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:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#define PRIME 104729 // A large prime number

typedef struct {

int x;

int y;

} Share;

// Function to compute (base^exp) % mod

int mod\_pow(int base, int exp, int mod) {

int result = 1;

base %= mod;

while (exp > 0) {

if (exp % 2 == 1)

result = (int)((1LL \* result \* base) % mod);

base = (int)((1LL \* base \* base) % mod);

exp /= 2;

}

return result;

}

// Function to compute modular inverse of a modulo mod

int mod\_inverse(int a, int mod) {

a %= mod;

for (int x = 1; x < mod; x++) {

if ((1LL \* a \* x) % mod == 1)

return x;

}

return -1; // No inverse exists

}

// Evaluates a polynomial at a point x

int evaluate\_polynomial(int \*coeffs, int k, int x) {

int y = 0;

for (int i = 0; i < k; i++) {

y = (y + (int)(1LL \* coeffs[i] \* mod\_pow(x, i, PRIME)) % PRIME) % PRIME;

}

return y;

}

// Splits the secret into n shares with a threshold of k

Share\* split\_secret(int secret, int k, int n) {

int \*coeffs = malloc(k \* sizeof(int));

coeffs[0] = secret;

for (int i = 1; i < k; i++) {

coeffs[i] = rand() % PRIME;

}

Share\* shares = malloc(n \* sizeof(Share));

for (int i = 0; i < n; i++) {

int x = i + 1;

int y = evaluate\_polynomial(coeffs, k, x);

shares[i].x = x;

shares[i].y = y;

}

free(coeffs);

return shares;

}

// Reconstructs the secret from k shares

int reconstruct\_secret(Share \*shares, int k) {

int secret = 0;

for (int i = 0; i < k; i++) {

int xi = shares[i].x;

int yi = shares[i].y;

int li = 1;

for (int j = 0; j < k; j++) {

if (i != j) {

int xj = shares[j].x;

int numerator = (-xj + PRIME) % PRIME;

int denominator = (xi - xj + PRIME) % PRIME;

int inv = mod\_inverse(denominator, PRIME);

li = (int)((1LL \* li \* numerator % PRIME) \* inv % PRIME);

}

}

secret = (secret + (int)(1LL \* yi \* li % PRIME)) % PRIME;

}

return secret;

}

int main() {

srand((unsigned int)time(NULL));

int secret = 12345;

int k = 3;

int n = 5;

printf("Original Secret: %d\n", secret);

Share\* shares = split\_secret(secret, k, n);

printf("Generated Shares:\n");

for (int i = 0; i < n; i++) {

printf("(%d, %d)\n", shares[i].x, shares[i].y);

}

// Use the first k shares to reconstruct

int recovered = reconstruct\_secret(shares, k);

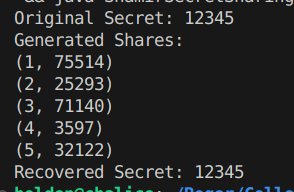
printf("Recovered Secret: %d\n", recovered);

free(shares);

return 0;

}

**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(k²)

**Space Complexity:**

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