

Implementation and Analysis of Classical Cipher Mechanisms in C++



Muhammad Ismail 2023452

Mahad Aqeel 2023286

Afeef Bari 2023356

Shayan 2023656

Information Security CY-211

Abstract

The following work focuses on analyzing the feasibility of using different kinds of classical cipher algorithms in C++. The primary focus is on three cipher types which include Symmetric, Asymmetric and Hybrid. Symmetric includes Caesar Cipher, Rail Fence Cipher, Playfair Cipher, Vigenère Cipher, OTP whereas asymmetric includes RSA encryption. The working of each cipher is described with regards to both encryption and decryption; the principles of mathematics underlying each cipher are also discussed on this basis of code snippets.

A general outline of the advantages and disadvantages of each cipher's algorithm is discussed and this is followed by the actual implementation of these ciphers using C++ programs. The project also discusses these ciphers in relation to the complexity and the time taken to encrypt and decrypt a message.

Furthermore, the program enables users to type plaintext and choose cipher to get the corresponding cipher text. The findings are expected to improve the knowledge of the casual reader on cryptographic concepts as well as the embedding of an actual code in C++. From these concepts students will be able to understand the current issues and developments taking place in information security.

TABLE OF CONTENT

Sr. No	Content
1.	Project Introduction
2.	Advantages and Disadvantages of Ciphering Mechanisms
3.	Flow Diagram
4.	Code Snippets
5.	Time Complexities
6.	Best Performing Encryption Technique
7.	Key Achievements of Project
8.	Future Improvements and Recommendations
9.	Final Thoughts

1. Project Introduction

The modern world is seeing itself involved in issues of information security more and more often. A major component of data security, cryptography is the science of protecting communications via the use of code. This project offers a sound and informative Cryptography System which can show diverse encryption methods, so that people have a better understanding of the basics of data protection.

How the Project Will Work

The project on hand for evaluation is a menu-based simple encryption system that is user input based where they can select and apply different types of encryption to the input text. The current model provides a user-friendly interface where the user may test various forms of encryption for different cryptography purposes.

Functional Description

Upon launching the system, users are presented with a main menu to select the type of encryption they wish to perform:

- **Symmetric Encryption:** Has a single key – encryption and decryption both are performed quickly and effectively yet the key has to be shared safely.
- **Asymmetric Encryption:** Uses a combination of a public and a private key improving the security of systems used in communication as well as the authentication phase but it is slow due to its complexity.
- **Hybrid Encryption:** An amalgamation of the two: E.g AES protects the transfer of a session key used for acceleration of data encryption. This approach is efficient enough, and at the same time provoking a very high level of security, and that is why this approach is suggested to be applied to secure communication systems.

After selecting a category, users can choose from a range of specific encryption methods, including but not limited to:

- Symmetric Methods: There is Caesar Cipher, Vigenère Cipher, Rail Fence Cipher, Playfair Cipher, One-Time Pad.
- Asymmetric Methods: RSA Encryption.
- Hybrid Methods: A mix of techniques under one category, symmetric and one in the asymmetric category.

2. Advantages and Disadvantages of the Cipher Mechanisms

Caesar Cipher

Advantages:

- Simple to implement and understand.
- Minimal computational power required.

- Useful for educational purposes to demonstrate basic encryption.

Disadvantages:

- Extremely insecure; can be broken easily using frequency analysis.
 - Fixed shift provides minimal complexity.
 - Not suitable for modern applications.
-

Vigenère Cipher

Advantages:

- More secure than Caesar Cipher due to polyalphabetic substitution.
- Resistant to simple frequency analysis.
- Easy to implement with manageable keys.

Disadvantages:

- Vulnerable to key-repetition attacks (e.g., Kasiski examination).
 - Key management can be challenging for longer texts.
 - Not suitable for large-scale secure communication.
-

Rail Fence Cipher

Advantages:

- Easy to implement and understand.
- Provides obfuscation through transposition.
- Effective for simple scenarios with low-security needs.

Disadvantages:

- Vulnerable to brute-force attacks if the number of rows is small.
 - Easily broken with known plaintext-ciphertext pairs.
 - Lacks key strength for modern cryptographic needs.
-

Playfair Cipher

Advantages:

- Encrypts digraphs (pairs of letters), making it more secure than monoalphabetic ciphers.
- Resistant to frequency analysis of individual letters.
- Simple to use with a predefined key matrix.

Disadvantages:

- Vulnerable to frequency analysis of digraphs.
 - Limited security compared to modern encryption methods.
 - Requires a pre-shared key for both parties.
-

One-Time Pad

Advantages:

- It is provably secure only if the key is indeed random and used only once.
- Provides perfect secrecy.
- Impossible to crack without access to the key.

Disadvantages:

- Must be used with a key as long as the message, the distribution of the key becomes complicated.
 - Vulnerable if the key is reused.
 - Impractical for frequent or large-scale communication.
-

RSA Encryption

Advantages:

- Provides strong security through large key sizes.
- Eliminates the need for pre-shared keys using public/private key pairs.
- Widely used in secure communications and digital signatures.

Disadvantages:

- Computationally expensive compared to symmetric encryption.
 - Key generation and management are complex.
 - Vulnerable to quantum computing attacks in the future.
-

Hybrid Method

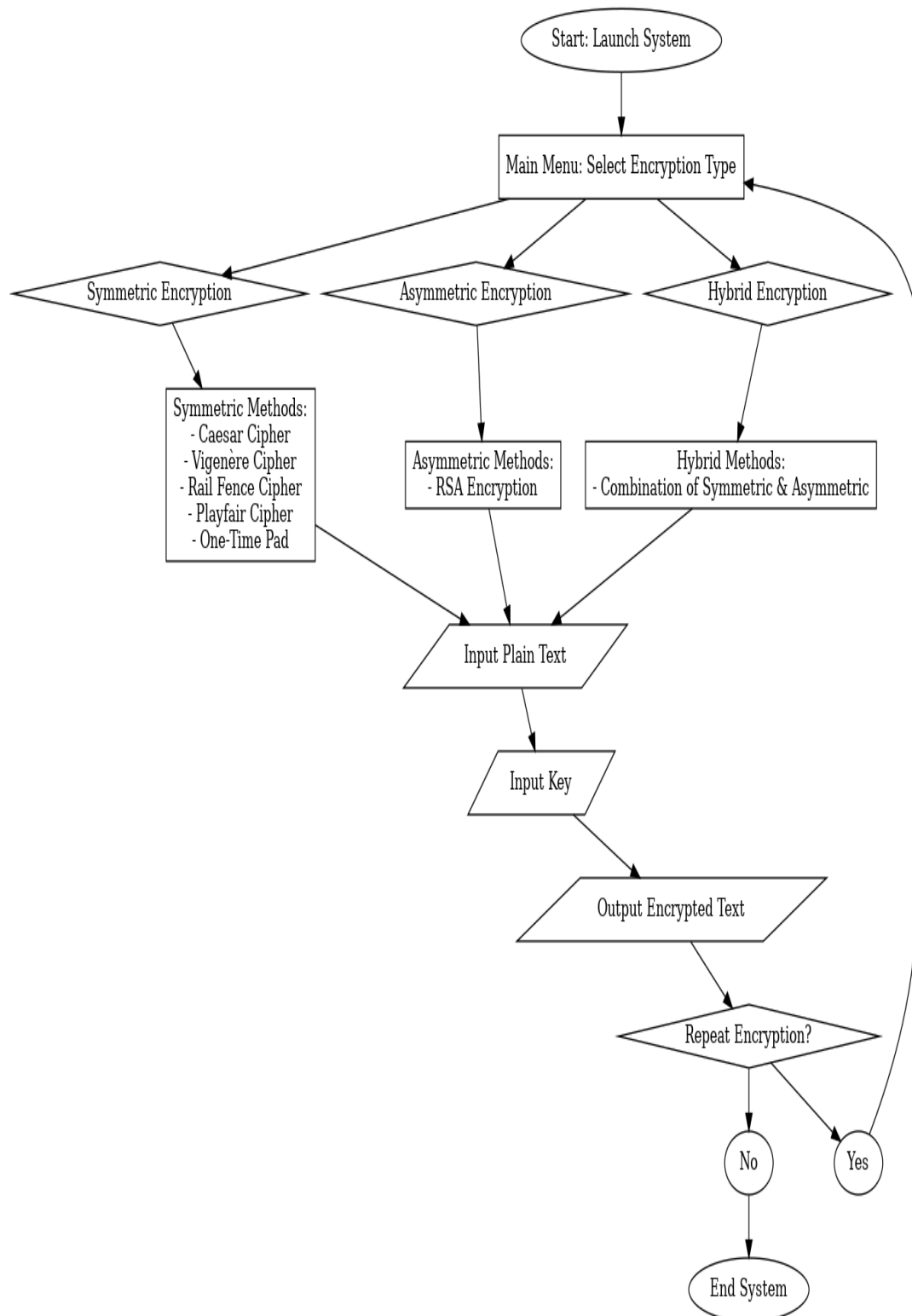
Advantages:

- Balances efficiency (symmetric) and security (asymmetric).
- Facilitates secure key exchange using asymmetric methods while encrypting data with faster symmetric methods.
- Suitable for modern communication protocols like HTTPS.

Disadvantages:

- Increases system complexity.
- Relies on proper implementation of both encryption types.
- Still vulnerable to attacks if one component is weak.

3. Flow Diagram



4. Code Snippets

```
using namespace std;

// Simple symmetric encryption using XOR
string xorEncrypt(const string &data, char key)
{
    string encrypted = data;
    for (size_t i = 0; i < data.size(); i++)
    {
        // XOR with the key
        encrypted[i] ^= key;
    }
    return encrypted;
}

// Simple RSA-like key generation
pair<int, int> generateKeys()
{
    // Example prime number
    int p = 61;

    // Example prime number
    int q = 53;

    // Modulus for public and private keys
    int n = p * q;

    // Euler's totient function
    int phi = (p - 1) * (q - 1);

    // Public exponent (must be coprime with phi)
    int e = 17;

    // Private exponent (calculated using modular inverse of e mod phi)
    int d = 2753;

    // Return public key (e, n)
    return {e, n};
}

// Simple RSA-like decryption
int rsaDecrypt(int cipher, int d, int n)
{
    int result = 1;
    for (int i = 0; i < d; i++)
    {
        // Modular exponentiation
        result = (result * cipher) % n;
    }
    return result;
}

// Convert string to integer
int stringToInt(const string &str)
{
    int result = 0;
    for (char c : str)
    {
        result = result * 256 + c;
    }
    return result;
}

// Convert integer back to string
string intToString(int num)
{
    string result;
    while (num > 0)
    {
        result = char(num % 256) + result;
        num /= 256;
    }
    return result;
}

// Function to encrypt the plaintext using Rail Fence Cipher
string railfenceencrypt(string plaintext, int key)
{
    if (key <= 1)
    {
        // No encryption for key <= 1
        return plaintext;
    }

    // Create a vector of strings for each rail
    vector<string> rail(key);

    // 1 means moving down, -1 means moving up
    int direction = 1;
    int row = 0;

    // Fill the rails with characters in a zigzag pattern
    for (char ch : plaintext)
    {
        rail[row] += ch;
        row += direction;
        if (row == 0 || row == key - 1)
        {
            direction *= -1;
        }
    }

    // Combine all rails into the ciphertext
    string ciphertext = "";
    for (const string &r : rail)
    {
        ciphertext += r;
    }

    return ciphertext;
}

// Function to decrypt the ciphertext using Rail Fence Cipher
string railfencedecrypt(string ciphertext, int key)
{
    if (key <= 1)
    {
        // No decryption for key <= 1
        return ciphertext;
    }

    // Track the number of characters in each rail
    vector<int> raillengths(key, 0);

    // 1 means moving down, -1 means moving up
    int direction = 1;
    int row = 0;

    // Fill the rails with characters in a zigzag pattern
    for (char ch : ciphertext)
    {
        raillengths[row]++;
        row += direction;
        if (row == 0 || row == key - 1)
        {
            direction *= -1;
        }
    }

    // Decrypt the ciphertext
    string plaintext = "";
    for (int i = 0; i < ciphertext.size(); i++)
    {
        int rail = i % key;
        if (raillengths[rail] > 0)
        {
            plaintext += ciphertext[i];
            raillengths[rail]--;
        }
    }

    return plaintext;
}
```



```
C:\Users\Admin\Desktop > @ ISProject > main()
135 string railfenceencrypt(string ciphertext, int key)
144 // 1 means moving down, -1 means moving up
145 int direction = 1;
146 int row = 0;
147
148 // Calculate the length of each rail based on the zigzag pattern
149 for (char ch : ciphertext)
150 {
151     raillengths[row]++;
152     row += direction;
153     if (row == 0 || row == key - 1)
154     {
155         direction *= -1;
156     }
157 }
158
159 // Fill the rails with characters from the ciphertext
160 vector<string> rail(key);
161 int index = 0;
162 for (int i = 0; i < key; i++)
163 {
164     rail[i] = ciphertext.substr(index, raillengths[i]);
165     index += raillengths[i];
166 }
167
168 // Reconstruct the plaintext by following the zigzag pattern
169 string plaintext = "";
170 row = 0;
171 direction = 1;
172 for (size_t i = 0; i < ciphertext.length(); i++)
173 {
174     // Take the first character of the current rail
175     plaintext += rail[row][0];
176     // Remove the used character
177     rail[row].substr(1);
178 }
179
180 // Function to generate the full key based on the plaintext length
181 string generateKey(string plaintext, string key)
182 {
183     int textlength = plaintext.length();
184     int keylength = key.length();
185     string fullkey = key;
186     // Repeat the key until it matches the length of the plaintext
187     for (int i = 0; fullkey.length() < textlength; i++)
188     {
189         fullkey += key[i % keylength];
190     }
191     return fullkey;
192 }
193
194 // Function to encrypt the plaintext using Vigenere Cipher
195 string vigenereencrypt(string plaintext, string key)
196 {
197     string ciphertext = "";
198     key = generateKey(plaintext, key);
199     for (size_t i = 0; i < plaintext.length(); i++)
200     {
201         if (isalpha(plaintext[i]))
202         {
203             char base = islower(plaintext[i]) ? 'a' : 'A';
204             char shift = islower(key[i]) ? 'a' : 'A';
205             ciphertext += (plaintext[i] - base + shift) % 26 + base;
206         }
207         else
208         {
209             ciphertext += plaintext[i];
210         }
211     }
212     return ciphertext;
213 }
214
215 // Function to decrypt the ciphertext using Vigenere Cipher
216 string vigeneredecrypt(string ciphertext, string key)
217 {
218     string plaintext = "";
219     key = generateKey(ciphertext, key);
220     for (size_t i = 0; i < ciphertext.length(); i++)
221     {
222         if (isalpha(ciphertext[i]))
223         {
224             char base = islower(ciphertext[i]) ? 'a' : 'A';
225             char shift = islower(key[i]) ? 'a' : 'A';
226             plaintext += (ciphertext[i] - base - shift + 26) % 26 + base;
227         }
228         else
229         {
230             plaintext += ciphertext[i];
231         }
232     }
233     return plaintext;
234 }
235
236 // Function to encrypt the plaintext using Caesar Cipher
237 string caesar_encrypt(string text, int shift)
238 {
239     string result = "";
240     for (char ch : text)
241     {
242         if (isalpha(ch))
243         {
244             char base = islower(ch) ? 'a' : 'A';
245             result += (ch - base + shift) % 26 + base;
246         }
247         else
248         {
249             result += ch;
250         }
251     }
252     return result;
253 }
254
255 // Function to decrypt the ciphertext using Caesar Cipher
256 string caesar_decrypt(string text, int shift)
257 {
258     // Reverse the shift for decryption
259     return caesar_encrypt(text, 26 - (shift % 26));
260 }
261
262 // Helper function to calculate determinant of a 2x2 matrix
263 int determinant(int matrix[2][2])
264 {
265     return (matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]);
266 }
267
268 // Helper function to find the adjugate of a 2x2 matrix
269 void adjugate(int matrix[2][2], int adj[2][2])
270 {
271     adj[0][0] = matrix[1][1];
272     adj[0][1] = -matrix[0][1];
273     adj[1][0] = -matrix[1][0];
274     adj[1][1] = matrix[0][0];
275 }
276
277 // Function to compute modular inverse of a number under modulo m for rsa
278 int modInverse(int a, int m)
279 {
280     a = a % m;
281     for (int x = 1; x < m; x++)
282     {
283         if ((a * x) % m == 1)
284             return x;
285     }
286     // Inverse does not exist
287     return -1;
288 }
289
290 // Efficient modular exponentiation
291 int modularExponentiation(int base, int exp, int mod)
292 {
293     int result = 1;
294     // Reduce base if it's greater than mod
295     base = base % mod;
296     while (exp > 0)
297     {
298         // If exponent is odd, multiply the base with the result
299         if (exp % 2 == 1)
300         {
301             result = (result * base) % mod;
302         }
303         // Square the base and halve the exponent
304         base = (base * base) % mod;
305         exp /= 2;
306     }
307     return result;
308 }
309
310 // Function to generate a random key of the same length as the plaintext
311 string generateRandomKey(const string &plaintext)
312 {
313     string key = "";
314     // Random number generator
315     mt19937 gen(static_cast<unsigned long>(time(nullptr)));
316     // Generate random values in the byte range (0-255)
317     uniform_int_distribution<int> dist(0, 255);
318     for (size_t i = 0; i < plaintext.length(); i++)
319     {
320         // Random number Generator
321         key += static_cast<char>(dist(gen));
322     }
323     return key;
324 }
325
326 // Function to generate the full key based on the plaintext length
327 string generateKey(string plaintext, string key)
328 {
329     int textlength = plaintext.length();
330     int keylength = key.length();
331     string fullkey = key;
332     // Repeat the key until it matches the length of the plaintext
333     for (int i = 0; fullkey.length() < textlength; i++)
334     {
335         fullkey += key[i % keylength];
336     }
337     return fullkey;
338 }
339
340 // Function to encrypt the plaintext using Vigenere Cipher
341 string vigenereencrypt(string plaintext, string key)
342 {
343     string ciphertext = "";
344     key = generateKey(plaintext, key);
345     for (size_t i = 0; i < plaintext.length(); i++)
346     {
347         if (isalpha(plaintext[i]))
348         {
349             char base = islower(plaintext[i]) ? 'a' : 'A';
350             char shift = islower(key[i]) ? 'a' : 'A';
351             ciphertext += (plaintext[i] - base + shift) % 26 + base;
352         }
353         else
354         {
355             ciphertext += plaintext[i];
356         }
357     }
358     return ciphertext;
359 }
360
361 // Function to decrypt the ciphertext using Vigenere Cipher
362 string vigeneredecrypt(string ciphertext, string key)
363 {
364     string plaintext = "";
365     key = generateKey(ciphertext, key);
366     for (size_t i = 0; i < ciphertext.length(); i++)
367     {
368         if (isalpha(ciphertext[i]))
369         {
370             char base = islower(ciphertext[i]) ? 'a' : 'A';
371             char shift = islower(key[i]) ? 'a' : 'A';
372             plaintext += (ciphertext[i] - base - shift + 26) % 26 + base;
373         }
374         else
375         {
376             plaintext += ciphertext[i];
377         }
378     }
379     return plaintext;
380 }
381
382 // Function to encrypt the plaintext using Caesar Cipher
383 string caesar_encrypt(string text, int shift)
384 {
385     string result = "";
386     for (char ch : text)
387     {
388         if (isalpha(ch))
389         {
390             char base = islower(ch) ? 'a' : 'A';
391             result += (ch - base + shift) % 26 + base;
392         }
393         else
394         {
395             result += ch;
396         }
397     }
398     return result;
399 }
400
401 // Function to decrypt the ciphertext using Caesar Cipher
402 string caesar_decrypt(string text, int shift)
403 {
404     // Reverse the shift for decryption
405     return caesar_encrypt(text, 26 - (shift % 26));
406 }
407
408 // Helper function to calculate determinant of a 2x2 matrix
409 int determinant(int matrix[2][2])
410 {
411     return (matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]);
412 }
413
414 // Helper function to find the adjugate of a 2x2 matrix
415 void adjugate(int matrix[2][2], int adj[2][2])
416 {
417     adj[0][0] = matrix[1][1];
418     adj[0][1] = -matrix[0][1];
419     adj[1][0] = -matrix[1][0];
420     adj[1][1] = matrix[0][0];
421 }
422
423 // Function to compute modular inverse of a number under modulo m for rsa
424 int modInverse(int a, int m)
425 {
426     a = a % m;
427     for (int x = 1; x < m; x++)
428     {
429         if ((a * x) % m == 1)
430             return x;
431     }
432     // Inverse does not exist
433     return -1;
434 }
435
436 // Efficient modular exponentiation
437 int modularExponentiation(int base, int exp, int mod)
438 {
439     int result = 1;
440     // Reduce base if it's greater than mod
441     base = base % mod;
442     while (exp > 0)
443     {
444         // If exponent is odd, multiply the base with the result
445         if (exp % 2 == 1)
446         {
447             result = (result * base) % mod;
448         }
449         // Square the base and halve the exponent
450         base = (base * base) % mod;
451         exp /= 2;
452     }
453     return result;
454 }
455
456 // Function to generate a random key of the same length as the plaintext
457 string generateRandomKey(const string &plaintext)
458 {
459     string key = "";
460     // Random number generator
461     mt19937 gen(static_cast<unsigned long>(time(nullptr)));
462     // Generate random values in the byte range (0-255)
463     uniform_int_distribution<int> dist(0, 255);
464     for (size_t i = 0; i < plaintext.length(); i++)
465     {
466         // Random number Generator
467         key += static_cast<char>(dist(gen));
468     }
469     return key;
470 }
471
472 // Function to generate the full key based on the plaintext length
473 string generateKey(string plaintext, string key)
474 {
475     int textlength = plaintext.length();
476     int keylength = key.length();
477     string fullkey = key;
478     // Repeat the key until it matches the length of the plaintext
479     for (int i = 0; fullkey.length() < textlength; i++)
480     {
481         fullkey += key[i % keylength];
482     }
483     return fullkey;
484 }
485
486 // Function to encrypt the plaintext using Vigenere Cipher
487 string vigenereencrypt(string plaintext, string key)
488 {
489     string ciphertext = "";
490     key = generateKey(plaintext, key);
491     for (size_t i = 0; i < plaintext.length(); i++)
492     {
493         if (isalpha(plaintext[i]))
494         {
495             char base = islower(plaintext[i]) ? 'a' : 'A';
496             char shift = islower(key[i]) ? 'a' : 'A';
497             ciphertext += (plaintext[i] - base + shift) % 26 + base;
498         }
499         else
500         {
501             ciphertext += plaintext[i];
502         }
503     }
504     return ciphertext;
505 }
506
507 // Function to decrypt the ciphertext using Vigenere Cipher
508 string vigeneredecrypt(string ciphertext, string key)
509 {
510     string plaintext = "";
511     key = generateKey(ciphertext, key);
512     for (size_t i = 0; i < ciphertext.length(); i++)
513     {
514         if (isalpha(ciphertext[i]))
515         {
516             char base = islower(ciphertext[i]) ? 'a' : 'A';
517             char shift = islower(key[i]) ? 'a' : 'A';
518             plaintext += (ciphertext[i] - base - shift + 26) % 26 + base;
519         }
520         else
521         {
522             plaintext += ciphertext[i];
523         }
524     }
525     return plaintext;
526 }
527
528 // Function to encrypt the plaintext using Caesar Cipher
529 string caesar_encrypt(string text, int shift)
530 {
531     string result = "";
532     for (char ch : text)
533     {
534         if (isalpha(ch))
535         {
536             char base = islower(ch) ? 'a' : 'A';
537             result += (ch - base + shift) % 26 + base;
538         }
539         else
540         {
541             result += ch;
542         }
543     }
544     return result;
545 }
546
547 // Function to decrypt the ciphertext using Caesar Cipher
548 string caesar_decrypt(string text, int shift)
549 {
550     // Reverse the shift for decryption
551     return caesar_encrypt(text, 26 - (shift % 26));
552 }
553
554 // Helper function to calculate determinant of a 2x2 matrix
555 int determinant(int matrix[2][2])
556 {
557     return (matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]);
558 }
559
560 // Helper function to find the adjugate of a 2x2 matrix
561 void adjugate(int matrix[2][2], int adj[2][2])
562 {
563     adj[0][0] = matrix[1][1];
564     adj[0][1] = -matrix[0][1];
565     adj[1][0] = -matrix[1][0];
566     adj[1][1] = matrix[0][0];
567 }
568
569 // Function to compute modular inverse of a number under modulo m for rsa
570 int modInverse(int a, int m)
571 {
572     a = a % m;
573     for (int x = 1; x < m; x++)
574     {
575         if ((a * x) % m == 1)
576             return x;
577     }
578     // Inverse does not exist
579     return -1;
580 }
581
582 // Efficient modular exponentiation
583 int modularExponentiation(int base, int exp, int mod)
584 {
585     int result = 1;
586     // Reduce base if it's greater than mod
587     base = base % mod;
588     while (exp > 0)
589     {
590         // If exponent is odd, multiply the base with the result
591         if (exp % 2 == 1)
592         {
593             result = (result * base) % mod;
594         }
595         // Square the base and halve the exponent
596         base = (base * base) % mod;
597         exp /= 2;
598     }
599     return result;
600 }
601
602 // Function to generate a random key of the same length as the plaintext
603 string generateRandomKey(const string &plaintext)
604 {
605     string key = "";
606     // Random number generator
607     mt19937 gen(static_cast<unsigned long>(time(nullptr)));
608     // Generate random values in the byte range (0-255)
609     uniform_int_distribution<int> dist(0, 255);
610     for (size_t i = 0; i < plaintext.length(); i++)
611     {
612         // Random number Generator
613         key += static_cast<char>(dist(gen));
614     }
615     return key;
616 }
617
618 // Function to generate the full key based on the plaintext length
619 string generateKey(string plaintext, string key)
620 {
621     int textlength = plaintext.length();
622     int keylength = key.length();
623     string fullkey = key;
624     // Repeat the key until it matches the length of the plaintext
625     for (int i = 0; fullkey.length() < textlength; i++)
626     {
627         fullkey += key[i % keylength];
628     }
629     return fullkey;
630 }
631
632 // Function to encrypt the plaintext using Vigenere Cipher
633 string vigenereencrypt(string plaintext, string key)
634 {
635     string ciphertext = "";
636     key = generateKey(plaintext, key);
637     for (size_t i = 0; i < plaintext.length(); i++)
638     {
639         if (isalpha(plaintext[i]))
640         {
641             char base = islower(plaintext[i]) ? 'a' : 'A';
642             char shift = islower(key[i]) ? 'a' : 'A';
643             ciphertext += (plaintext[i] - base + shift) % 26 + base;
644         }
645         else
646         {
647             ciphertext += plaintext[i];
648         }
649     }
650     return ciphertext;
651 }
652
653 // Function to decrypt the ciphertext using Vigenere Cipher
654 string vigeneredecrypt(string ciphertext, string key)
655 {
656     string plaintext = "";
657     key = generateKey(ciphertext, key);
658     for (size_t i = 0; i < ciphertext.length(); i++)
659     {
660         if (isalpha(ciphertext[i]))
661         {
662             char base = islower(ciphertext[i]) ? 'a' : 'A';
663             char shift = islower(key[i]) ? 'a' : 'A';
664             plaintext += (ciphertext[i] - base - shift + 26) % 26 + base;
665         }
666         else
667         {
668             plaintext += ciphertext[i];
669         }
670     }
671     return plaintext;
672 }
673
674 // Function to encrypt the plaintext using Caesar Cipher
675 string caesar_encrypt(string text, int shift)
676 {
677     string result = "";
678     for (char ch : text)
679     {
680         if (isalpha(ch))
681         {
682             char base = islower(ch) ? 'a' : 'A';
683             result += (ch - base + shift) % 26 + base;
684         }
685         else
686         {
687             result += ch;
688         }
689     }
690     return result;
691 }
692
693 // Function to decrypt the ciphertext using Caesar Cipher
694 string caesar_decrypt(string text, int shift)
695 {
696     // Reverse the shift for decryption
697     return caesar_encrypt(text, 26 - (shift % 26));
698 }
699
700 // Helper function to calculate determinant of a 2x2 matrix
701 int determinant(int matrix[2][2])
702 {
703     return (matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]);
704 }
705
706 // Helper function to find the adjugate of a 2x2 matrix
707 void adjugate(int matrix[2][2], int adj[2][2])
708 {
709     adj[0][0] = matrix[1][1];
710     adj[0][1] = -matrix[0][1];
711     adj[1][0] = -matrix[1][0];
712     adj[1][1] = matrix[0][0];
713 }
714
715 // Function to compute modular inverse of a number under modulo m for rsa
716 int modInverse(int a, int m)
717 {
718     a = a % m;
719     for (int x = 1; x < m; x++)
720     {
721         if ((a * x) % m == 1)
722             return x;
723     }
724     // Inverse does not exist
725     return -1;
726 }
727
728 // Efficient modular exponentiation
729 int modularExponentiation(int base, int exp, int mod)
730 {
731     int result = 1;
732     // Reduce base if it's greater than mod
733     base = base % mod;
734     while (exp > 0)
735     {
736         // If exponent is odd, multiply the base with the result
737         if (exp % 2 == 1)
738         {
739             result = (result * base) % mod;
740         }
741         // Square the base and halve the exponent
742         base = (base * base) % mod;
743         exp /= 2;
744     }
745     return result;
746 }
747
748 // Function to generate a random key of the same length as the plaintext
749 string generateRandomKey(const string &plaintext)
750 {
751     string key = "";
752     // Random number generator
753     mt19937 gen(static_cast<unsigned long>(time(nullptr)));
754     // Generate random values in the byte range (0-255)
755     uniform_int_distribution<int> dist(0, 255);
756     for (size_t i = 0; i < plaintext.length(); i++)
757     {
758         // Random number Generator
759         key += static_cast<char>(dist(gen));
760     }
761     return key;
762 }
763
764 // Function to generate the full key based on the plaintext length
765 string generateKey(string plaintext, string key)
766 {
767     int textlength = plaintext.length();
768     int keylength = key.length();
769     string fullkey = key;
770     // Repeat the key until it matches the length of the plaintext
771     for (int i = 0; fullkey.length() < textlength; i++)
772     {
773         fullkey += key[i % keylength];
774     }
775     return fullkey;
776 }
777
778 // Function to encrypt the plaintext using Vigenere Cipher
779 string vigenereencrypt(string plaintext, string key)
780 {
781     string ciphertext = "";
782     key = generateKey(plaintext, key);
783     for (size_t i = 0; i < plaintext.length(); i++)
784     {
785         if (isalpha(plaintext[i]))
786         {
787             char base = islower(plaintext[i]) ? 'a' : 'A';
788             char shift = islower(key[i]) ? 'a' : 'A';
789             ciphertext += (plaintext[i] - base + shift) % 26 + base;
790         }
791         else
792         {
793             ciphertext += plaintext[i];
794         }
795     }
796     return ciphertext;
797 }
798
799 // Function to decrypt the ciphertext using Vigenere Cipher
800 string vigeneredecrypt(string ciphertext, string key)
801 {
802     string plaintext = "";
803     key = generateKey(ciphertext, key);
804     for (size_t i = 0; i < ciphertext.length(); i++)
805     {
806         if (isalpha(ciphertext[i]))
807         {
808             char base = islower(ciphertext[i]) ? 'a' : 'A';
809             char shift = islower(key[i]) ? 'a' : 'A';
810             plaintext += (ciphertext[i] - base - shift + 26) % 26 + base;
811         }
812         else
813         {
814             plaintext += ciphertext[i];
815         }
816     }
817     return plaintext;
818 }
819
820 // Function to encrypt the plaintext using Caesar Cipher
821 string caesar_encrypt(string text, int shift)
822 {
823     string result = "";
824     for (char ch : text)
825     {
826         if (isalpha(ch))
827         {
828             char base = islower(ch) ? 'a' : 'A';
829             result += (ch - base + shift) % 26 + base;
830         }
831         else
832         {
833             result += ch;
834         }
835     }
836     return result;
837 }
838
839 // Function to decrypt the ciphertext using Caesar Cipher
840 string caesar_decrypt(string text, int shift)
841 {
842     // Reverse the shift for decryption
843     return caesar_encrypt(text, 26 - (shift % 26));
844 }
845
846 // Helper function to calculate determinant of a 2x2 matrix
847 int determinant(int matrix[2][2])
848 {
849     return (matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]);
850 }
851
852 // Helper function to find the adjugate of a 2x2 matrix
853 void adjugate(int matrix[2][2], int adj[2][2])
854 {
855     adj[0][0] = matrix[1][1];
856     adj[0][1] = -matrix[0][1];
857     adj[1][0] = -matrix[1][0];
858     adj[1][1] = matrix[0][0];
859 }
860
861 // Function to compute modular inverse of a number under modulo m for rsa
862 int modInverse(int a, int m)
863 {
864     a = a % m;
865     for (int x = 1; x < m; x++)
866     {
867         if ((a * x) % m == 1)
868             return x;
869     }
870     // Inverse does not exist
871     return -1;
872 }
873
874 // Efficient modular exponentiation
875 int modularExponentiation(int base, int exp, int mod)
876 {
877     int result = 1;
878     // Reduce base if it's greater than mod
879     base = base % mod;
880     while (exp > 0)
881     {
882         // If exponent is odd, multiply the base with the result
883         if (exp % 2 == 1)
884         {
885             result = (result * base) % mod;
886         }
887         // Square the base and halve the exponent
888         base = (base * base) % mod;
889         exp /= 2;
890     }
891     return result;
892 }
893
894 // Function to generate a random key of the same length as the plaintext
895 string generateRandomKey(const string &plaintext)
896 {
897     string key = "";
898     // Random number generator
899     mt19937 gen(static_cast<unsigned long>(time(nullptr)));
900     // Generate random values in the byte range (0-255)
901     uniform_int_distribution<int> dist(0, 255);
902     for (size_t i = 0; i < plaintext.length(); i++)
903     {
904         // Random number Generator
905         key += static_cast<char>(dist(gen));
906     }
907     return key;
908 }
909
910 // Function to generate the full key based on the plaintext length
911 string generateKey(string plaintext, string key)
912 {
913     int textlength = plaintext.length();
914     int keylength = key.length();
915     string fullkey = key;
916     // Repeat the key until it matches the length of the plaintext
917     for (int i = 0; fullkey.length() < textlength; i++)
918     {
919         fullkey += key[i % keylength];
920     }
921     return fullkey;
922 }
923
924 // Function to encrypt the plaintext using Vigenere Cipher
925 string vigenereencrypt(string plaintext, string key)
926 {
927     string ciphertext = "";
928     key = generateKey(plaintext, key);
929     for (size_t i = 0; i < plaintext.length(); i++)
930     {
931         if (isalpha(plaintext[i]))
932         {
933             char base = islower(plaintext[i]) ? 'a' : 'A';
934             char shift = islower(key[i]) ? 'a' : 'A';
935             ciphertext += (plaintext[i] - base + shift) % 26 + base;
936         }
937         else
938         {
939             ciphertext += plaintext[i];
940         }
941     }
942     return ciphertext;
943 }
944
945 // Function to decrypt the ciphertext using Vigenere Cipher
946 string vigeneredecrypt(string ciphertext, string key)
947 {
948     string plaintext = "";
949     key = generateKey(ciphertext, key);
950     for (size_t i = 0; i < ciphertext.length(); i++)
951     {
952         if (isalpha(ciphertext[i]))
953         {
954             char base = islower(ciphertext[i]) ? 'a' : 'A';
955             char shift = islower(key[i]) ? 'a' : 'A';
956             plaintext += (ciphertext[i] - base - shift + 26) % 26 + base;
957         }
958         else
959         {
960             plaintext += ciphertext[i];
961         }
962     }
963     return plaintext;
964 }
965
966 // Function to encrypt the plaintext using Caesar Cipher
967 string caesar_encrypt(string text, int shift)
968 {
969     string result = "";
970     for (char ch : text)
971     {
972         if (isalpha(ch))
973         {
974             char base = islower(ch) ? 'a' : 'A';
975             result += (ch - base + shift) % 26 + base;
976         }
977         else
978         {
979             result += ch;
980         }
981     }
982     return result;
983 }
984
985 // Function to decrypt the ciphertext using Caesar Cipher
986 string caesar_decrypt(string text, int shift)
987 {
988     // Reverse the shift for decryption
989     return caesar_encrypt(text, 26 - (shift % 26));
990 }
991
992 // Helper function to calculate determinant of a 2x2 matrix
993 int determinant(int matrix[2][2])
994 {
995     return (matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]);
996 }
997
998 // Helper function to find the adjugate of a 2x2 matrix
999 void adjugate(int matrix[2][2], int adj[2][2])
1000 {
1001     adj[0][0] = matrix[1][1];
1002     adj[0][1] = -matrix[0][1];
1003     adj[1][0] = -matrix[1][0];
1004     adj[1][1] = matrix[0][0];
1005 }
1006
1007 // Function to compute modular inverse of a number under modulo m for rsa
1008 int modInverse(int a, int m)
1009 {
1010     a = a % m;
1011     for (int x = 1; x < m; x++)
1012     {
1013         if ((a * x) % m == 1)
1014             return x;
1015     }
1016     // Inverse does not exist
1017     return -1;
1018 }
1019
1020 // Efficient modular exponentiation
1021 int modularExponentiation(int base, int exp, int mod)
1022 {
1023     int result = 1;
1024     // Reduce base if it's greater than mod
1025     base = base % mod;
1026     while (exp > 0)
1027     {
1028         // If exponent is odd, multiply the base with the result
1029         if (exp % 2 == 1)
1030         {
1031             result = (result * base) % mod;
1032         }
1033         // Square the base and halve the exponent
1034         base = (base * base) % mod;
1035         exp /= 2;
1036     }
1037     return result;
1038 }
1039
1040 // Function to generate a random key of the same length as the plaintext
1041 string generateRandomKey(const string &plaintext)
1042 {
1043     string key = "";
1044     // Random number generator
1045     mt19937 gen(static_cast<unsigned long>(time(nullptr)));
1046     // Generate random values in the byte range (0-255)
1047     uniform_int_distribution<int> dist(0, 255);
1048     for (size_t i = 0; i < plaintext.length(); i++)
1049     {
1050         // Random number Generator
1051         key += static_cast<char>(dist(gen));
1052     }
1053     return key;
1054 }
1055
1056 // Function to generate the full key based on the plaintext length
1057 string generateKey(string plaintext, string key)
1058 {
1059     int textlength = plaintext.length();
1060     int keylength = key.length();
1061     string fullkey = key;
1062     // Repeat the key until it matches the length of the plaintext
1063     for (int i = 0; fullkey.length() < textlength; i++)
1064     {
1065         fullkey += key[i % keylength];
1066     }
1067     return fullkey;
1068 }
1069
1070 // Function to encrypt the plaintext using Vigenere Cipher
1071 string vigenereencrypt(string plaintext, string key)
1072 {
1073     string ciphertext = "";
1074     key = generateKey(plaintext, key);
1075     for (size_t i = 0; i < plaintext.length(); i++)
1076     {
1077         if (isalpha(plaintext[i]))
1078         {
1079             char base = islower(plaintext[i]) ? 'a' : 'A';
1080             char shift = islower(key[i]) ? 'a' : 'A';
1081             ciphertext += (plaintext[i] - base + shift) % 26 + base;
1082         }
1083         else
1084         {
1085             ciphertext += plaintext[i];
1086         }
1087     }
1088     return ciphertext;
1089 }
1090
1091 // Function to decrypt the ciphertext using Vigenere Cipher
1092 string vigeneredecrypt(string ciphertext, string key)
1093 {
1094     string plaintext = "";
1095     key = generateKey(ciphertext, key);
1096     for (size_t i = 0; i < ciphertext.length(); i++)
1097     {
1098         if (isalpha(ciphertext[i]))
1099         {
1100             char base = islower(ciphertext[i]) ? 'a' : 'A';
1101             char shift = islower(key[i]) ? 'a' : 'A';
1102             plaintext += (ciphertext[i] - base - shift + 26) % 26 + base;
1103         }
1104         else
1105         {
1106             plaintext += ciphertext[i];
1107         }
1108     }
1109     return plaintext;
1110 }
1111
1112 // Function to encrypt the plaintext using Caesar Cipher
1113 string caesar_encrypt(string text, int shift)
1114 {
1115     string result = "";
1116     for (char ch : text)
1117     {
1118         if (isalpha(ch))
1119         {
1120             char base = islower(ch) ? 'a' : 'A';
1121             result += (ch - base + shift) % 26 + base;
1122         }
1123         else
1124         {
1125             result += ch;
1126         }
1127     }
1128     return result;
1129 }
1130
1131 // Function to decrypt the ciphertext using Caesar Cipher
1132 string caesar_decrypt(string text, int shift)
1133 {
1134     // Reverse the shift for decryption
1135     return caesar_encrypt(text, 26 - (shift % 26));
1136 }
1137
1138 // Helper function to calculate determinant of a 2x2 matrix
1139 int determinant(int matrix[2][2])
1140 {
1141     return (matrix[0][0] * matrix[1][1] - matrix[0][1] * matrix[1][0]);
1142 }
1143
1144 // Helper function to find the adjugate of a 2x2 matrix
1145 void adjugate(int matrix[2][2], int adj[2][2])
1146 {
1147     adj[0][0] = matrix[1][1];
1148     adj[0][1] = -matrix[0][1];
1149     adj[1][0] = -matrix[1][0];
1150     adj[1][1] = matrix[0][0];
1151 }
1152
1153 // Function to compute modular inverse of a number under modulo m for rsa
1154 int modInverse(int a, int m)
1155 {
1156     a = a % m;
1157     for (int x = 1; x < m; x++)
1158     {
1159         if ((a * x) % m == 1)
1160             return x;
1161     }
1162     // Inverse does not exist
1163     return -1;
1164 }
1165
1166 // Efficient modular exponentiation
1167 int modularExponentiation(int base, int exp, int mod)
1168 {
1169     int result = 1;
1170     // Reduce base if it's greater than mod
1171     base = base % mod;
1172     while (exp > 0)
1173     {
1174         // If exponent is odd, multiply the base with the result
1175         if (exp % 2 == 1)
1176         {
1177             result = (result * base) % mod;
1178         }
1179         // Square the base and halve the exponent
1180         base = (base * base) % mod;
1181         exp /= 2;
1182     }
1183     return result;
1184 }
1185
1186 // Function to generate a random key of the same length as the plaintext
1187 string generateRandomKey(const string &plaintext)
1188 {
1189     string key = "";
1190     // Random number generator
1191     mt19937 gen(static_cast<unsigned long>(time(nullptr)));
1192     // Generate random values in the byte range (0-255)
1193     uniform_int_distribution<int> dist(0, 255);
1194     for (size_t i = 0; i < plaintext.length(); i++)
1195     {
1196         // Random number Generator
1197         key += static_cast<char>(dist(gen));
1198     }
1199     return key;
1200 }
1201
1202 // Function to generate the full key based on the plaintext length
1203 string generateKey(string plaintext, string key)
1204 {
1205     int textlength = plaintext.length();
1206     int keylength = key.length();
1207     string fullkey = key;
1208     // Repeat the key until it matches the length of the plaintext
1209     for (int i = 0; fullkey.length() < textlength; i++)
1210     {
1211         fullkey += key[i % keylength];
1212     }
1213     return fullkey;
1214 }
1215
1216 // Function to encrypt the plaintext using Vigenere Cipher
1217 string vigenereencrypt(string plaintext, string key)
1218 {
1219     string ciphertext = "";
1220     key = generateKey(plaintext, key);
1221     for (size_t i = 0; i < plaintext.length(); i++)
1222     {
1223         if (isalpha(plaintext[i]))
1224         {
1225             char base = islower(plaintext[i]) ? 'a' : 'A';
1226             char shift = islower(key[i]) ? 'a' : 'A';
1227             ciphertext += (plaintext[i] - base + shift) % 26 + base;
1228         }
1229         else
1230         {
1231             ciphertext += plaintext[i];
1232         }
1233     }
1234     return ciphertext;
1235 }
1236
1237 // Function to decrypt the ciphertext using Vigenere Cipher
1238 string vigeneredecrypt(string ciphertext, string key
```

```
C:\Users> Admin > Desktop > C:\ISPjct.cpp > main()
346 }
347
348 // Function to encrypt the plaintext using the OTP
349 string encrypt(const string &plaintext, const string &key)
350 {
351     string ciphertext = "";
352
353     for (size_t i = 0; i < plaintext.length(); ++i)
354     {
355         // XOR each character of the plaintext with the key
356         ciphertext += plaintext[i] ^ key[i];
357     }
358     return ciphertext;
359 }
360
361 // Function to decrypt the ciphertext using the OTP
362 string decrypt(const string &ciphertext, const string &key)
363 {
364     string decryptedText = "";
365
366     for (size_t i = 0; i < ciphertext.length(); ++i)
367     {
368         // XOR again to get the original plaintext
369         decryptedText += ciphertext[i] ^ key[i];
370     }
371
372     return decryptedText;
373 }
374
375 // Function to compute the greatest common divisor (gcd)
376 int gcd(int a, int b)
377 {
378     if (b == 0)
379         return a;
380     return gcd(b, a % b);
381 }
382
383 // Function to generate RSA keys
384 void generateRSAKeys(int &n, int &e, int &d, int p, int q)
385 {
386     // Calculate n = p * q
387     n = p * q;
388
389     // Calculate Euler's totient function phi(n) = (p-1) * (q-1)
390     int phi_n = (p - 1) * (q - 1);
391
392     // Choose e such that 1 < e < phi(n) and gcd(e, phi(n)) = 1
393     e = 17;
394
395     // Calculate d such that (d * e) % phi(n) = 1
396     d = modInverse(e, phi_n);
397 }
```

```
C:\Users> Admin > Desktop > C:\ISPjct.cpp > main()
405 void generateRSAKeys(int &n, int &e, int &d, int p, int q)
406 {
407     // Calculate d such that (d * e) % phi(n) = 1 (d is modular inverse of e)
408     d = modInverse(e, phi_n);
409 }
410
411 // Function to convert a string to a vector of integers (based on ASCII value)
412 vector<int> stringToIntVector(const string &message)
413 {
414     vector<int> result;
415     for (char c : message)
416     {
417         result.push_back(static_cast<int>(c)); // Convert each character to its ASCII value
418     }
419     return result;
420 }
421
422 // Function to convert a vector of integers back to a string
423 string intVectorToString(const vector<int> &intVector)
424 {
425     string result;
426     for (int i : intVector)
427     {
428         // Convert each integer back to a character
429         result.push_back(static_cast<char>(i));
430     }
431     return result;
432 }
433
434 // Function to prepare the key matrix
435 void generateKeyMatrix(string key, char keyMatrix[5][5])
436 {
437     vector<bool> used(26, false);
438
439     // Treat 'J' and 'I' as the same
440     used['J' - 'A'] = true;
441
442     int row = 0, col = 0;
443
444     for (char c : key)
445     {
446         c = toupper(c);
447         if (used[c - 'A'] && !isalpha(c))
448             continue;
449         keyMatrix[row][col++] = c;
450         used[c - 'A'] = true;
451         if (col == 5)
452         {
453             col = 0;
454             row++;
455         }
456     }
457
458     for (char c = 'A'; c <= 'Z'; ++c)
459     {
460         if (used[c - 'A'])
461             continue;
462         keyMatrix[row][col++] = c;
463         used[c - 'A'] = true;
464         if (col == 5)
465         {
466             col = 0;
467             row++;
468         }
469     }
470 }
471
472 // Function to prepare the plaintext for Playfair cipher rules
473 string prepareText(string text)
474 {
475     string prepared = "";
476
477     for (char c : text)
478     {
479         if (isalpha(c))
480         {
481             c = toupper(c);
482             if (c == 'J')
483                 c = 'I';
484             prepared += c;
485         }
486     }
487
488     for (size_t i = 0; i < prepared.length() - 1; i += 2)
489     {
490         if (prepared[i] == prepared[i + 1])
491         {
492             // Add filler 'X' for duplicate letters
493             prepared.insert(i + 1, "X");
494         }
495         if (prepared.length() % 2 != 0)
496         {
497             // Add filler 'X' if length is odd
498             prepared += "X";
499         }
500     }
501     return prepared;
502 }
503
504 // Function to find position of a character in the key matrix
505 void findPosition(char keyMatrix[5][5], char c, int &row, int &col)
506 {
507     for (int i = 0; i < 5; ++i)
508     {
509         for (int j = 0; j < 5; ++j)
510         {
511             if (keyMatrix[i][j] == c)
512             {
513                 row = i;
514                 col = j;
515                 return;
516             }
517         }
518     }
519 }
```

```
C:\Users> Admin > Desktop > C:\ISPjct.cpp > main()
518 void findPosition(char keyMatrix[5][5], char c, int &row, int &col)
519 {
520     for (int j = 0; j < 5; ++j)
521     {
522         if (keyMatrix[i][j] == c)
523         {
524             row = i;
525             col = j;
526             return;
527         }
528     }
529 }
530
531 // Function to encrypt plaintext using Playfair cipher
532 string encrypt(string plaintext, char keyMatrix[5][5])
533 {
534     plaintext = prepareText(plaintext);
535     string ciphertext = "";
536
537     for (size_t i = 0; i < plaintext.length(); i += 2)
538     {
539         char first = plaintext[i];
540         char second = plaintext[i + 1];
541         int r1, c1, r2, c2;
542
543         findPosition(keyMatrix, first, r1, c1);
544         findPosition(keyMatrix, second, r2, c2);
545
546         if (r1 == r2)
547         {
548             // Same row
549             ciphertext += keyMatrix[r1][(c1 + 1) % 5];
550             ciphertext += keyMatrix[r2][(c2 + 1) % 5];
551         }
552         else if (c1 == c2)
553         {
554             // Same column
555             ciphertext += keyMatrix[(r1 + 1) % 5][c1];
556             ciphertext += keyMatrix[(r2 + 1) % 5][c2];
557         }
558         else
559         {
560             // Rectangle
561             ciphertext += keyMatrix[r1][c2];
562             ciphertext += keyMatrix[r2][c1];
563         }
564     }
565     return ciphertext;
566 }
```

Picture 1:

xorEncrypt(const string &data, char key)

- Performs simple symmetric encryption using the XOR operation.
- Iterates through each character of the input string data and applies XOR with the given key.
- Returns the XOR-encrypted string.

generateKeys()

- Generates RSA-like public and private keys for encryption and decryption.
- Uses two example prime numbers $p = 61$ and $q = 53$.
- Calculation procedure:
 - Modulus $n = p * q$.
 - Euler's totient function $\phi = (p-1) * (q-1)$.
 - Public exponent $e = 17$ (usually used in RSA and must be coprime to ϕ).
 - Private exponent $d = 2753$ (modular inverse of $e \bmod \phi$).
- Returns a pair of integers representing the public key (e, n) .

rsaEncrypt(int message, int e, int n)

- Performs RSA-like encryption using modular exponentiation.
- Encrypts the message by raising it to the power of e and taking modulo n .
- Returns the encrypted result.

rsaDecrypt(int cipher, int d, int n)

- Performs RSA-like decryption using modular exponentiation.
- Decrypts the cipher by raising it to the power of d and taking modulo n .
- Returns the decrypted message.

stringToInt(const string &str)

- Converts a string to an integer.
- Iterates through each character in the string, updating the result as $\text{result} = \text{result} * 256 + c$ (treating the string as a base-256 number).
- Returns the integer representation of the string.

Picture 2:

1. intToString(int num)

- Converts an integer back into a string.
- Iteratively extracts each byte (by using modulo 256) from the integer and prepends it to the resulting string.
- Returns the string representation of the integer.

2. railfenceencrypt(string plaintext, int key)

- Encrypts plaintext using the Rail Fence Cipher algorithm.
- If the key ≤ 1 , returns the plaintext without encryption.
- Creates a vector of strings (rail) to represent the zigzag pattern for the cipher.
- Traverses the plaintext:

- Places each character into the appropriate "rail" based on the current row and direction (up or down).
 - Reverses the direction when the top or bottom rail is reached.
 - Combines all the rails into a single ciphertext string.
 - Returns the encrypted ciphertext.
- 3. railfencedecrypt(string ciphertext, int key)**
- Decrypts a ciphertext encrypted using the Rail Fence Cipher.
 - If the key ≤ 1 , returns the ciphertext without decryption.
 - Sets up structures to track rail lengths and their positions to reconstruct the zigzag traversal.
 - Logic for decoding is likely implemented in the subsequent portion.

Picture 3:

railfencedecrypt(string ciphertext, int key)

- Decrypts ciphertext encrypted with the Rail Fence Cipher algorithm.
- Calculates the length of each rail based on the zigzag pattern.
- Fills the rails with characters from the ciphertext according to their lengths.
- Reconstructs the plaintext by following the zigzag traversal used during encryption.
- Returns the decrypted plaintext.

generateKey(string plaintext, string key)

- Generates a full key for use in the Vigenère Cipher by repeating the given key until its length matches the length of the plaintext.
- Takes the plaintext and key as input.
- Returns the full key as a string.

vigencrypt(string plaintext, string key)

- Encrypts plaintext using the Vigenère Cipher.
- Likely iterates through the plaintext, combining each character with the corresponding character in the key (based on the Vigenère Cipher logic).
- Generates key from plain text and string key
- Returns the encrypted plaintext.

Picture 4:

vigdecrypt(string plaintext, string key)

- Decrypts plain text with the Vigenère Cipher algorithm.
- Returns the decrypted text

caesar_encrypt(string plaintext, string key)

- Encrypts plaintext using the Caesar Cipher.
- Stores result in string
- Checks for capital letters in for loop and in IF condition
- Returns result which is encrypted text after iteration of for loop

caesar_decrypt(string plaintext, string key)

- Decrypts plaintext using the reverse of Caesar Cipher.

Picture 5:

determinant function:

- Computes the determinant of a 2x2 matrix.
- Formula used: $\det(A) = (a*d - b*c)$ for matrix $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$.

adjugate function:

- Calculates the adjugate (adjoint) of a 2x2 matrix.
- It swaps the diagonal elements and negates the off-diagonal elements of the input matrix.

modInverse function:

- Computes the modular multiplicative inverse of a modulo m.
- Uses a brute-force method to find an x such that $(a * x) \% m == 1$.
- Returns -1 if the modular inverse does not exist.

Exponentiation function:

- Implements efficient modular exponentiation.
- Computes $(base^{exp}) \% mod$ using the square-and-multiply method to handle large exponents efficiently.

generateRandomKey function:

- Generates a random string (key) of the same length as the input plaintext.
- Uses a random number generator to create random characters from the byte range (0-255).
- Builds the key by appending random characters iteratively.

Picture 6:

encrypt function:

- Implements one-time pad (OTP) encryption.
- XORs each character in the plaintext with the corresponding character in the key.
- Returns the resulting ciphertext as a string.

decrypt function:

- Reverses OTP encryption by XORing each character of the ciphertext with the corresponding character in the key.
- Returns the original plain text.

gcd function:

- By usage of the Euclidean Algorithm this function finds the GCD of two integers.
- Uses recursion for simplicity.

encrypt function (RSA):

- Encrypts an integer message using RSA encryption.
- Performs modular exponentiation $(\text{message}^e) \% n$.

decrypt function (RSA):

- Decrypts an integer ciphertext using RSA decryption.
- Computes $(\text{ciphertext}^d) \% n$.

generateRSAKeys function:

- Generates RSA public and private keys.
- Calculates: $n = p * q$ (product of two primes).
- Euler's totient function $\phi_n = (p-1) * (q-1)$.
- Choose e such that $1 < e < \phi_n$ and $\text{gcd}(e, \phi_n) == 1$.
- Computes the private key d , the modular inverse of e modulo ϕ_n .

Picture 7:**stringToIntVector Function:**

- Converts a string into a vector of integers based on ASCII values.
- Iterates over each character in the string, casting it to its integer value.
- Returns a vector of integers.

intVectorToString Function:

- Converts a vector of integers back into a string.
- Iterates over each integer in the vector, casting it back to a character.
- Returns the resulting string.

generateKeyMatrix Function:

- Generates a 5x5 key matrix for the Playfair cipher based on a given key.
- Uses a used array to track which characters are included in the matrix.
- Treats 'J' and 'I' as the same character (a common Playfair cipher rule).
- Fills the matrix row by row, first with characters from the key, then with the remaining letters of the alphabet (excluding duplicates).

Picture 8:**prepareText function**

- Converts all alphabetic characters in the input to uppercase.
- Replaces the letter 'J' with 'I' as per Playfair cipher convention.
- Detects consecutive duplicate letters in the input and inserts an 'X' between them.

- If the length of the prepared text is odd, adds an 'X' at the end to make the length even.
- Output: Returns the processed text ready for encryption.

findPosition function

- Finds the row and column indices of a given character in the Playfair cipher's key matrix.
- Iterates through the 5x5 key matrix to locate the character.
- Updates the references for row (row) and column (col) with the found indices.

encrypt function

- Prepares the plaintext using the prepareText function.
- Processes the prepared text two characters at a time:
- Uses the findPosition function in order to obtain the positions of the two characters in the key matrix.
- Applies Playfair cipher rules: If both characters are in the same row then the cipher replaces each with the character to its right in the same row if it is not full it will circle through to the first element.
- If both characters are in the same column, it substitutes each with the character below it (wrapping to the top if needed).
- If neither of the above, substitutes the characters with the ones at the opposite corners of the rectangle they form in the matrix.

5. Time Complexities

1. Symmetric Encryption

- Caesar Cipher:
 - *Time Complexity:* $O(n)$, where n is the length of the plaintext.
 - *Explanation:* Each character is shifted a fixed number of positions, which is a constant-time operation for each character.
- Vigenère Cipher:
 - *Time Complexity:* $O(n)$, where n is the length of the plaintext.
 - *Explanation:* Each character is shifted based on a key, but the shift value changes periodically. This still results in a linear time complexity.
- Rail Fence Cipher:
 - *Time Complexity:* $O(n)$, where n is the length of the plaintext.

- Explanation: The plaintext is written diagonally across a grid, and then read off row by row. This process involves iterating through the plaintext once and performing simple operations, leading to linear time complexity.
- Playfair Cipher:
 - Time Complexity: $O(n)$, where n is the length of the plaintext.
 - Explanation: Each pair of plaintext letters is encrypted using a 5x5 matrix. While the lookup and substitution operations might seem complex, they are performed a constant number of times for each plaintext pair, resulting in linear time complexity.
- One-Time Pad (OTP):
 - Time Complexity: $O(n)$, where n is the length of the plaintext.
 - Explanation: Each plaintext bit is XORed with a corresponding random key bit. This is a simple bitwise operation that takes constant time per bit, leading to linear time complexity.

2. Asymmetric Encryption (RSA-like)

- Key Generation:
 - Time Complexity: $O(k^3)$, where k is the key size in bits.
 - Explanation: Key generation involves finding large prime numbers and performing modular exponentiation, which are computationally intensive operations.
- Encryption:
 - Time Complexity: $O(k^3)$, where k is the key size in bits.
 - Explanation: Encryption involves modular exponentiation, which is the dominant operation.
- Decryption:
 - Time Complexity: $O(k^3)$, where k is the key size in bits.
 - Explanation: Similar to encryption, decryption involves modular exponentiation.

3. Hybrid Encryption

- Time Complexity: It depends on the specific symmetric and asymmetric algorithms used.
 - Explanation: Hybrid encryption involves both symmetric and asymmetric operations. The overall time complexity will depend on the efficiency of the chosen algorithms.

6. Best Performing Encryption Technique

Out of all the ciphering techniques used in our code, the most efficient out of all of them is perhaps the Caesar Cipher.

The following is the justification for our chosen technique:

Why Caesar Cipher is Most Efficient:

Time Complexity:

Caesar Cipher operates in $O(n)$. It performs a simple character shift operation, requiring minimal operations of computation.

Performance:

It involves straightforward arithmetic operations on characters, making it extremely fast, especially for small to medium-sized text.

Coding Mechanisms:

The implementation is simple, as seen in functions like `caesar_encrypt`. It uses a loop to iterate through the plaintext, applying the shift, with no need for additional data structures or computational overhead.

Comparison with Other Ciphers:

Vigenère Cipher:

It has the same time complexity, but slightly more complex due to the use of a keyword and handling multiple character shifts.

Rail Fence Cipher:

It has the same time complexity, but the zigzag pattern introduces additional overhead in managing positions.

Playfair Cipher:

This requires preprocessing to generate the key matrix and handle other functions, adding to complexity compared to Caesar.

RSA Encryption:

It has the highest time complexity due to modular arithmetic and is slower than all classical ciphers.

Final Verdict

The Caesar Cipher is the most efficient in terms of simplicity, speed, and coding mechanisms but, its efficiency comes at the cost of security, as it is highly vulnerable to attacks by third parties.

If efficiency is the primary criterion, Caesar Cipher stands out; for practical applications, more secure ciphers like Vigenère or hybrid systems should be considered.

7. Key Achievements of Project

1. Algorithmic Diversity

- Related operations have been successfully performed with many different encryption algorithms.
- Showed similar and distinguishable methods of encryption
- Demonstration on the adaptability of crypto techniques

2. Educational Significance

- Offered completely operational, first-hand training opportunity
- Connected principles of the existence of cryptography at the theoretical level with examples of its use
- Provided understandings into mathematical background of encryption

3. Technical Complexity

- Created the open, parameterizable and scalable cryptographic architecture
- Put in place high level of encryption and decryption functions
- Showed good programming practices in C++

8. Future Improvements and Recommendations

While the current implementation serves as an excellent educational tool, potential enhancements could include:

- Improving on the user interface and improved error checking
- Enabling of large messages since certain clients cannot handle large blocks of data, support for this kind of sizes is needed.
- Setting up a GUI based program which could be intricately integrated with softwares and used on an as needed basis.

For future developers and researchers:

- Keep abreast with the market trend with regards to encryption technologies.
- Consequently, there is a need to extend increased attention towards developing better and, conversely, safer use of communication.
- Cryptographic studies remain a very dynamic field, and such endeavours go along ways towards enhancing an understanding and usage of improved forms of secure communication systems.

9. Final Thoughts

Cryptography remains a critical field in our increasingly digital world. This project underscores the importance of understanding and implementing secure communication methods. By providing a hands-on exploration of encryption techniques, we contribute to the broader understanding of digital security and data protection.

The journey of cryptography is ongoing, with new challenges and innovations emerging constantly. This project serves as a stepping stone towards more advanced and secure communication systems.