DELHI TECHNOLOGICAL UNIVERSITY

CS305 –: Information Network & Security Lab File

Submitted by: **Submitted to:**

YUG BATHLA
23/CS/479
CSE1 (A1) (G3)

INDEX

EXPERIMENT – 1

Aim: To implement Caesar cipher encryption.
Encryption: Replace each plaintext letter with one a Fixed number of places down the alphabet.
Decryption: Replace each cipher text letter with one a fixed number of places up the alphabet.

Theory:

The **Caesar cipher** is one of the simplest and oldest **encryption techniques**, named after Julius Caesar, who used it in his private correspondence. It is a **substitution cipher** in which each letter in the plaintext is shifted a fixed number of positions down or up the alphabet.

1. **Encryption:**
   - Each letter of the plaintext is replaced by a letter located **a fixed number of positions down the alphabet**.
   - The fixed number is called the **key** or **shift**.
   - For example, with a shift of 3:
   - `Plaintext:  A B C D E ...`
   - `Ciphertext: D E F G H ...`

   Encryption Formula:

   ```
   C = (P + K) mod 26
   ```

   - `C` = Ciphertext letter (numeric value 0–25)
   - `P` = Plaintext letter (numeric value 0–25)
   - `K` = Key (shift value)
2. **Decryption:**
   - To retrieve the original text, each letter in the ciphertext is shifted **up by the same key value**.
   - **Decryption Formula:** `P = (C - K + 26) mod 26`
   - `P` = Original plaintext letter
   - `C` = Ciphertext letter
   - `K` = Key (shift value)

Characteristics

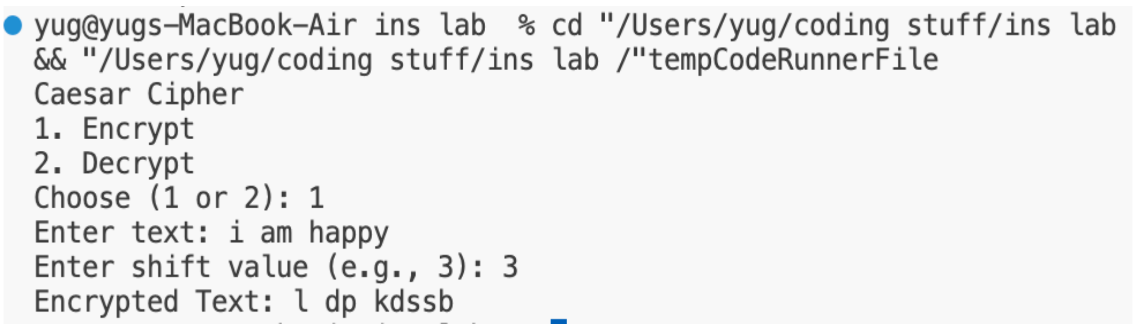
- **Symmetric Cipher:**
  The same key is used for encryption and decryption.
- **Alphabetic Substitution:**
  Only letters are substituted; spaces and punctuation may remain unchanged.
- **Fixed Shift:**
  All letters are shifted by the same amount.
-

Source code:

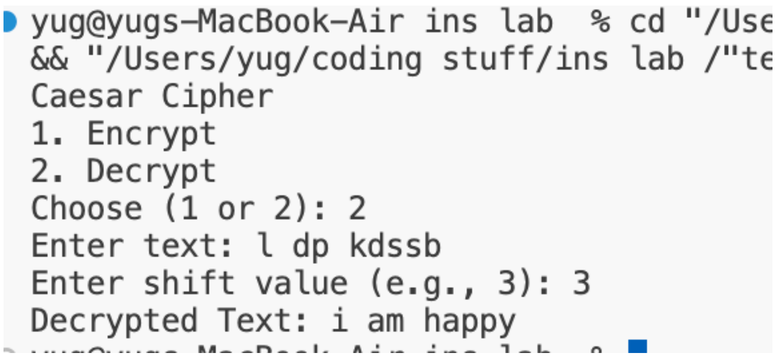
```
#include <iostream>
#include <string>
```

```cpp
using namespace std;

// Encrypt a message using Caesar Cipher
string encrypt(string text, int shift) {
    string result = "";

    for (char c : text) {
        if (isupper(c)) {
            result += char((c - 'A' + shift) % 26 + 'A');
        } else if (islower(c)) {
            result += char((c - 'a' + shift) % 26 + 'a');
        } else {
            result += c; // leave non-alphabet characters unchanged
        }
    }

    return result;
}

// Decrypt a message using Caesar Cipher
string decrypt(string text, int shift) {
    string result = "";

    for (char c : text) {
        if (isupper(c)) {
            result += char((c - 'A' - shift + 26) % 26 + 'A');
        } else if (islower(c)) {
            result += char((c - 'a' - shift + 26) % 26 + 'a');
        } else {
            result += c;
        }
    }

    return result;
}

int main() {
    string text;
    int shift, choice;

    cout << "Caesar Cipher\n";
    cout << "1. Encrypt\n2. Decrypt\nChoose (1 or 2): ";
    cin >> choice;
    cin.ignore(); // ignore newline character left in input buffer

    cout << "Enter text: ";
    getline(cin, text);

    cout << "Enter shift value (e.g., 3): ";
    cin >> shift;
    if (choice == 1) {
        cout << "Encrypted Text: " << encrypt(text, shift) << endl;
    } else if (choice == 2) {
        cout << "Decrypted Text: " << decrypt(text, shift) << endl;
    } else {
        cout << "Invalid choice.\n";
    }

    return 0;
}
```

Output:



```
yug@yugs-MacBook-Air ins lab  % cd "/Users/yug/coding stuff/ins lab
&& "/Users/yug/coding stuff/ins lab /"tempCodeRunnerFile
Caesar Cipher
1. Encrypt
2. Decrypt
Choose (1 or 2): 1
Enter text: i am happy
Enter shift value (e.g., 3): 3
Encrypted Text: l dp kdssb
```



```
yug@yugs-MacBook-Air ins lab  % cd "/Use
&& "/Users/yug/coding stuff/ins lab /"te
Caesar Cipher
1. Encrypt
2. Decrypt
Choose (1 or 2): 2
Enter text: l dp kdssb
Enter shift value (e.g., 3): 3
Decrypted Text: i am happy
```

EXPERIMENT – 2

Aim: To implement Monoalphabetic decryption. Encrypting and Decrypting works exactly the same for all monoalphabetic ciphers.
Encryption/Decryption: Every letter in the alphabet is represented by exactly one other letter in the key.

Theory:

A **Monoalphabetic Cipher** is a **substitution cipher** in which each letter of the plaintext is replaced by exactly **one unique letter** of the alphabet according to a **fixed key**. Unlike Caesar cipher, the shift doesn't have to be uniform; any mapping between plaintext and ciphertext letters is allowed.

Working Principle

1. **Encryption:**
   - Each plaintext letter is replaced by its corresponding letter in the key.
   - Example Key Mapping:
   - `Plain:  A B C D E F G H I J ...`
   - `Cipher: Q W E R T Y U I O P ...`
   - Plaintext "HELLO" → Ciphertext "ITSSG"
2. **Decryption:**
   - To decrypt, each ciphertext letter is replaced with its corresponding plaintext letter using the **reverse mapping**.

Characteristics

- **Fixed substitution:** Each letter has exactly **one corresponding ciphertext letter**.
- **Symmetric key:** Same key is required for encryption and decryption.
- **Alphabet-only substitution:** Typically, only letters are encrypted; numbers and punctuation remain unchanged.

Advantages

- Simple to implement and understand.
- Offers more variability than Caesar cipher.

Limitations

- Vulnerable to **frequency analysis**, as the mapping is static.
- Less secure for large texts without additional techniques.

Applications

- Used historically for confidential messages.
- Educational purposes for learning basic cryptography.

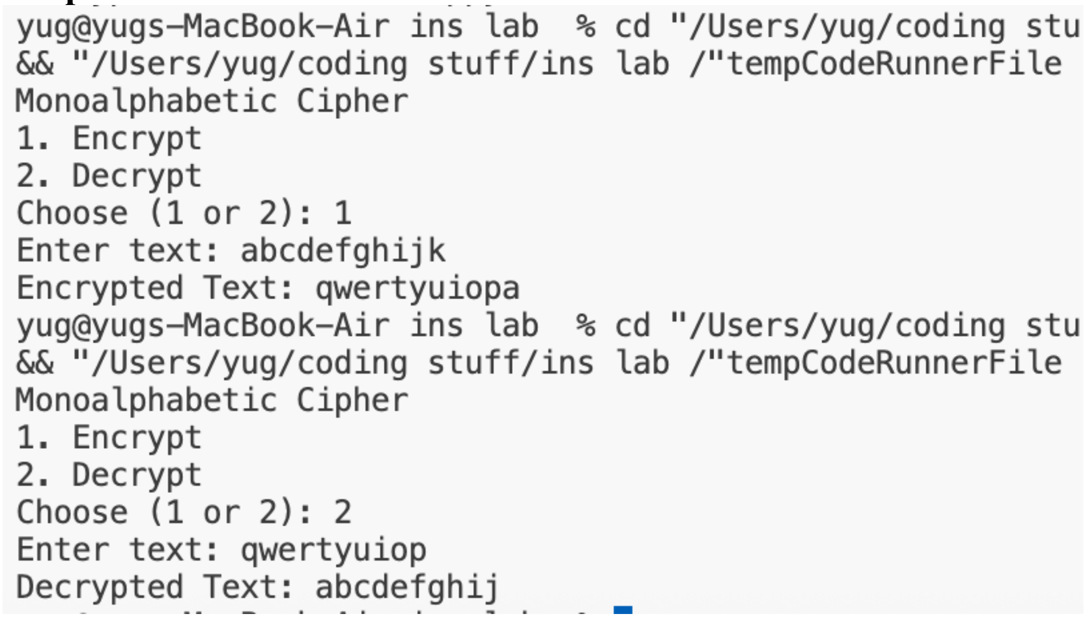
Source code:

```
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;

// Function to build mapping from standard to key alphabet
```

```cpp
unordered_map<char, char> buildMap(const string& from, const string& to) {
    unordered_map<char, char> map;
    for (int i = 0; i < 26; ++i) {
        map[from[i]] = to[i];
    }
    return map;
}

// Function to encrypt or decrypt text
string monoalphabeticCipher(const string& text, const unordered_map<char, char>&
map) {
    string result = "";
    for (char c : text) {
        if (isupper(c)) {
            result += toupper(map.at(tolower(c)));
        } else if (islower(c)) {
            result += map.at(c);
        } else {
            result += c; // Keep non-alphabetic characters unchanged
        }
    }
    return result;
}

int main() {
    string plainAlphabet = "abcdefghijklmnopqrstuvwxyz";

    // Monoalphabetic key (must be a permutation of 26 unique letters)
    string keyAlphabet =    "qwertyuiopasdfghjklzxcvbnm"; // key

    // Build encrypt and decrypt maps
    unordered_map<char, char> encryptMap = buildMap(plainAlphabet, keyAlphabet);
    unordered_map<char, char> decryptMap = buildMap(keyAlphabet, plainAlphabet);

    int choice;
    string input;

    cout << "Monoalphabetic Cipher\n";
    cout << "1. Encrypt\n2. Decrypt\nChoose (1 or 2): ";
    cin >> choice;
    cin.ignore(); // flush newline

    cout << "Enter text: ";
    getline(cin, input);

    if (choice == 1) {
        cout << "Encrypted Text: " << monoalphabeticCipher(input, encryptMap) <<
endl;
    } else if (choice == 2) {
        cout << "Decrypted Text: " << monoalphabeticCipher(input, decryptMap) <<
endl;
```

```
    } else {
        cout << "Invalid choice." << endl;
    }

    return 0;
}
```

Output:



```
yug@yugs-MacBook-Air ins lab  % cd "/Users/yug/coding stu
&& "/Users/yug/coding stuff/ins lab /"tempCodeRunnerFile
Monoalphabetic Cipher
1. Encrypt
2. Decrypt
Choose (1 or 2): 1
Enter text: abcdefghijk
Encrypted Text: qwertyuiopa
yug@yugs-MacBook-Air ins lab  % cd "/Users/yug/coding stu
&& "/Users/yug/coding stuff/ins lab /"tempCodeRunnerFile
Monoalphabetic Cipher
1. Encrypt
2. Decrypt
Choose (1 or 2): 2
Enter text: qwertyuiop
Decrypted Text: abcdefghij
```

EXPERIMENT – 3

Aim: To implement Play fair cipher encryption-decryption.

Theory:

The **Playfair Cipher** is a **digraph substitution cipher** invented by Charles Wheatstone (and promoted by Lord Playfair) in 1854.

- Instead of encrypting single letters, **two letters at a time (digraphs)** are encrypted, making it more secure than monoalphabetic ciphers.

Working Principle

1. **Key Table Creation**
    - A **5×5 matrix** is filled with letters of a key (replacing J with I), followed by the remaining letters of the alphabet in order.
    - Example:
    - `K E Y W O`
    - `R D A B C`
    - `F G H I L`
    - `M N P Q S`
    - `T U V X Z`
2. **Encryption Rules**
    - Break plaintext into digraphs (pairs of letters). If a digraph has identical letters, insert a filler (like X).
    - For each digraph:
        1. **Same row:** Replace each letter with the letter to its **right** (wrap around at end).
        2. **Same column:** Replace each letter with the letter **below** (wrap around at bottom).
        3. **Rectangle:** Replace each letter with the letter in the **same row but the column of the other letter**.
3. **Decryption Rules**
    - Reverse the encryption rules:
        1. **Same row:** Letter to **left**.
        2. **Same column:** Letter **above**.
        3. **Rectangle:** Swap columns as in encryption.

Characteristics

- Encrypts **pairs of letters (digraphs)** instead of single letters.
- Provides better **security** than monoalphabetic cipher.
- Uses a **5×5 key table**, combining I/J into a single letter.

Advantages

- Harder to break using simple frequency analysis.
- Encrypts more than one letter at a time, improving security.

Limitations

- Slightly complex compared to monoalphabetic cipher.
- Still vulnerable to **modern cryptanalysis**.

Applications

- Historically used for military communication.
- Good for **educational purposes** to demonstrate digraph encryption.

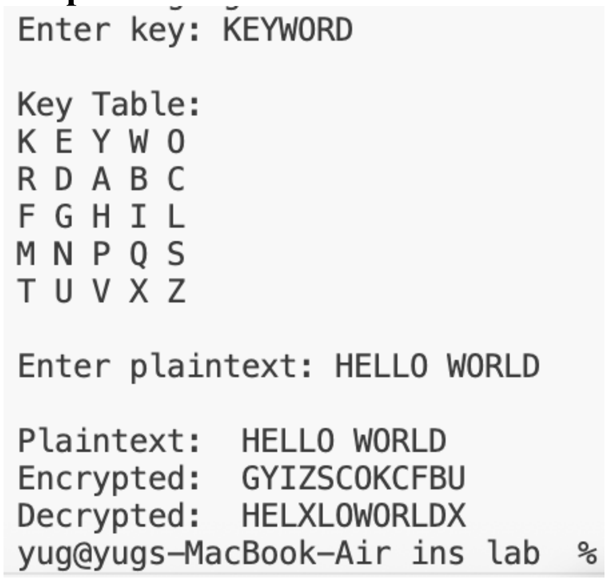
Source code:

```cpp
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <cctype>

using namespace std;

class PlayfairCipher {
    vector<vector<char>> keyTable;
    map<char, pair<int, int>> pos; // letter -> (row, col)

public:
    PlayfairCipher(string key) {
        createKeyTable(key);
    }

    void createKeyTable(string key) {
        vector<bool> used(26, false);
        keyTable.assign(5, vector<char>(5, ' '));
        string filteredKey;

        // Uppercase, replace J with I, remove duplicates
        for (char c : key) {
            c = toupper(static_cast<unsigned char>(c));
            if (c == 'J') c = 'I';
            if (c < 'A' || c > 'Z') continue;
            if (!used[c - 'A']) {
                used[c - 'A'] = true;
                filteredKey.push_back(c);
            }
        }

        // Add remaining letters
        for (char c = 'A'; c <= 'Z'; c++) {
            if (c == 'J') continue;
            if (!used[c - 'A']) {
                used[c - 'A'] = true;
                filteredKey.push_back(c);
            }
        }

        // Fill 5x5 table
        int idx = 0;
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                keyTable[i][j] = filteredKey[idx];
                pos[filteredKey[idx]] = {i, j};
                idx++;
            }
        }
    }

    string prepareText(string text, bool forEncryption) {
        string processed;
        for (char c : text) {
            c = toupper(static_cast<unsigned char>(c));
            if (c == 'J') c = 'I';
            if (c >= 'A' && c <= 'Z') processed.push_back(c);
        }

        if (forEncryption) {
            string result;
            for (size_t i = 0; i < processed.size(); i++) {
                result.push_back(processed[i]);
                if (i + 1 == processed.size()) {
                    result.push_back('X'); // padding
                } else if (processed[i] == processed[i + 1]) {
                    result.push_back('X');
                } else {
                    result.push_back(processed[i + 1]);
                    i++;
                }
            }
            return result;
        }
        return processed; // For decryption, no digraph processing needed
    }

    string encrypt(string plaintext) {
        string text = prepareText(plaintext, true);
        string cipher;
        for (size_t i = 0; i < text.size(); i += 2) {
            char a = text[i], b = text[i + 1];
            pair<int, int> p1 = pos[a];
            pair<int, int> p2 = pos[b];
            int r1 = p1.first, c1 = p1.second;
            int r2 = p2.first, c2 = p2.second;
```

```
            if (r1 == r2) { // Same row
                cipher.push_back(keyTable[r1][(c1 + 1) % 5]);
                cipher.push_back(keyTable[r2][(c2 + 1) % 5]);
            } else if (c1 == c2) { // Same column
                cipher.push_back(keyTable[(r1 + 1) % 5][c1]);
                cipher.push_back(keyTable[(r2 + 1) % 5][c2]);
            } else { // Rectangle
                cipher.push_back(keyTable[r1][c2]);
                cipher.push_back(keyTable[r2][c1]);
            }
        }
        return cipher;
    }

    string decrypt(string ciphertext) {
        string text = prepareText(ciphertext, false);
        string plain;
        for (size_t i = 0; i < text.size(); i += 2) {
            char a = text[i], b = text[i + 1];
            auto [r1, c1] = pos[a];
            auto [r2, c2] = pos[b];

            if (r1 == r2) { // Same row
                plain.push_back(keyTable[r1][(c1 + 4) % 5]);
                plain.push_back(keyTable[r2][(c2 + 4) % 5]);
            } else if (c1 == c2) { // Same column
                plain.push_back(keyTable[(r1 + 4) % 5][c1]);
                plain.push_back(keyTable[(r2 + 4) % 5][c2]);
            } else { // Rectangle
                plain.push_back(keyTable[r1][c2]);
                plain.push_back(keyTable[r2][c1]);
            }
        }
        return plain;
    }

    void printKeyTable() {
        for (auto &row : keyTable) {
            for (char c : row) cout << c << ' ';
            cout << "\n";
        }
    }
};

int main() {
    string key, plaintext;

    cout << "Enter key: ";
    getline(cin, key);

    PlayfairCipher cipher(key);
```

```cpp
    cout << "\nKey Table:\n";
    cipher.printKeyTable();

    cout << "\nEnter plaintext: ";
    getline(cin, plaintext);

    string encrypted = cipher.encrypt(plaintext);
    string decrypted = cipher.decrypt(encrypted);

    cout << "\nPlaintext:  " << plaintext;
    cout << "\nEncrypted:  " << encrypted;
    cout << "\nDecrypted:  " << decrypted << "\n";

    return 0;
}
```

Output:



```
Enter key: KEYWORD

Key Table:
K E Y W O
R D A B C
F G H I L
M N P Q S
T U V X Z

Enter plaintext: HELLO WORLD

Plaintext:  HELLO WORLD
Encrypted:  GYIZSCOKCFBU
Decrypted:  HELXLOWORLDX
yug@yugs-MacBook-Air ins lab  %
```

EXPERIMENT – 4

Aim: To implement Polyalphabetic cipher encryption decryption.
Encryption/Decryption: Based on substitution, using multiple substitution Alphabets

Theory:

A **polyalphabetic cipher** uses **multiple substitution alphabets** to encrypt plaintext, unlike monoalphabetic ciphers which use a single substitution.

- Famous example: **Vigenère cipher**.

Working Principle

1. A **key (sequence of letters)** determines which substitution alphabet to use.
2. Each letter of plaintext is encrypted using a **different Caesar shift** according to the key.
3. Encryption formula:
4. `C_i = (P_i + K_i) mod 26`
   - `C_i` = Ciphertext letter
   - `P_i` = Plaintext letter
   - `K_i` = Key letter corresponding to that position
5. Decryption formula:
6. `P_i = (C_i - K_i + 26) mod 26`

Characteristics

- Uses **multiple Caesar shifts**, making frequency analysis harder.
- Symmetric key cipher.
- Plaintext letters are **spread over multiple alphabets**.

Advantages

- More secure than monoalphabetic substitution.
- Reduces predictability of letter frequencies.

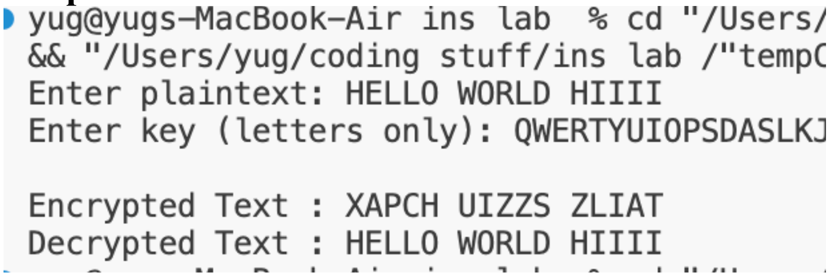
Applications

- Vigenère cipher for secure communication.
- Educational demonstration of **polyalphabetic substitution**.

Source code:

```cpp
#include <iostream>
#include <string>
using namespace std;

// Function to generate repeating key (ignores non-alphabet characters)
string generateKey(const string &text, const string &key) {
    string newKey;
    int j = 0; // index for key
    for (size_t i = 0; i < text.size(); i++) {
        if (isalpha(text[i])) {
            newKey.push_back(key[j % key.size()]);
            j++;
        } else {
            newKey.push_back(text[i]); // keep spaces/punctuation aligned
        }
    }
    return newKey;
}

// Encrypt function
string encryptText(const string &text, const string &key) {
    string cipher_text;
    for (size_t i = 0; i < text.size(); i++) {
        if (isupper(text[i])) {
            char x = ((text[i] - 'A') + (toupper(key[i]) - 'A')) % 26 + 'A';
            cipher_text.push_back(x);
        } else if (islower(text[i])) {
            char x = ((text[i] - 'a') + (tolower(key[i]) - 'a')) % 26 + 'a';
            cipher_text.push_back(x);
        } else {
            cipher_text.push_back(text[i]); // leave spaces/punctuation
        }
    }
    return cipher_text;
}

// Decrypt function
string decryptText(const string &cipher_text, const string &key) {
    string orig_text;
    for (size_t i = 0; i < cipher_text.size(); i++) {
        if (isupper(cipher_text[i])) {
            char x = ((cipher_text[i] - 'A') - (toupper(key[i]) - 'A') + 26) % 26 +
'A';
            orig_text.push_back(x);
        } else if (islower(cipher_text[i])) {
            char x = ((cipher_text[i] - 'a') - (tolower(key[i]) - 'a') + 26) % 26 +
'a';
            orig_text.push_back(x);
        } else {
            orig_text.push_back(cipher_text[i]);
        }
    }
    return orig_text;
}

int main() {
    string text, keyword;
    cout << "Enter plaintext: ";
    getline(cin, text);

    cout << "Enter key (letters only): ";
    cin >> keyword;

    string key = generateKey(text, keyword);
    string cipher_text = encryptText(text, key);

    cout << "\nEncrypted Text : " << cipher_text << endl;
    cout << "Decrypted Text : " << decryptText(cipher_text, key) << endl;

    return 0;
}
```

Output:



```
yug@yugs-MacBook-Air ins lab  % cd "/Users/
&& "/Users/yug/coding stuff/ins lab /"tempC
Enter plaintext: HELLO WORLD HIIII
Enter key (letters only): QWERTYUIOPSDASLKJ

Encrypted Text : XAPCH UIZZS ZLIAT
Decrypted Text : HELLO WORLD HIIII
```

EXPERIMENT – 5

Aim: To implement Hill- cipher encryption decryption

Theory:

The **Hill cipher** is a **polygraphic substitution cipher** based on **linear algebra**.

- Encrypts plaintext in blocks (vectors) of size **n** using an **n×n key matrix**.
- Developed by Lester Hill in 1929.

Working Principle

1. Represent plaintext letters as **numbers (A=0, B=1, … Z=25)**.
2. Divide plaintext into blocks of size **n**.
3. Encryption formula:
4. `C = (K * P) mod 26`
   - `C` = Ciphertext vector
   - `K` = Key matrix (invertible modulo 26)
   - `P` = Plaintext vector
5. Decryption formula:
6. `P = (K^-1 * C) mod 26`
   - `K^-1` = Modular inverse of key matrix modulo 26

Characteristics

- **Polygraphic cipher**: Encrypts multiple letters at once.
- Requires **invertible key matrix**.
- Provides better security than monoalphabetic ciphers.

Applications

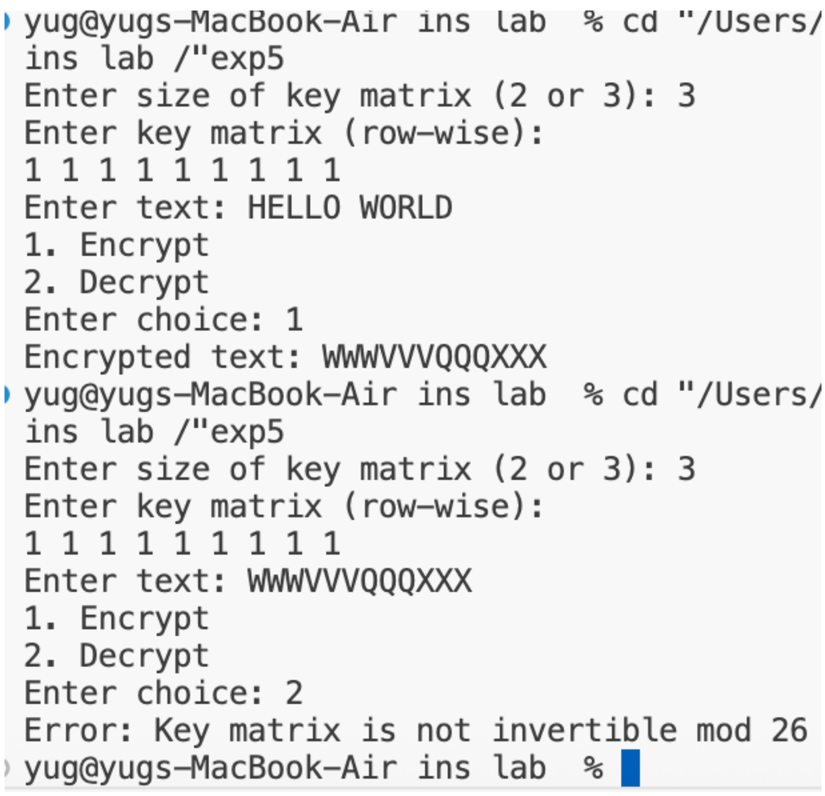
- Classical cryptography education.
- Demonstrates **linear algebra in encryption**.

Source code :

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

// Function to get modulo inverse of a number under mod 26
int modInverse(int a, int m) {
    a = a % m;
    for (int x = 1; x < m; x++) {
        if ((a * x) % m == 1) return x;
    }
    return -1;
}
```

```cpp
// Function to get determinant of matrix (only for 2x2 and 3x3)
int determinant(const vector<vector<int>> &mat, int n) {
    if (n == 2)
        return (mat[0][0]*mat[1][1] - mat[0][1]*mat[1][0]);
    else if (n == 3)
        return (mat[0][0]*(mat[1][1]*mat[2][2] - mat[1][2]*mat[2][1])
                - mat[0][1]*(mat[1][0]*mat[2][2] - mat[1][2]*mat[2][0])
                + mat[0][2]*(mat[1][0]*mat[2][1] - mat[1][1]*mat[2][0]));
    return 0;
}

// Function to get adjoint of 2x2 or 3x3 matrix
vector<vector<int>> adjoint(const vector<vector<int>> &mat, int n) {
    vector<vector<int>> adj(n, vector<int>(n));
    if (n == 2) {
        adj[0][0] = mat[1][1];
        adj[0][1] = -mat[0][1];
        adj[1][0] = -mat[1][0];
        adj[1][1] = mat[0][0];
    } else if (n == 3) {
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                vector<vector<int>> temp(2, vector<int>(2));
                int r = 0;
                for (int k = 0; k < n; k++) {
                    if (k == i) continue;
                    int c = 0;
                    for (int l = 0; l < n; l++) {
                        if (l == j) continue;
                        temp[r][c] = mat[k][l];
                        c++;
                    }
                    r++;
                }
                adj[j][i] = ((temp[0][0]*temp[1][1] - temp[0][1]*temp[1][0]) *
(((i+j)%2==0)?1:-1));
            }
        }
    }
    return adj;
}

// Function to get inverse of matrix mod 26
vector<vector<int>> inverseMatrix(const vector<vector<int>> &mat, int n) {
    int det = determinant(mat, n);
    det = (det % 26 + 26) % 26;
    int invDet = modInverse(det, 26);
    if (invDet == -1) {
        throw runtime_error("Key matrix is not invertible mod 26");
    }
    vector<vector<int>> adj = adjoint(mat, n);
    vector<vector<int>> inv(n, vector<int>(n));
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            int val = adj[i][j] * invDet;
            val = (val % 26 + 26) % 26;
            inv[i][j] = val;
        }
    }
    return inv;
}

// Function to multiply matrix and vector mod 26
vector<int> multiply(const vector<vector<int>> &mat, const vector<int> &vec, int n)
{
    vector<int> res(n);
    for (int i = 0; i < n; i++) {
        int sum = 0;
        for (int j = 0; j < n; j++) {
            sum += mat[i][j] * vec[j];
        }
        res[i] = (sum % 26 + 26) % 26;
    }
    return res;
}

string processText(const string &text, int n) {
    string t = text;
    t.erase(remove_if(t.begin(), t.end(), [](char c){ return !isalpha(c); }),
t.end());
    transform(t.begin(), t.end(), t.begin(), ::toupper);
    while (t.size() % n != 0) t += 'X';
    return t;
}

string encrypt(const string &plaintext, const vector<vector<int>> &key, int n) {
    string pt = processText(plaintext, n);
    string ct;
    for (size_t i = 0; i < pt.size(); i += n) {
        vector<int> block(n);
        for (int j = 0; j < n; j++) block[j] = pt[i+j] - 'A';
        vector<int> enc = multiply(key, block, n);
        for (int j = 0; j < n; j++) ct += (enc[j] + 'A');
    }
    return ct;
}

string decrypt(const string &ciphertext, const vector<vector<int>> &key, int n) {
    vector<vector<int>> invKey = inverseMatrix(key, n);
    string ct = processText(ciphertext, n);
    string pt;
    for (size_t i = 0; i < ct.size(); i += n) {
        vector<int> block(n);
        for (int j = 0; j < n; j++) block[j] = ct[i+j] - 'A';
        vector<int> dec = multiply(invKey, block, n);
        for (int j = 0; j < n; j++) pt += (dec[j] + 'A');
    }
    return pt;
}

int main() {
    int n;
    cout << "Enter size of key matrix (2 or 3): ";
    cin >> n;
    if (n != 2 && n != 3) {
        cout << "Only 2x2 or 3x3 matrices supported.\n";
        return 1;
    }
    vector<vector<int>> key(n, vector<int>(n));
    cout << "Enter key matrix (row-wise):\n";
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cin >> key[i][j];
    cin.ignore();
    string text;
    cout << "Enter text: ";
    getline(cin, text);
    int choice;
    cout << "1. Encrypt\n2. Decrypt\nEnter choice: ";
    cin >> choice;
    try {
        if (choice == 1) {
            string ct = encrypt(text, key, n);
            cout << "Encrypted text: " << ct << endl;
        } else if (choice == 2) {
            string pt = decrypt(text, key, n);
            cout << "Decrypted text: " << pt << endl;
        } else {
            cout << "Invalid choice.\n";
        }
    } catch (const exception &e) {
        cout << "Error: " << e.what() << endl;
    }
    return 0;
}
```

Output:



```
yug@yugs-MacBook-Air ins lab  % cd "/Users/
ins lab /"exp5
Enter size of key matrix (2 or 3): 3
Enter key matrix (row-wise):
1 1 1 1 1 1 1 1 1
Enter text: HELLO WORLD
1. Encrypt
2. Decrypt
Enter choice: 1
Encrypted text: WWWVVVQQQXXX
yug@yugs-MacBook-Air ins lab  % cd "/Users/
ins lab /"exp5
Enter size of key matrix (2 or 3): 3
Enter key matrix (row-wise):
1 1 1 1 1 1 1 1 1
Enter text: WWWVVVQQQXXX
1. Encrypt
2. Decrypt
Enter choice: 2
Error: Key matrix is not invertible mod 26
yug@yugs-MacBook-Air ins lab  %
```

EXPERIMENT – 6

Aim: To implement S-DES sub key Generation

Theory:

Simplified Data Encryption Standard (**S-DES**) is a simplified version of the DES encryption algorithm designed for **educational purposes**.

- S-DES operates on **8-bit plaintext** using a **10-bit key**.
- It uses **two 8-bit subkeys (K1 and K2)** generated from the original key.

Key Generation Process

1. **Input 10-bit key**.
2. **Permutation P10**: Rearrange the 10 bits according to a fixed permutation.
3. **Split into two halves**: Left (L) and Right (R), each 5 bits.
4. **Left shift (LS-1)**: Circular left shift on each half.
5. **Permutation P8**: Select 8 bits from the shifted halves to form **K1**.
6. **Left shift (LS-2)**: Circular left shift by 2 bits on each half.
7. **Permutation P8**: Form **K2**.

Characteristics

- Generates **two subkeys** for the encryption rounds.
- Used in the S-DES **Feistel structure** for encryption/decryption.

Applications

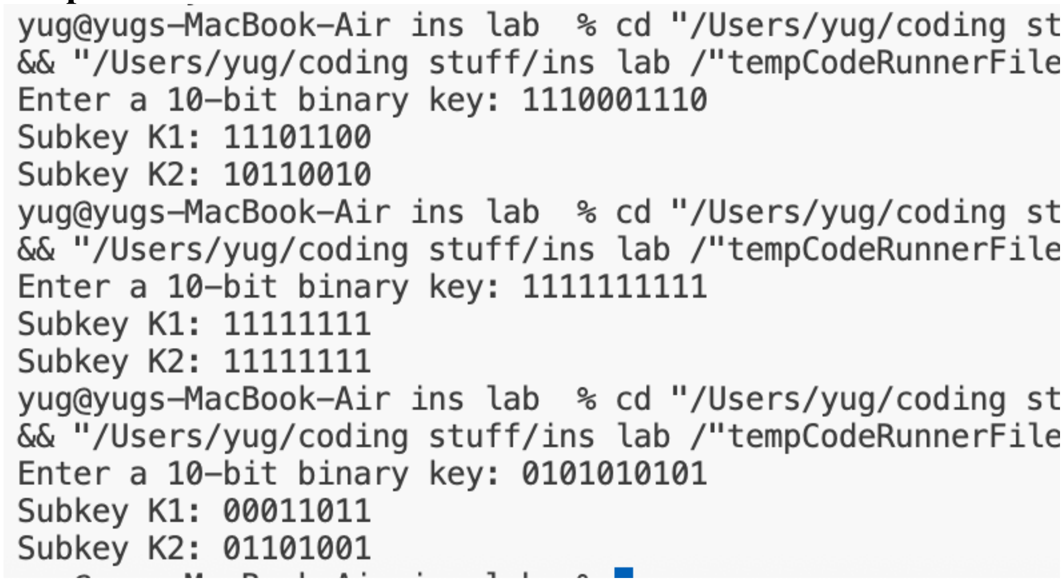
- Educational purposes to demonstrate **key scheduling** and Feistel encryption.

Source code:

```cpp
#include <iostream>
#include <string>
#include <vector>

using namespace std;

// Function to permute the key based on the given permutation table
string permute(const string& key, const vector<int>& table) {
    string permutedKey;
    for (int i : table) {
        permutedKey += key[i - 1]; // -1 for zero-based indexing
    }
    return permutedKey;
}

// Function to perform left shift on the given bits
string leftShift(const string& bits) {
    return bits.substr(1) + bits[0]; // Shift left
}
```

```cpp
// Function to generate subkeys K1 and K2 from the given key
void generateSubkeys(const string& key, string& K1, string& K2) {
    // P10 and P8 permutation tables
    vector<int> P10 = {3, 5, 2, 7, 4, 10, 1, 9, 8, 6};
    vector<int> P8 = {6, 3, 7, 4, 8, 5, 10, 9};

    // Step 1: Permute the key using P10
    string permutedKey = permute(key, P10);

    // Step 2: Split the key into two halves
    string leftHalf = permutedKey.substr(0, 5);
    string rightHalf = permutedKey.substr(5, 5);

    // Step 3: Generate K1
    leftHalf = leftShift(leftHalf);
    rightHalf = leftShift(rightHalf);
    K1 = permute(leftHalf + rightHalf, P8);

    // Step 4: Generate K2
    leftHalf = leftShift(leftHalf);
    rightHalf = leftShift(rightHalf);
    K2 = permute(leftHalf + rightHalf, P8);
}

int main() {
    string key;

    // Input 10-bit binary key
    cout << "Enter a 10-bit binary key: ";
    cin >> key;

    // Validate key length
    if (key.length() != 10) {
        cout << "Error: Key must be exactly 10 bits long." << endl;
        return 1;
    }

    // Variables to hold the subkeys
    string K1, K2;

    // Generate subkeys
    generateSubkeys(key, K1, K2);

    // Output the subkeys
    cout << "Subkey K1: " << K1 << endl;
    cout << "Subkey K2: " << K2 << endl;

    return 0;
}
```

Output:



```
yug@yugs-MacBook-Air ins lab  % cd "/Users/yug/coding st
&& "/Users/yug/coding stuff/ins lab /"tempCodeRunnerFile
Enter a 10-bit binary key: 1110001110
Subkey K1: 11101100
Subkey K2: 10110010
yug@yugs-MacBook-Air ins lab  % cd "/Users/yug/coding st
&& "/Users/yug/coding stuff/ins lab /"tempCodeRunnerFile
Enter a 10-bit binary key: 1111111111
Subkey K1: 11111111
Subkey K2: 11111111
yug@yugs-MacBook-Air ins lab  % cd "/Users/yug/coding st
&& "/Users/yug/coding stuff/ins lab /"tempCodeRunnerFile
Enter a 10-bit binary key: 0101010101
Subkey K1: 00011011
Subkey K2: 01101001
```

EXPERIMENT – 7

Aim: Write a program to implement Diffie-Hallman key exchange algorithm
Theory:

The **Diffie–Hellman Key Exchange (DHKE)** algorithm allows two parties — traditionally called **Alice** and **Bob** — to **securely establish a shared secret key** over an insecure communication channel.

This shared key can later be used to **encrypt** and **decrypt** messages (for example, in symmetric encryption like AES).

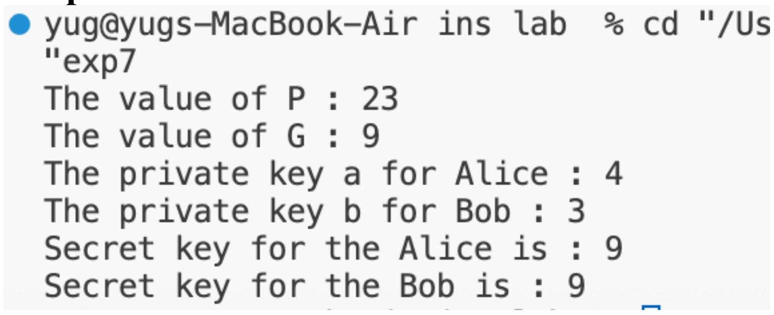
DHKE relies on the **difficulty of the discrete logarithm problem** — that is, given $g^x \bmod p$, it is computationally infeasible to find $x$if $p$is large enough.

Steps:

1. Both users agree on a **prime number (p)** and a **primitive root (g)** publicly.
2. Alice selects a private key **a**, and Bob selects **b**.
3. Alice computes **A = g^a mod p**, Bob computes **B = g^b mod p**, and they exchange A and B.
4. Alice computes **S = B^a mod p**, Bob computes **S = A^b mod p**.
5. Both get the **same shared secret key (S)**.

Source code:
```cpp
#include <cmath>
#include <iostream>

using namespace std;

// Power function to return value of a ^ b mod P
long long int power(long long int a, long long int b,
                    long long int P)
{
    if (b == 1)
        return a;

    else
        return (((long long int)pow(a, b)) % P);
}

// Driver program
int main()
{
    long long int P, G, x, a, y, b, ka, kb;

    // Both the persons will be agreed upon the
    // public keys G and P
    P = 23; // A prime number P is taken
    cout << "The value of P : " << P << endl;

    G = 9; // A primitive root for P, G is taken
    cout << "The value of G : " << G << endl;

    // Alice will choose the private key a
    a = 4; // a is the chosen private key
    cout << "The private key a for Alice : " << a << endl;

    x = power(G, a, P); // gets the generated key

    // Bob will choose the private key b
    b = 3; // b is the chosen private key
    cout << "The private key b for Bob : " << b << endl;

    y = power(G, b, P); // gets the generated key

    // 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
    cout << "Secret key for the Alice is : " << ka << endl;

    cout << "Secret key for the Bob is : " << kb << endl;

    return 0;
}
```

Output:



```
● yug@yugs-MacBook-Air ins lab  % cd "/Us
  "exp7
  The value of P : 23
  The value of G : 9
  The private key a for Alice : 4
  The private key b for Bob : 3
  Secret key for the Alice is : 9
  Secret key for the Bob is : 9
```

EXPERIMENT – 8

Aim: Write a program to implement RSA encryption-decryption.

Theory:

The **RSA algorithm** is an **asymmetric cryptographic technique** used for **secure data transmission**.
It uses **two keys** — a **public key** for encryption and a **private key** for decryption.

Steps:

1. **Key Generation:**
    - Choose two large prime numbers **p** and **q**.
    - Compute **n = p × q**.
    - Compute **φ(n) = (p − 1)(q − 1)**.
    - Choose an integer **e** such that `1 < e < φ(n)` and **gcd(e, φ(n)) = 1**.
    - Compute **d**, the modular inverse of **e** (i.e., `d × e ≡ 1 mod φ(n)`).
    - Public key: **(e, n)**
      Private key: **(d, n)**
2. **Encryption:**
    - Convert the plaintext message **M** into an integer.
    - Compute ciphertext **C = M^e mod n**.
3. **Decryption:**
    - Compute plaintext **M = C^d mod n** using the private key.

Features:

- Based on the **difficulty of factoring large prime numbers**.
- Provides **confidentiality**, **authentication**, and **digital signatures**.
- Widely used in **secure communication**, **SSL/TLS**, and **digital certificates**.

Advantages:

- High security for key exchange and digital signatures.
- Public key can be shared freely without revealing private key.

Limitations:

- Slower compared to symmetric algorithms like AES.
- Requires large key sizes for modern security.

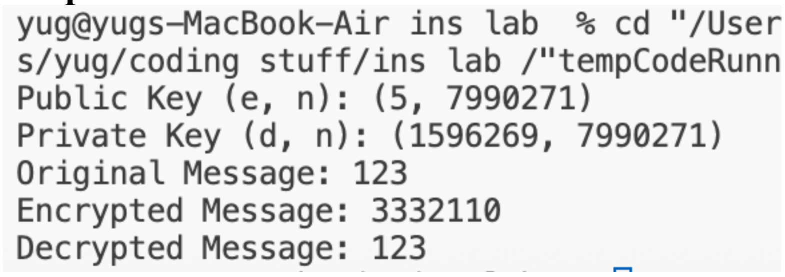
Source code:

```
#include <iostream>
#include <cmath>
#include <algorithm>

using namespace std;

// Function to compute base^expo mod m
```

```
int power(int base, int expo, int m) {
    int res = 1;
    base = base % m;
    while (expo > 0) {
        if (expo & 1)
            res = (res * 1LL * base) % m;
        base = (base * 1LL * base) % m;
        expo = expo / 2;
    }
    return res;
}

// Function to find modular inverse of e modulo phi(n)
// Here we are calculating phi(n) using Hit and Trial Method
// but we can optimize it using Extended Euclidean Algorithm
int modInverse(int e, int phi) {
    for (int d = 2; d < phi; d++) {
        if ((e * d) % phi == 1)
            return d;
    }
    return -1;
}

// Compute GCD (since __gcd is not standard on all compilers)
int gcdCustom(int a, int b) {
    while (b != 0) {
        int t = b;
        b = a % b;
        a = t;
    }
    return a;
}

// RSA Key Generation
void generateKeys(int &e, int &d, int &n) {
    int p = 7919;
    int q = 1009;

    n = p * q;
    int phi = (p - 1) * (q - 1);

    // Choose e, where 1 < e < phi(n) and gcd(e, phi(n)) == 1
    for (e = 2; e < phi; e++) {
        if (gcdCustom(e, phi) == 1)
            break;
    }

    // Compute d such that e * d ≡ 1 (mod phi(n))
    d = modInverse(e, phi);
}
```

```
// Encrypt message using public key (e, n)
int encrypt(int m, int e, int n) {
    return power(m, e, n);
}

// Decrypt message using private key (d, n)
int decrypt(int c, int d, int n) {
    return power(c, d, n);
}

int main() {
    int e, d, n;

    // Key Generation
    generateKeys(e, d, n);

    cout << "Public Key (e, n): (" << e << ", " << n << ")\n";
    cout << "Private Key (d, n): (" << d << ", " << n << ")\n";

    // Message
    int M = 123;
    cout << "Original Message: " << M << endl;

    // Encrypt the message
    int C = encrypt(M, e, n);
    cout << "Encrypted Message: " << C << endl;

    // Decrypt the message
    int decrypted = decrypt(C, d, n);
    cout << "Decrypted Message: " << decrypted << endl;

    return 0;
}
```

Output:



```
yug@yugs-MacBook-Air ins lab  % cd "/User
s/yug/coding stuff/ins lab /"tempCodeRunn
Public Key (e, n): (5, 7990271)
Private Key (d, n): (1596269, 7990271)
Original Message: 123
Encrypted Message: 3332110
Decrypted Message: 123
```

EXPERIMENT – 9

Aim: Write a program to generate SHA-1 hash

Theory:

SHA-1 (Secure Hash Algorithm - 1) is a **cryptographic hash function** that takes an input message and produces a **160-bit (20-byte) hash value**, commonly represented as a **40-digit hexadecimal number**.
It is a **one-way function**, meaning the original data cannot be retrieved from the hash.

Working Principle:

1. **Input Processing:**
   The message is divided into 512-bit blocks.
2. **Padding:**
   A single '1' bit is added, followed by '0's, to make the message length congruent to 448 mod 512, then append the original length (64 bits).
3. **Initialization:**
   SHA-1 uses **five 32-bit words** as initial hash values.
4. **Processing:**
   Each block is processed in **80 rounds** using bitwise operations, rotations, and logical functions.
5. **Output:**
   After all blocks are processed, the five hash values are concatenated to produce the **final 160-bit hash**.

Applications:

- Digital signatures
- File integrity verification
- Password storage and verification
- Version control systems (e.g., Git uses SHA-1)

Advantages:

- Produces a fixed-size output for any input.
- Simple and efficient to implement.

Limitations:

- **Not collision-resistant** — can produce the same hash for different inputs.
- Replaced by more secure algorithms like **SHA-256** and **SHA-3**.

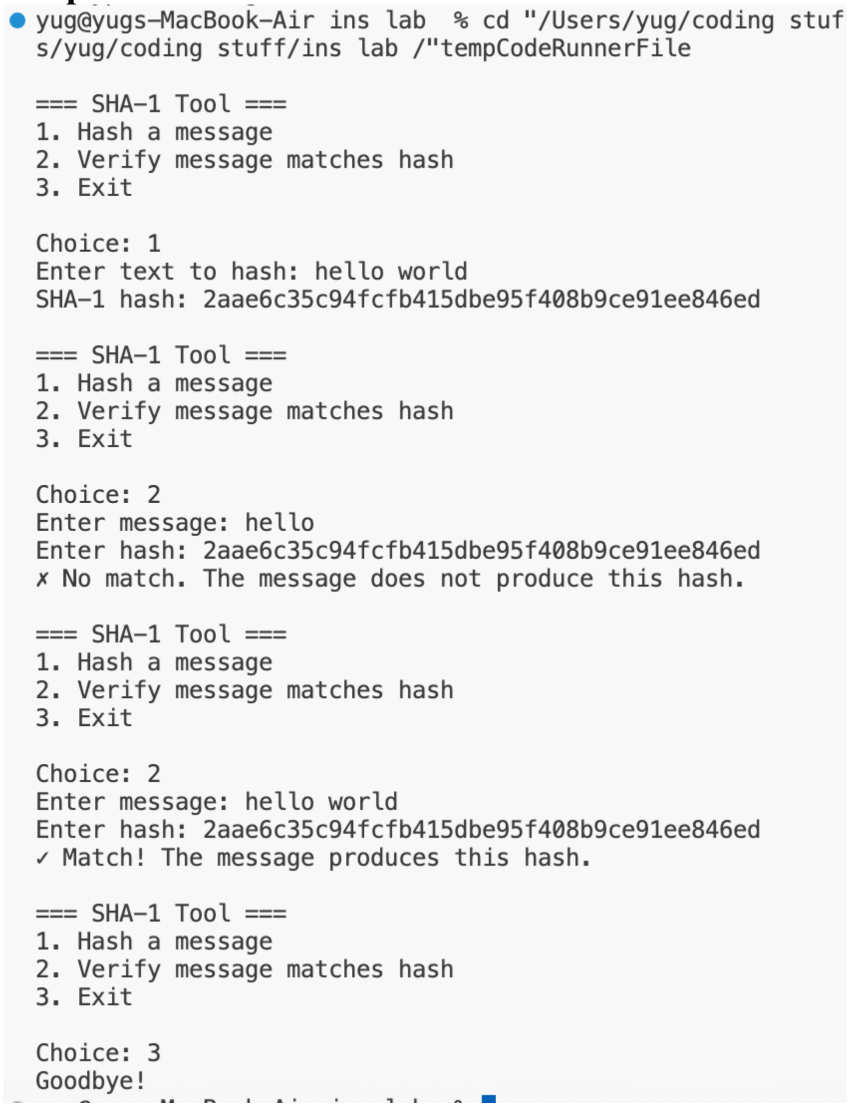
Source code:

```cpp
#include <iostream>
#include <sstream>
#include <iomanip>
#include <vector>
#include <cstring>
```

```
#include <fstream>
#include <algorithm>

class SHA1 {
public:
    SHA1() { reset(); }

    void update(const std::string &s) {
        update(reinterpret_cast<const unsigned char*>(s.c_str()), s.size());
    }
    std::string final() {
        unsigned char digest[20];
        finalize(digest);
        std::ostringstream oss;
        for (int i = 0; i < 20; ++i)
            oss << std::hex << std::setw(2) << std::setfill('0') << (int)digest[i];
        return oss.str();
    }
    void reset() {
        h0 = 0x67452301;
        h1 = 0xEFCDAB89;
        h2 = 0x98BADCFE;
        h3 = 0x10325476;
        h4 = 0xC3D2E1F0;
        buffer.clear();
        bit_len = 0;
        finalized = false;
    }
private:
    uint32_t h0, h1, h2, h3, h4;
    std::vector<unsigned char> buffer;
    uint64_t bit_len = 0;
    bool finalized = false;
    static uint32_t leftrotate(uint32_t value, uint32_t bits) {
        return (value << bits) | (value >> (32 - bits));
    }
    void process_block(const unsigned char *block) {
        uint32_t w[80];
        for (int i = 0; i < 16; ++i) {
            w[i]  = (block[i * 4 + 0] << 24);
            w[i] |= (block[i * 4 + 1] << 16);
            w[i] |= (block[i * 4 + 2] << 8);
            w[i] |= (block[i * 4 + 3]);
        }
        for (int i = 16; i < 80; ++i)
            w[i] = leftrotate(w[i - 3] ^ w[i - 8] ^ w[i - 14] ^ w[i - 16], 1);
        uint32_t a = h0, b = h1, c = h2, d = h3, e = h4;
        for (int i = 0; i < 80; ++i) {
            uint32_t f, k;
            if (i < 20) { f = (b & c) | ((~b) & d); k = 0x5A827999; }
            else if (i < 40) { f = b ^ c ^ d; k = 0x6ED9EBA1; }
            else if (i < 60) { f = (b & c) | (b & d) | (c & d); k = 0x8F1BBCDC; }
            else { f = b ^ c ^ d; k = 0xCA62C1D6; }
            uint32_t temp = leftrotate(a, 5) + f + e + k + w[i];
            e = d;
            d = c;
            c = leftrotate(b, 30);
            b = a;
            a = temp;
        }
        h0 += a;
        h1 += b;
        h2 += c;
        h3 += d;
        h4 += e;
    }
    void update(const unsigned char *data, size_t len) {
        bit_len += len * 8;
        for (size_t i = 0; i < len; ++i) {
            buffer.push_back(data[i]);
            if (buffer.size() == 64) {
                process_block(buffer.data());
                buffer.clear();
            }
        }
    }
    void finalize(unsigned char digest[20]) {
        if (finalized) return;
        finalized = true;
        buffer.push_back(0x80);
        while ((buffer.size() % 64) != 56)
            buffer.push_back(0x00);
        unsigned char length_bytes[8];
        for (int i = 0; i < 8; ++i)
            length_bytes[7 - i] = (bit_len >> (i * 8)) & 0xFF;
        buffer.insert(buffer.end(), length_bytes, length_bytes + 8);
        for (size_t i = 0; i < buffer.size(); i += 64)
            process_block(&buffer[i]);
        uint32_t h[5] = {h0, h1, h2, h3, h4};
        for (int i = 0; i < 5; ++i) {
            digest[i * 4 + 0] = (h[i] >> 24) & 0xFF;
            digest[i * 4 + 1] = (h[i] >> 16) & 0xFF;
            digest[i * 4 + 2] = (h[i] >> 8) & 0xFF;
            digest[i * 4 + 3] = h[i] & 0xFF;
        }
    }
};
std::string hashString(const std::string& input) {
    SHA1 sha;
    sha.update(input);
    return sha.final();
}
```

```cpp
void toLowerCase(std::string& str) {
    std::transform(str.begin(), str.end(), str.begin(), ::tolower);
}
bool verifyHash(const std::string& input, const std::string& target_hash) {
    std::string computed = hashString(input);
    std::string target = target_hash;
    toLowerCase(computed);
    toLowerCase(target);
    return computed == target;
}
int main() {
    while (true) {
        std::cout << "\n=== SHA-1 Tool ===" << std::endl;
        std::cout << "1. Hash a message" << std::endl;
        std::cout << "2. Verify message matches hash" << std::endl;
        std::cout << "3. Exit" << std::endl;
        std::cout << "\nChoice: ";

        int choice;
        std::cin >> choice;
        std::cin.ignore();

        if (choice == 1) {
            std::string input;
            std::cout << "Enter text to hash: ";
            std::getline(std::cin, input);
            std::cout << "SHA-1 hash: " << hashString(input) << std::endl;

        } else if (choice == 2) {
            std::string input, hash;
            std::cout << "Enter message: ";
            std::getline(std::cin, input);
            std::cout << "Enter hash: ";
            std::getline(std::cin, hash);
            if (verifyHash(input, hash)) {
                std::cout << "✓ Match! The message produces this hash." <<
std::endl;
            } else {
                std::cout << "✗ No match. The message does not produce this hash."
<< std::endl;
            }
        } else if (choice == 3) {
            std::cout << "Goodbye!" << std::endl;
            break;
        } else {
            std::cout << "Invalid choice." << std::endl;
        }
    }
    return 0;
}
```

Output:



```
yug@yugs-MacBook-Air ins lab  % cd "/Users/yug/coding stuf
s/yug/coding stuff/ins lab /"tempCodeRunnerFile

=== SHA-1 Tool ===
1. Hash a message
2. Verify message matches hash
3. Exit

Choice: 1
Enter text to hash: hello world
SHA-1 hash: 2aae6c35c94fcfb415dbe95f408b9ce91ee846ed

=== SHA-1 Tool ===
1. Hash a message
2. Verify message matches hash
3. Exit

Choice: 2
Enter message: hello
Enter hash: 2aae6c35c94fcfb415dbe95f408b9ce91ee846ed
✗ No match. The message does not produce this hash.

=== SHA-1 Tool ===
1. Hash a message
2. Verify message matches hash
3. Exit

Choice: 2
Enter message: hello world
Enter hash: 2aae6c35c94fcfb415dbe95f408b9ce91ee846ed
✓ Match! The message produces this hash.

=== SHA-1 Tool ===
1. Hash a message
2. Verify message matches hash
3. Exit

Choice: 3
Goodbye!
```

EXPERIMENT – 10

Aim: Write a program to implement a digital signature algorithm

Theory:

A **Digital Signature** is a **cryptographic technique** that ensures the **authenticity**, **integrity**, and **non-repudiation** of a message.
It allows a sender to sign a message using a **private key**, and the receiver to verify it using the **public key**.

The **Digital Signature Algorithm (DSA)** is a **public key algorithm** based on **modular arithmetic** and **discrete logarithms**.

Working Principle:

1. Key Generation:

- Choose a **prime modulus p** and a **subprime q**, where q divides (p−1).
- Select a **generator g** of order q modulo p.
- Choose a random **private key x** ($1 < x < q$).
- Compute **public key** $y = g^x \bmod p$.

2. Signature Generation:

- Select a random number **k** ($1 < k < q$).
- Compute $r = (g^k \bmod p) \bmod q$.
- Compute $s = (k^{-1} \times (H(m) + x \times r)) \bmod q$,
  where H(m) is the hash of the message.

The **signature** is the pair **(r, s)**.

3. Signature Verification:

- Compute $w = s^{-1} \bmod q$.
- Compute $u_1 = (H(m) \times w) \bmod q$, $u_2 = (r \times w) \bmod q$.
- Compute $v = ((g^{u1} \times y^{u2}) \bmod p) \bmod q$.
- If **v == r**, the signature is **valid**

Applications

- Email and document authentication
- Software distribution
- Secure financial transactions
- Digital certificates (used in SSL/TLS)

Advantages:

- Provides **data integrity** and **authentication**.
- Ensures **non-repudiation** (sender cannot deny sending).

Limitations:

- More computationally expensive than symmetric encryption.
- Requires careful random number generation for **k**, else security is compromised.

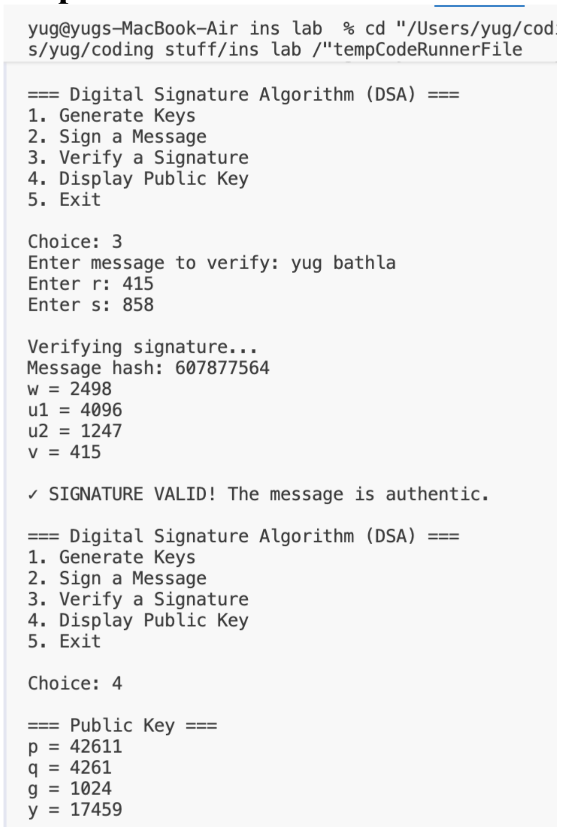
Source code:

```cpp
#include <iostream>
#include <string>
#include <cmath>
#include <ctime>
#include <sstream>
#include <iomanip>
#include <vector>

// Simple big integer class for DSA operations (limited to 64-bit for simplicity)
class BigInt {
public:
    long long value;

    BigInt(long long v = 0) : value(v) {}

    BigInt operator+(const BigInt& other) const {
        return BigInt(value + other.value);
    }

    BigInt operator-(const BigInt& other) const {
        return BigInt(value - other.value);
    }

    BigInt operator*(const BigInt& other) const {
        return BigInt(value * other.value);
    }

    BigInt operator%(const BigInt& other) const {
        return BigInt(value % other.value);
    }

    bool operator==(const BigInt& other) const {
        return value == other.value;
    }

    bool operator<(const BigInt& other) const {
        return value < other.value;
    }
};

// Modular exponentiation: (base^exp) % mod
long long modPow(long long base, long long exp, long long mod) {
    long long result = 1;
    base = base % mod;
    while (exp > 0) {
        if (exp % 2 == 1) {
            result = (result * base) % mod;
        }
        exp = exp >> 1;
        base = (base * base) % mod;
    }

    return result;
}

// Extended Euclidean Algorithm for modular inverse
long long modInverse(long long a, long long m) {
    long long m0 = m, x0 = 0, x1 = 1;

    if (m == 1) return 0;

    while (a > 1) {
        long long q = a / m;
        long long t = m;

        m = a % m;
        a = t;
        t = x0;

        x0 = x1 - q * x0;
        x1 = t;
    }

    if (x1 < 0) x1 += m0;

    return x1;
}

// Simple primality test (Miller-Rabin would be better for production)
bool isPrime(long long n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;

    for (long long i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0)
            return false;
    }

    return true;
}

// Simple hash function (in production, use SHA-256)
long long simpleHash(const std::string& message) {
    long long hash = 0;
```

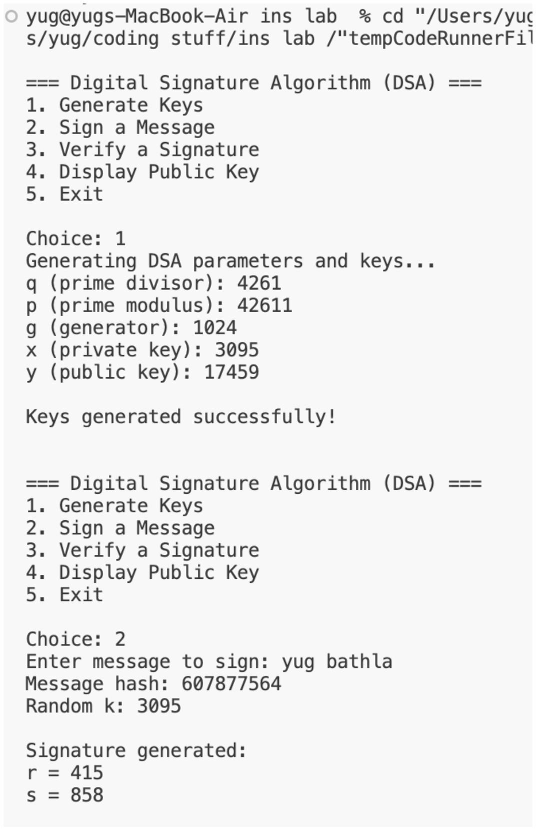
```cpp
    for (char c : message) {
        hash = (hash * 31 + c) % 1000000007;
    }
    return hash;
}

class DSA {
private:
    long long p;  // Prime modulus
    long long q;  // Prime divisor of p-1
    long long g;  // Generator
    long long x;  // Private key
    long long y;  // Public key

    // Generate a prime number
    long long generatePrime(long long min, long long max) {
        srand(time(0));
        for (int attempts = 0; attempts < 1000; attempts++) {
            long long candidate = min + rand() % (max - min);
            if (isPrime(candidate)) {
                return candidate;
            }
        }
        return 0;
    }

    // Find a generator g
    long long findGenerator() {
        for (long long h = 2; h < p; h++) {
            g = modPow(h, (p - 1) / q, p);
            if (g > 1) {
                return g;
            }
        }
        return 2;
    }

public:
    DSA() : p(0), q(0), g(0), x(0), y(0) {}

    // Generate DSA parameters and keys
    void generateKeys() {
        std::cout << "Generating DSA parameters and keys..." << std::endl;

        // Generate prime q (smaller prime)
        q = generatePrime(1000, 5000);
        std::cout << "q (prime divisor): " << q << std::endl;

        // Generate prime p such that q divides (p-1)
        for (int i = 2; i < 100; i++) {
            long long candidate = i * q + 1;
            if (isPrime(candidate)) {
                p = candidate;
                break;
            }
        }
        std::cout << "p (prime modulus): " << p << std::endl;

        // Find generator g
        g = findGenerator();
        std::cout << "g (generator): " << g << std::endl;

        // Generate private key x (random number < q)
        srand(time(0) + 1);
        x = (rand() % (q - 1)) + 1;
        std::cout << "x (private key): " << x << std::endl;

        // Calculate public key y = g^x mod p
        y = modPow(g, x, p);
        std::cout << "y (public key): " << y << std::endl;

        std::cout << "\nKeys generated successfully!\n" << std::endl;
    }

    // Sign a message
    std::pair<long long, long long> sign(const std::string& message) {
        if (p == 0 || q == 0) {
            std::cout << "Error: Keys not generated yet!" << std::endl;
            return {0, 0};
        }

        // Hash the message
        long long h = simpleHash(message);
        std::cout << "Message hash: " << h << std::endl;

        // Generate random k (1 < k < q)
        srand(time(0) + rand());
        long long k = (rand() % (q - 2)) + 2;
        std::cout << "Random k: " << k << std::endl;

        // Calculate r = (g^k mod p) mod q
        long long r = modPow(g, k, p) % q;

        // Calculate s = (k^-1 * (h + x*r)) mod q
        long long k_inv = modInverse(k, q);
        long long s = (k_inv * (h + x * r)) % q;

        // Make sure r and s are not zero
        if (r == 0 || s == 0) {
            std::cout << "Error: Invalid signature (r or s is zero),
regenerating..." << std::endl;
            return sign(message);
```

```cpp
        }
        std::cout << "\nSignature generated:" << std::endl;
        std::cout << "r = " << r << std::endl;
        std::cout << "s = " << s << std::endl;

        return {r, s};
    }
    // Verify a signature
    bool verify(const std::string& message, long long r, long long s) {
        if (p == 0 || q == 0) {
            std::cout << "Error: Keys not generated yet!" << std::endl;
            return false;
        }
        // Check if r and s are in valid range
        if (r <= 0 || r >= q || s <= 0 || s >= q) {
            std::cout << "Invalid signature: r or s out of range" << std::endl;
            return false;
        }
        // Hash the message
        long long h = simpleHash(message);
        std::cout << "Message hash: " << h << std::endl;
        // Calculate w = s^-1 mod q
        long long w = modInverse(s, q);
        std::cout << "w = " << w << std::endl;
        // Calculate u1 = (h * w) mod q
        long long u1 = (h * w) % q;
        std::cout << "u1 = " << u1 << std::endl;
        // Calculate u2 = (r * w) mod q
        long long u2 = (r * w) % q;
        std::cout << "u2 = " << u2 << std::endl;
        // Calculate v = ((g^u1 * y^u2) mod p) mod q
        long long v1 = modPow(g, u1, p);
        long long v2 = modPow(y, u2, p);
        long long v = ((v1 * v2) % p) % q;
        std::cout << "v = " << v << std::endl;
        // Signature is valid if v == r
        return v == r;
    }

    void displayPublicKey() {
        std::cout << "\n=== Public Key ===" << std::endl;
        std::cout << "p = " << p << std::endl;
        std::cout << "q = " << q << std::endl;
        std::cout << "g = " << g << std::endl;
        std::cout << "y = " << y << std::endl;
    }
};

int main() {
    DSA dsa;
    std::string message;
    long long r = 0, s = 0;

    while (true) {
        std::cout << "\n=== Digital Signature Algorithm (DSA) ===" << std::endl;
        std::cout << "1. Generate Keys" << std::endl;
        std::cout << "2. Sign a Message" << std::endl;
        std::cout << "3. Verify a Signature" << std::endl;
        std::cout << "4. Display Public Key" << std::endl;
        std::cout << "5. Exit" << std::endl;
        std::cout << "\nChoice: ";
        int choice;
        std::cin >> choice;
        std::cin.ignore();
        if (choice == 1) {
            dsa.generateKeys();
        } else if (choice == 2) {
            std::cout << "Enter message to sign: ";
            std::getline(std::cin, message);
            auto signature = dsa.sign(message);
            r = signature.first;
            s = signature.second;
        } else if (choice == 3) {
            std::cout << "Enter message to verify: ";
            std::getline(std::cin, message);
            std::cout << "Enter r: ";
            std::cin >> r;
            std::cout << "Enter s: ";
            std::cin >> s;
            std::cin.ignore();
            std::cout << "\nVerifying signature..." << std::endl;
            bool isValid = dsa.verify(message, r, s);
            if (isValid) {
                std::cout << "\n✓ SIGNATURE VALID! The message is authentic." <<
std::endl;
            } else {
                std::cout << "\n✗ SIGNATURE INVALID! The message may have been
tampered with." << std::endl;
            }

        } else if (choice == 4) {
            dsa.displayPublicKey();

        } else if (choice == 5) {
            std::cout << "Goodbye!" << std::endl;
            break;

        } else {
            std::cout << "Invalid choice." << std::endl;
        }
    }
    return 0;
}
```

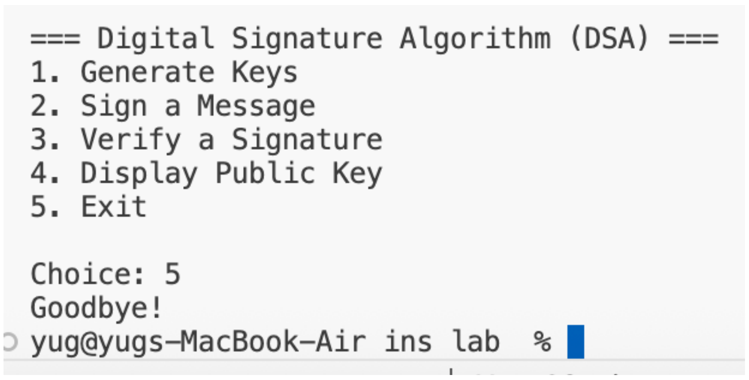
Output:



```
yug@yugs-MacBook-Air ins lab  % cd "/Users/yug/cod
s/yug/coding stuff/ins lab /"tempCodeRunnerFile

=== Digital Signature Algorithm (DSA) ===
1. Generate Keys
2. Sign a Message
3. Verify a Signature
4. Display Public Key
5. Exit

Choice: 3
Enter message to verify: yug bathla
Enter r: 415
Enter s: 858

Verifying signature...
Message hash: 607877564
w = 2498
u1 = 4096
u2 = 1247
v = 415

✓ SIGNATURE VALID! The message is authentic.

=== Digital Signature Algorithm (DSA) ===
1. Generate Keys
2. Sign a Message
3. Verify a Signature
4. Display Public Key
5. Exit

Choice: 4

=== Public Key ===
p = 42611
q = 4261
g = 1024
y = 17459
```



```
yug@yugs-MacBook-Air ins lab  % cd "/Users/yug
s/yug/coding stuff/ins lab /"tempCodeRunnerFil

=== Digital Signature Algorithm (DSA) ===
1. Generate Keys
2. Sign a Message
3. Verify a Signature
4. Display Public Key
5. Exit

Choice: 1
Generating DSA parameters and keys...
q (prime divisor): 4261
p (prime modulus): 42611
g (generator): 1024
x (private key): 3095
y (public key): 17459

Keys generated successfully!


=== Digital Signature Algorithm (DSA) ===
1. Generate Keys
2. Sign a Message
3. Verify a Signature
4. Display Public Key
5. Exit

Choice: 2
Enter message to sign: yug bathla
Message hash: 607877564
Random k: 3095

Signature generated:
r = 415
s = 858
```



```
=== Digital Signature Algorithm (DSA) ===
1. Generate Keys
2. Sign a Message
3. Verify a Signature
4. Display Public Key
5. Exit

Choice: 5
Goodbye!
yug@yugs-MacBook-Air ins lab  %
```