Shamir Secret Sharing Scheme

Aim:

To securely split a secret into multiple parts (shares) such that only a minimum threshold of shares is required to reconstruct the original secret using polynomial interpolation.

Description:

Shamir's Secret Sharing is a cryptographic algorithm that divides a secret into n parts, with a threshold k such that any k parts can reconstruct the secret. This is achieved by evaluating a randomly generated polynomial (with the secret as the constant term) at n different points. The scheme ensures that fewer than k shares provide no information about the secret.

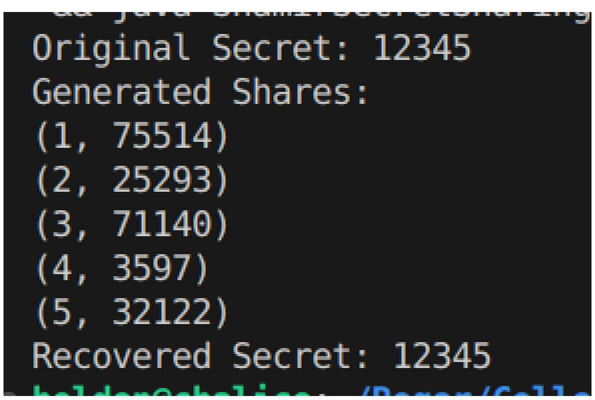
Code:

```
import java.math.BigInteger;
import java.security.SecureRandom;
import java.util.*;

public class ShamirSecretSharing {

    private static final SecureRandom random = new SecureRandom();
    private static final BigInteger PRIME = new BigInteger("104729"); // A large prime
number

    // Represents a share (x, y)
    public static class Share {
    public final BigInteger x, y;

    public Share(BigInteger x, BigInteger y) {
    this.x = x;
    this.y = y;
    }

    @Override
    public String toString() {
    return "(" + x + ", " + y + ")";
    }
    }

    // Splits the secret into n shares with a threshold of k
    public static List<Share> splitSecret(BigInteger secret, int k, int n) {
    List<BigInteger> coefficients = new ArrayList<>();
    coefficients.add(secret); // a0 = secret
```

```
// Generate random coefficients for the polynomial
for (int i = 1; i < k; i++) {
coefficients.add(new BigInteger(PRIME.bitLength(), random).mod(PRIME));
}

List<Share> shares = new ArrayList<>();
for (int i = 1; i <= n; i++) {
BigInteger x = BigInteger.valueOf(i);
BigInteger y = evaluatePolynomial(coefficients, x);
shares.add(new Share(x, y));
}

return shares;
}

// Evaluates a polynomial at point x
private static BigInteger evaluatePolynomial(List<BigInteger> coeffs, BigInteger x) {
BigInteger y = BigInteger.ZERO;
for (int i = 0; i < coeffs.size(); i++) {
BigInteger term = coeffs.get(i).multiply(x.pow(i)).mod(PRIME);
y = y.add(term).mod(PRIME);
}
return y;
}

// Reconstructs the secret from k shares using Lagrange interpolation
public static BigInteger reconstructSecret(List<Share> shares) {
BigInteger secret = BigInteger.ZERO;

for (int i = 0; i < shares.size(); i++) {
BigInteger xi = shares.get(i).x;
BigInteger yi = shares.get(i).y;

BigInteger li = BigInteger.ONE;
for (int j = 0; j < shares.size(); j++) {
    if (i != j) {
    BigInteger xj = shares.get(j).x;
    BigInteger numerator = xj.negate().mod(PRIME);
    BigInteger denominator = xi.subtract(xj).mod(PRIME);
    li                                                        =
li.multiply(numerator).multiply(denominator.modInverse(PRIME)).mod(PRIME);
    }
}

secret = secret.add(yi.multiply(li)).mod(PRIME);
}
```

```
        return secret;
        }

        // Demo
        public static void main(String[] args) {
        BigInteger secret = new BigInteger("12345");
        int k = 3; // Minimum required shares to reconstruct
        int n = 5; // Total shares

        System.out.println("Original Secret: " + secret);

        List<Share> shares = splitSecret(secret, k, n);
        System.out.println("Generated Shares:");
        for (Share s : shares) {
        System.out.println(s);
        }

        // Pick any k shares to reconstruct
        List<Share> subset = shares.subList(0, k);
        BigInteger recovered = reconstructSecret(subset);

        System.out.println("Recovered Secret: " + recovered);
        }
}
```

Output:



Code Explanation (in brief):

1. splitSecret: Generates a random polynomial of degree k-1 with the secret as the constant term. Evaluates the polynomial at n different x values to produce the shares.

2. evaluatePolynomial: Computes the value of the polynomial at a specific x.
3. reconstructSecret: Reconstructs the original secret using Lagrange interpolation on any k shares.
4. main: Demonstrates splitting and reconstructing a secret using the functions above.

Time Complexity:

- Splitting (splitSecret):
  - Generating coefficients: O(k)
  - Evaluating polynomial for n values: O(n * k)
  - Total: O(nk)
- Reconstruction (reconstructSecret):
  - Lagrange interpolation over k points: $O(k^2)$

Space Complexity:

- Splitting:
  - Storing coefficients: O(k)
  - Storing shares: O(n)
- Reconstruction:
  - Using only k shares: O(k)
  - Overall: O(n + k)