1. Simulation of Caesar Cipher

Title

Simulation of Caesar Cipher Encryption and Decryption

Algorithm

1. **Input**: Plaintext and shift key (k).
2. **Encryption**:
    - For each character in the plaintext:
        - If it's a letter, shift it forward by k positions in the alphabet (mod 26 for wrap-around).
        - Non-alphabetic characters remain unchanged.
3. **Decryption**:
    - For each character in the ciphertext:
        - Shift it backward by k positions (mod 26).
        - Non-alphabetic characters remain unchanged.
4. **Output**: Ciphertext (for encryption) or plaintext (for decryption).

Source Code

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>

void caesar_encrypt(char *text, int key) {

  for (int i = 0; text[i] != '\0'; i++) {

    if (isalpha(text[i])) {

      char offset = isupper(text[i]) ? 'A' : 'a';

      text[i] = (char)(((text[i] - offset + key) % 26) + offset);

    }  }}
```

```
void caesar_decrypt(char *text, int key) {
    caesar_encrypt(text, 26 - key); // Decryption is encryption with inverse key
}
int main() {
    char text[100];
    int key;
    printf("Enter plaintext: ");
    fgets(text, sizeof(text), stdin);
    text[strcspn(text, "\n")] = '\0'; // Remove newline
    printf("Enter shift key: ");
    scanf("%d", &key);

    printf("Plaintext: %s\n", text);
    caesar_encrypt(text, key);
    printf("Ciphertext: %s\n", text);
    caesar_decrypt(text, key);
    printf("Decrypted text: %s\n", text);
    return 0;
}
```

Output

```
Enter plaintext: Hello World
Enter shift key: 3
Plaintext: Hello World
Ciphertext: Khoor Zruog
Decrypted text: Hello World
```

Conclusion

The Caesar Cipher is a simple substitution cipher that shifts letters by a fixed number. It is easy to implement but insecure due to its small keyspace (only 25 possible keys), making it vulnerable to brute-force attacks. This simulation demonstrates basic encryption and decryption effectively.

2. Simulation of Hill Cipher

Title

Simulation of Hill Cipher Encryption and Decryption

Algorithm

1. **Input**: Plaintext and 2x2 key matrix (must be invertible modulo 26).
2. **Encryption**:
    - Convert plaintext to numbers (A=0, B=1, ..., Z=25).
    - Group into pairs (add padding if needed).
    - For each pair, multiply by the key matrix modulo 26 to get ciphertext numbers.
    - Convert numbers back to letters.
3. **Decryption**:
    - Compute the inverse of the key matrix modulo 26.
    - Multiply ciphertext pairs by the inverse matrix modulo 26.
    - Convert results back to plaintext.
4. **Output**: Ciphertext or plaintext.

Source Code

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>


void hill_encrypt(char *text, int key[2][2]) {

  int len = strlen(text), i;

  for (i = 0; i < len - 1; i += 2) {

    int p1 = toupper(text[i]) - 'A';

    int p2 = toupper(text[i + 1]) - 'A';
```

```
        text[i] = (char)((key[0][0] * p1 + key[0][1] * p2) % 26 + 'A');
        text[i + 1] = (char)((key[1][0] * p1 + key[1][1] * p2) % 26 + 'A');
    }}
int mod_inverse(int a, int m) {
    for (int x = 1; x < m; x++)
        if ((a * x) % m == 1) return x;
    return -1; // No inverse exists
}
void hill_decrypt(char *text, int key[2][2]) {
    int det = (key[0][0] * key[1][1] - key[0][1] * key[1][0]) % 26;
    if (det < 0) det += 26;
    int inv_det = mod_inverse(det, 26);
    if (inv_det == -1) {
        printf("Key matrix is not invertible!\n");
        return;
    }
    int inv_key[2][2] = {
        {(key[1][1] * inv_det) % 26, (-key[0][1] * inv_det) % 26},
        {(-key[1][0] * inv_det) % 26, (key[0][0] * inv_det) % 26}
    };
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            if (inv_key[i][j] < 0) inv_key[i][j] += 26;
    hill_encrypt(text, inv_key);}
int main() {
    char text[100];
```

```
    int key[2][2];
    printf("Enter plaintext (even length, letters only): ");
    scanf("%s", text);
    printf("Enter 2x2 key matrix (4 integers):\n");
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            scanf("%d", &key[i][j]);
    printf("Plaintext: %s\n", text);
    hill_encrypt(text, key);
    printf("Ciphertext: %s\n", text);
    hill_decrypt(text, key);
    printf("Decrypted text: %s\n", text);
    return 0;
}
```

OUTPUT

```
Enter plaintext (even length, letters only): HELLO
Enter 2x2 key matrix (4 integers):
5 8
17 3
Plaintext: HELLO
Ciphertext: ZSLHL
Decrypted text: HELLO
```

Conclusion

The Hill Cipher uses matrix multiplication for encryption, offering more security than the Caesar Cipher due to its larger keyspace. However, it requires an invertible key matrix and is still vulnerable to known-plaintext attacks. This simulation highlights linear algebra's role in cryptography.

3. Simulation of Rail Fence Cipher

Title

Simulation of Rail Fence Cipher Encryption and Decryption

Algorithm

1. **Input**: Plaintext and number of rails (key).
2. **Encryption**:
    - Create a rail matrix with the specified number of rails.
    - Write the plaintext in a zigzag pattern across the rails.
    - Read the rails row by row to form the ciphertext.
3. **Decryption**:
    - Calculate the positions of characters in the rail matrix.
    - Reconstruct the zigzag pattern and read diagonally to recover the plaintext.
4. **Output**: Ciphertext or plaintext.

Source Code

```
#include <stdio.h>
#include <string.h>

void rail_fence_encrypt(char *text, int key, char *result) {
  int len = strlen(text);
  char rail[key][len];
  for (int i = 0; i < key; i++)
    for (int j = 0; j < len; j++)
      rail[i][j] = '\0';

  int row = 0, col = 0, dir = 0; // dir: 0=down, 1=up
```

```
    for (int i = 0; i < len; i++) {
        rail[row][col++] = text[i];
        if (row == 0) dir = 0;
        if (row == key - 1) dir = 1;
        row += dir ? -1 : 1;
    }

    int k = 0;
    for (int i = 0; i < key; i++)
        for (int j = 0; j < len; j++)
            if (rail[i][j] != '\0')
                result[k++] = rail[i][j];
    result[k] = '\0';
}

void rail_fence_decrypt(char *text, int key, char *result) {
    int len = strlen(text);
    char rail[key][len];
    int pos[len];

    for (int i = 0; i < key; i++)
        for (int j = 0; j < len; j++)
            rail[i][j] = '\0';

    int row = 0, col = 0, dir = 0, k = 0;
    for (int i = 0; i < len; i++) {
```

```
        pos[i] = col;
        rail[row][col] = '*';
        if (row == 0) dir = 0;
        if (row == key - 1) dir = 1;
        row += dir ? -1 : 1;
        col++;
    }

    k = 0;
    for (int i = 0; i < key; i++)
        for (int j = 0; j < len; j++)
            if (rail[i][j] == '*')
                rail[i][j] = text[k++];

    row = 0, col = 0, dir = 0, k = 0;
    for (int i = 0; i < len; i++) {
        result[k++] = rail[row][col++];
        if (row == 0) dir = 0;
        if (row == key - 1) dir = 1;
        row += dir ? -1 : 1;
    }
    result[k] = '\0';
}

int main() {
    char text[100], result[100];
```

```
    int key;
    printf("Enter plaintext: ");
    scanf("%s", text);
    printf("Enter number of rails: ");
    scanf("%d", &key);

    rail_fence_encrypt(text, key, result);
    printf("Ciphertext: %s\n", result);
    rail_fence_decrypt(result, key, text);
    printf("Decrypted text: %s\n", text);
    return 0;
}
```

Output

Enter plaintext: HELLOWORLD

Enter number of rails: 3

Ciphertext: HLOOLWEDL

Decrypted text: HELLOWORLD

Conclusion

The Rail Fence Cipher rearranges letters based on a zigzag pattern, making it simple but less secure due to predictable patterns. This simulation demonstrates transposition ciphers and their reversible nature, though they are easily broken with frequency analysis.

4. Simulation of Playfair Cipher

Title

Simulation of Playfair Cipher Encryption and Decryption

Algorithm

1. **Input**: Plaintext and 5x5 key matrix (generated from a keyword).
2. **Key Matrix Creation**:
    - Fill a 5x5 matrix with the keyword (removing duplicates, treating I/J as one).
    - Fill remaining cells with unused letters.
3. **Encryption**:
    - Break plaintext into digraphs (add filler like 'X' if needed).
    - For each digraph:
        - If in the same row, replace with letters to the right (wrap around).
        - If in the same column, replace with letters below (wrap around).
        - If different rows/columns, form a rectangle and take opposite corners.
4. **Decryption**:
    - Same as encryption, but shift left (for rows) or up (for columns).
5. **Output**: Ciphertext or plaintext.

Source Code

```
#include <stdio.h>

#include <string.h>

#include <ctype.h>


void create_playfair_matrix(char *key, char matrix[5][5]) {

  int used[26] = {0}, k = 0, i, j;

  used['J' - 'A'] = 1; // Treat I/J as same
```

```
    for (i = 0; i < 5; i++)
        for (j = 0; j < 5; j++)
            matrix[i][j] = '\0';

    for (i = 0; key[i]; i++) {
        char c = toupper(key[i]);
        if (!used[c - 'A']) {
            matrix[k / 5][k % 5] = c;
            used[c - 'A'] = 1;
            k++;
        }
    }
    for (char c = 'A'; c <= 'Z'; c++)
        if (!used[c - 'A']) {
            matrix[k / 5][k % 5] = c;
            used[c - 'A'] = 1;
            k++;
        }
}

void find_position(char matrix[5][5], char c, int *row, int *col) {
    if (c == 'J') c = 'I';
    for (int i = 0; i < 5; i++)
        for (int j = 0; j < 5; j++)
            if (matrix[i][j] == c) {
                *row = i;
```

```
            *col = j;
            return;
        }
    }
}

void playfair_encrypt(char *text, char matrix[5][5], char *result) {
    int len = strlen(text), k = 0;
    for (int i = 0; i < len; i += 2) {
        char a = toupper(text[i]), b = toupper(text[i + 1]);
        int r1, c1, r2, c2;
        find_position(matrix, a, &r1, &c1);
        find_position(matrix, b, &r2, &c2);

        if (r1 == r2) {
            result[k++] = matrix[r1][(c1 + 1) % 5];
            result[k++] = matrix[r2][(c2 + 1) % 5];
        } else if (c1 == c2) {
            result[k++] = matrix[(r1 + 1) % 5][c1];
            result[k++] = matrix[(r2 + 1) % 5][c2];
        } else {
            result[k++] = matrix[r1][c2];
            result[k++] = matrix[r2][c1];
        }
    }
    result[k] = '\0';
}
```

```
void playfair_decrypt(char *text, char matrix[5][5], char *result) {
    int len = strlen(text), k = 0;
    for (int i = 0; i < len; i += 2) {
        char a = toupper(text[i]), b = toupper(text[i + 1]);
        int r1, c1, r2, c2;
        find_position(matrix, a, &r1, &c1);
        find_position(matrix, b, &r2, &c2);

        if (r1 == r2) {
            result[k++] = matrix[r1][(c1 - 1 + 5) % 5];
            result[k++] = matrix[r2][(c2 - 1 + 5) % 5];
        } else if (c1 == c2) {
            result[k++] = matrix[(r1 - 1 + 5) % 5][c1];
            result[k++] = matrix[(r2 - 1 + 5) % 5][c2];
        } else {
            result[k++] = matrix[r1][c2];
            result[k++] = matrix[r2][c1];
        }
    }
    result[k] = '\0';
}

int main() {
    char key[100], text[100], result[100];
    char matrix[5][5];
```

```
    printf("Enter keyword: ");
    scanf("%s", key);
    printf("Enter plaintext (even length, letters only): ");
    scanf("%s", text);

    create_playfair_matrix(key, matrix);
    playfair_encrypt(text, matrix, result);
    printf("Ciphertext: %s\n", result);
    playfair_decrypt(result, matrix, text);
    printf("Decrypted text: %s\n", text);
    return 0;
}
```

Output

```
Enter keyword: MONARCHY
Enter plaintext (even length, letters only): HELLO
Ciphertext: FJHLTK
Decrypted text: HELLO
```

Conclusion

The Playfair Cipher improves on monoalphabetic ciphers by encrypting digraphs, making it harder to break with frequency analysis. However, it is still vulnerable to cryptanalysis with sufficient ciphertext. This simulation shows its complexity and effectiveness for small-scale encryption.

5. Simulation of RSA Algorithm

Title

Simulation of RSA Encryption and Decryption

Algorithm

1. **Input**: Two prime numbers (p, q) and a message (m).
2. **Key Generation**:
    - Compute n = p * q and ϕ(n) = (p-1) * (q-1).
    - Choose e (public key) such that 1 < e < ϕ(n) and gcd(e, ϕ(n)) = 1.
    - Compute d (private key) such that d * e ≡ 1 (mod ϕ(n)).
3. **Encryption**:
    - For message m, compute ciphertext c = m^e mod n.
4. **Decryption**:
    - Compute plaintext m = c^d mod n.
5. **Output**: Ciphertext and decrypted message.

Source Code

```
#include <stdio.h>

long long mod_pow(long long base, long long exp, long long mod) {
  long long result = 1;
  base %= mod;
  while (exp > 0) {
    if (exp & 1) result = (result * base) % mod;
    base = (base * base) % mod;
    exp >>= 1;
  }
  return result;
}
```

```
int gcd(int a, int b) {
   while (b) {
      a %= b;
      int temp = a;
      a = b;
      b = temp;
   }
   return a;
}

int main() {
   int p, q, e, m;
   printf("Enter two prime numbers (p, q): ");
   scanf("%d %d", &p, &q);
   printf("Enter message (number < p*q): ");
   scanf("%d", &m);
   printf("Enter public exponent e: ");
   scanf("%d", &e);

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

   if (gcd(e, phi) != 1) {
      printf("Invalid e: not coprime with phi(n)\n");
      return 1;
```

```
    }

    // Find d such that d * e = 1 mod phi
    long long d = 1;
    while ((d * e) % phi != 1) d++;

    long long c = mod_pow(m, e, n); // Encrypt
    long long decrypted = mod_pow(c, d, n); // Decrypt

    printf("Public key: (e=%d, n=%lld)\n", e, n);
    printf("Private key: (d=%lld, n=%lld)\n", d, n);
    printf("Ciphertext: %lld\n", c);
    printf("Decrypted message: %lld\n", decrypted);
    return 0;
}
```

```
Output
Enter two prime numbers (p, q): 3 11
Enter message (number < p*q): 25
Enter public exponent e: 7
Public key: (e=7, n=33)
Private key: (d=3, n=33)
Ciphertext: 16
Decrypted message: 25
```

Conclusion

The RSA algorithm leverages the difficulty of factoring large numbers for security. This simulation uses small primes for simplicity, but real-world RSA requires large primes for robustness. It demonstrates public-key cryptography effectively, though computation is intensive for large numbers.

6. Simulation of Diffie-Hellman Key Exchange Algorithm

Title

Simulation of Diffie-Hellman Key Exchange

Algorithm

1. **Input**: Public prime number (p), base (g), private keys for Alice (a) and Bob (b).
2. **Key Exchange**:
    - Alice computes A = g^a mod p and sends A to Bob.
    - Bob computes B = g^b mod p and sends B to Alice.
    - Alice computes shared key K = B^a mod p.
    - Bob computes shared key K = A^b mod p.
3. **Output**: Public values and shared secret key.

Source Code

```
#include <stdio.h>
long long mod_pow(long long base, long long exp, long long mod) {
  long long result = 1;
  base %= mod;
  while (exp > 0) {
    if (exp & 1) result = (result * base) % mod;
    base = (base * base) % mod;
    exp >>= 1;    }
  return result; }
int main() {
  long long p, g, a, b;
  printf("Enter prime number (p): ");
  scanf("%lld", &p);
   printf("Enter base (g): ");
```

```
    scanf("%lld", &g);
    printf("Enter Alice's private key (a): ");
    scanf("%lld", &a);
    printf("Enter Bob's private key (b): ");
    scanf("%lld", &b);
    long long A = mod_pow(g, a, p); // Alice's public key
    long long B = mod_pow(g, b, p); // Bob's public key
    long long key_alice = mod_pow(B, a, p); // Alice's shared key
    long long key_bob = mod_pow(A, b, p); // Bob's shared key
    printf("Public values: p=%lld, g=%lld\n", p, g);
    printf("Alice's public key: %lld\n", A);
    printf("Bob's public key: %lld\n", B);
    printf("Shared secret key (Alice): %lld\n", key_alice);
    printf("Shared secret key (Bob): %lld\n", key_bob);
    return 0; }
```

OUTPUT

```
Enter prime number (p): 23
Enter base (g): 5
Enter Alice's private key (a): 6
Enter Bob's private key (b): 15
Public values: p=23, g=5
Alice's public key: 8
Bob's public key: 19
Shared secret key (Alice): 2
Shared secret key (Bob): 2
```

Conclusion: The Diffie-Hellman algorithm enables secure key exchange over an insecure channel using modular exponentiation. This simulation shows how both parties compute the same shared key without revealing their private keys. Its security relies on the discrete logarithm problem, though it requires large primes in practice.