e, f, g) + SHA512_K[t]
+ w[t];
        uint64 T2 = Sigma0(a) + Maj(a, b, c);
        h_val = g;
        g = f;
        f = e;
        e = d + T1;
        d = c;
        c = b;
        b = a;
        a = T1 + T2;

        // Save intermediate values
        stateValues.push_back(a);
        stateValues.push_back(b);
        stateValues.push_back(c);
        stateValues.push_back(d);
        stateValues.push_back(e);
        stateValues.push_back(f);
        stateValues.push_back(g);
        stateValues.push_back(h_val);
    }

    // Save the intermediate results
    intermediateResults.push_back(stateValues);

    // Update hash values
    h[0] += a;
    h[1] += b;
```

```
        h[2] += c;
        h[3] += d;
        h[4] += e;
        h[5] += f;
        h[6] += g;
        h[7] += h_val;
    }

public:
    SHA512() {
        // Initialize hash values (first 64 bits of the fractional parts of
the square roots of the first 8 primes)
        h[0] = 0x6a09e667f3bcc908;
        h[1] = 0xbb67ae8584caa73b;
        h[2] = 0x3c6ef372fe94f82b;
        h[3] = 0xa54ff53a5f1d36f1;
        h[4] = 0x510e527fade682d1;
        h[5] = 0x9b05688c2b3e6c1f;
        h[6] = 0x1f83d9abfb41bd6b;
        h[7] = 0x5be0cd19137e2179;
    }

    void reset() {
        // Reset the hash values to initial state
        h[0] = 0x6a09e667f3bcc908;
        h[1] = 0xbb67ae8584caa73b;
        h[2] = 0x3c6ef372fe94f82b;
        h[3] = 0xa54ff53a5f1d36f1;
        h[4] = 0x510e527fade682d1;
        h[5] = 0x9b05688c2b3e6c1f;
        h[6] = 0x1f83d9abfb41bd6b;
        h[7] = 0x5be0cd19137e2179;

        message.clear();
        intermediateResults.clear();
    }

    string hash(const string& input) {
        reset();
        padMessage(input);

        // Process the message in 1024-bit blocks
        for (size_t i = 0; i < message.size() / 128; i++) {
            processBlock(i);
```

```
        }

        // Convert the hash values to a string
        stringstream ss;
        for (int i = 0; i < 8; i++) {
            ss << hex << setw(16) << setfill('0') << h[i];
        }

        return ss.str();
    }

    void printIntermediateResults() {
        cout << "Intermediate results for SHA-512:" << endl;
        for (size_t block = 0; block < intermediateResults.size();
block++) {
            cout << "Block " << block + 1 << ":" << endl;

            // Print initial state
            cout << "  Initial state:" << endl;
            cout << "    a: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][0] << endl;
            cout << "    b: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][1] << endl;
            cout << "    c: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][2] << endl;
            cout << "    d: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][3] << endl;
            cout << "    e: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][4] << endl;
            cout << "    f: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][5] << endl;
            cout << "    g: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][6] << endl;
            cout << "    h: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][7] << endl;

            // Print message schedule
            cout << "  Message schedule:" << endl;
            for (int i = 0; i < 80; i++) {
                cout << "    W[" << setw(2) << setfill(' ') << dec << i
<< "]: "
                     << hex << setw(16) << setfill('0') <<
intermediateResults[block][8 + i] << endl;
            }
```

```
            // Print compression function rounds
            cout << "  Compression function rounds:" << endl;
            for (int round = 0; round < 80; round++) {
                cout << "    Round " << setw(2) << setfill(' ') << dec <<
round << ":" << endl;
                cout << "      a: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][88 + round * 8 + 0] << endl;
                cout << "      b: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][88 + round * 8 + 1] << endl;
                cout << "      c: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][88 + round * 8 + 2] << endl;
                cout << "      d: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][88 + round * 8 + 3] << endl;
                cout << "      e: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][88 + round * 8 + 4] << endl;
                cout << "      f: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][88 + round * 8 + 5] << endl;
                cout << "      g: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][88 + round * 8 + 6] << endl;
                cout << "      h: " << hex << setw(16) << setfill('0') <<
intermediateResults[block][88 + round * 8 + 7] << endl;
            }
        }
    }
};

// MD5 Implementation
class MD5 {
private:
    uint32 a0, b0, c0, d0; // Initial hash values
    vector<uint8> message; // Message after padding
    vector<vector<uint32>> intermediateResults;

    void padMessage(const string& input) {
        // Convert input string to bytes
        for (char c : input) {
            message.push_back(static_cast<uint8>(c));
        }

        size_t originalLength = message.size();
        size_t originalLengthBits = originalLength * 8;

        // Append the bit '1' to the message
```

```
        message.push_back(0x80);

        // Append 0 ≤ k < 512 bits '0', so that the resulting message
length (in bits)
        // is congruent to 448 (mod 512)
        while ((message.size() * 8) % 512 != 448) {
            message.push_back(0);
        }

        // Append the length of the original message (before padding)
as a 64-bit little-endian integer
        for (int i = 0; i < 8; i++) {
            message.push_back((originalLengthBits >> (i * 8)) & 0xFF);
        }
    }

    void processBlock(size_t block) {
        vector<uint32> stateValues;
        uint32 a = a0, b = b0, c = c0, d = d0;
        uint32 M[16];

        // Save initial state
        stateValues.push_back(a);
        stateValues.push_back(b);
        stateValues.push_back(c);
        stateValues.push_back(d);

        // Break chunk into sixteen 32-bit words
        for (int i = 0; i < 16; i++) {
            M[i] = message[block * 64 + i * 4] |
                  (message[block * 64 + i * 4 + 1] << 8) |
                  (message[block * 64 + i * 4 + 2] << 16) |
                  (message[block * 64 + i * 4 + 3] << 24);
            stateValues.push_back(M[i]); // Save message words
        }

        // Main loop
        for (int i = 0; i < 64; i++) {
            uint32 F_val, g;

            if (i < 16) {
                F_val = F(b, c, d);
                g = i;
            } else if (i < 32) {
```

```
                F_val = G(b, c, d);
                g = (5 * i + 1) % 16;
            } else if (i < 48) {
                F_val = H(b, c, d);
                g = (3 * i + 5) % 16;
            } else {
                F_val = I(b, c, d);
                g = (7 * i) % 16;
            }

            uint32 temp = d;
            d = c;
            c = b;
            b = b + ROTL((a + F_val + MD5_K[i] + M[g]), MD5_S[i]);
            a = temp;

            // Save intermediate values
            stateValues.push_back(a);
            stateValues.push_back(b);
            stateValues.push_back(c);
            stateValues.push_back(d);
        }

        // Save the intermediate results
        intermediateResults.push_back(stateValues);

        // Add the chunk's hash to the result
        a0 += a;
        b0 += b;
        c0 += c;
        d0 += d;
    }

public:
    MD5() {
        // Initialize variables (in little-endian)
        a0 = 0x67452301;
        b0 = 0xefcdab89;
        c0 = 0x98badcfe;
        d0 = 0x10325476;
    }

    void reset() {
        // Reset the hash values to initial state
```

```
        a0 = 0x67452301;
        b0 = 0xefcdab89;
        c0 = 0x98badcfe;
        d0 = 0x10325476;

        message.clear();
        intermediateResults.clear();
    }

    string hash(const string& input) {
        reset();
        padMessage(input);

        // Process the message in 512-bit blocks
        for (size_t i = 0; i < message.size() / 64; i++) {
            processBlock(i);
        }

        // Convert the hash values to a string (in little-endian)
        uint8 digest[16];
        digest[0] = a0 & 0xFF;
        digest[1] = (a0 >> 8) & 0xFF;
        digest[2] = (a0 >> 16) & 0xFF;
        digest[3] = (a0 >> 24) & 0xFF;
        digest[4] = b0 & 0xFF;
        digest[5] = (b0 >> 8) & 0xFF;
        digest[6] = (b0 >> 16) & 0xFF;
        digest[7] = (b0 >> 24) & 0xFF;
        digest[8] = c0 & 0xFF;
        digest[9] = (c0 >> 8) & 0xFF;
        digest[10] = (c0 >> 16) & 0xFF;
        digest[11] = (c0 >> 24) & 0xFF;
        digest[12] = d0 & 0xFF;
        digest[13] = (d0 >> 8) & 0xFF;
        digest[14] = (d0 >> 16) & 0xFF;
        digest[15] = (d0 >> 24) & 0xFF;

        return bytesToHexString(digest, 16);
    }

    void printIntermediateResults() {
        cout << "Intermediate results for MD5:" << endl;
        for (size_t block = 0; block < intermediateResults.size();
block++) {
```

```
        cout << "Block " << block + 1 << ":" << endl;

        // Print initial state
        cout << "  Initial state:" << endl;
        cout << "    A: " << hex << setw(8) << setfill('0') <<
intermediateResults[block][0] << endl;
        cout << "    B: " << hex << setw(8) << setfill('0') <<
intermediateResults[block][1] << endl;
        cout << "    C: " << hex << setw(8) << setfill('0') <<
intermediateResults[block][2] << endl;
        cout << "    D: " << hex << setw(8) << setfill('0') <<
intermediateResults[block][3] << endl;

        // Print message words
        cout << "  Message words:" << endl;
        for (int i = 0; i < 16; i++) {
            cout << "    M[" << setw(2) << setfill(' ') << dec << i
<< "]: "
                 << hex << setw(8) << setfill('0') <<
intermediateResults[block][4 + i] << endl;
        }

        // Print rounds
        cout << "  Rounds:" << endl;
        for (int round = 0; round < 64; round++) {
            cout << "    Round " << setw(2) << setfill(' ') << dec <<
round << ":" << endl;
            cout << "      A: " << hex << setw(8) << setfill('0') <<
intermediateResults[block][20 + round * 4 + 0] << endl;
            cout << "      B: " << hex << setw(8) << setfill('0') <<
intermediateResults[block][20 + round * 4 + 1] << endl;
            cout << "      C: " << hex << setw(8) << setfill('0') <<
intermediateResults[block][20 + round * 4 + 2] << endl;
            cout << "      D: " << hex << setw(8) << setfill('0') <<
intermediateResults[block][20 + round * 4 + 3] << endl;
        }
    }
  }
};
```

```
// Simple implementation of Digital Signature Standard (DSS)
// This is a simplified simulation of DSS using a smaller prime
class DSS {
    private:
        // Small prime numbers for demonstration
        uint64 p; // A prime modulus
        uint64 q; // A prime divisor of p-1
        uint64 g; // A generator of order q in the multiplicative group of
integers modulo p
        uint64 x; // Private key
        uint64 y; // Public key = g^x mod p

        // Modular exponentiation (unchanged)
        uint64 modExp(uint64 base, uint64 exponent, uint64 modulus)
{
            uint64 result = 1;
            base = base % modulus;

            while (exponent > 0) {
                if (exponent % 2 == 1) {
                    result = (result * base) % modulus;
                }

                exponent = exponent >> 1;
                base = (base * base) % modulus;
            }

            return result;
        }

        // Modular inverse (unchanged)
        uint64 modInverse(uint64 a, uint64 m) {
            int64_t m0 = m;
            int64_t a0 = a;
            int64_t y = 0, x = 1;

            if (m == 1) return 0;

            while (a0 > 1) {
                int64_t q = a0 / m0;
                int64_t t = m0;

                m0 = a0 % m0;
                a0 = t;
```

```
            t = y;

            y = x - q * y;
            x = t;
        }

        if (x < 0) x += m;

        return x;
    }

    // Hash function (modified for more reliable hashing)
    uint64 hash(const string& message) {
        // For demonstration, use a simple consistent hash
        uint64 h = 0;
        for (char c : message) {
            h = (h * 31 + c) % q;
        }
        return h != 0 ? h : 1; // Ensure h is not zero
    }

public:
    DSS() {
        // More carefully chosen parameters
        p = 23;        // A prime
        q = 11;        // A prime dividing p-1
        g = 2;         // A generator

        // Generate a private key
        x = 3;         // Private key (normally random)

        // Compute the public key
        y = modExp(g, x, p);
    }

    void generateKeys() {
        cout << "Generating DSS keys (using small primes for
demonstration)..." << endl;
        cout << "Domain parameters:" << endl;
        cout << "  p = " << p << " (prime modulus)" << endl;
        cout << "  q = " << q << " (prime divisor of p-1)" << endl;
        cout << "  g = " << g << " (generator)" << endl;

        // Generate a new private key
```

```
        x = 3;
        cout << "Private key x = " << x << endl;

        // Compute the public key
        y = modExp(g, x, p);
        cout << "Public key y = " << y << " (= g^x mod p)" <<
endl;
    }

    pair<uint64, uint64> sign(const string& message) {
        // Key changes in signing process
        uint64 k = 7;  // Careful selection of k

        // Ensure 1 < k < q
        if (k <= 1 || k >= q) {
            throw runtime_error("Invalid k value");
        }

        // Calculate r = (g^k mod p) mod q
        uint64 r = modExp(g, k, p) % q;

        // Calculate s = k^-1 * (hash(message) + x*r) mod q
        uint64 hm = hash(message);
        uint64 kInv = modInverse(k, q);
        uint64 s = (kInv * (hm + x * r)) % q;

        // Ensure s is not zero
        if (s == 0) {
            throw runtime_error("Signature calculation resulted in s =
0");
        }

        return make_pair(r, s);
    }

    bool verify(const string& message, uint64 r, uint64 s) {
        // Verify signature parameters
        if (r <= 0 || r >= q || s <= 0 || s >= q) {
            cout << "Invalid r or s values" << endl;
            return false;
        }

        // Calculate w = s^-1 mod q
        uint64 w = modInverse(s, q);
```

```
        // Calculate u1 = hash(message) * w mod q
        uint64 hm = hash(message);
        uint64 u1 = (hm * w) % q;

        // Calculate u2 = r * w mod q
        uint64 u2 = (r * w) % q;

        // Calculate v = ((g^u1 * y^u2) mod p) mod q
        uint64 v1 = modExp(g, u1, p);
        uint64 v2 = modExp(y, u2, p);
        uint64 v = ((v1 * v2) % p) % q;

        // Debug print statements
        cout << "Verification details:" << endl;
        cout << "  Message hash h(m) = " << hm << endl;
        cout << "  w = s^-1 mod q = " << w << endl;
        cout << "  u1 = h(m) * w mod q = " << u1 << endl;
        cout << "  u2 = r * w mod q = " << u2 << endl;
        cout << "  v = ((g^u1 * y^u2) mod p) mod q = " << v <<
endl;
        cout << "  r (expected) = " << r << endl;

        // Verify signature
        return v == r;
    }

    void simulateSignAndVerify(const string& message) {
        cout << "Digital Signature Standard (DSS) Simulation" <<
endl;
        cout <<
"===================================" <<
endl;
        cout << "Message: \"" << message << "\"" << endl;

        try {
            // Step 1: Generate keys
            generateKeys();

            // Step 2: Sign the message
            cout << "\nSigning process:" << endl;
            auto signature = sign(message);
            uint64 r = signature.first;
            uint64 s = signature.second;
```

```
            cout << "  Signature components:" << endl;
            cout << "    r = " << r << endl;
            cout << "    s = " << s << endl;

            // Step 3: Verify the signature
            cout << "\nVerification process:" << endl;
            bool verified = verify(message, r, s);

            cout << "\nVerification result: " << (verified ? "VALID
signature" : "INVALID signature") << endl;

            // Demonstrate invalid signature
            cout << "\nDemonstrating an invalid signature:" << endl;
            bool invalidVerified = verify(message + "tampered", r, s);
            cout << "  Verification result for tampered message: " <<
(invalidVerified ? "VALID signature" : "INVALID signature") << endl;
        }
        catch (const exception& e) {
            cout << "Error: " << e.what() << endl;
        }
    }
};

// Main application
void printMenu() {
    cout << "\n==== Asymmetric Cryptography Menu ====" <<
endl;
    cout << "1. Calculate SHA-512 hash" << endl;
    cout << "2. Calculate MD5 hash" << endl;
    cout << "3. Simulate Digital Signature Standard (DSS)" << endl;
    cout << "4. Run test cases" << endl;
    cout << "5. Exit" << endl;
    cout << "Enter your choice: ";
}

void runTestCases() {
    cout << "\n==== Running Test Cases ====" << endl;

    // Test vectors for SHA-512
    cout << "\nSHA-512 Test Cases:" << endl;
    vector<pair<string, string>> sha512Tests = {
        {"",
"cf83e1357eefb8bdf1542850d66d8007d620e4050b5715dc83f4a92
```

```
1d36ce9ce47d0d13c5d85f2b0ff8318d2877eec2f63b931bd47417a8
1a538327af927da3e"},
        {"abc",
"ddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee
64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce8
0e2a9ac94fa54ca49f"},
        {"The quick brown fox jumps over the lazy dog",
"07e547d9586f6a73f73fbac0435ed76951218fb7d0c8d788a309d78
5436bbb642e93a252a954f23912547d1e8a3b5ed6e1bfd709782123
3fa0538f3db854fee6"}
    };

    SHA512 sha512;
    for (const auto& test : sha512Tests) {
        string input = test.first;
        string expectedOutput = test.second;
        string actualOutput = sha512.hash(input);

        cout << "Input: \"" << input << "\"" << endl;
        cout << "Expected: " << expectedOutput << endl;
        cout << "Actual:   " << actualOutput << endl;
        cout << "Result:   " << (expectedOutput == actualOutput ?
"PASS" : "FAIL") << endl << endl;
    }

    // Test vectors for MD5
    cout << "MD5 Test Cases:" << endl;
    vector<pair<string, string>> md5Tests = {
        {"", "d41d8cd98f00b204e9800998ecf8427e"},
        {"abc", "900150983cd24fb0d6963f7d28e17f72"},
        {"The quick brown fox jumps over the lazy dog",
"9e107d9d372bb6826bd81d3542a419d6"}
    };

    MD5 md5;
    for (const auto& test : md5Tests) {
        string input = test.first;
        string expectedOutput = test.second;
        string actualOutput = md5.hash(input);

        cout << "Input: \"" << input << "\"" << endl;
        cout << "Expected: " << expectedOutput << endl;
        cout << "Actual:   " << actualOutput << endl;
```

```
        cout << "Result:   " << (expectedOutput == actualOutput ?
"PASS" : "FAIL") << endl << endl;
    }

    // Test DSS with a simple message
    cout << "DSS Test Case:" << endl;
    DSS dss;
    string message = "Test message for DSS";
    dss.simulateSignAndVerify(message);
}

void runSHA512() {
    string input;
    cout << "\nEnter a message to hash with SHA-512: ";
    cin.ignore();
    getline(cin, input);

    SHA512 sha512;
    string hash = sha512.hash(input);

    cout << "\nSHA-512 hash: " << hash << endl;

    // Print intermediate results
    char showIntermediate;
    cout << "\nDo you want to see intermediate results? (y/n): ";
    cin >> showIntermediate;

    if (showIntermediate == 'y' || showIntermediate == 'Y') {
        sha512.printIntermediateResults();
    }
}

void runMD5() {
    string input;
    cout << "\nEnter a message to hash with MD5: ";
    cin.ignore();
    getline(cin, input);

    MD5 md5;
    string hash = md5.hash(input);

    cout << "\nMD5 hash: " << hash << endl;

    // Print intermediate results
```

```
    char showIntermediate;
    cout << "\nDo you want to see intermediate results? (y/n): ";
    cin >> showIntermediate;

    if (showIntermediate == 'y' || showIntermediate == 'Y') {
        md5.printIntermediateResults();
    }
}

void runDSS() {
    string input;
    cout << "\nEnter a message to sign with DSS: ";
    cin.ignore();
    getline(cin, input);

    DSS dss;
    dss.simulateSignAndVerify(input);
}

int main() {
    int choice;
    bool running = true;

    cout << "Asymmetric Cryptography Simulation" << endl;
    cout <<
"==================================" << endl;

    while (running) {
        printMenu();
        cin >> choice;

        switch (choice) {
            case 1:
                runSHA512();
                break;
            case 2:
                runMD5();
                break;
            case 3:
                runDSS();
                break;
            case 4:
                runTestCases();
                break;
```

```
            case 5:
                running = false;
                cout << "\nExiting program. Goodbye!" << endl;
                break;
            default:
                cout << "\nInvalid choice. Please try again." << endl;
        }
    }

    return 0;
}
```

Asymmetric Algorithms Test Cases Output:

SHA-512 Test Cases

Normal Input Test Cases

1. Standard ASCII string
    - Input: "Hello, World!"
    - Purpose: Validate hash generation for typical text
2. Alphanumeric string
    - Input: "OpenAI ChatGPT 2024"
    - Purpose: Test hash generation with mixed character types
3. Sentence with punctuation
    - Input: "Cryptography is fascinating!"
    - Purpose: Verify handling of punctuation marks

1.

```
Asymmetric Cryptography Simulation
==================================

==== Asymmetric Cryptography Menu ====
1. Calculate SHA-512 hash
2. Calculate MD5 hash
3. Simulate Digital Signature Standard (DSS)
4. Run test cases
5. Exit
Enter your choice: 1

Enter a message to hash with SHA-512: Hello, World!

SHA-512 hash: 374d794a95cdcfd8b35993185fef9ba368f160d8daf432d08ba9f1ed1e5abe6cc69291e0fa2fe0006a52570ef18c19def4e617c33ce52ef0a6e5fbe318cb0387

Do you want to see intermediate results? (y/n): y
```

```
Do you want to see intermediate results? (y/n): y
Intermediate results for SHA-512:
Block 1:
  Initial state:
    a: 6a09e667f3bcc908
    b: bb67ae8584caa73b
    c: 3c6ef372fe94f82b
    d: a54ff53a5f1d36f1
    e: 510e527fade682d1
    f: 9b05688c2b3e6c1f
    g: 1f83d9abfb41bd6b
    h: 5be0cd19137e2179
  Message schedule:
    W[ 0]: 48656c6c6f2c2057
    W[ 1]: 6f726c6421800000
    W[ 2]: 0000000000000000
    W[ 3]: 0000000000000000
    W[ 4]: 0000000000000000
    W[ 5]: 0000000000000000
    W[ 6]: 0000000000000000
    W[ 7]: 0000000000000000
    W[ 8]: 0000000000000000
    W[ 9]: 0000000000000000
    W[10]: 0000000000000000
    W[11]: 0000000000000000
    W[12]: 0000000000000000
    W[13]: 0000000000000000
    W[14]: 0000000000000000
    W[15]: 0000000000000068
    W[16]: 7f6e0cf32bcea057
    W[17]: 6f7f6c6421800341
    W[18]: 2e87304753445d43
    W[19]: 7a2eb37f710a9e36
    W[20]: ff2bfb2b61671104
    W[21]: 835bee73a3ff35ea
    W[22]: 1883e952d9d878b7
    W[23]: 7ddd5942d886c853
    W[24]: 3aeb12881a0f02a2
    W[25]: 649f010fb7c05fed
    W[26]: b21632d7034bc890
    W[27]: 2dc3d415c26319f0
    W[28]: 6ec7c6a52408c1b8
    W[29]: 261b9399186c7461
    W[30]: 558f812e248952cb
    W[31]: e1ea9d3a701b38aa
    W[32]: 618ea34e0d2ef969
    W[33]: 618bd048bfe295cd
    W[34]: 3ac23b93d3e878f7
    W[35]: c2ebfcf30a05b55f
    W[36]: a953dcf8566d3d37
    W[37]: b753906b47aabaf2
    W[38]: d774af731ecdf856
    W[39]: 8dc64cf65d25679d
    W[40]: 044b782155249e13
    W[41]: 29352f1a7e0a73cb
```

2.

```
==== Asymmetric Cryptography Menu ====
1. Calculate SHA-512 hash
2. Calculate MD5 hash
3. Simulate Digital Signature Standard (DSS)
4. Run test cases
5. Exit
Enter your choice: 1

Enter a message to hash with SHA-512: Karan 2004 La4

SHA-512 hash: eecaac43f676c665901981412dcb108f9738ca011d53ac20167da2351672b33ee1a7edf8f7e9be4d146c239acd21806a56690f1c66ac1b51f4dfea2a7b451de0

Do you want to see intermediate results? (y/n): y
Intermediate results for SHA-512:
Block 1:
  Initial state:
    a: 6a09e667f3bcc908
    b: bb67ae8584caa73b
    c: 3c6ef372fe94f82b
    d: a54ff53a5f1d36f1
    e: 510e527fade682d1
    f: 9b05688c2b3e6c1f
    g: 1f83d9abfb41bd6b
    h: 5be0cd19137e2179
  Message schedule:
    W[ 0]: 4b6172616e203230
    W[ 1]: 3034204c61348000
    W[ 2]: 0000000000000000
    W[ 3]: 0000000000000000
    W[ 4]: 0000000000000000
    W[ 5]: 0000000000000000
    W[ 6]: 0000000000000000
    W[ 7]: 0000000000000000
    W[ 8]: 0000000000000000
    W[ 9]: 0000000000000000
    W[10]: 0000000000000000
    W[11]: 0000000000000000
    W[12]: 0000000000000000
    W[13]: 0000000000000000
    W[14]: 0000000000000000
    W[15]: 0000000000000070
    W[16]: 63abbea852594fb0
    W[17]: 3042204c61348381
    W[18]: 352557cd445612f6
    W[19]: 12a02cea7c294221
    W[20]: 6ba0ed91bd586770
    W[21]: bd0fc5b44d27fb85
    W[22]: 5047e24fb1846e25
    W[23]: 79a663bd93171a17
    W[24]: 3efcc848afe11a23
    W[25]: 64bcc323e6d951a7
    W[26]: e6fa03a640e782ae
    W[27]: 79e0f415de8dfc16
    W[28]: 812def982b733ee6
    W[29]: c1aace926f1bdab7
```

3.

```
Asymmetric Cryptography Simulation
==================================

==== Asymmetric Cryptography Menu ====
1. Calculate SHA-512 hash
2. Calculate MD5 hash
3. Simulate Digital Signature Standard (DSS)
4. Run test cases
5. Exit
Enter your choice: 1

Enter a message to hash with SHA-512: Cryptography is fascinating!

SHA-512 hash: cd67cdf3cd95b46eef080d1eb2692092a7190ccc5f34953eef6efec644753ed1bd6a771afe3b84ae436e90a9c9ee7b61bf28b51d3c287e0ea9bf95b622ab9dcd

Do you want to see intermediate results? (y/n): n
```

Edge Case Test

Cases

1. Empty String

    - Input: ""
    - Purpose: Verify behavior with zero-length input

```
==== Asymmetric Cryptography Menu ====
1. Calculate SHA-512 hash
2. Calculate MD5 hash
3. Simulate Digital Signature Standard (DSS)
4. Run test cases
5. Exit
Enter your choice: 1

Enter a message to hash with SHA-512:

SHA-512 hash: cf83e1357eefb8bdf1542850d66d8007d620e4050b5715dc83f4a921d36ce9ce47d0d13c5d85f2b0ff8318d2877eec2f63b931bd47417a81a538327af927da3e

Do you want to see intermediate results? (y/n): n
```

2. Single Space

    - Input: " "
    - Purpose: Test hash generation with minimal whitespace

```
==== Asymmetric Cryptography Menu ====
1. Calculate SHA-512 hash
2. Calculate MD5 hash
3. Simulate Digital Signature Standard (DSS)
4. Run test cases
5. Exit
Enter your choice: 1

Enter a message to hash with SHA-512:

SHA-512 hash: f90ddd77e400dfe6a3fcf479b00b1ee29e7015c5bb8cd70f5f15b4886cc339275ff553fc8a053f8ddc7324f45168cffaf81f8c3ac93996f6536eef38e5e40768

Do you want to see intermediate results? (y/n): n
```

MD5 Test Cases

Normal Input Test Cases

1. Standard Text

    - Input: "Secure Hashing Algorithm"
    - Purpose: Basic hash generation validation

```
==== Asymmetric Cryptography Menu ====
1. Calculate SHA-512 hash
2. Calculate MD5 hash
3. Simulate Digital Signature Standard (DSS)
4. Run test cases
5. Exit
Enter your choice: 2

Enter a message to hash with MD5: Secure Hashing Algorithm

MD5 hash: 3857684adde65968b1858bcbba912497

Do you want to see intermediate results? (y/n): y
Intermediate results for MD5:
Block 1:
  Initial state:
    A: 67452301
    B: efcdab89
    C: 98badcfe
    D: 10325476
  Message words:
    M[ 0]: 75636553
    M[ 1]: 48206572
    M[ 2]: 69687361
    M[ 3]: 4120676e
    M[ 4]: 726f676c
    M[ 5]: 6d687469
    M[ 6]: 00000080
    M[ 7]: 00000000
    M[ 8]: 00000000
    M[ 9]: 00000000
    M[10]: 00000000
    M[11]: 00000000
    M[12]: 00000000
    M[13]: 00000000
    M[14]: 000000c0
    M[15]: 00000000
  Rounds:
    Round  0:
      A: 10325476
      B: 56d290af
      C: efcdab89
      D: 98badcfe
    Round  1:
      A: 98badcfe
      B: 8ab401af
      C: 56d290af
      D: efcdab89
    Round  2:
      A: efcdab89
      B: 62871de9
      C: 8ab401af
      D: 56d290af
```

```
    D: 81fc89d1
  Round 53:
    A: 81fc89d1
    B: 2dbe0ab1
    C: e2127fe6
    D: 2d83e202
  Round 54:
    A: 2d83e202
    B: 9cf2da9d
    C: 2dbe0ab1
    D: e2127fe6
  Round 55:
    A: e2127fe6
    B: 4b2847eb
    C: 9cf2da9d
    D: 2dbe0ab1
  Round 56:
    A: 2dbe0ab1
    B: a0b331d1
    C: 4b2847eb
    D: 9cf2da9d
  Round 57:
    A: 9cf2da9d
    B: aa41d923
    C: a0b331d1
    D: 4b2847eb
  Round 58:
    A: 4b2847eb
    B: 1dcd884f
    C: aa41d923
    D: a0b331d1
  Round 59:
    A: a0b331d1
    B: 38df5fdd
    C: 1dcd884f
    D: aa41d923
  Round 60:
    A: aa41d923
    B: e3233437
    C: 38df5fdd
    D: 1dcd884f
  Round 61:
    A: 1dcd884f
    B: 86f23d44
    C: e3233437
    D: 38df5fdd
  Round 62:
    A: 38df5fdd
    B: 32d0a8b3
    C: 86f23d44
    D: e3233437
  Round 63:
    A: e3233437
    B: 788c3b54
    C: 32d0a8b3
    D: 86f23d44
```

2. Alphanumeric String

- Input: "MD5 Hash Test 2024"
- Purpose: Test mixed character type handling

```
==== Asymmetric Cryptography Menu ====
1. Calculate SHA-512 hash
2. Calculate MD5 hash
3. Simulate Digital Signature Standard (DSS)
4. Run test cases
5. Exit
Enter your choice: 2

Enter a message to hash with MD5: MD5 Hash Test 2025

MD5 hash: 441ab56fa1e5fbc3df4bc227c0c80902

Do you want to see intermediate results? (y/n): y
Intermediate results for MD5:
Block 1:
  Initial state:
    A: 67452301
    B: efcdab89
    C: 98badcfe
    D: 10325476
  Message words:
    M[ 0]: 2035444d
    M[ 1]: 68736148
    M[ 2]: 73655420
    M[ 3]: 30322074
    M[ 4]: 00803532
    M[ 5]: 00000000
    M[ 6]: 00000000
    M[ 7]: 00000000
    M[ 8]: 00000000
    M[ 9]: 00000000
    M[10]: 00000000
    M[11]: 00000000
    M[12]: 00000000
    M[13]: 00000000
    M[14]: 00000090
    M[15]: 00000000
  Rounds:
    Round  0:
      A: 10325476
      B: bfc20e04
      C: efcdab89
      D: 98badcfe
    Round  1:
      A: 98badcfe
      B: 2442ef1a
      C: bfc20e04
      D: efcdab89
    Round  2:
      A: efcdab89
      B: 85372f39
      C: 2442ef1a
      D: bfc20e04
    Round  3:
      A: bfc20e04
      B: 06ff4f2b
```

3. Technical String

- Input: "Cryptographic Hash Function"
- Purpose: Verify hash generation for technical terminology

```
==== Asymmetric Cryptography Menu ====
1. Calculate SHA-512 hash
2. Calculate MD5 hash
3. Simulate Digital Signature Standard (DSS)
4. Run test cases
5. Exit
Enter your choice: 2

Enter a message to hash with MD5: Cryptographic Hash Function

MD5 hash: a99a923eaf82f3695f8a2115317ae31e

Do you want to see intermediate results? (y/n): y
Intermediate results for MD5:
Block 1:
  Initial state:
    A: 67452301
    B: efcdab89
    C: 98badcfe
    D: 10325476
  Message words:
    M[ 0]: 70797243
    M[ 1]: 72676f74
    M[ 2]: 69687061
    M[ 3]: 61482063
    M[ 4]: 46206873
    M[ 5]: 74636e75
    M[ 6]: 806e6f69
    M[ 7]: 00000000
    M[ 8]: 00000000
    M[ 9]: 00000000
    M[10]: 00000000
    M[11]: 00000000
    M[12]: 00000000
    M[13]: 00000000
    M[14]: 000000d8
    M[15]: 00000000
  Rounds:
    Round  0:
      A: 10325476
      B: e1d908ac
      C: efcdab89
      D: 98badcfe
    Round  1:
      A: 98badcfe
      B: b75aaf00
      C: e1d908ac
      D: efcdab89
    Round  2:
      A: efcdab89
      B: 44e0cf41
      C: b75aaf00
      D: e1d908ac
    Round  3:
      A: e1d908ac
      B: e68eda8b
```

Digital Signature Test Cases

1.Standard Message

```
==== Asymmetric Cryptography Menu ====
1. Calculate SHA-512 hash
2. Calculate MD5 hash
3. Simulate Digital Signature Standard (DSS)
4. Run test cases
5. Exit
Enter your choice: 3

Enter a message to sign with DSS: Karan
Digital Signature Standard (DSS) Simulation
====================================
Message: "Karan"
Generating DSS keys (using small primes for demonstration)...
Domain parameters:
  p = 17 (prime modulus)
  q = b (prime divisor of p-1)
  g = 2 (generator)
Private key x = 3
Public key y = 8 (= g^x mod p)

Signing process:
  Signature components:
    r = 2
    s = 3

Verification process:
Verification details:
  Message hash h(m) = 4
  w = s^-1 mod q = 4
  u1 = h(m) * w mod q = 5
  u2 = r * w mod q = 8
  v = ((g^u1 * y^u2) mod p) mod q = 2
  r (expected) = 2

Verification result: VALID signature

Demonstrating an invalid signature:
Verification details:
  Message hash h(m) = 2
  w = s^-1 mod q = 4
  u1 = h(m) * w mod q = 8
  u2 = r * w mod q = 8
  v = ((g^u1 * y^u2) mod p) mod q = 1
  r (expected) = 2
  Verification result for tampered message: INVALID signature
```

2. Technical Communication

```
==== Asymmetric Cryptography Menu ====
1. Calculate SHA-512 hash
2. Calculate MD5 hash
3. Simulate Digital Signature Standard (DSS)
4. Run test cases
5. Exit
Enter your choice: 3

Enter a message to sign with DSS: Cryptographic Authentication Prot
Digital Signature Standard (DSS) Simulation
====================================
Message: "Cryptographic Authentication Protocol"
Generating DSS keys (using small primes for demonstration)...
Domain parameters:
  p = 17 (prime modulus)
  q = b (prime divisor of p-1)
  g = 2 (generator)
Private key x = 3
Public key y = 8 (= g^x mod p)

Signing process:
  Signature components:
    r = 2
    s = 1

Verification process:
Verification details:
  Message hash h(m) = 1
  w = s^-1 mod q = 1
  u1 = h(m) * w mod q = 1
  u2 = r * w mod q = 2
  v = ((g^u1 * y^u2) mod p) mod q = 2
  r (expected) = 2

Verification result: VALID signature

Demonstrating an invalid signature:
Verification details:
  Message hash h(m) = 4
  w = s^-1 mod q = 1
  u1 = h(m) * w mod q = 4
  u2 = r * w mod q = 2
  v = ((g^u1 * y^u2) mod p) mod q = 1
  r (expected) = 2
  Verification result for tampered message: INVALID signature
```

Running Some Test Cases :

```
==== Asymmetric Cryptography Menu ====
1. Calculate SHA-512 hash
2. Calculate MD5 hash
3. Simulate Digital Signature Standard (DSS)
4. Run test cases
5. Exit
Enter your choice: 4

==== Running Test Cases ====

SHA-512 Test Cases:
Input: ""
Expected: cf83e1357eefb8bdf1542850d66d8007d620e4050b5715dc83f4a921d36ce9ce47d0d13c5d85f2b0ff8318d2877eec2f63b931bd47417a81a538327af927da3e
Actual:   cf83e1357eefb8bdf1542850d66d8007d620e4050b5715dc83f4a921d36ce9ce47d0d13c5d85f2b0ff8318d2877eec2f63b931bd47417a81a538327af927da3e
Result:   PASS

Input: "abc"
Expected: ddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f
Actual:   ddaf35a193617abacc417349ae20413112e6fa4e89a97ea20a9eeee64b55d39a2192992a274fc1a836ba3c23a3feebbd454d4423643ce80e2a9ac94fa54ca49f
Result:   PASS

Input: "The quick brown fox jumps over the lazy dog"
Expected: 07e547d9586f6a73f73fbac0435ed76951218fb7d0c8d788a309d785436bbb642e93a252a954f23912547d1e8a3b5ed6e1bfd7097821233fa0538f3db854fee6
Actual:   07e547d9586f6a73f73fbac0435ed76951218fb7d0c8d788a309d785436bbb642e93a252a954f23912547d1e8a3b5ed6e1bfd7097821233fa0538f3db854fee6
Result:   PASS

MD5 Test Cases:
Input: ""
Expected: d41d8cd98f00b204e9800998ecf8427e
Actual:   d41d8cd98f00b204e9800998ecf8427e
Result:   PASS

Input: "abc"
Expected: 900150983cd24fb0d6963f7d28e17f72
Actual:   900150983cd24fb0d6963f7d28e17f72
Result:   PASS

Input: "The quick brown fox jumps over the lazy dog"
Expected: 9e107d9d372bb6826bd81d3542a419d6
Actual:   9e107d9d372bb6826bd81d3542a419d6
Result:   PASS

DSS Test Case:
Digital Signature Standard (DSS) Simulation
====================================
Message: "Test message for DSS"
Generating DSS keys (using small primes for demonstration)...
Domain parameters:
  p = 17 (prime modulus)
  q = b (prime divisor of p-1)
  g = 2 (generator)
Private key x = 3
Public key y = 8 (= g^x mod p)
```