ДОДАТОК А

(довідковий)

ВИХІДНИЙ КОД ПРОГРАМИ

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "params.h"
#include "poly.h"
#include "pack.h"
#include "rng.h"
#include "hash.h"
#include "gen_polys.h"

void qtesla_hash_H(unsigned char* c_bin, poly *v, const unsigned char* rand)
{
    unsigned char t[PARAM_N * PARAM_K + HM_BYTES];
    int32_t cL, i, j, ii = 0;
     Dstu8845Ctx*pctx = dstu8845_alloc();
    //1  i: = 0
    for (j = 0; j < PARAM_K; j++)
    for (i = 0; i < PARAM_N; i++)
    {
        /*
            2  if v[i] > PARAM_Q/2 then
            3       v[i] := v[i] - PARAM_Q
            4  end if
        */
        if (v[j][i] > PARAM_Q / 2)
            v[j][i] -= PARAM_Q;
        //5  cL = v[i] mod(1 << PARAM_D);
        cL = v[j][i] & ((1 << PARAM_D) - 1);
        // 6  if cL > 2 (d-1) then cL -= 2d
        if (cL > (1 << (PARAM_D - 1)))
            cL -= (1 << PARAM_D);
        // 7 t[i] := ((v[i] - cL) / 2PARAM_D) mod 256;
```

```
        t[ii++] = ((v[j][i] - cL) >> PARAM_D) & 0xFF;
        // 8 i : = i + 1
        // 9 if i < PARAM_N then goto step 2

    }
    // 10 t:= t || rand
    memcpy(t + PARAM_N * PARAM_K, rand, HM_BYTES);
    //11 seedbytes_init(λ, ctx, t, PARAM_N + HM_BYTES)
    seedbytes_init(LAMBDA, pctx, t, PARAM_N * PARAM_K + HM_BYTES);
    //12 randombytes(ctx, c, CRYPTO_C_BYTES);   Додатковий буфер
    randombytes(pctx, c_bin, CRYPTO_C_BYTES);
     dstu8845_free(pctx);
}

static int qtesla_test_correctness(poly v)
{
  int32_t mask, left, val;
  uint32_t t0, t1;
  for (int i=0; i<PARAM_N; i++) {
    mask = (PARAM_Q/2 - v[i]) >> (32-1);
    val = ((v[i]-PARAM_Q) & mask) | (v[i] & ~mask);
    t0 = (uint32_t)(~(Abs(val) - (PARAM_Q/2 - PARAM_REJECTION))) >> (32-1);
    left = val;
    val = (val + (1<<(PARAM_D-1))-1) >> PARAM_D;
    val = left - (val << PARAM_D);

    t1  =  (uint32_t)(~(Abs(val)  -  ((1<<(PARAM_D-1))-PARAM_REJECTION)))  >>
(32-1);
    if ((t0 | t1) == 1)
      return 1;
  }
  return 0;
}

static int qtesla_check_z(poly z)
{
  for (int i=0; i<PARAM_N; i++) {
    if (z[i] < -(PARAM_B-PARAM_U) || z[i] > (PARAM_B-PARAM_U))
```

```
        return 1;
    }
    return 0;
}

int qtesla_sign(Dstu8845Ctx* ctx, unsigned char *sm, unsigned long long
*smlen, const unsigned char *m, unsigned long long mlen, const unsigned char*
sk)
{
    unsigned char c[CRYPTO_C_BYTES], randomness[CRYPTO_SEEDBYTES],
randomness_input[CRYPTO_RANDOMBYTES+CRYPTO_SEEDBYTES+HM_BYTES],
seeds[2*CRYPTO_SEEDBYTES];
    uint32_t pos_list[PARAM_H];
    int16_t sign_list[PARAM_H], s[PARAM_N], e[PARAM_K][PARAM_N];
    poly y, v [PARAM_K], Sc, Ec, z, a [PARAM_K];
    int res = 0, j;
    //int nonce = 0;  // Initialize domain separator for sampling y
    Dstu8845Ctx* pctx0 = dstu8845_alloc();
    qtesla_decode_sk(seeds, s, e, sk);
    randombytes(ctx, randomness_input+CRYPTO_RANDOMBYTES, CRYPTO_SEEDBYTES);
    memcpy(randomness_input, &seeds[CRYPTO_SEEDBYTES], CRYPTO_SEEDBYTES);
    hash64(randomness_input + CRYPTO_RANDOMBYTES + CRYPTO_SEEDBYTES, m, mlen);
    hash32(randomness, randomness_input, CRYPTO_RANDOMBYTES + CRYPTO_SEEDBYTES
+ HM_BYTES);
    // ctx0
    seedbytes_init(LAMBDA, pctx0, seeds, CRYPTO_SEEDBYTES);
    qtesla_gen_a(pctx0, a/*, seeds*/);
    seedbytes_init(LAMBDA, pctx0, randomness, CRYPTO_SEEDBYTES);
    while (1) {
        qtesla_gen_y(pctx0, y/*, ++nonce*/);
        for (j = 0; j < PARAM_K; j++)
          poly_mul(v[j], a[j], y);
        qtesla_hash_H(c, v, randomness_input + CRYPTO_RANDOMBYTES +
CRYPTO_SEEDBYTES);
        qtesla_gen_c(pos_list, sign_list, c);
        sparse_mul16(Sc, s, pos_list, sign_list);
        poly_add(z, y, Sc);
        if (qtesla_check_z(z) != 0) {
```

```
          continue;
        }
        for (j = 0; j < PARAM_K; j++)
        {
          sparse_mul16(Ec, e[j], pos_list, sign_list);
          poly_sub_correct(v[j], v[j], Ec);
          if (qtesla_test_correctness(v[j]) != 0) {
              break;
          }
        }
         if (j == PARAM_K)
              break;
    }
     for (unsigned long long i = 0; i < mlen; i++)
        sm[CRYPTO_BYTES+i] = m[i];
     *smlen = CRYPTO_BYTES + mlen;
     qtesla_encode_sig(sm, c, z);
       dstu8845_free(pctx0);
    return 0;
  }

  int qtesla_verify(unsigned char *m, unsigned long long *mlen, const unsigned char *sm, unsigned long long smlen, const unsigned char *pk)
  {
    unsigned char c[CRYPTO_C_BYTES], c_sig[CRYPTO_C_BYTES], seed[CRYPTO_SEEDBYTES], hm[HM_BYTES];
    uint32_t pos_list[PARAM_H];
    int16_t sign_list[PARAM_H];
    int32_t pk_t[PARAM_K][PARAM_N];
    poly w [PARAM_K], z, a [PARAM_K], Tc;
    int32_t j;
    if (smlen < CRYPTO_BYTES) return -1;
    qtesla_decode_sig(c, z, sm);
    if (qtesla_check_z(z) != 0) return -2;                 // Check norm of z
    qtesla_decode_pk(pk_t, seed, pk);

    Dstu8845Ctx *pctx0 = dstu8845_alloc();
    seedbytes_init(LAMBDA, pctx0, seed, CRYPTO_SEEDBYTES);
```

```
  qtesla_gen_a(pctx0, a/*, seed*/);
  qtesla_gen_c(pos_list, sign_list, c);
  for (j = 0; j < PARAM_K; j++)
  {
      sparse_mul32(Tc, pk_t[j], pos_list, sign_list);
      poly_mul(w[j], a[j], z);
      poly_sub_reduce(w[j], w[j], Tc);                    // Compute w = az
- tc
  }
  hash64(hm, sm + CRYPTO_BYTES, smlen - CRYPTO_BYTES);
  qtesla_hash_H(c_sig, w, hm);
  if (memcmp(c, c_sig, CRYPTO_C_BYTES)) return -3;
  *mlen = smlen-CRYPTO_BYTES;
  for (unsigned long long i = 0; i < *mlen; i++)
    m[i] = sm[CRYPTO_BYTES+i];
  dstu8845_free(pctx0);
  return 0;
}
```

Фрагмент коду 1 - Підпис та перевірка підпису

```
static const int64_t cdt_v[CDT_ROWS*CDT_COLS] = {
     0x0000000000000000LL, // 0
     0x023A1B3F94933202LL, // 1  160470696614638082
     0x06AD3C4C19410B24LL, // 2
     0x0B1D1E95803CBB73LL, // 3
     0x0F879D85E7AB7F6FLL, // 4
     0x13EA9C5C52732915LL, // 5
     0x18440933FFD2011BLL, // 6
     0x1C91DFF191E15D07LL, // 7
     0x20D22D0F2017900DLL, // 8
     0x25031040C1E626EFLL, // 9
     0x2922BEEBA163019DLL, // 10
     0x2D2F866A3C5122D3LL, // 11
     0x3127CE192059EF64LL, // 12
     0x350A1928231CB01ALL, // 13
     0x38D5082CD4FCC414LL, // 14
```

```
0x3C875A73B33ADA6BLL, // 15
0x401FEF0E67CD47D3LL, // 16
0x439DC59E3077B59CLL, // 17
0x46FFFEDA4FC0A316LL, // 18
0x4A45DCD32E9CAA91LL, // 19
0x4D6EC2F3922E5C24LL, // 20
0x507A35C1FB354670LL, // 21
0x5367DA64EA5F1C63LL, // 22
0x563775ED5B93E26ELL, // 23
0x58E8EC6B50CB95F8LL, // 24
0x5B7C3FD0B999197DLL, // 25
0x5DF18EA7664D810ELL, // 26
0x6049129F03B5CD6DLL, // 27
0x62831EF856A48427LL, // 28
0x64A01ED314BA206FLL, // 29
0x66A09363CA89DAA3LL, // 30
0x688512173EF213F5LL, // 31
0x6A4E42A8B137E138LL, // 32
0x6BFCDD302C5B888ALL, // 33
0x6D91A82DF797EAB8LL, // 34
0x6F0D7697EBA6A51DLL, // 35
0x707125ED27F05CF1LL, // 36
0x71BD9C544C184D8DLL, // 37
0x72F3C6C7FB380322LL, // 38
0x74149755088E5CC6LL, // 39
0x7521036D434271D4LL, // 40
0x761A02516A02B0CELL, // 41
0x77008B9461817A43LL, // 42
0x77D595B95BC6A0FELL, // 43
0x789A14EE338BB727LL, // 44
0x794EF9E2D7C53213LL, // 45
0x79F530BE414FE24DLL, // 46
0x7A8DA03110886732LL, // 47
0x7B1928A59B3AA79ELL, // 48
0x7B98A38CE58D06AELL, // 49
0x7C0CE2C7BAD3164ALL, // 50
0x7C76B02ADDE64EF2LL, // 51
0x7CD6CD1D13EE98F2LL, // 52
```

```
0x7D2DF24DA06E2473LL, // 53
0x7D7CCF81A5CD98B9LL, // 54
0x7DC40B76C24FB5D4LL, // 55
0x7E0443D92DE22661LL, // 56
0x7E3E0D4B91401720LL, // 57
0x7E71F37EC9C1DE8DLL, // 58
0x7EA07957CE6B9051LL, // 59
0x7ECA1921F1AF6404LL, // 60
0x7EEF44CBC73DA35BLL, // 61
0x7F10662D0574233DLL, // 62
0x7F2DDF53CDDCD427LL, // 63
0x7F480AD7DF028A76LL, // 64
0x7F5F3C324B0F66B2LL, // 65
0x7F73C018698C18A7LL, // 66
0x7F85DCD8D69F8939LL, // 67
0x7F95D2B96ED3DA10LL, // 68
0x7FA3DC55532D71BBLL, // 69
0x7FB02EFA1DDDC61ELL, // 70
0x7FBAFB038BAE76E4LL, // 71
0x7FC46C34F918B3E3LL, // 72
0x7FCCAA102B95464CLL, // 73
0x7FD3D828F7D49092LL, // 74
0x7FDA16756C11CF83LL, // 75
0x7FDF819A3A7BFE69LL, // 76
0x7FE4333332A5FEBDLL, // 77
0x7FE84217AA0DE2B3LL, // 78
0x7FEBC29AC3100A8BLL, // 79
0x7FEEC6C78F0D514ELL, // 80
0x7FF15E9914396F2ALL, // 81
0x7FF3982E4982FB97LL, // 82
0x7FF57FFA236862D1LL, // 83
0x7FF720EFD36F4850LL, // 84
0x7FF884AB61732BC7LL, // 85
0x7FF9B396CA3B383CLL, // 86
0x7FFAB50BD1DD3633LL, // 87
0x7FFB8F72BA84114BLL, // 88
0x7FFC485E115A3388LL, // 89
0x7FFCE4A3C3B92B98LL, // 90
```

```
0x7FFD6873AE755E4ALL, // 91
0x7FFDD76BD840FDA1LL, // 92
0x7FFE34AA86CE6870LL, // 93
0x7FFE82DE5CA6A885LL, // 94
0x7FFEC454ABAA26DFLL, // 95
0x7FFEFB0625FADB89LL, // 96
0x7FFF28A214B1160FLL, // 97
0x7FFF4E983945429DLL, // 98
0x7FFF6E217C168A6ALL, // 99
0x7FFF884787F2B986LL, // 100
0x7FFF9DEB70088602LL, // 101
0x7FFFAFCB7B419E48LL, // 102
0x7FFFBE882DABB8F8LL, // 103
0x7FFFCAA8A65BDA07LL, // 104
0x7FFFD49E66188754LL, // 105
0x7FFFDCC891191605LL, // 106
0x7FFFE376BC4B0583LL, // 107
0x7FFFE8EB54D33209LL, // 108
0x7FFFED5DAEE78F4ELL, // 109
0x7FFFF0FBC7A6933DLL, // 110
0x7FFFF3EBC43A9213LL, // 111
0x7FFFF64D375FC4CCLL, // 112
0x7FFFF83A354A0431LL, // 113
0x7FFFF9C83CE9BB0DLL, // 114
0x7FFFFB08FCAC61A6LL, // 115
0x7FFFFC0AF80A1A6FLL, // 116
0x7FFFFCDA127DDE76LL, // 117
0x7FFFFD8003E62E56LL, // 118
0x7FFFFE04B9BF9C5BLL, // 119
0x7FFFFE6EA82EF9BDLL, // 120
0x7FFFFEC30D64CD46LL, // 121
0x7FFFFF0629856684LL, // 122
0x7FFFFF3B6CEEE3F1LL, // 123
0x7FFFFF659E6F7BA6LL, // 124
0x7FFFFF86FAC1036ALL, // 125
0x7FFFFFA14E69EDE9LL, // 126
0x7FFFFFB60AF6ACB7LL, // 127
0x7FFFFFC65857AECFLL, // 128
```

```
0x7FFFFFD3230F314FLL, // 129
0x7FFFFFDD27BE0A17LL, // 130
0x7FFFFFE4FC86CDFFLL, // 131
0x7FFFFFEB18AA9E4CLL, // 132
0x7FFFFFEFDAB1FD73LL, // 133
0x7FFFFFF38D65D499LL, // 134
0x7FFFFFF66BD0EB8CLL, // 135
0x7FFFFFF8A4782371LL, // 136
0x7FFFFFFA5BEF7C27LL, // 137
0x7FFFFFFBAEEB0B4CLL, // 138
0x7FFFFFFCB3E55903LL, // 139
0x7FFFFFFD7C6FE192LL, // 140
0x7FFFFFFE163E99E3LL, // 141
0x7FFFFFFE8BFC2558LL, // 142
0x7FFFFFFEE5F1CE80LL, // 143
0x7FFFFFFF2A8C31FDLL, // 144
0x7FFFFFFF5EC3CD18LL, // 145
0x7FFFFFFF866F376BLL, // 146
0x7FFFFFFFA483A906LL, // 147
0x7FFFFFFFBB4780C4LL, // 148
0x7FFFFFFFCC79BEB2LL, // 149
0x7FFFFFFFD970CBE1LL, // 150
0x7FFFFFFFE3326D21LL, // 151
0x7FFFFFFFEA865AB8LL, // 152
0x7FFFFFFFF004A7C8LL, // 153
0x7FFFFFFFF420E4F9LL, // 154
0x7FFFFFFFF732B791LL, // 155
0x7FFFFFFFF97C764FLL, // 156
0x7FFFFFFFFB303DDDLL, // 157
0x7FFFFFFFFC73D5A3LL, // 158
0x7FFFFFFFFD63AA57LL, // 159
0x7FFFFFFFFE15140DLL, // 160
0x7FFFFFFFFE981196LL, // 161
0x7FFFFFFFFEF89992LL, // 162
0x7FFFFFFFFF3F9A0CLL, // 163
0x7FFFFFFFFF73BA0BLL, // 164
0x7FFFFFFFFF99EBBBLL, // 165
0x7FFFFFFFFFB5DAA0LL, // 166
```

```
0x7FFFFFFFFFCA3E7BLL, // 167
0x7FFFFFFFFFD91985LL, // 168
0x7FFFFFFFFFE3E70ALL, // 169
0x7FFFFFFFFFEBBE45LL, // 170
0x7FFFFFFFFFF16C5CLL, // 171
0x7FFFFFFFFFF587BELL, // 172
0x7FFFFFFFFFF87E7FLL, // 173
0x7FFFFFFFFFFAA108LL, // 174
0x7FFFFFFFFFFC29F5LL, // 175
0x7FFFFFFFFFFD43E8LL, // 176
0x7FFFFFFFFFFE0DD7LL, // 177
0x7FFFFFFFFFFE9E31LL, // 178
0x7FFFFFFFFFFF0530LL, // 179
0x7FFFFFFFFFFF4E88LL, // 180
0x7FFFFFFFFFFF82AALL, // 181
0x7FFFFFFFFFFFA7A6LL, // 182
0x7FFFFFFFFFFFC1D6LL, // 183
0x7FFFFFFFFFFFD458LL, // 184
0x7FFFFFFFFFFFE166LL, // 185
0x7FFFFFFFFFFFEA97LL, // 186
0x7FFFFFFFFFFFF10CLL, // 187
0x7FFFFFFFFFFFF594LL, // 188
0x7FFFFFFFFFFFF8C0LL, // 189
0x7FFFFFFFFFFFFAF7LL, // 190
0x7FFFFFFFFFFFFC83LL, // 191
0x7FFFFFFFFFFFFD96LL, // 192
0x7FFFFFFFFFFFFE56LL, // 193
0x7FFFFFFFFFFFFEDALL, // 194
0x7FFFFFFFFFFFFF36LL, // 195
0x7FFFFFFFFFFFFF75LL, // 196
0x7FFFFFFFFFFFFFA1LL, // 197
0x7FFFFFFFFFFFFFBFLL, // 198
0x7FFFFFFFFFFFFFD4LL, // 199
0x7FFFFFFFFFFFFFE2LL, // 200
0x7FFFFFFFFFFFFFECLL, // 201
0x7FFFFFFFFFFFFFF2LL, // 202
0x7FFFFFFFFFFFFFF7LL, // 203
0x7FFFFFFFFFFFFFFALL, // 204
```

```
      0x7FFFFFFFFFFFFFFCLL, // 205
      0x7FFFFFFFFFFFFFFDLL, // 206
      0x7FFFFFFFFFFFFFFELL, // 207
      0x7FFFFFFFFFFFFFFFLL, // 208
}; // cdt_v
#define DFIELD ((int64_t)(~(uint64_t)0 >> 1))
#define PRODIFF(diff, a_u, a_v, k) { \
    diff = (diff + (a_v[k] & DFIELD) - (a_u[k] & DFIELD)) >> (64-1); \
}
#define PROSWAP(swap, diff, a_u, a_v, k) { \
    swap = (a_u[k] ^ a_v[k]) & diff; a_u[k] ^= swap; a_v[k] ^= swap; \
}
#define PROSWAPG(swap, diff, g_u, g_v) { \
    swap = (g_u ^ g_v) & (int32_t)diff; g_u ^= swap; g_v ^= swap; \
}
#define MINMAX0(swap, diff, a_u, a_v) { \
    PRODIFF(diff, a_u, a_v, 0); \
    PROSWAP(swap, diff, a_u, a_v, 0); \
}
#if CDT_COLS > 1
#define MINMAX1(swap, diff, a_u, a_v) { \
        PRODIFF(diff, a_u, a_v, 1); \
        MINMAX0(swap, diff, a_u, a_v); \
        PROSWAP(swap, diff, a_u, a_v, 1); \
    }
#else
#define MINMAX1(swap, diff, a_u, a_v) MINMAX0(swap, diff, a_u, a_v)
#endif
#if CDT_COLS > 2
#define MINMAX2(swap, diff, a_u, a_v) { \
        PRODIFF(diff, a_u, a_v, 2); \
        MINMAX1(swap, diff, a_u, a_v); \
        PROSWAP(swap, diff, a_u, a_v, 2); \
    }
#else
#define MINMAX2(swap, diff, a_u, a_v) MINMAX1(swap, diff, a_u, a_v)
#endif
#if CDT_COLS > 3
```

```
#define MINMAX3(swap, diff, a_u, a_v) { \
        PRODIFF(diff, a_u, a_v, 3); \
        MINMAX2(swap, diff, a_u, a_v); \
        PROSWAP(swap, diff, a_u, a_v, 3); \
    }
#else
#define MINMAX3(swap, diff, a_u, a_v) MINMAX2(swap, diff, a_u, a_v)
#endif
#if CDT_COLS > 4
#define MINMAX4(swap, diff, a_u, a_v) { \
        PRODIFF(diff, a_u, a_v, 4); \
        MINMAX3(swap, diff, a_u, a_v); \
        PROSWAP(swap, diff, a_u, a_v, 4); \
    }
#else
#define MINMAX4(swap, diff, a_u, a_v) MINMAX3(swap, diff, a_u, a_v)
#endif
#if CDT_COLS <= 5
     // TODO: improve MINIMAX performance:
#define MINIMAX(a_u, a_v, g_u, g_v) { \
        int64_t diff = 0, swapa; int32_t swapg; \
        MINMAX4(swapa, diff, a_u, a_v); \
        PROSWAPG(swapg, diff, g_u, g_v); \
    }
#else
#error "Unsupported precision"
#endif
#define MINMAXG(a_u, a_v) {\
    int32_t diff = ((a_v & 0x7FFFFFFFL) - (a_u & 0x7FFFFFFFL)) >> (RADIX32-
1); \
    int32_t swap = (a_u ^ a_v) & diff; a_u ^= swap; a_v ^= swap; \
}

// Сортування обєднанням
static void knuthMergeExchangeKG(int64_t a[/*n*CDT_COLS*/], int32_t
g[/*n*/], unsigned int n)
{
     unsigned int t = 1;
```

```
	while (t < n - t) {
		t += t;
	}
	for (unsigned int p = t; p > 0; p >>= 1) {
		int64_t *ap = a + p * CDT_COLS;
		int64_t *a_i = a, *ap_i = ap;
		int32_t *gp = g + p;
		for (unsigned int i = 0; i < (unsigned int)(n - p); i++, a_i +=
CDT_COLS, ap_i += CDT_COLS) {
			if (!(i & p)) {
				MINIMAX(a_i, ap_i, g[i], gp[i]);
			}
		}
		for (unsigned int q = t; q > p; q >>= 1) {
			int64_t *ap_i = ap, *aq_i = a + q * CDT_COLS;
			int32_t *gq = g + q;
			for (unsigned int i = 0; i < (unsigned int)(n - q); i++,
ap_i += CDT_COLS, aq_i += CDT_COLS) {
				if (!(i & p)) {
					MINIMAX(ap_i, aq_i, gp[i], gq[i]);
				}
			}
		}
	}
}

// Сортування обєднанням
static void knuthMergeExchangeG(int32_t a[/*n*/], size_t n) {
	if (n <= 1) {
		return;
	}
	size_t t = 1;
	while (t < n - t) {
		t += t;
	}
	for (size_t p = t; p > 0; p >>= 1) {
		int32_t *ap = a + p;
		for (size_t i = 0; i < n - p; i++) {
```

```
                if (!(i & p)) {
                        MINMAXG(a[i], ap[i]);
                }
        }
        for (size_t q = t; q > p; q >>= 1) {
                int32_t *aq = a + q;
                for (size_t i = 0; i < n - q; i++) {
                        if (!(i & p)) {
                                MINMAXG(ap[i], aq[i]);
                        }
                }
        }
    }
}

typedef int64_t          sdigit_t;       // Signed 64-bit digit
#define CHUNK_SIZE 512
// Генерація полінома з коефіцієнтами згідно розподілу Гауса
void  qtesla_gen_gauss_poly/*qtesla_kmxGauss*/(Dstu8845Ctx*  ctx,  int32_t
z[]/*, const unsigned char* seed*/)
{
    sdigit_t sampk[(CHUNK_SIZE + CDT_ROWS) * CDT_COLS];
    int32_t sampg[CHUNK_SIZE + CDT_ROWS];
    randombytes(ctx,   (uint8_t*)sampk,   CHUNK_SIZE   *   CDT_COLS   *
sizeof(sdigit_t));
    memcpy(sampk + CHUNK_SIZE * CDT_COLS, cdt_v, CDT_ROWS * CDT_COLS *
sizeof(sdigit_t));
    for (int32_t i = 0; i < CHUNK_SIZE; i++)
        sampg[i] = i << 16;
    for (int32_t i = 0; i < CDT_ROWS; i++)
        sampg[CHUNK_SIZE + i] = 0xFFFF0000L ^ i;
    knuthMergeExchangeKG(sampk, sampg, CHUNK_SIZE + CDT_ROWS);
    int32_t prev_inx = 0;
    for (int i = 0; i < CHUNK_SIZE + CDT_ROWS; i++) {
        int32_t curr_inx = sampg[i] & 0xFFFFL;
        prev_inx ^= (curr_inx ^ prev_inx) & ((prev_inx - curr_inx) >>
(RADIX32 - 1));
```

```
		int32_t neg = (int32_t)(sampk[i * CDT_COLS] >> (RADIX - 1));  // 
Only the (so far unused) msb of the leading word
		sampg[i] |= ((neg & -prev_inx) ^ (~neg & prev_inx)) & 0xFFFFL;
	}
	knuthMergeExchangeG(sampg, CHUNK_SIZE + CDT_ROWS);
	for (int i = 0; i < CHUNK_SIZE; i++) {
		z[i] = (sampg[i] << (RADIX32 - 16)) >> (RADIX32 - 16);
	}
}

// Перевірка норми для малих поліномів
static int qtesla_check_ES(poly p, unsigned int bound)
{
	unsigned int i, j, sum = 0, limit = PARAM_N;
	int32_t temp, mask, list[PARAM_N];
	for (j = 0; j < PARAM_N; j++)
		list[j] = Abs(p[j]);
	for (j = 0; j < PARAM_H; j++) {
		for (i = 0; i < limit - 1; i++) {
			// If list[i+1] > list[i] then exchange contents
			mask = (list[i + 1] - list[i]) >> (32 - 1);
			temp = (list[i + 1] & mask) | (list[i] & ~mask);
			list[i + 1] = (list[i] & mask) | (list[i + 1] & ~mask);
			list[i] = temp;
		}
		sum += (unsigned int)list[limit - 1];
		limit -= 1;
	}
	if (sum > bound)
		return 1;
	return 0;
}

// Генерація ключів
int qtesla_keypair(Dstu8845Ctx* ctx, unsigned char *pk, unsigned char *sk)
{
	unsigned char randomness[CRYPTO_RANDOMBYTES], randomness_extended[4 * 
CRYPTO_SEEDBYTES];
```

```
poly s, e[PARAM_K], a[PARAM_K], t[PARAM_K];
int i, j;
Dstu8845Ctx *ctx0 = dstu8845_alloc();
int counts = 1 +  // SEED_S
     1 +     // SEED_E
     1 +     // SEED_A
     1;      // SEED_Y
randombytes(ctx, randomness, CRYPTO_RANDOMBYTES);
seedbytes_init(LAMBDA, ctx0, randomness, CRYPTO_RANDOMBYTES);
randombytes(ctx0, randomness_extended, counts * CRYPTO_SEEDBYTES);
seedbytes_init(LAMBDA,       ctx0,       randomness_extended       /*+
CRYPTO_SEEDBYTES*/, CRYPTO_SEEDBYTES);
do {  // Sample the secret polynomial
     qtesla_gen_gauss_poly(ctx0, s);
} while (qtesla_check_ES(s, PARAM_KEYGEN_BOUND_S) != 0);
uint8_t* prand = randomness_extended + CRYPTO_SEEDBYTES;
seedbytes_init(LAMBDA, ctx0, prand, CRYPTO_SEEDBYTES);
for (j = 0; j < PARAM_K; j++)
{
     for (i = 0; i < PARAM_K; i++)
     {
          do {
               qtesla_gen_gauss_poly(ctx0, e[i]);
          } while (qtesla_check_ES(e[i], PARAM_KEYGEN_BOUND_E) != 0);
     }
}
prand += CRYPTO_SEEDBYTES;
seedbytes_init(LAMBDA, ctx0, prand , CRYPTO_SEEDBYTES);
qtesla_gen_a(ctx0, a);
for (i = 0; i < PARAM_K; i++)
{
     poly_mul(t[i], a[i], s);
     poly_add_correct(t[i], t[i], e[i]);
}
qtesla_encode_sk(sk, s, e, prand);
qtesla_encode_pk(pk, t, prand);
dstu8845_free(ctx);
return 0; }
```

Фрагмент коду 2 – Генерація ключів та передбачення для розподілу Гауса

ДОДАТОК Б

(обов’язковий)

СЛАЙДИ ПРЕЗЕНТАЦІЇ

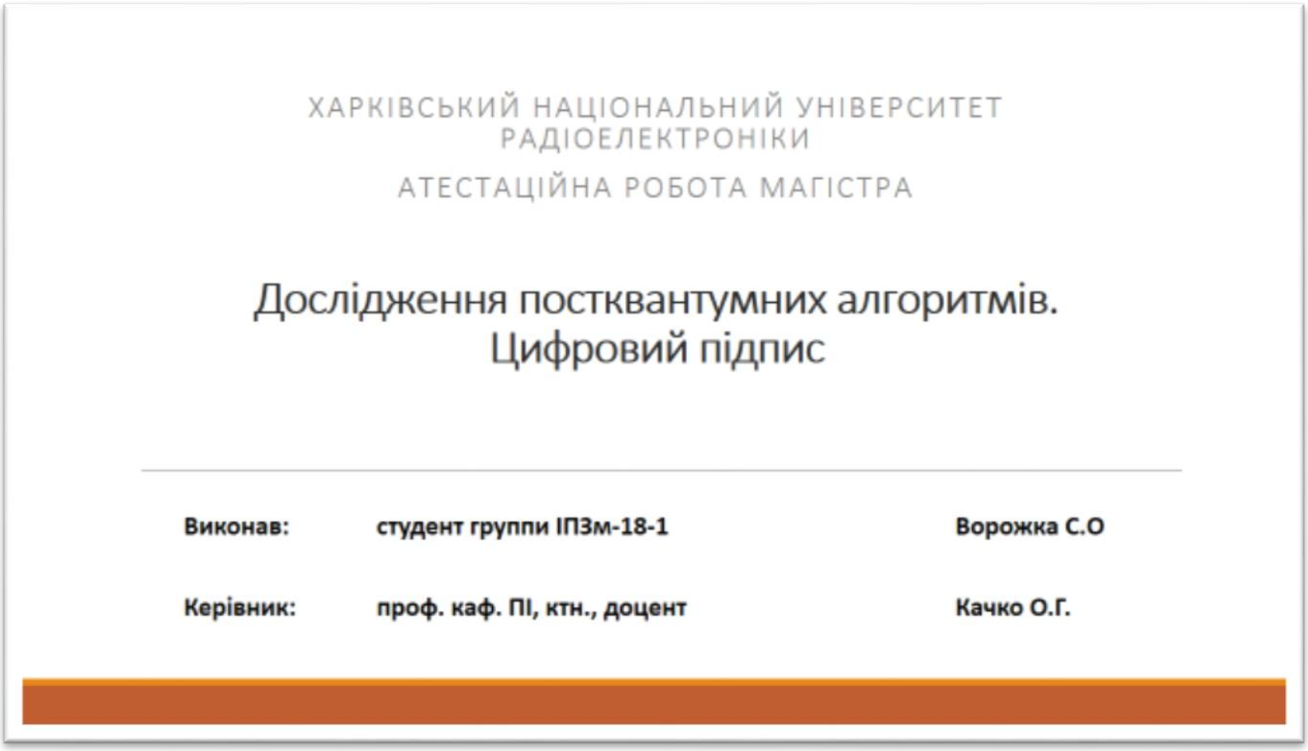

Рисунок Б.2 – Слайд 1

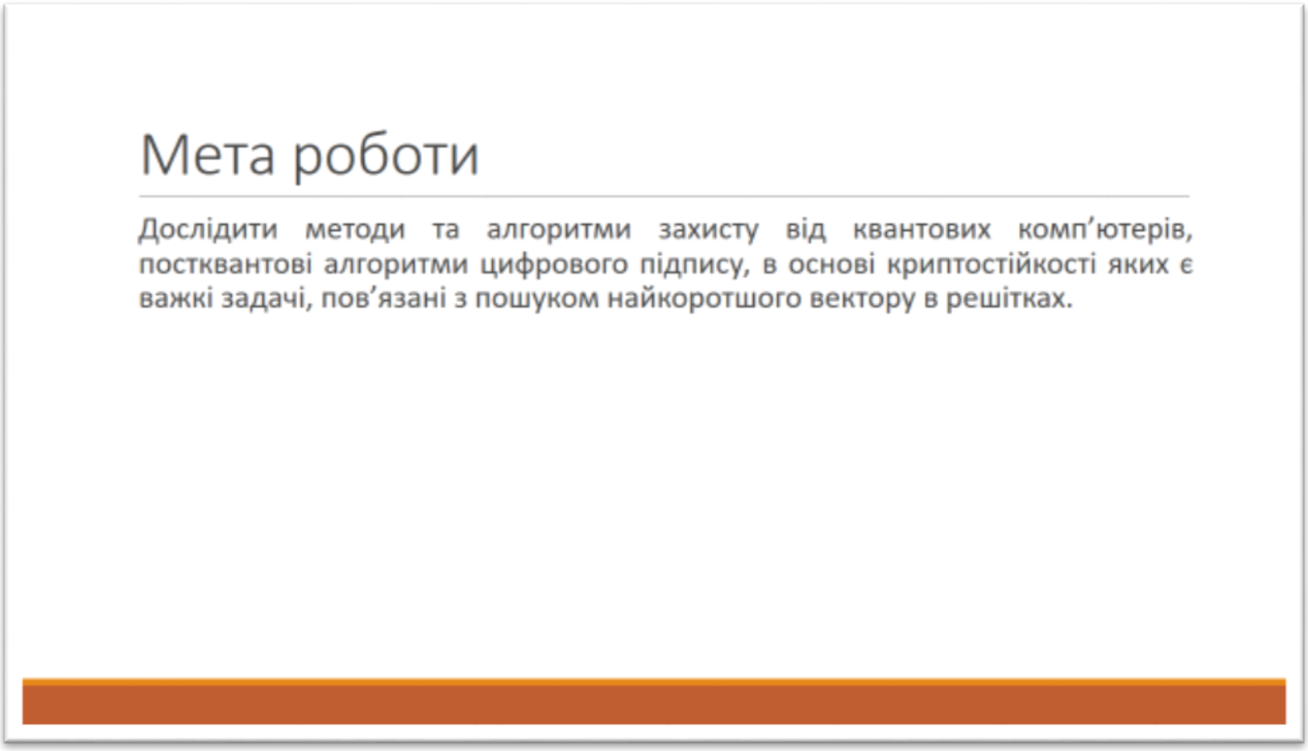

Рисунок Б.2 – Слайд 2

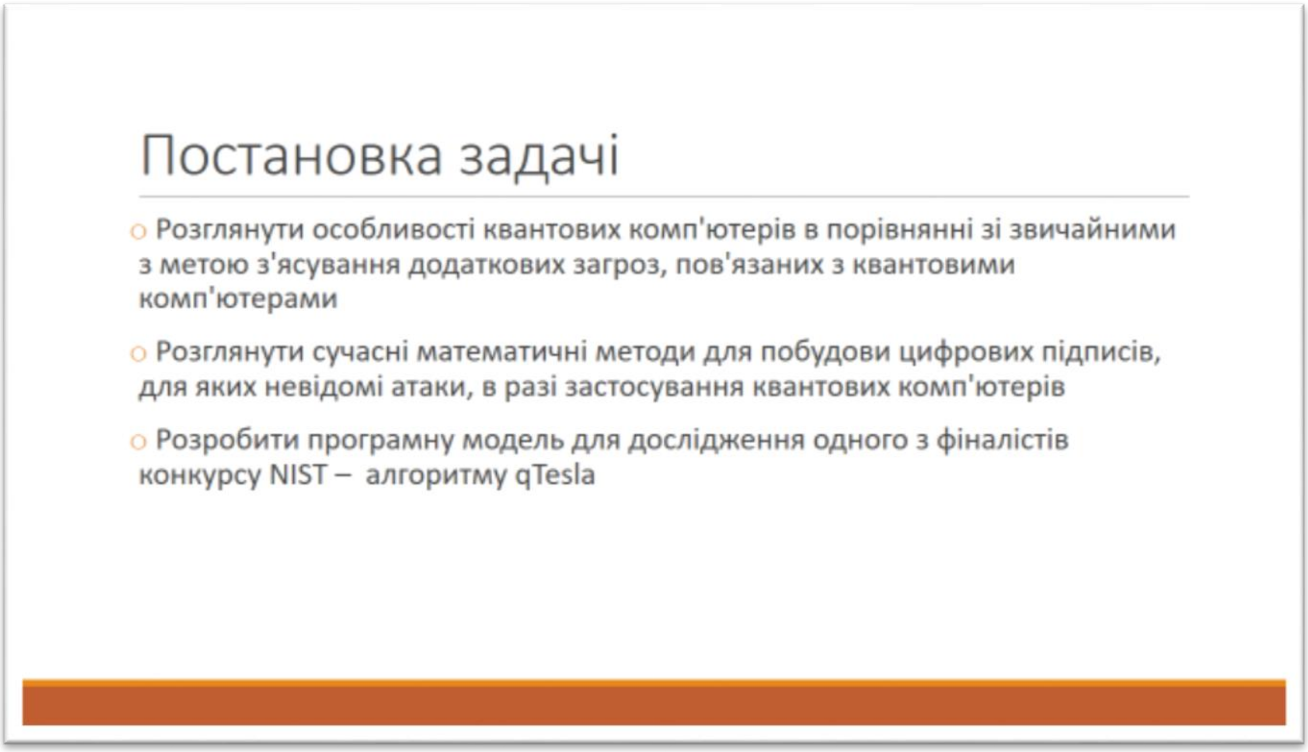

Рисунок Б.2 – Слайд 3

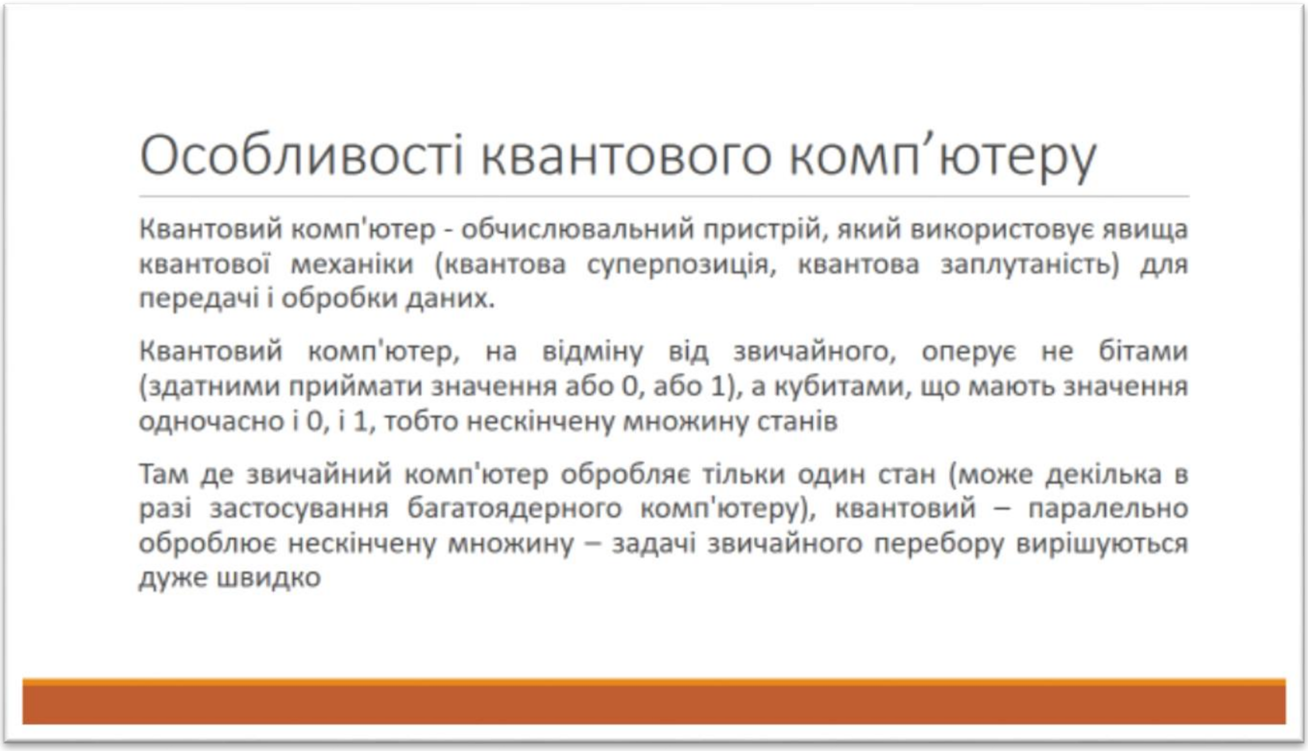

Рисунок Б.2 – Слайд 4

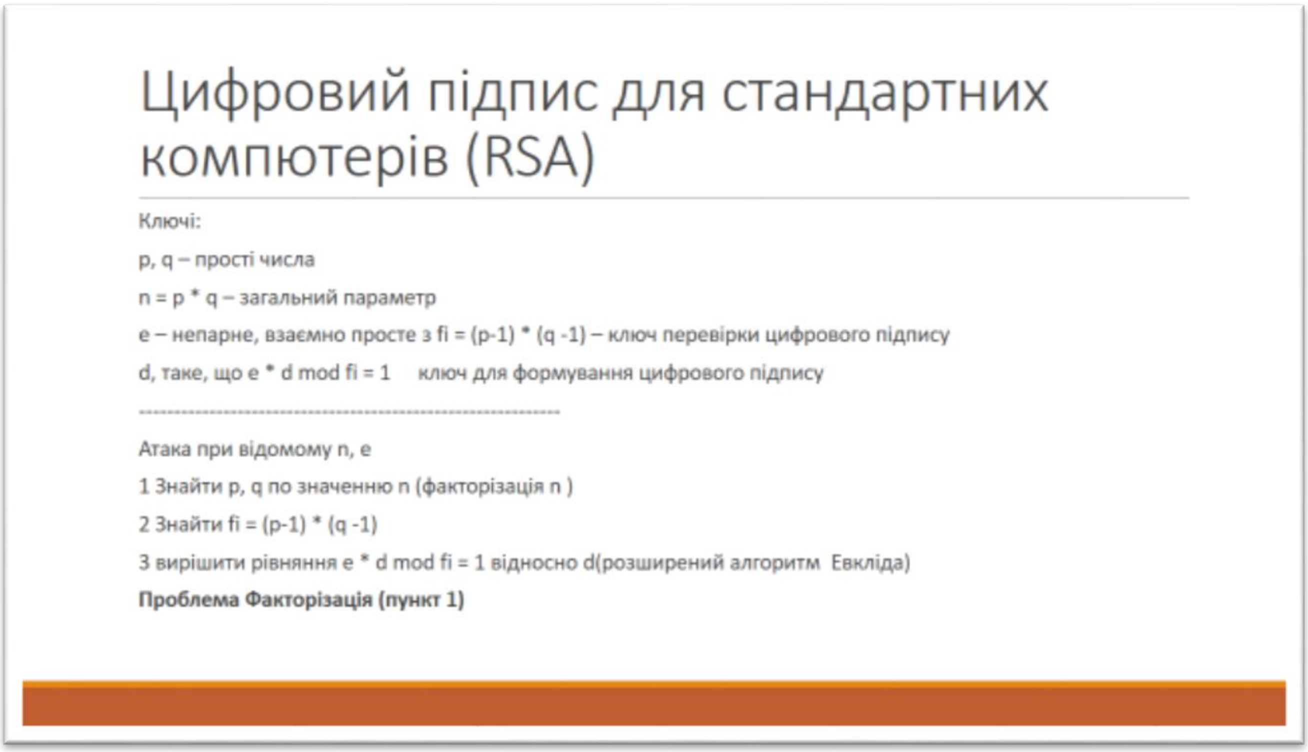

Рисунок Б.2 – Слайд 5

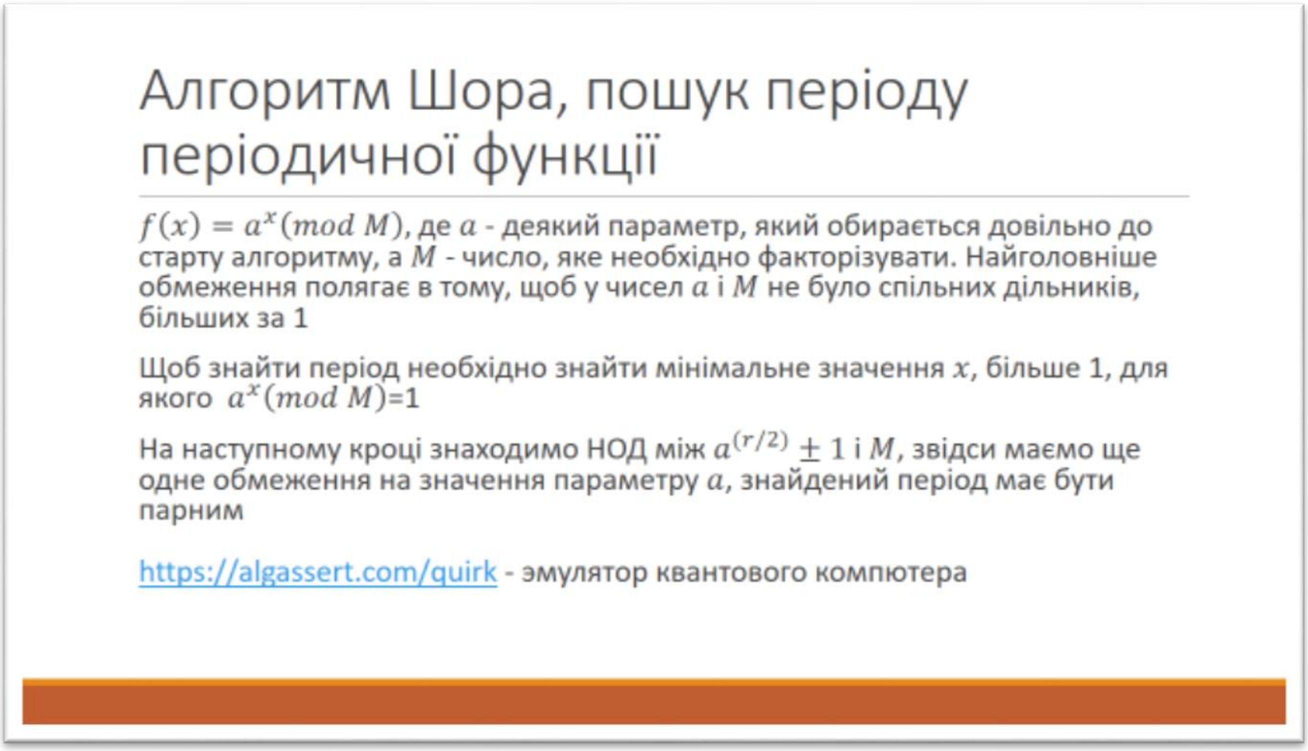

Рисунок Б.2 – Слайд 6

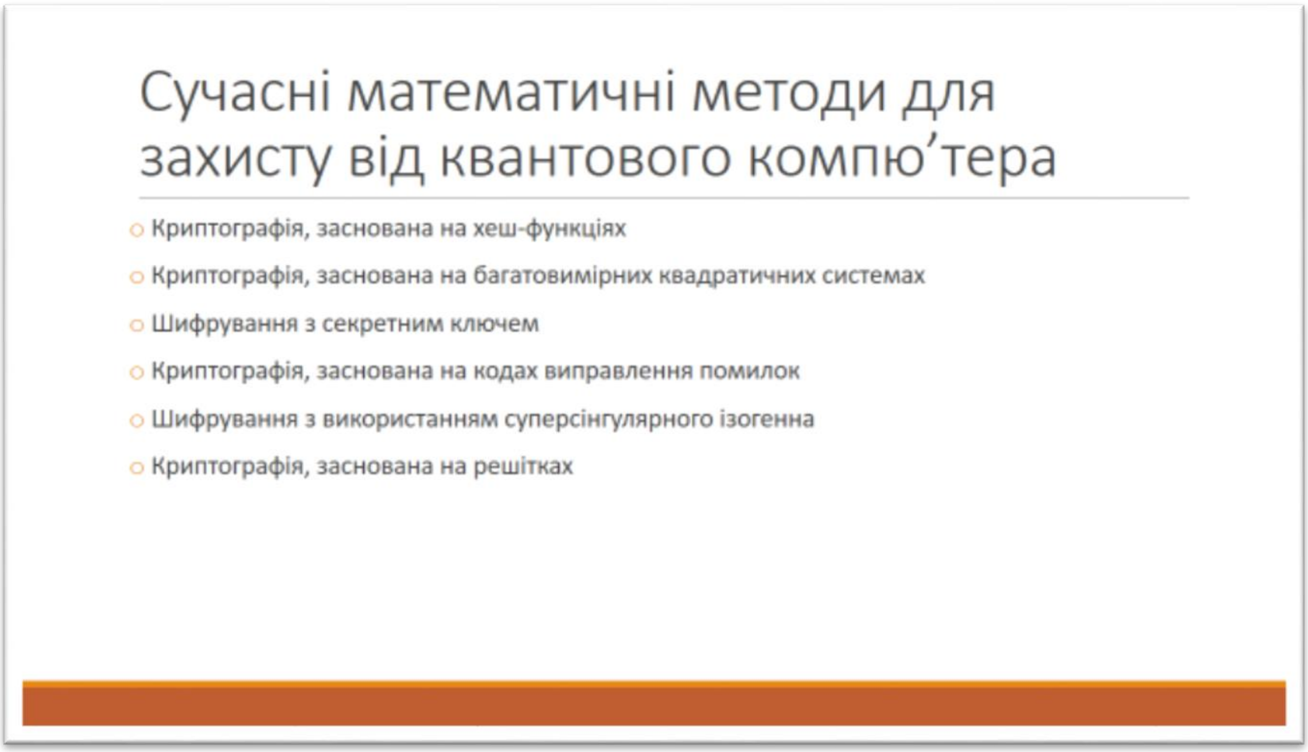

Рисунок Б.2 – Слайд 7

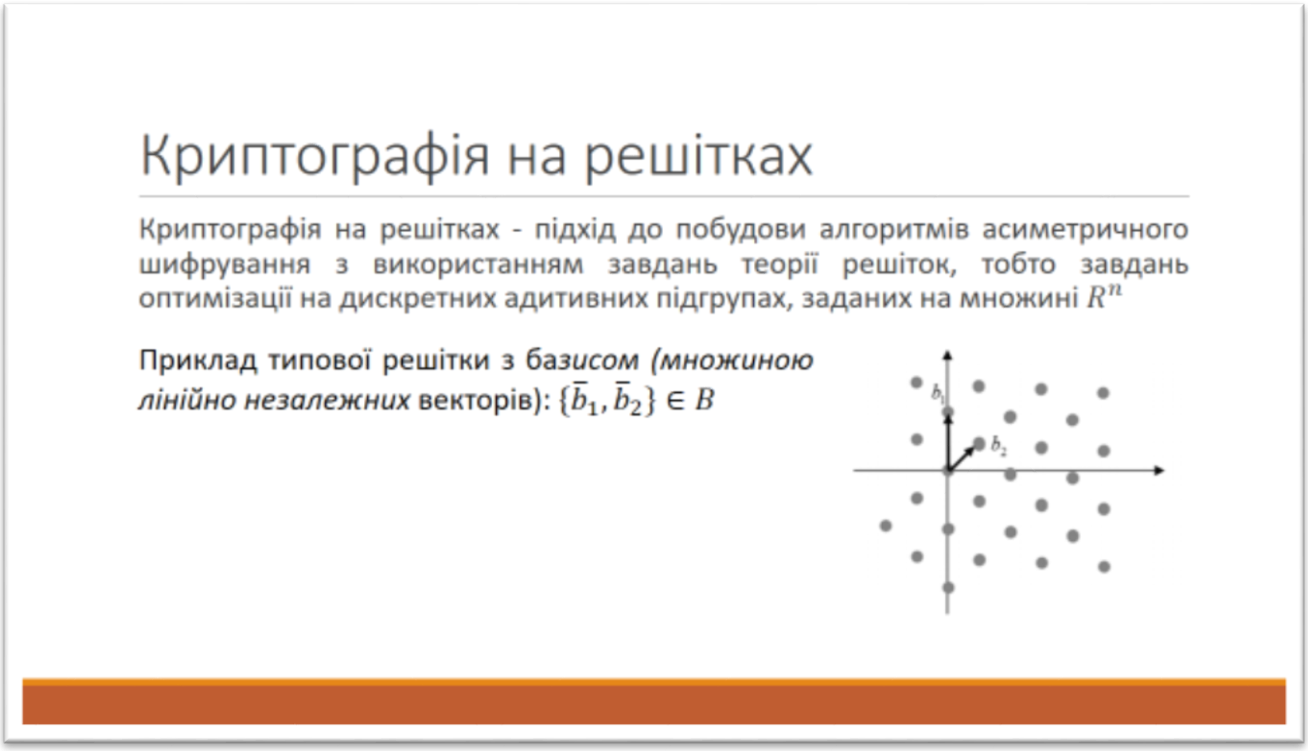

Рисунок Б.2 – Слайд 8

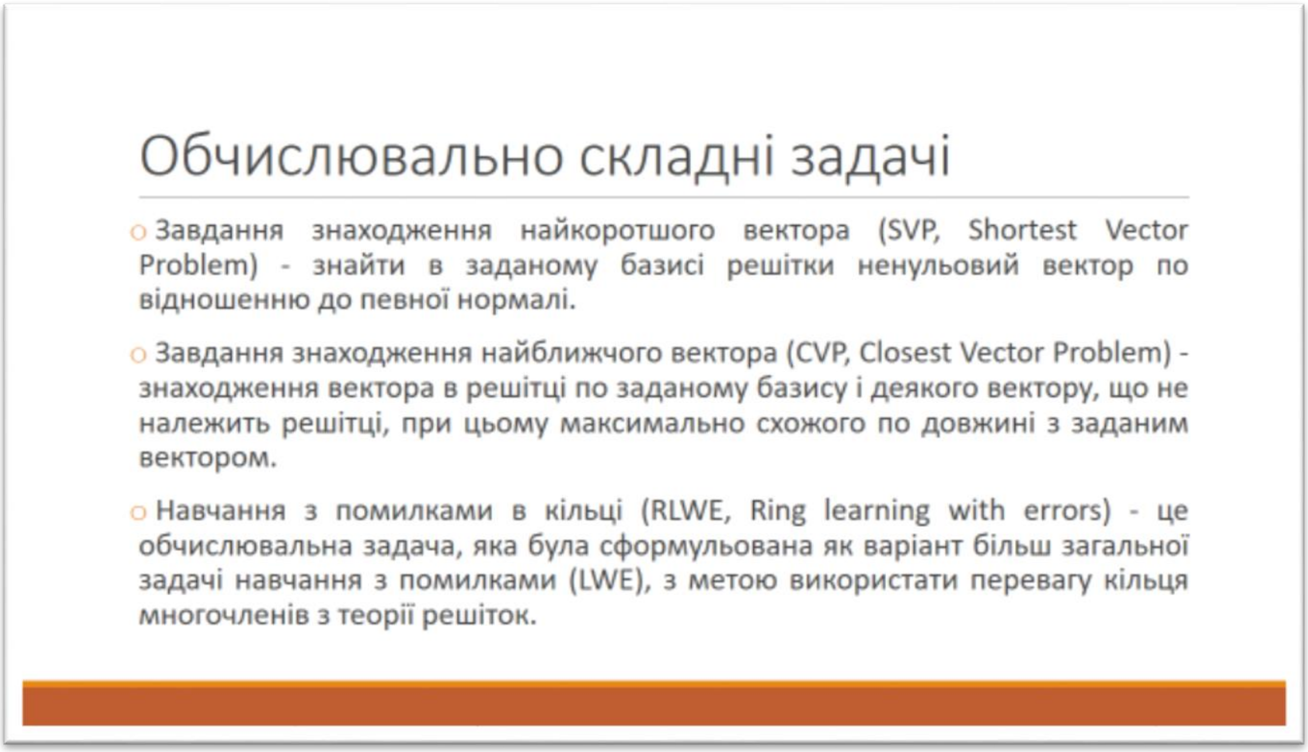

Рисунок Б.2 – Слайд 9

Рисунок Б.2 – Слайд 10

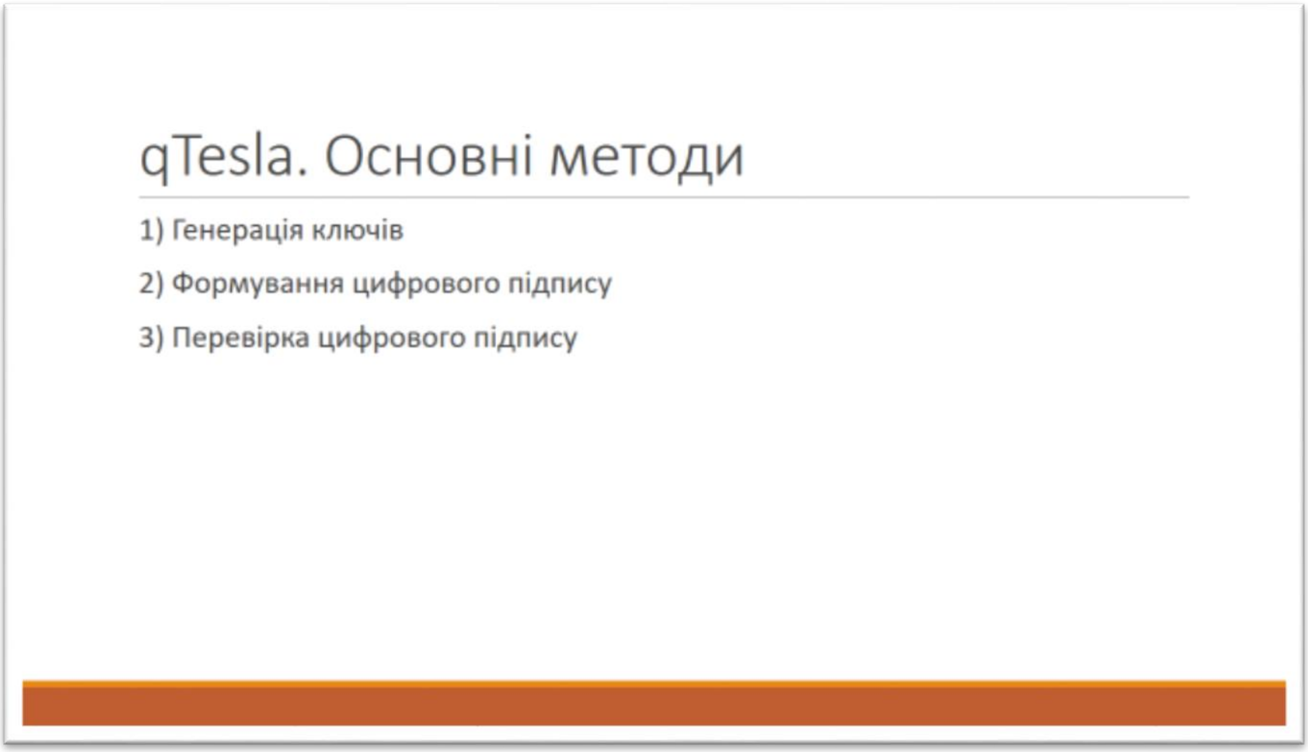

Рисунок Б.2 – Слайд 11

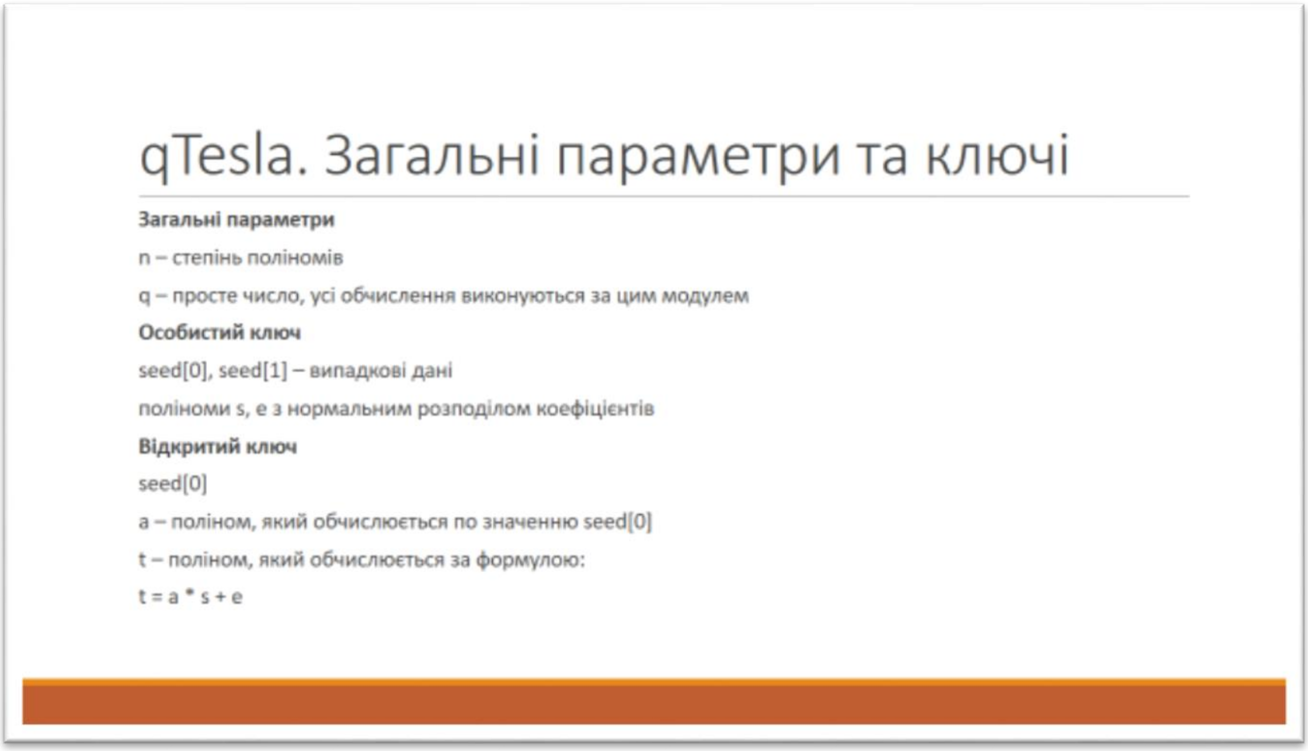

Рисунок Б.2 – Слайд 12

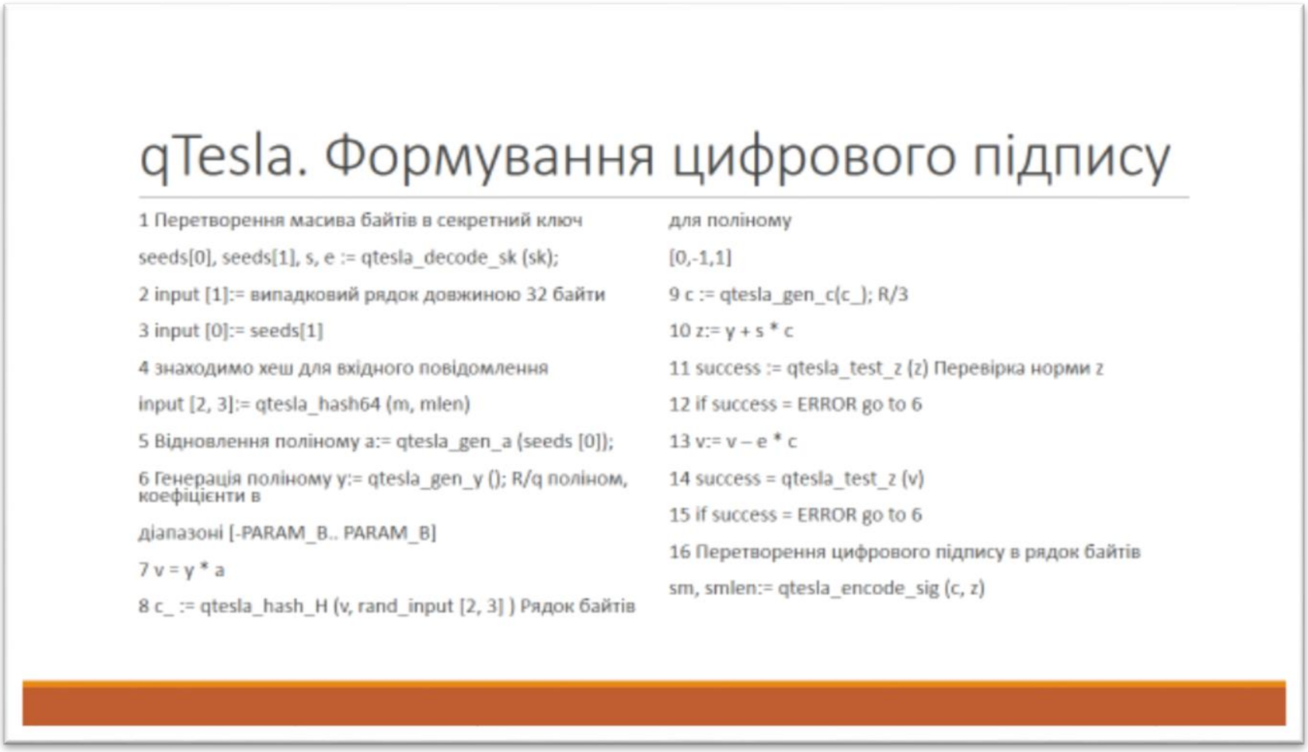

Рисунок Б.2 – Слайд 13

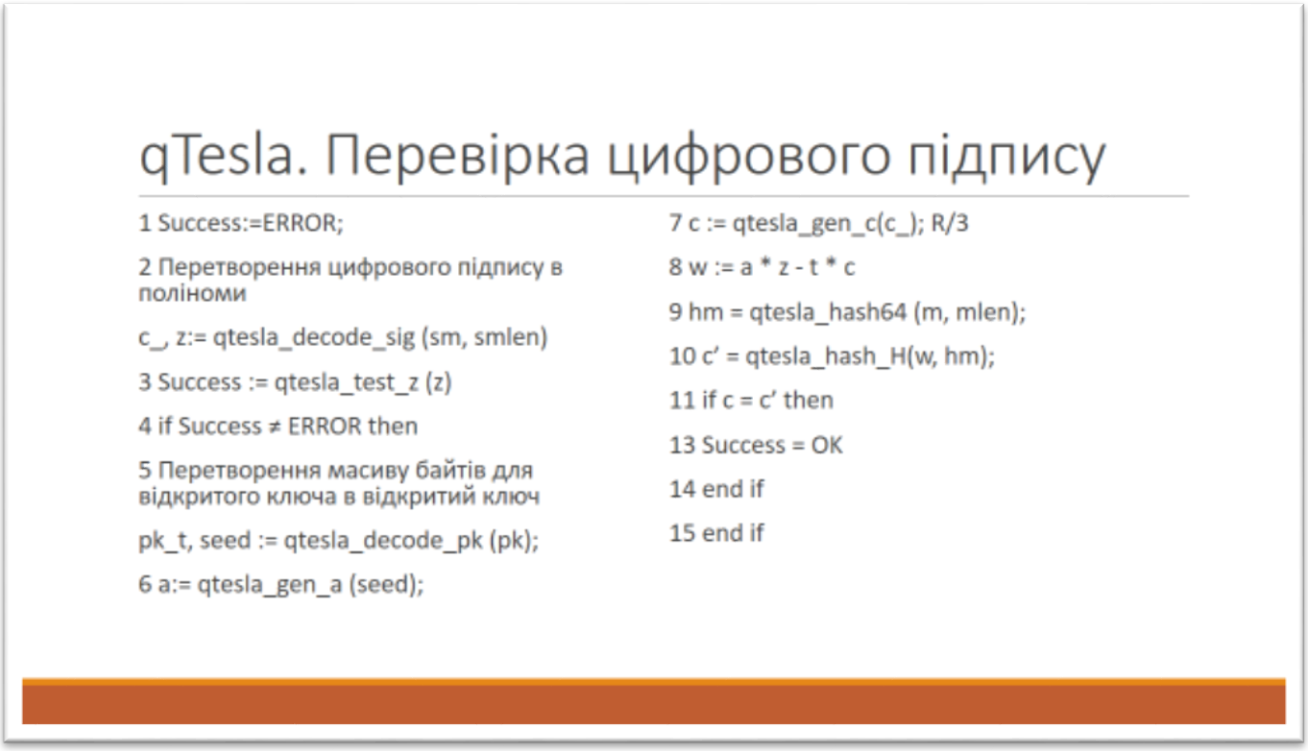

Рисунок Б.2 – Слайд 14

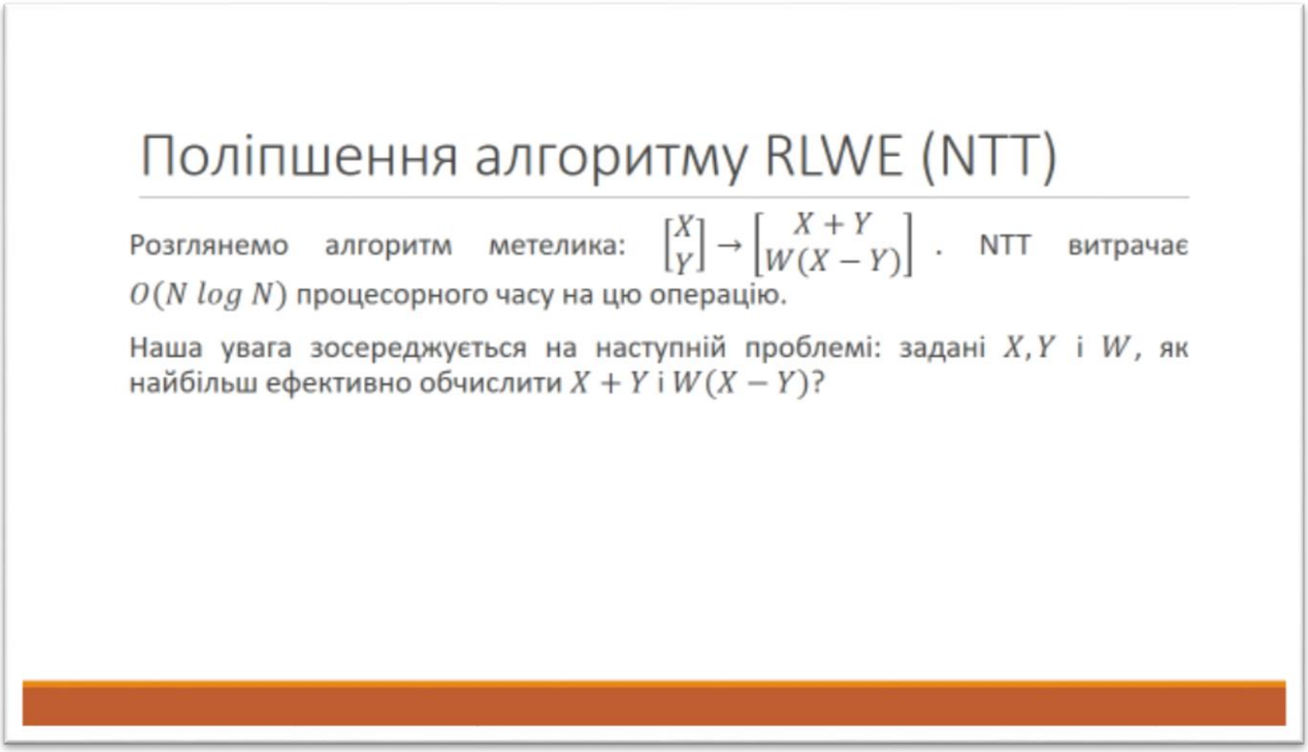

Рисунок Б.2 – Слайд 15

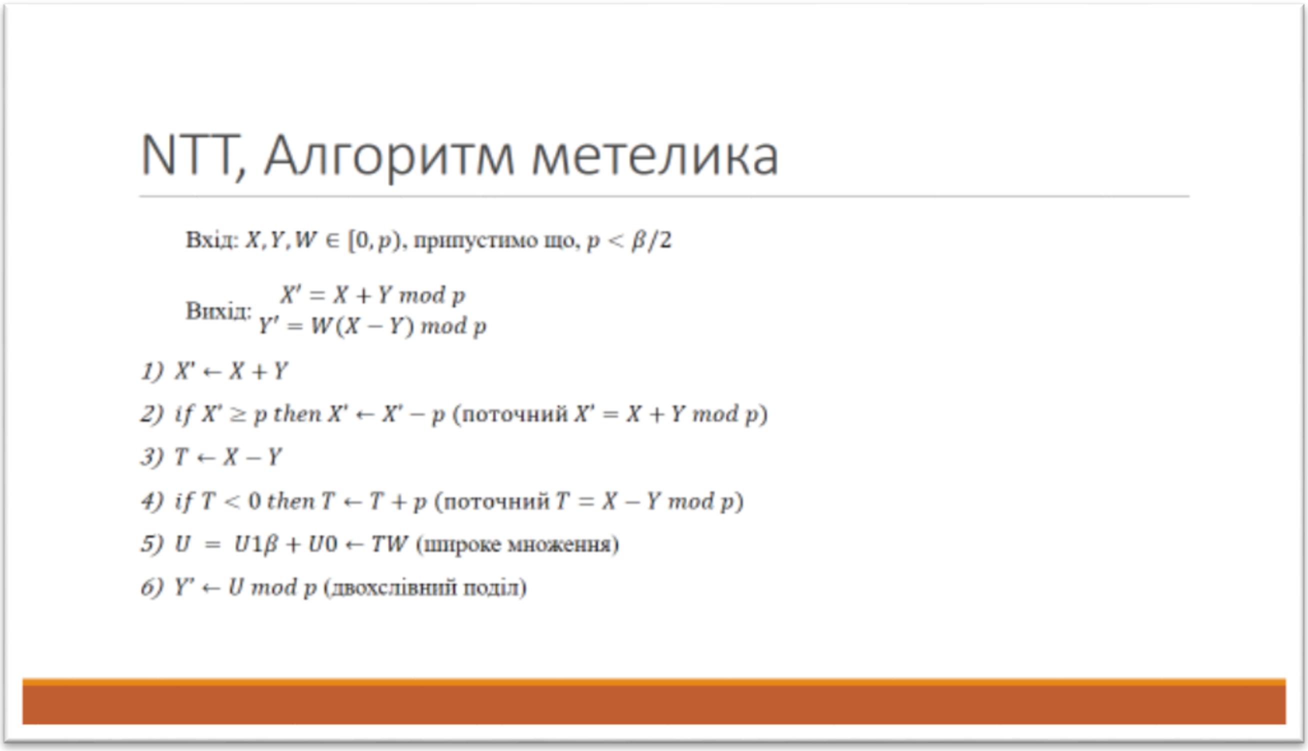

Рисунок Б.2 – Слайд 16

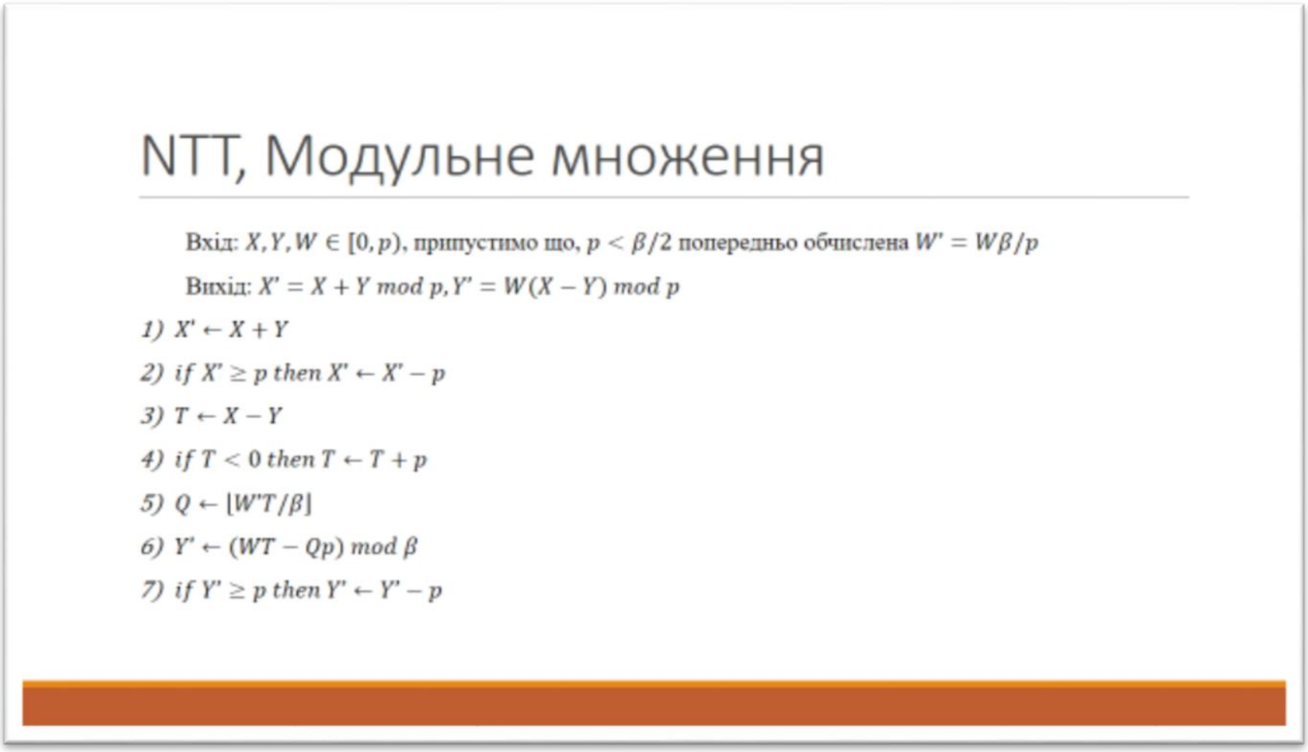

Рисунок Б.2 – Слайд 17

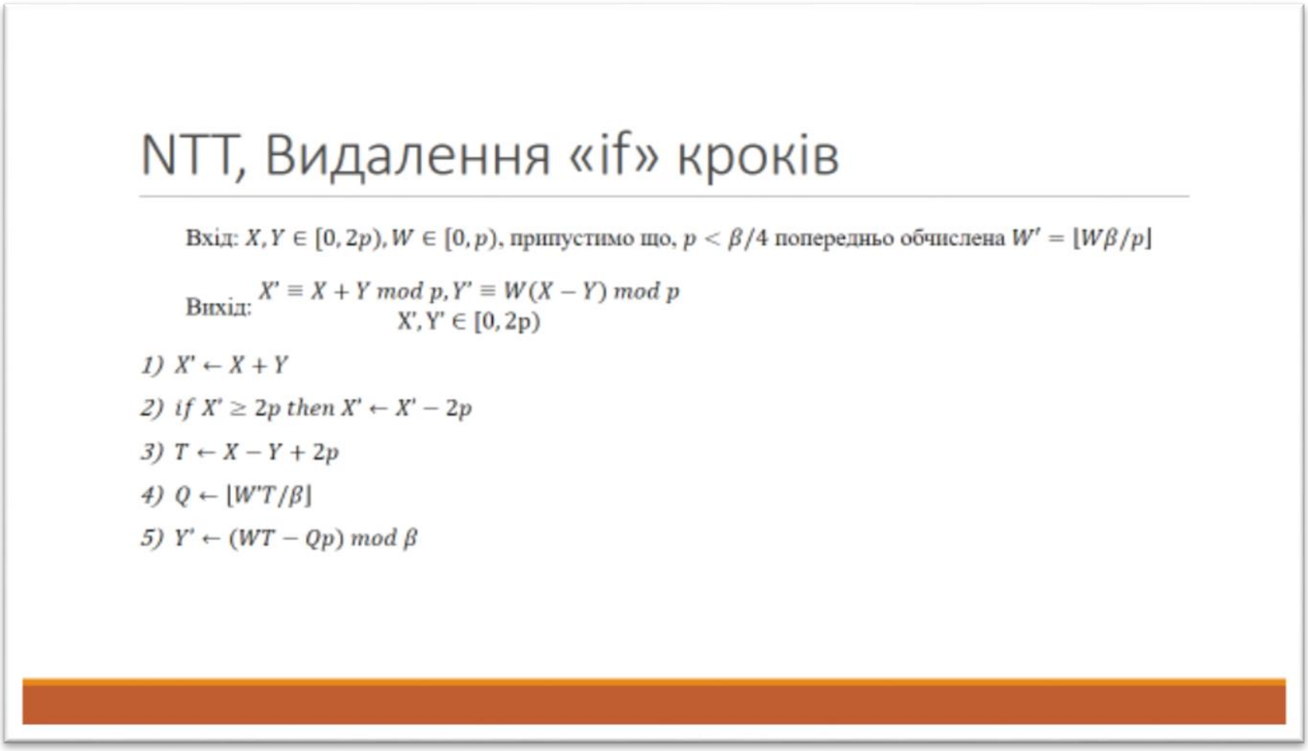

Рисунок Б.2 – Слайд 18

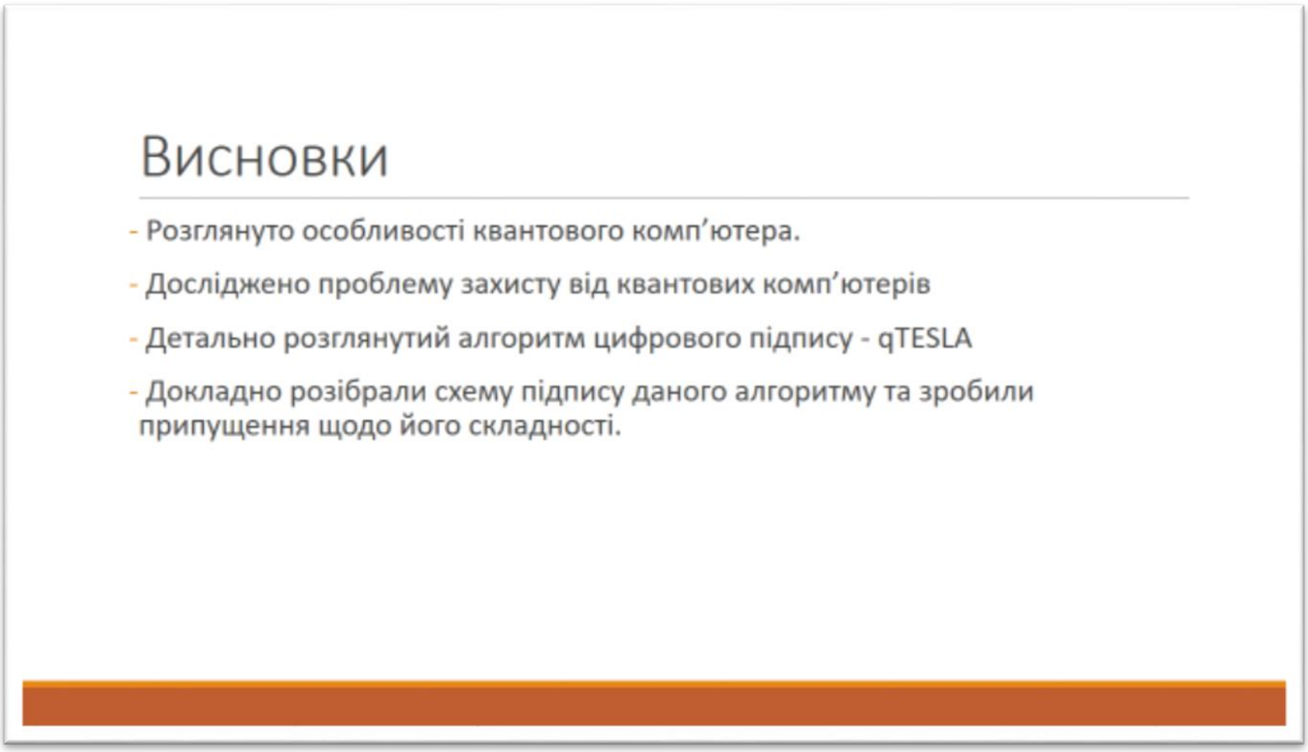

Рисунок Б.2 – Слайд 19