

Volume 2

+300 Java Algorithms

Mastering the Art of Problem-Solving

Hernando Abella

Aluna Publishing House is united by our shared passion for education, languages, and technology. Our mission is to provide the ultimate learning experience when it comes to books. We believe that books are not just words on paper; they are gateways to knowledge, imagination, and enlightenment.

Through our collective expertise, we aim to bridge the gap between traditional learning and the digital age, harnessing the power of technology to make books more accessible, interactive, and enjoyable. We are dedicated to creating a platform that fosters a love for reading, language, and lifelong learning. Join us on our journey as we embark on a quest to redefine the way you experience books.

Let's unlock the limitless potential of knowledge, one page at a time.



In every line of code, they have woven a story of innovation and creativity.

Your code is the melody that gives life to projects.

May they continue creating and programming with passion!

Thank you for allowing me to be part of your journey.

With gratitude,
Hernando Abella
Author of *+300 Java Algorithms*



Download your bonus books @
www.hernandoabellacom

This book is Part 2 of ***300+ Java Algorithms***.

While **Part 1** focused on building strong algorithmic foundations and core problem-solving patterns in Java, **Part 2** continues with more advanced topics, complex data structures, and challenging algorithmic scenarios.

This volume is designed for readers who already understand the basics of algorithms and are ready to go deeper—focusing on optimization, scalability, and real-world problem-solving.

If you have not yet completed **Part 1**, it is strongly recommended to start there before proceeding with this book.

Table of Contents

Baconian Cipher.....	16
Blowfish.....	19
Caesar.....	22
Columnar Transposition Cipher.....	25
DES.....	29
ECC.....	32
Hill Cipher.....	34
Mono Alphabetic.....	37
Permutation Cipher.....	40
Polybius.....	44
Product Cipher.....	46
Rail Fence Cipher.....	49
Simple Sub Cipher.....	51
Vigenere.....	54
XOR Cipher.....	56
Conversions.....	58
Any Base to Decimal.....	59
Base64.....	61
Binary To Decimal.....	66
Binary To Hexadecimal.....	69
Binary To Octal.....	71
Decimal To Any Base.....	73
Decimal To Binary.....	75
Decimal To Hexadecimal.....	77
Decimal To Octal.....	79
Hex To Oct.....	80
HexaDecimal To Binary.....	81

HexaDecimal To Decimal	82
IP Converter	83
IPv6 Converter	85
Integer To English	88
Morse Code Converter	91
Number To Words	95
Octal To Binary	98
Octal To Hexadecimal	99
Phonetic Alphabet Converter	100
Rgb Hsv Conversion	103
Temperature Converter	106
Time Converter	108
Units Converter	111
Words To Number	117
Compression	120
 Arithmetic Coding	121
 LZ77	123
 LZ78	126
 LZW	128
 Move To Front	130
Devutils	133
 Process Details	134
 Large Tree Node	136
 Node	139
 Simple Node	142
 Simple Tree Node	144
 Run Length Encoding	146

Divide and Conquer.....	148
Binary Exponentiation.....	149
Closest Pair.....	151
Counting Inversions.....	153
Median Of Two Sorted Arrays.....	155
Skyline Algorithm.....	157
Strassen Matrix Multiplication.....	160
Tiling Problem.....	164
Dynamic Programming.....	166
Abbreviation.....	167
BruteForce Knapsack.....	169
Climbing Stairs.....	171
Damerau Levenshtein Distance.....	173
Dice Throw.....	175
Edit Distance.....	177
Fibonacci.....	179
Kadane Algorithm.....	181
Knapsack Memoization.....	182
Levenshtein Distance.....	184
Matrix Chain Multiplication.....	186
Matrix Chain Recursive Top Down Memoisation....	188
Maximum Product Subarray.....	190
Maximum Sum of Non-Adjacent Eleme.....	192
Minimum Path Sum.....	193
Optimal Job Scheduling.....	195
Partition Problem.....	197
Regex Matching.....	199

Shortest Common SupersequenceLe	201
Subset Count	203
Subset Sum	205
Tree Matching	207
Tribonacci	209
Wildcard Matching	210
Wine Problem	212
Geometry	214
Bentley Ottmann	215
Bresenham Line	217
Convex Hull	219
Graham Scan	222
Midpoint Circle	225
Midpoint Ellipse	227
Wu's Line	230
Graph	232
Bron Kerbosch	233
Constrained Shortest Path	235
Dinic	238
Edmonds Karp	242
Hierholzer Algorithm	247
Hierholzer Eulerian Path	250
Hopcroft Karp	253
Hungarian Algorithm	257
Stoer Wagner	261
Traveling Salesman	264
Greedy Algorithms	267

Activity Selection	268
Bandwidth Allocation	271
Binary Addition	273
Coin Change	275
Digit Separation	277
Egyptian Fraction	279
Job Sequencing	280
Merge Intervals	283
Optimal File Merging	286
Stock Profit Calculator	288
Io	290
Buffered Reader	291
Lineclipping	292
Cohen Sutherland	293
Liang Barsky	296
Maths	298
Absolute Max	299
Absolute Min	301
Absolute Value	303
Amicable Number	304
Area	306
Auto Correlation	308
Automorphic Number	310
Average	311
Ceil	312
Chinese Remainder Theorem	313
Combinations	315

Determinant Of Matrix.....	317
Digital Root.....	320
Eulers Function.....	322
Factorial.....	324
Factorial Recursion.....	325
Fibonacci Number Check.....	326
Find Max Recursion.....	328
Find Min Recursion.....	329
Happy Number.....	330
Means.....	332
Median.....	333
MinValue.....	334
Perfect Cube.....	335
Perfect Number.....	337
Perfect Square.....	339
Power Using Recursion.....	341
Quadratic Equation Solver.....	342
Reverse Number.....	344
Smith Number.....	345
Sum Of Digits.....	347
Sum Of Squares.....	348
Volume.....	349
Matrix.....	351
 Inverse Of Matrix.....	352
 LU Decomposition.....	355
 Matrix Multiplication.....	358
 Matrix Rank.....	360
 Matrix Transpose.....	362

Median Of Matrix	364
Mirror Of Matrix	366
Print A Matrix in Spiral Order	368
Rotate Matrix By 90 Degrees	370
Solve System	372
Misc	374
Color Contrast Ratio	374
Map Reduce	376
Median Of Running Array	379
Median Of Running Array Byte	381
Palindrome Prime	383
Palindrome Singly Linked List	385
Range In Sorted Array	388
Shuffle Array	391
Sparsity	392
Three Sum Problem	394
Two Sum Problem	397
Physics	399
Coulombs Law	400
Damped Oscillator	402
Elastic Collision 2D	404
Gravitation	408
Ground To Ground Projectile Motion	410
Kinematics	412
Projectile Motion	414
Simple Pendulum RK4	416
Puzzles and Games	418

Tower Of Hanoi.....	419
Word Boggle.....	421
Randomized.....	424
Karger Min Cut.....	424
Monte Carlo Integration.....	427
Randomized Closest Pair.....	429
Randomized Matrix Multiplication Verification.....	432
Randomized Quick Sort.....	435
Reservoir Sampling.....	437
Recursion.....	439
Dice Thrower.....	440
Fibonacci Series.....	442
Generate Subsets.....	443
Sylvester Sequence.....	445
Scheduling.....	446
Aging Scheduling.....	447
Fair Share Scheduling.....	449
Gang Scheduling.....	452
Job Scheduling with Deadline.....	454
Multi Agent Scheduling.....	456
Non-Preemptive Priority Scheduling.....	458
Preemptive Priority Scheduling.....	460
Proportional Fair Scheduling.....	463
Sliding Window.....	466
Longest Subarray with Sum Less Or E.....	467
Longest Substring Without Repeating.....	469
Max Sum K Size Subarray.....	471

Maximum Sliding Window	473
Min Sum K Size Subarray	475
Minimum Window Substring	477
Shortest Coprime Segment	479
Strings	481
Anagrams	482
Check Vowels	483
Count Char	484
Count Words	486
Hamming Distance	487
Isogram	488
Lower	490
Palindrome	491
Pangram	493
Reverse String	495
Reverse Words in String	496
String Compression	498
Valid Parentheses	500
Word Ladder	502

Baconian Cipher

The **Baconian cipher** is a steganographic substitution cipher invented by **Francis Bacon**.

It encodes each letter of the alphabet using a group of **5 characters** made up of only **A and B**.

That means every letter from **A-Z** becomes a 5-letter sequence like:

A = AAAAA

B = AAAAB

C = AAABA

...

Z = BABBB

So a word like:

HELLO

becomes a sequence of A's and B's.

It is useful for:

- Hidden messages
- Encoding inside text formatting
- Learning binary-style encoding

```
import java.util.HashMap;
import java.util.Map;

public class Main {

    private static final Map<Character, String> baconMap = new
    HashMap<>();
    private static final Map<String, Character> reverseMap = new
    HashMap<>();

    // Initialize Baconian mappings
    static {
```

```

String[] codes = {
    "AAAAA", "AAAAB", "AAABA", "AAABB", "AABAA", "AABAB",
    "AABBA", "AABBB", "ABAAB", "ABAAB", "ABABA", "ABABB",
    "ABBAA", "ABBAB", "ABBBA", "ABBBB", "BAAAA", "BAAAB",
    "BAABA", "BAABB", "BABAA", "BABAB", "BABBA", "BABBB",
    "BBAAA", "BBAAB"
};

for (int i = 0; i < 26; i++) {
    char letter = (char) ('A' + i);
    baconMap.put(letter, codes[i]);
    reverseMap.put(codes[i], letter);
}
}

// Encrypt
public static String encrypt(String text) {
    text = text.toUpperCase();
    StringBuilder result = new StringBuilder();

    for (char c : text.toCharArray()) {
        if (Character.isLetter(c)) {
            result.append(baconMap.get(c)).append(" ");
        }
    }
    return result.toString().trim();
}

// Decrypt
public static String decrypt(String text) {
    StringBuilder result = new StringBuilder();
    String[] tokens = text.split(" ");

    for (String token : tokens) {
        if (reverseMap.containsKey(token)) {

```

```
        result.append(reverseMap.get(token));
    }
}
return result.toString();
}

public static void main(String[] args) {

    String message = "HELLO";

    String encrypted = encrypt(message);
    String decrypted = decrypt(encrypted);

    System.out.println("Original : " + message);
    System.out.println("Encrypted : " + encrypted);
    System.out.println("Decrypted : " + decrypted);
}
}

// Original : HELLO
// Encrypted : AABBB AABAA ABABB ABABB ABBBA
// Decrypted : HELLO
```

Blowfish

Blowfish is a fast, symmetric block cipher designed by Bruce Schneier.

It uses:

- Block size: **64 bits**
- Key size: **32 – 448 bits**
- Feistel network with **16 rounds**
- Very fast in software and fully supported by Java Cryptography API (JCE)

It is considered secure for learning and legacy systems, but modern systems typically prefer **AES** because Blowfish's block size is small.

```
import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;

public class Main {

    private static final String ALGORITHM = "Blowfish";

    // Generate key from a string
    public static SecretKey generateKey(String key) {
        return new SecretKeySpec(key.getBytes(), ALGORITHM);
    }

    // Encrypt text
    public static String encrypt(String plainText, SecretKey
secretKey) throws Exception {
        Cipher cipher = Cipher.getInstance(ALGORITHM);
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
```

```

        byte[] encryptedBytes =
cipher.doFinal(plainText.getBytes());
        return
Base64.getEncoder().encodeToString(encryptedBytes);
}

// Decrypt text
public static String decrypt(String cipherText, SecretKey
secretKey) throws Exception {
    Cipher cipher = Cipher.getInstance(ALGORITHM);
    cipher.init(Cipher.DECRYPT_MODE, secretKey);

    byte[] decodedBytes =
Base64.getDecoder().decode(cipherText);
    byte[] decryptedBytes = cipher.doFinal(decodedBytes);

    return new String(decryptedBytes);
}

public static void main(String[] args) {

try {

String message = "Hello Blowfish";
String key = "SuperSecretKey";

SecretKey secretKey = generateKey(key);

String encrypted = encrypt(message, secretKey);
String decrypted = decrypt(encrypted, secretKey);

System.out.println("Original : " + message);
System.out.println("Encrypted: " + encrypted);
System.out.println("Decrypted: " + decrypted);
}

```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

// Original : Hello Blowfish
// Encrypted: 7dnP4T1jy859XFPzwxDRGg==
// Decrypted: Hello Blowfish
```

Caesar

The **Caesar Cipher** is one of the simplest and most famous substitution ciphers.

Each letter in the plaintext is shifted by a fixed number of positions in the alphabet.

If the shift is **3**:

A → D
B → E
C → F

...

X → A
Y → B
Z → C

Formula:

Encryption:

$$E(x) = (x + shift) \bmod 26$$

Decryption:

$$D(x) = (x - shift) \bmod 26$$

It was used by **Julius Caesar** in military communication.

It is **not secure**, but perfect for understanding substitution ciphers.

```
public class Main {  
  
    // Encrypt text  
    public static String encrypt(String text, int shift) {  
        StringBuilder result = new StringBuilder();  
  
        for (char ch : text.toCharArray()) {  
  
            if (Character.isUpperCase(ch)) {  
                result.append((char) ((ch - 65 + shift) % 26 + 65));  
            } else if (Character.isLowerCase(ch)) {  
                result.append((char) ((ch - 97 + shift) % 26 + 97));  
            } else {  
                result.append(ch);  
            }  
        }  
        return result.toString();  
    }  
}
```

```

        char encrypted = (char) ((ch - 'A' + shift) % 26
+ 'A');
        result.append(encrypted);
    }
    else if (Character.isLowerCase(ch)) {
        char encrypted = (char) ((ch - 'a' + shift) % 26
+ 'a');
        result.append(encrypted);
    }
    else {
        result.append(ch); // keep spaces and punctuation
    }
}

return result.toString();
}

// Decrypt text
public static String decrypt(String text, int shift) {
    return encrypt(text, 26 - (shift % 26));
}

public static void main(String[] args) {

    String message = "HELLO WORLD";
    int shift = 3;

    String encrypted = encrypt(message, shift);
    String decrypted = decrypt(encrypted, shift);

    System.out.println("Original : " + message);
    System.out.println("Encrypted: " + encrypted);
    System.out.println("Decrypted: " + decrypted);
}
}

```

```
// Original : HELLO WORLD  
// Encrypted: KHOOR ZRUOG  
// Decrypted: HELLO WORLD
```

Columnar Transposition Cipher

The Columnar Transposition Cipher is a classic transposition cipher where the plaintext is written into a grid row-by-row using a keyword to determine the order in which columns are read. The keyword is sorted alphabetically, and the columns are read in that sorted order to produce the ciphertext.

Unlike substitution ciphers, this method does **not change the letters**, only their **positions**, which is why it's called a *transposition* cipher.

How it works (encryption):

1. Choose a keyword (for example: ZEBRA).
2. Write the plaintext in rows under the keyword.
3. Number the letters in the keyword based on alphabetical order.
4. Read the columns in the order of the sorted keyword indexes.
5. Combine those columns to get the ciphertext.

Example with keyword **ZEBRA**:

Keyword: Z E B R A
Order: 5 3 2 4 1

Plaintext: ATTACKATDAWN

Grid:

Z E B R A
A T T A C
K A T D A
W N X X X (X used for padding)

Read by order → A B E R Z

Ciphertext: CAXTATNTTDWA

```
import java.util.*;  
  
public class Main {  
  
    // Get the order of columns from the key  
    private static int[] getKeyOrder(String key) {
```

```

int len = key.length();
Integer[] order = new Integer[len];

for (int i = 0; i < len; i++) {
    order[i] = i;
}

Arrays.sort(order, Comparator.comparingInt(i ->
key.charAt(i)));

int[] result = new int[len];
for (int i = 0; i < len; i++) {
    result[order[i]] = i;
}
return result;
}

// Encryption
public static String encrypt(String text, String key) {
    text = text.replaceAll("\\s+", "").toUpperCase();

    int keyLen = key.length();
    int rows = (int) Math.ceil((double) text.length() /
keyLen);

    char[][] matrix = new char[rows][keyLen];

    int index = 0;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < keyLen; j++) {
            if (index < text.length()) {
                matrix[i][j] = text.charAt(index++);
            } else {
                matrix[i][j] = 'X'; // padding
            }
        }
    }
}

```

```

        }
    }

    int[] order = getKeyOrder(key);
    StringBuilder cipher = new StringBuilder();

    for (int k = 0; k < keyLen; k++) {
        for (int col = 0; col < keyLen; col++) {
            if (order[col] == k) {
                for (int row = 0; row < rows; row++) {
                    cipher.append(matrix[row][col]);
                }
            }
        }
    }
    return cipher.toString();
}

// Decryption
public static String decrypt(String cipher, String key) {
    int keyLen = key.length();
    int rows = (int) Math.ceil((double) cipher.length() /
keyLen);

    char[][] matrix = new char[rows][keyLen];

    int[] order = getKeyOrder(key);
    int index = 0;

    for (int k = 0; k < keyLen; k++) {
        for (int col = 0; col < keyLen; col++) {
            if (order[col] == k) {
                for (int row = 0; row < rows; row++) {
                    if (index < cipher.length()) {

```

```

        matrix[row][col] =
cipher.charAt(index++);
    }
}
}
}

StringBuilder plain = new StringBuilder();
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < keyLen; j++) {
        plain.append(matrix[i][j]);
    }
}

return plain.toString().replaceAll("X+$", ""); // remove
padding
}

// Test
public static void main(String[] args) {
    String key = "ZEBRA";
    String text = "ATTACK AT DAWN";

    String encrypted = encrypt(text, key);
    System.out.println("Encrypted: " + encrypted);

    String decrypted = decrypt(encrypted, key);
    System.out.println("Decrypted: " + decrypted);
}
}

// Encrypted: CAXTTXTANADXAKW
// Decrypted: ATTACKATDAWN

```

DES

DES is a **symmetric-key block cipher** developed in the 1970s. It encrypts data in **64-bit blocks** using a **56-bit key** (plus 8 parity bits). DES applies **16 rounds** of substitution and permutation operations (Feistel structure) to transform plaintext into ciphertext.

Today, DES is considered **insecure** for modern systems because its 56-bit key is too short and can be brute-forced. It is mostly used for learning or legacy support. Modern replacements are **AES** and **3DES**.

Key points:

- Symmetric encryption (same key for encrypt/decrypt)
- Block size: **64 bits**
- Key size: **56 bits**
- Structure: **Feistel network, 16 rounds**
- Status: **Insecure / deprecated for real security**

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import java.util.Base64;

public class Main {

    // Generate DES Key
    public static SecretKey generateKey() throws Exception {
        KeyGenerator keyGenerator =
        KeyGenerator.getInstance("DES");
        keyGenerator.init(56); // DES uses 56-bit keys
        return keyGenerator.generateKey();
    }

    // Encrypt using DES
    public static String encrypt(String plaintext, SecretKey key)
throws Exception {
```

```

    Cipher cipher =
Cipher.getInstance("DES/ECB/PKCS5Padding");
    cipher.init(Cipher.ENCRYPT_MODE, key);

    byte[] encryptedBytes =
cipher.doFinal(plaintext.getBytes());
    return
Base64.getEncoder().encodeToString(encryptedBytes);
}

// Decrypt using DES
public static String decrypt(String encryptedText, SecretKey
key) throws Exception {
    Cipher cipher =
Cipher.getInstance("DES/ECB/PKCS5Padding");
    cipher.init(Cipher.DECRYPT_MODE, key);

    byte[] decodedBytes =
Base64.getDecoder().decode(encryptedText);
    byte[] decryptedBytes = cipher.doFinal(decodedBytes);

    return new String(decryptedBytes);
}

public static void main(String[] args) {
try {
    String message = "HELLO WORLD";

    SecretKey key = generateKey();

    String encrypted = encrypt(message, key);
    String decrypted = decrypt(encrypted, key);

    System.out.println("Original: " + message);
    System.out.println("Encrypted: " + encrypted);
}

```

```
        System.out.println("Decrypted: " + decrypted);

    } catch (Exception e) {
        e.printStackTrace();
    }
}

// Original: HELLO WORLD
// Encrypted: 0enQj79jIBQT+XcPwclWvDQ==
// Decrypted: HELLO WORLD
```

ECC

Elliptic Curve Cryptography (ECC) is a public-key cryptography system based on the mathematics of elliptic curves over finite fields.

It provides the same security level as RSA/DH but with much smaller keys, making it faster and more efficient.

ECC is used in:

- HTTPS / TLS
- Bitcoin & blockchain wallets
- Secure messaging (Signal, WhatsApp)
- Mobile devices & IoT

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.spec.ECGenParameterSpec;
import java.util.Base64;

public class Main {

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

        // Create ECC Key Pair Generator
        KeyPairGenerator keyPairGenerator =
KeyPairGenerator.getInstance("EC");
        ECGenParameterSpec ecSpec = new
ECGenParameterSpec("secp256r1");

        keyPairGenerator.initialize(ecSpec);

        // Generate Key Pair
        KeyPair keyPair = keyPairGenerator.generateKeyPair();
```

```
PublicKey publicKey = keyPair.getPublic();
PrivateKey privateKey = keyPair.getPrivate();

// Show keys
System.out.println("Public Key:");
System.out.println(Base64.getEncoder().encodeToString(publicKey.getEncoded()));

System.out.println("\nPrivate Key:");
System.out.println(Base64.getEncoder().encodeToString(privateKey.getEncoded()));
}

}

// Public Key:
//
MFkwEwYHCoZIzj0CAQYIKoZIzj0DAQcDQgAESQiQpFUp7TEJXHYCJXnMT0fCoC/Zb
51MtctPZnnmroy2q1RyRLqfpXkUlcW15GFBxInR+GMPeW4JY42wnK4Zgw==

// Private Key:
//
MEECAQAwEwYHCoZIzj0CAQYIKoZIzj0DAQcEJzA1AgEBBCDrGX+aU4RIt0csT9xse
vu63nbL89DkcEpjvzbTnNSkgg==
```

Hill Cipher

The Hill Cipher is a classical polygraphic substitution cipher that uses linear algebra (matrix multiplication) to encrypt blocks of letters.

- Letters are converted to numbers: A=0, B=1, ..., Z=25
- A **key matrix** (2×2 , 3×3 , etc.) is used

Encryption:

$$C = (K \times P) \mod 26$$

Decryption requires the matrix inverse modulo 26

Important: The key matrix **must be invertible mod 26**, otherwise decryption is impossible.

Example (2×2 Hill Cipher)

Key matrix:

$$K = \begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix}$$

Plaintext: "HI"

Convert:

$$H = 7, I = 8$$

Multiply:

$$\begin{bmatrix} 3 & 3 \\ 2 & 5 \end{bmatrix} \begin{bmatrix} 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 45 \\ 54 \end{bmatrix} \mod 26 = \begin{bmatrix} 19 \\ 2 \end{bmatrix}$$

$$19 = T, 2 = C$$

Encrypted text = "TC"

```
public class Main {
```

```

static int[][] key = {
    {3, 3},
    {2, 5}
};

// Encrypt function
public static String encrypt(String text) {
    text = text.replaceAll("\s+", "").toUpperCase();

    if (text.length() % 2 != 0) {
        text += "X"; // padding
    }

    StringBuilder cipher = new StringBuilder();

    for (int i = 0; i < text.length(); i += 2) {

        int x1 = text.charAt(i) - 'A';
        int x2 = text.charAt(i + 1) - 'A';

        int c1 = (key[0][0] * x1 + key[0][1] * x2) % 26;
        int c2 = (key[1][0] * x1 + key[1][1] * x2) % 26;

        cipher.append((char) (c1 + 'A'));
        cipher.append((char) (c2 + 'A'));
    }

    return cipher.toString();
}

public static void main(String[] args) {

    String message = "HELLO";
    String encrypted = encrypt(message);
}

```

```
        System.out.println("Plaintext : " + message);
        System.out.println("Encrypted : " + encrypted);
    }
}

// Plaintext : HELLO
// Encrypted : HIOZHN
```

Mono Alphabetic

A Monoalphabetic Cipher is a substitution cipher where each letter in the plaintext is replaced by another fixed letter from the alphabet.

One fixed mapping is used for the entire message

Example:

A → Q, B → W, C → E, D → R, ...

- Simple to implement
- Easy to crack using **frequency 37an37isis**

It is more secure than Caesar (which shifts letters), but still **not safe for real security**.

Example Mapping

Plain: ABCDEFGHIJKLMNOPQRSTUVWXYZ

Cipher: QWERTYUIOPASDFGHJKLMNPQRSTUVWXYZ

Plaintext: HELLO

Encrypted: ITSSG

```
import java.util.HashMap;

public class Main {

    static final String PLAIN  = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
    static final String CIPHER = "QWERTYUIOPASDFGHJKLMNPQRSTUVWXYZ";

    static HashMap<Character, Character> encryptMap = new
    HashMap<>();
    static HashMap<Character, Character> decryptMap = new
    HashMap<>();
```

```

// Initialize the maps
static {
    for (int i = 0; i < PLAIN.length(); i++) {
        encryptMap.put(PLAIN.charAt(i), CIPHER.charAt(i));
        decryptMap.put(CIPHER.charAt(i), PLAIN.charAt(i));
    }
}

// Encrypt
public static String encrypt(String text) {
    StringBuilder result = new StringBuilder();
    text = text.toUpperCase();

    for (char ch : text.toCharArray()) {
        if (Character.isLetter(ch))
            result.append(encryptMap.get(ch));
        else
            result.append(ch);
    }
    return result.toString();
}

// Decrypt
public static String decrypt(String text) {
    StringBuilder result = new StringBuilder();
    text = text.toUpperCase();

    for (char ch : text.toCharArray()) {
        if (Character.isLetter(ch))
            result.append(decryptMap.get(ch));
        else
            result.append(ch);
    }
    return result.toString();
}

```

```
public static void main(String[] args) {  
  
    String message = "HELLO WORLD";  
    String encrypted = encrypt(message);  
    String decrypted = decrypt(encrypted);  
  
    System.out.println("Plaintext : " + message);  
    System.out.println("Encrypted : " + encrypted);  
    System.out.println("Decrypted : " + decrypted);  
}  
}  
  
// Plaintext : HELLO WORLD  
// Encrypted : ITSSG VGKSR  
// Decrypted : HELLO WORLD
```

Permutation Cipher

The **Permutation Cipher** (also called **Transposition Cipher**) works by **rearranging (permuting) the positions of characters** in a fixed-size block using a secret key (a permutation of positions).

Unlike substitution ciphers (Caesar, Monoalphabetic), a permutation cipher **does not change the letters themselves** — it only **reorders them**.

Example for block size 5:

Key (permutation):

[3, 1, 4, 2, 5]

This means:

Position: 1 2 3 4 5

Becomes: 3 1 4 2 5

Plain block: HELLO

Reordered : L H O E O → depending on the key

It is more secure than a simple Caesar shift, but still breakable by modern methods.

How it works:

1. Choose a permutation key as an array (example: {3,1,4,2,5})
2. Split the plaintext into blocks of N
3. Rearrange characters in each block according to the key
4. To decrypt, use the inverse permutation

```
import java.util.Arrays;
```

```
public class Main {
```

```
    // Permutation key: positions start at 1 (human-friendly)
    // 1 -> 3, 2 -> 1, 3 -> 4, 4 -> 2
```

```
static final int[] KEY = {3, 1, 4, 2};
static final int BLOCK_SIZE = KEY.length;

// ====== ENCRYPT ======
public static String encrypt(String text) {

    text = text.replaceAll("\s+", "").toUpperCase();

    // Pad with X if necessary
    while (text.length() % BLOCK_SIZE != 0) {
        text += "X";
    }

    StringBuilder result = new StringBuilder();

    for (int i = 0; i < text.length(); i += BLOCK_SIZE) {

        char[] block = text.substring(i, i +
BLOCK_SIZE).toCharArray();
        char[] encrypted = new char[BLOCK_SIZE];

        for (int j = 0; j < BLOCK_SIZE; j++) {
            // KEY[j] - 1 is the source index
            encrypted[j] = block[KEY[j] - 1];
        }

        result.append(encrypted);
    }

    return result.toString();
}

// ====== DECRYPT ======
public static String decrypt(String text) {
```

```

StringBuilder result = new StringBuilder();

// Build inverse key
int[] inverseKey = new int[BLOCK_SIZE];
for (int i = 0; i < BLOCK_SIZE; i++) {
    inverseKey[KEY[i] - 1] = i + 1;
}

for (int i = 0; i < text.length(); i += BLOCK_SIZE) {

    char[] block = text.substring(i, i +
BLOCK_SIZE).toCharArray();
    char[] decrypted = new char[BLOCK_SIZE];

    for (int j = 0; j < BLOCK_SIZE; j++) {
        decrypted[j] = block[inverseKey[j] - 1];
    }

    result.append(decrypted);
}

return result.toString();
}

public static void main(String[] args) {

    String message = "HELLO WORLD";
    String encrypted = encrypt(message);
    String decrypted = decrypt(encrypted);

    System.out.println("Key : " +
Arrays.toString(KEY));
    System.out.println("Plaintext : " + message);
    System.out.println("Encrypted : " + encrypted);
    System.out.println("Decrypted : " + decrypted);
}

```

```
    }  
}  
  
// Key      : [3, 1, 4, 2]  
// Plaintext : HELLO WORLD  
// Encrypted  : LHLEOORWXLXD  
// Decrypted   : HELLOWORLDXX
```

Polybius

The Polybius Cipher is a classical **substitution cipher** that uses a **5×5 grid** of letters. Each letter is replaced by its **row and column numbers** inside the square. The letters **I** and **J** are usually combined so that 25 letters fit in the grid. It is one of the earliest examples of encoding letters into numbers.

```
import java.util.HashMap;
import java.util.Map;

public class Main {

    private static final char[][] SQUARE = {
        {'A', 'B', 'C', 'D', 'E'},
        {'F', 'G', 'H', 'I', 'K'}, // I/J combined
        {'L', 'M', 'N', 'O', 'P'},
        {'Q', 'R', 'S', 'T', 'U'},
        {'V', 'W', 'X', 'Y', 'Z'}
    };

    private static final Map<Character, String> MAP = new
    HashMap<>();

    static {
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                MAP.put(SQUARE[i][j], "" + (i + 1) + (j + 1));
            }
        }
        MAP.put('J', "24"); // treat J as I
    }

    public static String encrypt(String text) {
        text = text.toUpperCase().replaceAll("[^A-Z]", "");
        StringBuilder result = new StringBuilder();
    }
}
```

```
        for (char c : text.toCharArray()) {
            result.append(MAP.get(c)).append(" ");
        }

        return result.toString().trim();
    }

    public static void main(String[] args) {
        System.out.println(encrypt("HELLO")); // Output: 23 15
31 31 34
    }
}

// 23 15 31 31 34
```

Product Cipher

A Product Cipher is a classical encryption method that combines **two or more ciphers**—most commonly **substitution** and **transposition**—to create a stronger cipher. The idea is that while substitution hides the letters and transposition hides the positions, combining both makes breaking the cipher significantly harder.

A common simple example is using a **Caesar Cipher** (substitution) followed by a **Columnar Transposition Cipher** (transposition).

Below is a minimal and fully functional Java snippet that performs:

1. Caesar shift (substitution)
2. Columnar transposition (transposition)

```
public class Main {

    // Step 1: Caesar substitution
    public static String caesarEncrypt(String text, int shift) {
        StringBuilder result = new StringBuilder();

        for (char c : text.toUpperCase().toCharArray()) {
            if (c >= 'A' && c <= 'Z') {
                char shifted = (char) ((c - 'A' + shift) % 26 +
'A');
                result.append(shifted);
            }
        }

        return result.toString();
    }

    // Step 2: Columnar transposition
    public static String columnarEncrypt(String text, int cols) {
        StringBuilder result = new StringBuilder();
    }
}
```

```

        int rows = (int) Math.ceil(text.length() / (double)
cols);

        char[][] grid = new char[rows][cols];

        int index = 0;
        for (int r = 0; r < rows; r++) {
            for (int c = 0; c < cols; c++) {
                if (index < text.length()) {
                    grid[r][c] = text.charAt(index++);
                } else {
                    grid[r][c] = 'X'; // padding
                }
            }
        }

        // Read column by column
        for (int c = 0; c < cols; c++) {
            for (int r = 0; r < rows; r++) {
                result.append(grid[r][c]);
            }
        }

        return result.toString();
    }

    // Full Product Cipher: Caesar + Transposition
    public static String encrypt(String message, int shift, int
columns) {
        String step1 = caesarEncrypt(message, shift);
        return columnarEncrypt(step1, columns);
    }

    public static void main(String[] args) {
        String plaintext = "HELLOWORLD";

```

```
String encrypted = encrypt(plaintext, 3, 4);

System.out.println("Plaintext: " + plaintext);
System.out.println("Encrypted: " + encrypted);
}

// Plaintext: HELLOWORLD
// Encrypted: KROHZGORXOUX
```

Rail Fence Cipher

The Rail Fence Cipher is a classical **transposition cipher** that writes the message in a zig-zag pattern across multiple "rails" (rows). After writing the zig-zag, the ciphertext is formed by reading row by row. It changes the order of characters without changing the characters themselves.

```
public class Main {

    // Encrypt using Rail Fence Cipher
    public static String encrypt(String text, int rails) {
        if (rails <= 1) return text;

        StringBuilder[] fence = new StringBuilder[rails];
        for (int i = 0; i < rails; i++) fence[i] = new
StringBuilder();

        int row = 0;
        boolean down = true;

        for (char c : text.toCharArray()) {
            fence[row].append(c);

            if (down) {
                if (row == rails - 1) {
                    down = false;
                    row--;
                } else {
                    row++;
                }
            } else {
                if (row == 0) {
                    down = true;
                    row++;
                } else {

```

```
        row--;
    }
}
}

StringBuilder encrypted = new StringBuilder();
for (StringBuilder sb : fence) encrypted.append(sb);

return encrypted.toString();
}

public static void main(String[] args) {
    String plaintext = "HELLOWORLD";
    String encrypted = encrypt(plaintext, 3);

    System.out.println("Plaintext: " + plaintext);
    System.out.println("Encrypted: " + encrypted);
}
}

// Plaintext: HELLOWORLD
// Encrypted: HOELWRDLO
```