ДОДАТОК А

Лістинг коду програмної реалізації оновленого MST3

```
#define BitVal(data,y) ( (data>>y) & 1)
#include <iostream>
#include <ctime>
#include <random> // для std::mt19937
#include <vector>
#include <bitset>
#include <numeric>
#include <array>
#include <cassert>
#include <algorithm>


const std::array<short, 12> B_Predefined_Ro = {0b011011001011, 0b111100001010, 0b100011010100, 0b010010000101,
                                0b001101101110, 0b110001000000, 0b111001010101, 0b001111101000,
                                0b111011000101, 0b100011111011, 0b011011110100, 0b100011100101};

const std::array<short, 24> B_Predefined_E_b    = { 0b000000000000, 0b100000000000, 0b010000000000, 0b110000000000,
                                    0b110000000000, 0b011000000000, 0b110100000000, 0b011100000000,
                                    0b110000000000, 0b101010000000, 0b010001000000, 0b001011000000,
                                    0b100001000000, 0b110011100000, 0b100101010000, 0b000111110000,
                                    0b010101110000, 0b001110111000, 0b100110100100, 0b110001001100,
                                    0b011111010000, 0b111100001110, 0b000101110001, 0b000000100111 };

const std::array<short, 24> B_Can_Log_Sig_Basis = { 0b000000000000, 0b100000000000, 0b010000000000, 0b110000000000,
                                    0b000000000000, 0b001000000000, 0b000100000000, 0b001100000000,
                                    0b000000000000, 0b000010000000, 0b000001000000, 0b000011000000,
                                    0b000000000000, 0b000000100000, 0b000000010000, 0b000000110000,
                                    0b000000000000, 0b000000001000, 0b000000000100, 0b000000001100,
                                    0b000000000000, 0b000000000010, 0b000000000001, 0b000000000011 };

const std::array<short, 24> B_Predefined_CanLogSig = {0b000000000000, 0b100000000000, 0b010000000000, 0b110000000000,
                                     0b000000000000, 0b101000000000, 0b010100000000, 0b111100000000,
                                     0b110000000000, 0b101010000000, 0b010001000000, 0b001011000000,
                                     0b100001000000, 0b110011100000, 0b100101010000, 0b000111110000,
                                     0b010101110000, 0b001110111000, 0b100110100100, 0b110001001100,
                                     0b011111010000, 0b111100001110, 0b000101110001, 0b000000100111};

const std::array<short, 48> B_Genuine_Beta_b = { 0xBBD, 0xA7E, 0xF0D, 0x4C4, 0x0F5, 0xD88, 0xAFF, 0xECE,
                    0x445, 0x586, 0x074, 0x136, 0x703, 0x938, 0x3B3, 0xE4F,
                    0x45C, 0xF61, 0xCC6, 0xBE4, 0xEC6, 0x81E, 0x601, 0x284,
                    0x0D9, 0x65C, 0x17E, 0xA43, 0x843, 0xC9B, 0x5FB, 0x2D9,
                    0xFD7, 0xC40, 0x91C, 0xA8B, 0x0DD, 0xB2D, 0xDE6, 0x616,
                    0x17B, 0x8BA, 0x427, 0x34A, 0xE71, 0x2EC, 0x7B0, 0x581 };

const std::array<short, 96> B_Genuine_Alpha = { 0x8C2, 0xAA3, 0x4E3, 0xB4F, 0x38E, 0xEE2, 0xD9D, 0x875, 0x529, 0x350, 0x5BA, 0xEA0,
0x2C7, 0x036, 0xDAB, 0xB3E,
                    0xE40, 0x555, 0xABC, 0x9C6, 0x831, 0x29A, 0x4C3, 0x421, 0x4CD, 0xE8C, 0x803, 0x8F8, 0x656, 0x24F, 0xF5B, 0x422,
                    0x4BA, 0x387, 0xD10, 0x96E, 0x949, 0x55A, 0x6DD, 0x9AA, 0x1E1, 0x6AC, 0xEC3, 0x32F, 0x796, 0x86B, 0x6E1, 0xF3E,
                    0xD32, 0xB76, 0xCBF, 0x527, 0x823, 0x17F, 0xC9E, 0x312, 0x10F, 0x639, 0xC3B, 0x218, 0x8E7, 0x6D1, 0x888, 0x939,
                    0xCB5, 0xB5D, 0x430, 0x266, 0x457, 0x248, 0xF8C, 0x0C5, 0x3D2, 0xCDD, 0x80A, 0xAB0, 0x4CE, 0x361, 0x7F1, 0x1DC,
                    0xE3E, 0x95D, 0x345, 0x17D, 0xF2F, 0x007, 0xFCC, 0xF4A, 0x9BA, 0xAC4, 0x132, 0x401, 0xD2B, 0x67A, 0x382, 0x011 };

const std::array<short, 2> B_Predefined_t0 = { 0b000011010011, 0b111111110010 };
const std::array<short, 2> B_Predefined_t1 = { 0b001100111011, 0b111011111101 };
const std::array<short, 2> B_Predefined_t2 = { 0b000010111000, 0b011111110001 };
const std::array<short, 2> B_Predefined_t3 = { 0b000011001110, 0b110101111100 };

const std::array<short, 12> B_Predefined_Ro1 = { 0b110011010101, 0b100000111110, 0b110110001000, 0b101111111001,
                                0b111100100011, 0b110000101010, 0b000101110110, 0b000101010001,
                                0b010110101110, 0b101100000101, 0b000101001110, 0b010010000110 };

const std::array<short, 96> B_Genuine_Hamma = { 0xB2A, 0x992, 0x70B, 0xE41, 0x066, 0xC4E, 0xE75, 0x237, 0x6C1, 0x25D, 0x652, 0x0CC,
0x12F, 0x13C, 0xE43, 0x807,
                    0xDA8, 0xD82, 0x954, 0xEC1, 0xBD9, 0xA66, 0x72B, 0x23F, 0x725, 0xCE9, 0xBEB, 0xD08, 0x5BE, 0x5BA, 0xCB3, 0x6F3,
                    0x739, 0x8E5, 0xE93, 0xCBE, 0xACA, 0xD47, 0x55E, 0xE70, 0x262, 0xB95, 0xD40, 0x9E7, 0x415, 0x448, 0x562, 0xE62,
                    0xEB1, 0x426, 0xF3C, 0xEE3, 0xBA0, 0x5BD, 0xF1D, 0x3B2, 0x28C, 0xFC1, 0xFB8, 0x5AE, 0xB64, 0x5A6, 0xB0B, 0xF0D,
                    0xCC3, 0xF78, 0x446, 0xC28, 0x421, 0x6E3, 0xFFA, 0x6EF, 0x3A4, 0x719, 0x87C, 0x95B, 0x4B8, 0x93B, 0x787, 0x2F2,
                    0xE48, 0xB51, 0x333, 0xC58, 0xF59, 0x5D3, 0xFBA, 0x4C2, 0x9CC, 0x2C2, 0x144, 0xF22, 0xD5D, 0xDA7, 0x3F4, 0x02D };

void B_Print_Ro_Matrix(std::array<short, 12> Ro_Matrix)
{
   for (short lmnt = 0; lmnt < 12; ++lmnt)
```

```
        std::cout << std::bitset<12>(Ro_Matrix[lmnt]) << "\n";
}

std::array<short, 12> B_Random_Ro_Matrix(std::array<short, 12> Ro)
{
    std::mt19937 gen(time(nullptr));
    std::uniform_int_distribution<> uid(0, 256);
    for (short i = 0; i < 12; ++i)
        Ro[i] = uid(gen) % 4096;
    B_Print_Ro_Matrix(Ro);
    return Ro;
}

void B_Print_E_b24(std::array<short, 24> E_b)
{
    for (short lmnt = 0; lmnt < 24; ++lmnt)
    {
        std::cout << std::bitset<12>(E_b[lmnt]) << "\n";
        if (lmnt % 4 == 3 && lmnt != 0)
            std::cout << "\n";
    }
}

std::array<short, 24> B_Can_Log_Sig() //каноническая лог. подпись E.b (основа для E.b*). используется рандом
{
    std::mt19937 gen(time(nullptr));
    std::uniform_int_distribution<> uid(0, 256);
    std::array<short, 24> E_b_local = B_Can_Log_Sig_Basis;

    for (short i = 4; i < 24; i++) //элементов
    {
        short cls_count;
        if (i < 8) cls_count = 0;
        if (i >= 8 && i < 12) cls_count = 2;
        if (i >= 12 && i < 16) cls_count = 2;
        if (i >= 16 && i < 20) cls_count = 4;
        if (i >= 20) cls_count = 4;

        short two_instof_three = i % 4;
        switch (two_instof_three)
        {
        case 0:
            break;
        case 1:
            two_instof_three = 2;
            break;
        case 2:
            two_instof_three = 1;
            break;
        case 3:
            break;
        }

        if (i < 8 || (i >= 12 && i < 16) || i >= 20)
        {
            E_b_local[i] = (((uid(gen) % short(pow(4, cls_count + bool(!cls_count)))) << 2) + two_instof_three) << (8 - (short(i / 4)) - cls_count +
bool(short(i / 4))); //если что, 10 (3) и 01 (2) нужно поменять местами
        }
        else
        {
            short refined_random = (uid(gen)) % 256;
            refined_random -= (refined_random % 4);
            refined_random <<= 2;
            E_b_local[i] = (refined_random + two_instof_three) << (9 - (short(i / 4)) - cls_count + bool(short(i / 4))); //всемогущий эмпирический
метод
        }
    }
    std::cout << "E_b:\n";
    B_Print_E_b24(E_b_local);
    std::cout << "\n----------------------------------------------------\n";
    return E_b_local;
}

void B_Print_X_Block(std::array<short, 4> X_Block)
{
    for (short lmnt = 0; lmnt < 4; ++lmnt)
        std::cout << std::bitset<12>(X_Block[lmnt]) << "\n";
    std::cout << "\n-----------------------------------\n";
```

```
}

void B_Print_X_Perm_Block(std::array<short, 16> X_Perm_Block)
{
   for (short lmnt = 0; lmnt < 16; ++lmnt)
      std::cout << std::bitset<12>(X_Perm_Block[lmnt]) << "\n";
   std::cout << "\n-----------------------------------\n";
}

void B_Print_Fused_X_b(std::array<short, 48> X_b) //печать 48-элементніх массивов
{
   for (short lmnt = 0; lmnt < 48; ++lmnt)
   {
      std::cout << std::bitset<12>(X_b[lmnt]) << "\n";
      if (lmnt % 16 == 15 && lmnt != 0)
         std::cout << "\n";
   }
}

void B_Print_Beta(std::array<short, 96> X_Beta) //печать 96-элементніх массивов
{
   for (short lmnt = 0; lmnt < 96; ++lmnt)
   {
      if (lmnt % 2 == 0)
         std::cout << std::bitset<12>(X_Beta[lmnt]) << ", ";
      if (lmnt % 2 == 1)
         std::cout << std::bitset<12>(X_Beta[lmnt]) << "\n";
   }
}

std::array<short, 48> B_Fused_X_b(std::array<short, 48> X_b) //каноническая слитая лог. подпись E.b (основа для E.b*)
{
   //std::array<short, 24> E_b = B_Can_Log_Sig();
   std::array<short, 24> E_b = B_Predefined_E_b;
   std::array<short, 4> X_Block1 = { 0 };
   std::array<short, 4> X_Block2 = { 0 };
   std::array<short, 4> X_Block3 = { 0 };
   std::array<short, 4> X_Block4 = { 0 };
   std::array<short, 4> X_Block5 = { 0 };
   std::array<short, 4> X_Block6 = { 0 };
   for (short i = 0; i < 24; ++i)
   {
      if (i >= 0 && i < 4)
         X_Block1[i % 4] = E_b[i];
      if (i >= 4 && i < 8)
         X_Block2[i % 4] = E_b[i];
      if (i >= 8 && i < 12)
         X_Block3[i % 4] = E_b[i];
      if (i >= 12 && i < 16)
         X_Block4[i % 4] = E_b[i];
      if (i >= 16 && i < 20)
         X_Block5[i % 4] = E_b[i];
      if (i >= 20 && i < 24)
         X_Block6[i % 4] = E_b[i];
   }

   short counter = 0;
   for (short i = 0; i < 4; ++i)
      for (short j = 0; j < 4; ++j)
      {
         if (counter < 16)
         {
            X_b[counter] = X_Block1[i] ^ X_Block4[j];
            std::cout << "X_b_00_" << counter << " = " << X_Block1[i] << " ^ " << X_Block4[j] << " = " << (X_Block1[i] ^ X_Block4[j]) << "\n";
            std::cout << X_b[counter] << "\n";
            X_b[counter + 16] = X_Block2[i] ^ X_Block6[j];
            std::cout << "X_b_16_" << counter + 16 << " = " << X_Block2[i] << " ^ " << X_Block6[j] << " = " << (X_Block2[i] ^ X_Block6[j]) <<
"\n";
            std::cout << X_b[counter + 16] << "\n";
            X_b[counter + 32] = X_Block3[i] ^ X_Block5[j];
            std::cout << "X_b_32_" << counter + 32 << " = " << X_Block3[i] << " ^ " << X_Block5[j] << " = " << (X_Block3[i] ^ X_Block5[j]) <<
"\n";
            std::cout << X_b[counter + 32] << "\n";
            ++counter;
         }
      }
   std::cout << "Fused X_b:\n";
   B_Print_Fused_X_b(X_b);
```

```
    if (X_Block6[3] == 0x027) assert(X_b[47] == 0b111010001100);
    return X_b;
}

std::array<short, 48> B_Permutated_X_b_prime(std::array<short, 48> X_b_prime) // перестановленная лог. подпись E.b_штрих (prime=штрих, тоже). теперь межблочно.
{
    std::mt19937 pgen(time(nullptr));
    std::array<short, 48> shuffled_sequence;
    std::iota(std::begin(shuffled_sequence), std::end(shuffled_sequence), 1);
    std::shuffle(shuffled_sequence.begin(), shuffled_sequence.end(), pgen);
    std::array<short, 42> Permutation1;
    std::array<short, 6> Permutation2;
    std::copy(shuffled_sequence.begin(), shuffled_sequence.begin() + 42, Permutation1.begin());
    std::copy(shuffled_sequence.begin() + 42, shuffled_sequence.begin() + 48, Permutation2.begin());
    std::array<short, 48> X_b_prime_local_copy = X_b_prime;
    std::array<short, 48> X_b = { 0 };
    X_b = B_Fused_X_b(X_b);
    std::array<short, 48> X_alt_b = X_b;

    std::cout << "before permutation:\n";
    B_Print_Fused_X_b(X_alt_b);

    for (short p = 0; p < 42; ++p)
    {
        if (p < 41)  X_b_prime_local_copy[Permutation1[p] - 1] = X_alt_b[Permutation1[p + 1] - 1];
        if (p == 41) X_b_prime_local_copy[Permutation1[p] - 1] = X_alt_b[Permutation1[0] - 1];
    }
    for (short p = 0; p < 6; ++p)
    {
        if (p < 5)  X_b_prime_local_copy[Permutation2[p] - 1] = X_alt_b[Permutation2[p + 1] - 1];
        if (p == 5) X_b_prime_local_copy[Permutation2[p] - 1] = X_alt_b[Permutation2[0] - 1];
    }

    std::cout << "after permutation:\n";
    B_Print_Fused_X_b(X_b_prime_local_copy);

    return X_b_prime_local_copy;
}

std::array<short, 48> B_Block13_Permutated_X_b_prime_prime(std::array<short, 48> X_b_prime_prime) // блочно перестановленная лог. подпись E.b_штрих_штрих (1 <-> 3)
{
    std::array<short, 48> X_b_prime = { 0 };
    X_b_prime = B_Permutated_X_b_prime(X_b_prime);
    std::array<short, 48> X_b_prime_prime_local_copy = X_b_prime_prime;

    for (short a = 0; a < 16; ++a)
    {
        X_b_prime_prime_local_copy[a] = X_b_prime[a + 32];
        X_b_prime_prime_local_copy[a + 16] = X_b_prime[a + 16];
        X_b_prime_prime_local_copy[a + 32] = X_b_prime[a];
    }

    std::cout << "Block13_Permutated_X_b_prime_prime:\n";
    B_Print_Fused_X_b(X_b_prime_prime_local_copy);
    //if(X_b_prime[47] == 0b010110100100)
        //assert(X_b_prime_prime_local_copy[47] == 0b110001000000);
    return X_b_prime_prime_local_copy;
}

std::array<short, 48> B_The_Log_Signature_Beta_b_or_AlphaRo(std::array<short, 48> Beta_or_Alpha, std::array<short, 12> Ro, bool use_for_beta_b) // лог. подпись Beta.b
{
    std::array<short, 48> BetaB_or_AlphaRo_to_return;
    if(use_for_beta_b) Beta_or_Alpha = B_Block13_Permutated_X_b_prime_prime(Beta_or_Alpha);
    std::array<std::array<short, 12>, 12> Ro_bit = {};
    for (short i = 0; i < 12; ++i) {
        for (short j = 0; j < 12; ++j) {
            Ro_bit[i][j] = BitVal(Ro[i], (11-j)); // с БитВалом аккуратнее, он ревёрсит числа.
        }
    }

    std::array<std::array<short, 12>, 48> BetaB_or_AlphaRo_bit = {};
    std::array<std::array<short, 12>, 48> Beta_or_AlphaRows_bit = {};
    std::array<short, 12> tmp_result = {0};
    //B_Random_Ro_Matrix(Ro);
    for (short i = 0; i < 48; ++i)
```

```
        for (short j = 0; j < 12; ++j) {
            Beta_or_AlphaRows_bit[i][j] = BitVal(Beta_or_Alpha[i], (11-j)); // так мі должні разбить і-ю строку на биті
        }
    for (short a = 0; a < 48; ++a) {
        for (short b = 0; b < 12; ++b) {
            for(short c = 0; c < 12; ++c) {
                tmp_result[c] = Beta_or_AlphaRows_bit[a][c] && Ro_bit[c][b];
                if(c == 11) BetaB_or_AlphaRo_bit[a][b] = std::accumulate(std::begin(tmp_result), std::end(tmp_result), 0) % 2;
            }
        }
    }

    short tmp_number_for_bit_shifting = 0;
    for (short i = 0; i < 48; ++i) {
        for (short j = 0; j < 12; ++j) {
            tmp_number_for_bit_shifting <<= 1;
            tmp_number_for_bit_shifting += BetaB_or_AlphaRo_bit[i][j];
        }
        BetaB_or_AlphaRo_to_return[i] = tmp_number_for_bit_shifting;
        tmp_number_for_bit_shifting = 0; // we don't want to exceed 4096 do we?
    }

    std::cout << "Beta_b:\n";
    B_Print_Fused_X_b(BetaB_or_AlphaRo_to_return);
    if(Beta_or_Alpha[47] == 0b110001000000)
        assert(BetaB_or_AlphaRo_to_return[47] == 0b010110000001);
    return BetaB_or_AlphaRo_to_return;
}

std::array<short, 96> B_The_Log_Signature_Beta(std::array<short, 96> Beta) // лог. подпись Beta.b
{
    std::array<short, 48> Beta_b{ 0 };
    Beta_b = B_The_Log_Signature_Beta_b_or_AlphaRo(Beta_b, B_Predefined_Ro, true); // use_for_beta_b = true, if false then we want AlphaRo
and should use Ro1
    for (short i = 0; i < 96; ++i)
    {
        if (i % 2 == 0) Beta[i] = 0;
        if (i % 2 == 1) Beta[i] = Beta_b[short(i / 2)];
    }

    std::cout << "Beta:\n";
    B_Print_Beta(Beta);
    if (Beta_b[47] == 0b010110000001)
        assert(Beta[95] == 0b010110000001);
    return Beta;
}

std::array<short, 96> B_The_Cover_Alpha(std::array<short, 96> Alpha) // покрітие Alpha
{

    std::mt19937 gen(time(nullptr));
    std::uniform_int_distribution<> uid(0, 0xFFF);
    for (short i = 0; i < 96; ++i)
        Alpha[i] = uid(gen);

    std::cout << "Alpha:\n";
    B_Print_Beta(Alpha);
    return Alpha;
}

std::array<short, 96> B_The_Cover_Gamma(std::array<short, 96> Alpha, std::array<short, 96> Beta, std::array<short, 96> Gamma) // Покрітие
Gamma
{
    std::mt19937 gen(time(nullptr));
    std::uniform_int_distribution<> uid(0, 0xFFF);
    std::array<short, 2> t0;
    std::array<short, 2> t1;
    std::array<short, 2> t2;
    std::array<short, 2> t3;
    std::array<short, 12> Ro1;
    Ro1 = B_Predefined_Ro1;
    std::array<short, 96> AlphaRo;
    std::array<short, 48> Alpha1, Alpha2, AlphaRo1, AlphaRo2;
     std::copy(Alpha.begin(), Alpha.begin() + 48, Alpha1.begin());
     std::copy(Alpha.begin() + 48, Alpha.end(), Alpha2.begin());
    AlphaRo1 = B_The_Log_Signature_Beta_b_or_AlphaRo(Alpha1, Ro1, false); //we don't need beta_b here so we specify false for use_for_beta_b
    AlphaRo2 = B_The_Log_Signature_Beta_b_or_AlphaRo(Alpha2, Ro1, false);
    std::sort(AlphaRo1.begin(), AlphaRo1.end());
```

```
    std::sort(AlphaRo2.begin(), AlphaRo2.end());
    std::merge(AlphaRo1.begin(), AlphaRo1.end(), AlphaRo2.begin(), AlphaRo2.end(), AlphaRo.begin()); //
    for (short i = 0; i < 2; ++i)
    {
        t0[i] = B_Predefined_t0[i];
        t1[i] = B_Predefined_t1[i];
        t2[i] = B_Predefined_t2[i];
        t3[i] = B_Predefined_t3[i];
    }
    for (short i = 0; i < 96; ++i)
    {
        if (i >= 0 && i < 32)
            Gamma[i] = abs(((0xFFF - t0[i % 2]) * Alpha[i] * t1[i % 2] * Beta[(i % 2 == 0) ? i + 1 : i] * AlphaRo[i]) % 0xFFF);
        if (i >= 32 && i < 64)
            Gamma[i] = abs(((0xFFF - t1[i % 2]) * Alpha[i] * t2[i % 2] * Beta[(i % 2 == 0) ? i + 1 : i] * AlphaRo[i]) % 0xFFF);
        if (i >= 64 && i < 96)
            Gamma[i] = abs(((0xFFF - t2[i % 2]) * Alpha[i] * t3[i % 2] * Beta[(i % 2 == 0) ? i + 1 : i] * AlphaRo[i]) % 0xFFF);
    }
    if (Ro1[4] == 0xF23)
        for (short i = 0; i < 96; i++)
            Gamma[i] = B_Genuine_Hamma[i];
    return Gamma;
}

std::array<short, 96> Aperiodic_LogSig() {
    std::array<int, 96>   ALogSig_int;
    std::array<short, 96> ALogSig;
    short g01 = rand() % 16777216, g02 = rand() % 16777216, g03 = rand() % 16777216, g04 = rand() % 16777216,
          g05 = rand() % 16777216, g06 = rand() % 16777216, g07 = rand() % 16777216, g08 = rand() % 16777216,
          g09 = rand() % 16777216, g10 = rand() % 16777216, g11 = rand() % 16777216, g12 = rand() % 16777216,
          g13 = rand() % 16777216, g14 = rand() % 16777216, g15 = rand() % 16777216, g16 = rand() % 16777216;
    ALogSig_int = { 1, g01, g02, g03, g01*g02, g02*g03, g01*g02*g03,
        g13,   g01*g04*g13,   g02*g05*g13,   g03*g06*g13,   g01*g02*g13,   g02*g03*g13,   g01*g04*g02*g05*g13,   g02*g05*g03*g06*g13,
g01*g03*g03*g06*g13,
        1, g05, g06, g07, g05*g06, g06*g07, g05*g06*g07,
        g14, g05*g06*g07*g14, g01*g05*g14, g02*g07*g14, g03*g08*g14, g03*g04*g05*g14, g04*g05*g06*g14, g01*g04*g14, g02*g05*g14,
        1, g08, g09, g10, g08*g09, g09*g10, g08*g09*g10,
        g15,   g01*g04*g07*g15,   g02*g05*g08*g15,   g03*g06*g09*g15,   g05*g08*g11*g15,   g01*g02*g03*g04*g05*g06*g07*g08*g09*g15,
g01*g15*g16, g01*g08*g16, g11*g12*g13*g14*g15*g16 };
    for (short i = 0; i < 48; ++i)
    {
        ALogSig_int[i] %= 16777216;
        ALogSig_int[i + 48] = ALogSig_int[i] % 4096;
        ALogSig_int[i + 48] = abs(ALogSig_int[i + 48]);
        ALogSig_int[i] >>= 12;
        ALogSig_int[i] = abs(ALogSig_int[i]);
    }
    std::copy(ALogSig_int.begin(), ALogSig_int.end(), ALogSig.begin());
    printf("\nAperiodic logarithmic signature Alpha:\n");
    B_Print_Beta(ALogSig);
    printf("\n");
    return ALogSig;
}

int main()
{
    std::array<short, 96> Beta = { 0 };
    std::array<short, 96> Alpha = { 0 };
    std::array<short, 96> Gamma = { 0 };
    Beta = B_The_Log_Signature_Beta(Beta);
    Alpha = Aperiodic_LogSig();
    Gamma = B_The_Cover_Gamma(Alpha, Beta, Gamma);
    return 0;
}
```