ДОДАТОК А

(довідковий)

Вихідний код програми

```
#include <stdlib.h>
#include <stdint.h>
#include "crypto_stream_salsa20.h"
#include "rng.h"
#include "shred.h"

#define RAND_LEN_BYTES (4096)

static int init = 0;
static unsigned char key[crypto_stream_salsa20_KEYBYTES];
static unsigned char nonce[crypto_stream_salsa20_NONCEBYTES] = {0};
static unsigned char randpool[RAND_LEN_BYTES];
static uint16_t randpos = RAND_LEN_BYTES;


void
fastrandombytes(unsigned char *r, unsigned long long rlen)
{
  unsigned long long n=0;
  uint8_t i;
  if(!init)
  {
    randombytes(key, crypto_stream_salsa20_KEYBYTES);
    init = 1;
  }
  crypto_stream_salsa20(r,rlen,nonce,key);

  // Increase 64-bit counter (nonce)
  for(i=0;i<8;i++)
    n ^= ((unsigned long long)nonce[i]) << 8*i;
  n++;
  for(i=0;i<8;i++)
```

```
    nonce[i] = (n >> 8*i) & 0xff;
}


void rng_cleanup()
{
  if(init)
  {
    init = 0;
    shred(key, crypto_stream_salsa20_KEYBYTES);
    shred(nonce, crypto_stream_salsa20_KEYBYTES);
    shred(randpool, crypto_stream_salsa20_KEYBYTES);
  }
}


void
rng_init()
{
  fastrandombytes(randpool, RAND_LEN_BYTES);
  randpos = 0;
}


void
rng_uint16(uint16_t *r)
{
  if(randpos >= (RAND_LEN_BYTES - sizeof(uint16_t)))
  {
    fastrandombytes(randpool, RAND_LEN_BYTES);
    randpos = 0;
  }
  *r = (uint16_t)(randpool[randpos++] & 0xff) << 8;
  *r |= (uint16_t)(randpool[randpos++] & 0xff);

  return;
}
```

```
void
rng_uint64(uint64_t *r)
{
  if(randpos >= RAND_LEN_BYTES - sizeof(uint64_t))
  {
    fastrandombytes(randpool, RAND_LEN_BYTES);
    randpos = 0;
  }
  *r  = ((uint64_t)(randpool[randpos++] & 0xff)) << 070;
  *r |= ((uint64_t)(randpool[randpos++] & 0xff)) << 060;
  *r |= ((uint64_t)(randpool[randpos++] & 0xff)) << 050;
  *r |= ((uint64_t)(randpool[randpos++] & 0xff)) << 040;
  *r |= ((uint64_t)(randpool[randpos++] & 0xff)) << 030;
  *r |= ((uint64_t)(randpool[randpos++] & 0xff)) << 020;
  *r |= ((uint64_t)(randpool[randpos++] & 0xff)) << 010;
  *r |= ((uint64_t)(randpool[randpos++] & 0xff));

  return;
}
```

Фрагмент коду 1 – Випадкова генерація байтів

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "param.h"
#include "poly.h"
#include "../common/fastrandombytes.h"
#include "../common/crypto_hash_sha512.h"


/*
 * memory requirement: 6 ring elements
 */
void
```

```
keygen(
          uint16_t  *F,     /* output secret key f */
          uint16_t  *g,     /* optional output secret key g */
          uint16_t  *h,     /* output public key h */
          uint16_t  *buf,
    const PARAM_SET *param)
{
    int16_t     i;
    uint16_t    *f;
    uint16_t    *f_inv;
    uint16_t    *localbuf;

    f           = buf;
    f_inv       = f     + param->padN;
    /* three ring elements for karatsuba */
    localbuf    = f_inv + param->padN;

    do{
        /* generate f = pF+1 until f is invertible mod 2*/
        trinary_poly_gen(F, param->N, param->d);
        for (i=0;i<param->N;i++)
            f[i] = param->p*F[i];
        f[0]++;
    }while (ntru_ring_inv(f, param->N, localbuf, f_inv) == -1);


    /* compute f^-1 by lifting f_inv mod 2 to f_inv mod q*/
    ring_lift_inv_pow2(f_inv, f, param, localbuf);

    /* generate g*/
    trinary_poly_gen(g, param->N, param->d);

    for (i=0;i<param->N;i++)
    {
        f[i] = f[i] & 0x7FF;
        g[i] = g[i] & 0x7FF;
    }
```

```
    /* compute h = f^-1*g */
    ntru_ring_mult_coefficients(f_inv, g, param, localbuf, h);

    memset(buf, 0, sizeof(uint16_t)*param->padN*6);
    return;
}

/*
 * memory requirement: 5 ring elements
 * checking if h = g/f
 */
int check_keys(
    const uint16_t  *F,
    const uint16_t  *g,
    const uint16_t  *h,
          uint16_t  *buf,
    const PARAM_SET *param)
{
    int16_t  i;
    uint16_t *f, *grec, *localbuf;

    memset(buf, 0, sizeof(uint16_t)*param->padN*5);
    f           = buf;
    grec        = f     +   param->padN;
    localbuf    = grec  +   param->padN;

    for (i=0;i<param->N;i++)
        f[i] = F[i]*param->p;
    f[0]++;
    ntru_ring_mult_coefficients(f, h, param, localbuf, grec);

    for(i=0;i<param->N;i++)
    {
        if (grec[i]!=g[i])
        {
            printf("checking keys error for %dth coefficients: %d %d\n", i,
grec[i], g[i]);
            return -1;
```

```
        }
    }

    memset(buf, 0, sizeof(uint16_t)*param->padN*5);
    return 0;
}

/* check if message is a valid trinary poly for kem */
int
check_m (
    const uint16_t *m,
    const uint16_t N)
{
    uint16_t i;
    for(i=0;i<N;i++)
    {
        if(m[i]!=1 && m[i]!=65535 && m[i]!=0)
        {
            printf("checking message error for %dth coefficients: %d \n", i,
m[i]);
            return -1;
        }
    }
    return 0;
}


/*
 * memory requirement: 5 ring elements
 */

int encrypt_kem(
    const uint16_t  *m,     /* input binary message */
    const uint16_t  *h,     /* input public key */
          uint16_t  *c,     /* output ciphertext */
          uint16_t  *buf,
    const PARAM_SET *param)
{
```

```
    if (check_m(m, param->N) == -1 )
    {
        printf("error message\n");
        return -1;
    }
    uint16_t    i;
    uint16_t    *r, *t, *localbuf;


    r           = buf;
    t           = r + param->padN;
    /* three ring elements for karatsuba */
    localbuf    = t + param->padN;

    trinary_poly_gen(r, param->N, param->d);

    ntru_ring_mult_coefficients(r, h, param, localbuf, t);

    for (i=0;i<param->N;i++)
        c[i] = (t[i]*param->p + m[i]) & (param->q-1);

    memset(buf, 0, sizeof(uint16_t)*param->padN*5);
    return 0;
}

/*
 * lift the message back to a trinary polynomial
 * by mod q then mod p
 */
static void
lift_msg(
    uint16_t    *m,
    PARAM_SET   *param)
{
    uint16_t    i;
    int         tmp;
    for (i=0;i<param->N;i++)
    {
```

```
        tmp = m[i] % param->q;
        if (tmp>param->q/2)
            tmp -= param->q;

        tmp = tmp % param->p;
        if (tmp == 2)
            tmp = -1;
        if (tmp == -2)
            tmp = 1;
        m[i] = tmp;
    }


    return;
}


/*
 * memory requirement: 4 ring elements
 */
int decrypt_kem(
    uint16_t    *m,     /* output trinary message */
    uint16_t    *F,     /* input public key */
    uint16_t    *c,     /* input ciphertext */
    uint16_t    *buf,
    PARAM_SET   *param)
{

    uint16_t    *f, *localbuf, i;

    f           = buf;
    localbuf    = f + param->padN;

    for (i=0;i<param->N;i++)
        f[i] = F[i]*param->p;
    f[0]++;

    /* compute e = c * f */
```

```
    ntru_ring_mult_coefficients(c, f, param, localbuf, m);

    /* recover m = e mod p */
    lift_msg(m, param);

    memset(buf, 0, sizeof(uint16_t)*param->padN*4);
    return 0;
}



/*
 * check if a message length is valid for ntruencrypt-cca
 * then convert the message into a binary polynomial and
 * pad the message with a random binary string p
 */
int
pad_msg(
          uint16_t  *m,     /* output message */
    const char      *msg,   /* input message string */
    const size_t    msg_len,/* input length of the message */
    const PARAM_SET *param)
{
    if (msg_len > param->max_msg_len)
    {
        printf("error: message too long");
        return -1;
    }
    uint16_t    *pad;
    uint16_t    i,j;
    char        tmp;
    memset(m, 0, sizeof(uint16_t)*param->N);

    /* generate the pad of a degree 167 trinary polynomial*/
    pad = m + param->N - 167;
    trinary_poly_gen(pad, 167, 56);

    /* convert the message length into coefficients */
```

```
    pad -= 8;
    tmp = msg_len;
    for(j=0;j<8;j++)
    {
        pad[j] = tmp & 1;
        tmp >>= 1;
    }
    /* form the message binary polynomial */
    for (i=0;i<msg_len;i++)
    {
        tmp = msg[i];
        for(j=0;j<8;j++)
        {
            m[i*8+j] = tmp & 1;
            tmp >>= 1;
        }
    }
    return 0;
}


/*
 * converting a binary polynomial into a char string
 * return the length of the message string
 */
int
recover_msg(
          char      *msg,   /* output message string */
    const uint16_t  *m,     /* input binary message */
    const PARAM_SET *param)
{
    char    tmp;
    int     msg_len;
    uint16_t i,j;
    msg_len = 0;

    for (j=0;j<8;j++)
    {
```

```
        msg_len += (m[param->N - 167 - 8 + j]<<j);
    }

    if (msg_len > param->max_msg_len)
    {
        printf("error: message too long");
        return -1;
    }

    for (i=0;i<msg_len;i++)
    {
        tmp = 0;
        for (j=0;j<8;j++)
        {
            tmp += (m[i*8+j]<<j);
        }
        msg[i] = tmp;
    }
    return msg_len;
}



/*
 * generate a balanced trinary r from msg and h
 * memory requirement: 2 * LENGTH_OF_HASH
 */
int
generate_r(
          uint16_t  *r,     /* output r */
    const uint16_t  *msg,   /* input binary message */
    const uint16_t  *h,     /* input public key */
          uint16_t  *buf,
    const PARAM_SET *param)
{
    uint16_t i;
    for (i=0;i<param->N;i++)
    {
```

```
        if (msg[i]!=0 && msg[i]!=1 && (msg[i]%param->q)!=param->q-1)
        {
            printf("invalid messages\n");
            return -1;
        }
    }
    unsigned char *seed = (unsigned char*) buf;
    memset(seed, 0, sizeof(unsigned char)* LENGTH_OF_HASH*2);

    /* hash message/public key into a string 'seed'*/
    crypto_hash_sha512(seed, (unsigned char*)msg, param->N*2);
    crypto_hash_sha512(seed+LENGTH_OF_HASH, (unsigned char*)h, param->N*2);


    /* use the seed to generate r */
    trinary_poly_gen_w_seed(r, param->N, param->d, seed, LENGTH_OF_HASH*2);

    memset(seed, 0, sizeof(unsigned char)* LENGTH_OF_HASH*2);

    return 0;
}



/*
 * input a message msg, output msg \trixor hash(rh)
 * memory requirements: LENGTH_OF_HASH + 1 ring element
 */
int
mask_m(
          uint16_t  *msg,   /* in/output binary message */
    const uint16_t  *rh,
          uint16_t  *buf,
    const PARAM_SET *param)
{
    unsigned char   *seed;
    uint16_t        *mask;
    uint16_t        i;
```

```
    memset(buf,          0,          sizeof(uint16_t)*param->padN          +
sizeof(char)*LENGTH_OF_HASH);
    seed  = (unsigned char*) buf;
    mask  = (uint16_t *) (seed + LENGTH_OF_HASH);

    crypto_hash_sha512(seed,      (unsigned      char*)      rh,      param-
>N*sizeof(uint16_t)/sizeof(unsigned char));

    rand_tri_poly_from_seed(mask, param->N, seed, LENGTH_OF_HASH);

    for (i=0;i<param->N;i++)
    {
        if (mask[i] == 65535)
            msg[i] --;
        else if (mask[i] == 1)
            msg[i] ++;

        if (msg[i] == 65534)
            msg[i] =  1;
        if (msg[i] == 2)
            msg[i] =  -1;
    }
    memset(buf,          0,          sizeof(uint16_t)*param->padN          +
sizeof(char)*LENGTH_OF_HASH);
    return 0;
}



/*
 * input a message msg, output msg \trixor hash(rh)
 * memory requirements: LENGTH_OF_HASH + 1 ring element
 */
static int
unmask_m(
          uint16_t  *msg,   /* in/output binary message */
    const uint16_t  *rh,
```

```
          uint16_t  *buf,
    const PARAM_SET *param)
{
    unsigned char   *seed;
    uint16_t        *mask;
    uint16_t        i;

    memset(buf,          0,          sizeof(uint16_t)*param->padN          +
sizeof(char)*LENGTH_OF_HASH);
    seed  = (unsigned char*) buf;
    mask  = (uint16_t *) (seed + LENGTH_OF_HASH);


    crypto_hash_sha512(seed,      (unsigned      char*)      rh,      param-
>N*sizeof(uint16_t)/sizeof(unsigned char));

    rand_tri_poly_from_seed(mask, param->N, seed, LENGTH_OF_HASH);

    for (i=0;i<param->N;i++)
    {
        if (mask[i] == 65535)
            msg[i] ++;
        else if (mask[i] == 1)
            msg[i] --;

        if (msg[i] == 65534)
            msg[i] =  1;
        if (msg[i] == 2)
            msg[i] =  -1;
    }
    memset(buf,          0,          sizeof(uint16_t)*param->padN          +
sizeof(char)*LENGTH_OF_HASH);
    return 0;
}
/*
 * CCA-2 secure encryption algorithm using NAEP
 * memory requirement: 6 ring elements
 */
```

```
void
encrypt_cca(
          uint16_t  *c,     /* output ciphertext */
    const char      *msg,   /* input message: a string of chars */
    const size_t    msg_len,/* input the length of the message */
    const uint16_t  *h,     /* input public key */
          uint16_t  *buf,
    const PARAM_SET *param)
{
    uint16_t    i;
    uint16_t    *r, *t, *m, *localbuf;

    m           = buf;
    r           = buf   + param->padN;
    t           = r     + param->padN;
    localbuf    = t     + param->padN;

    /* pad the message */
    if (pad_msg( m, msg, msg_len, param) == -1)
        return;

    /* generate r from the message */
    if (generate_r(r, m, h, localbuf,param) == -1)
        return;

    /* compute r*h */
    ntru_ring_mult_coefficients(r, h, param, localbuf, t);
    for (i=0;i<param->N;i++)
    {
        t[i] *= param->p;
        t[i] &= (param->q-1);
    }

    /* mask the message with hash(r*h) */
    mask_m (m, t, localbuf, param);


    for (i=0;i<param->N;i++)
```

```
        c[i] = (t[i] + m[i]) & (param->q-1);

    memset(buf,0, sizeof(uint16_t)*param->padN*6);

    return ;
}


/*
 * CCA-2 secure encryption algorithm using NAEP
 * return the length of the message
 * memory requirement: 7 ring elements
 */
int decrypt_cca(
          char      *msg,  /* output message: a string of chars */
    const uint16_t  *F,    /* input public key */
    const uint16_t  *h,    /* input public key */
    const uint16_t  *c,    /* input ciphertext */
          uint16_t  *buf,
    const PARAM_SET *param)
{
    uint16_t i, msg_len;
    uint16_t *f, *m, *t, *r, *t_rec, *localbuf;

    memset(buf, 0, sizeof(int16_t)*param->padN*8);

    f           = buf;
    m           = f     + param->padN;
    t           = m     + param->padN;
    r           = t     + param->padN;
    t_rec       = r     + param->padN;
    localbuf    = t_rec + param->padN;


    for (i=0;i<param->N;i++)
        f[i] = F[i]*param->p;
    f[0]++;
```

```
/* compute e = c * f */
ntru_ring_mult_coefficients(c, f, param, localbuf, m);

/* recover m = e mod p */
lift_msg(m, param);

/* recover r*h */
for (i=0;i<param->padN;i++)
    t[i] = (c[i] - m[i]) & (param->q-1);

/* unmask m with hash(r*h) */
unmask_m (m, t, localbuf, param);

/* recover r from hash(m) */
if (generate_r(r, m, h, localbuf,param) == -1)
{
    memset(buf,0, sizeof(uint16_t)*param->padN*7);
    return -1;
}

/* check if recovered r is correct */
ntru_ring_mult_coefficients(r, h, param, localbuf, t_rec);

for(i=0;i<param->N;i++)
{
    if (((param->p*t_rec[i] - t[i]) & (param->q-1)) !=0)
    {
        printf("error: \n");
        printf("r: \n");
        for (i=0;i<param->padN;i++)
            printf("%d, ", r[i]);
        printf("\n");
        printf("h: \n");
        for (i=0;i<param->padN;i++)
            printf("%d, ", h[i]);
        printf("\n");
        printf("t_rec: \n");
```

```
            for (i=0;i<param->padN;i++)
                printf("%d, ", t_rec[i]);
            printf("\n");
            printf("t: \n");
            for (i=0;i<param->padN;i++)
                printf("%d, ", t[i]);
            printf("\n");
            printf("c: \n");
            for (i=0;i<param->padN;i++)
                printf("%d, ", c[i]);
            printf("\n");

            memset(buf,0, sizeof(uint16_t)*param->padN*8);
            return -1;
        }
    }

    /* convert the message polynomial into char string */
    msg_len = recover_msg(msg, m, param);
    memset(buf,0, sizeof(uint16_t)*param->padN*8);
    return msg_len;
}
```

Фрагмент коду 2 – Алгоритм несиметричного шифрування NTRU-KEM

ДОДАТОК Б

(обов’язковий)

СЛАЙДИ ПРЕЗЕНТАЦІЇ

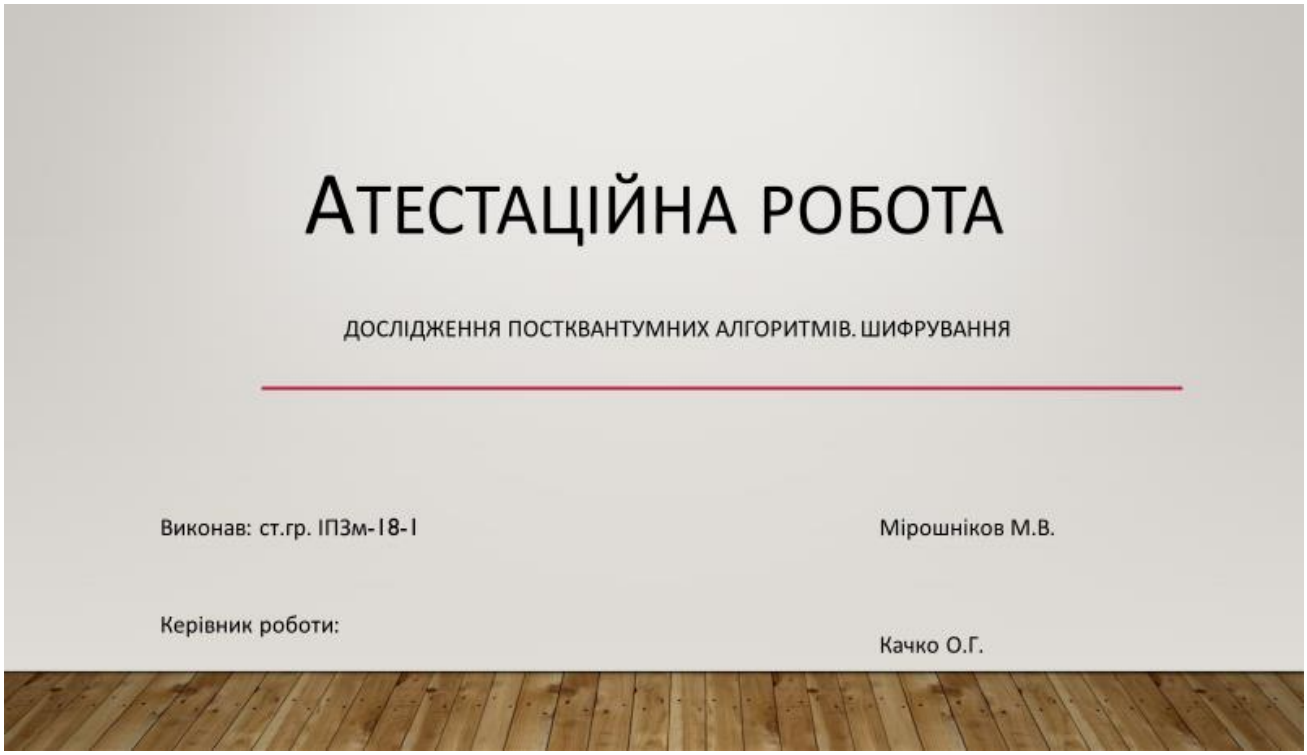

Рисунок Б.2 – Слайд 1

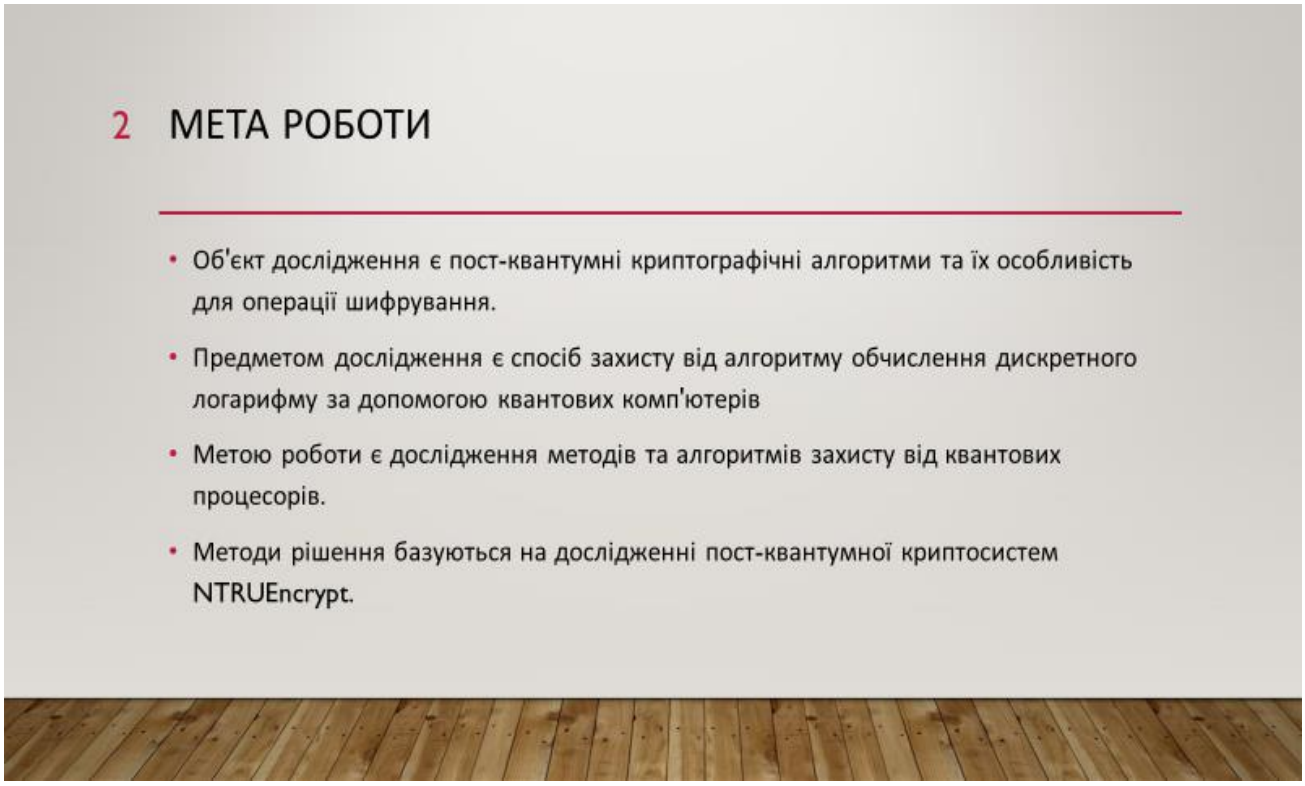

Рисунок Б.2 – Слайд 2

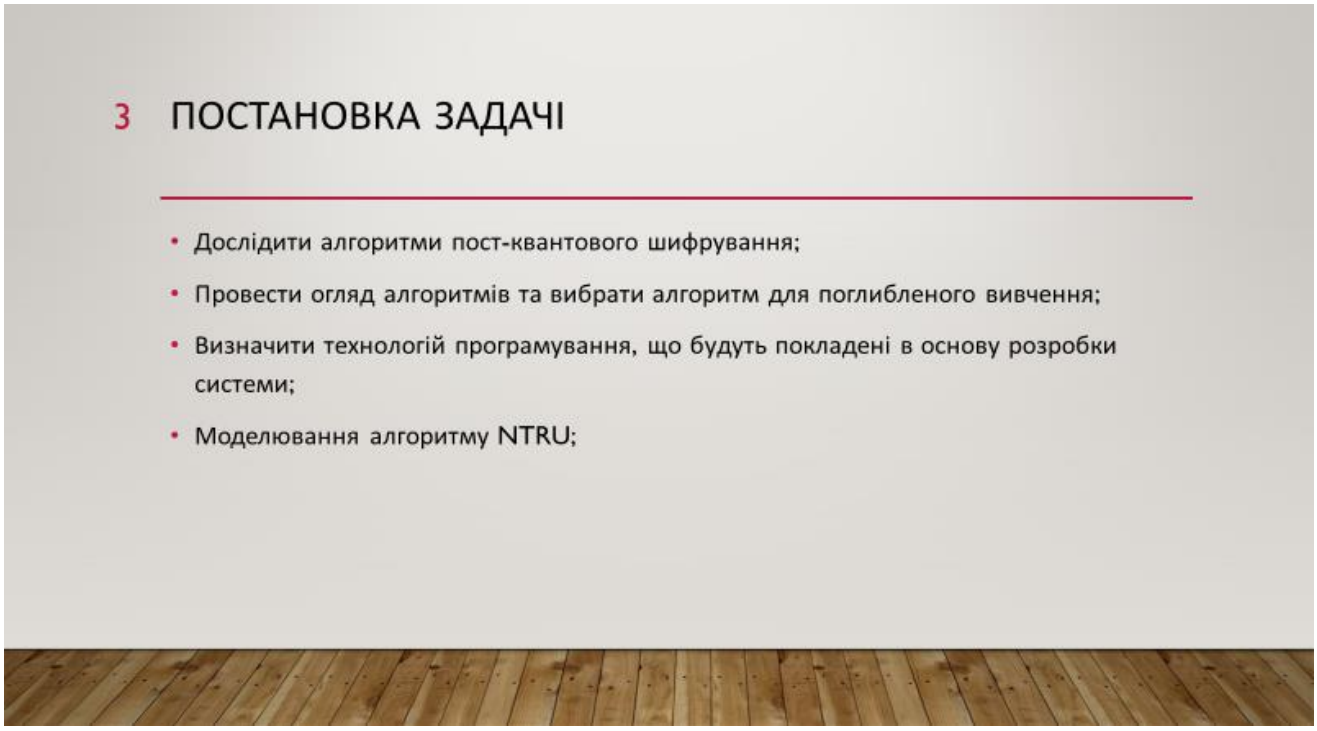

Рисунок Б.2 – Слайд 3

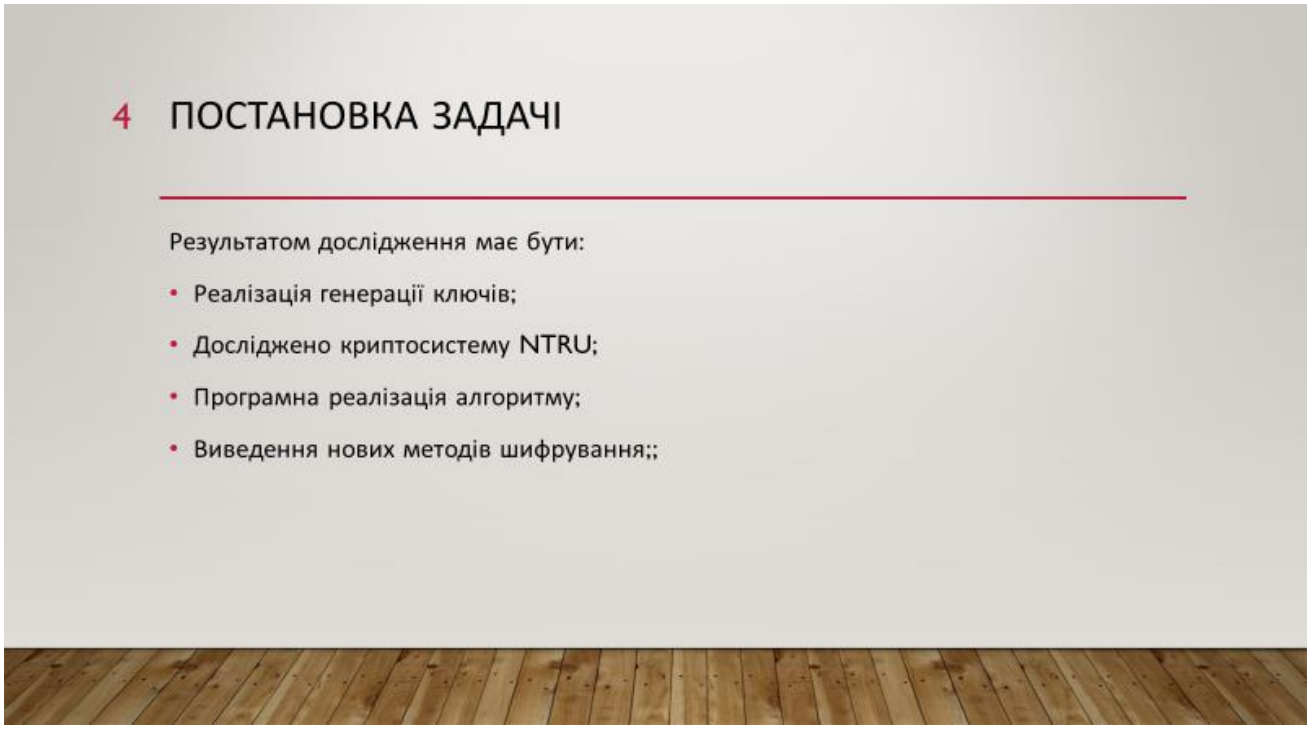

Рисунок Б.2 – Слайд 4

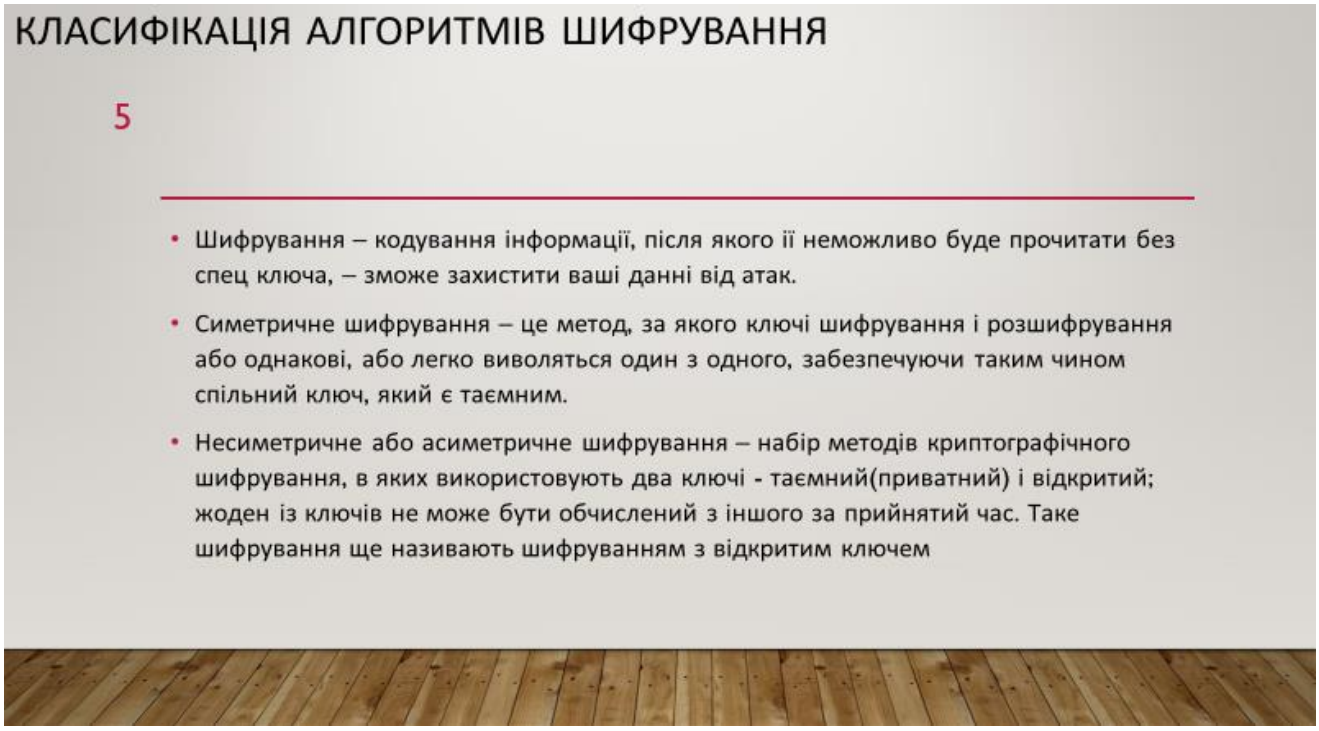

Рисунок Б.2 – Слайд 5

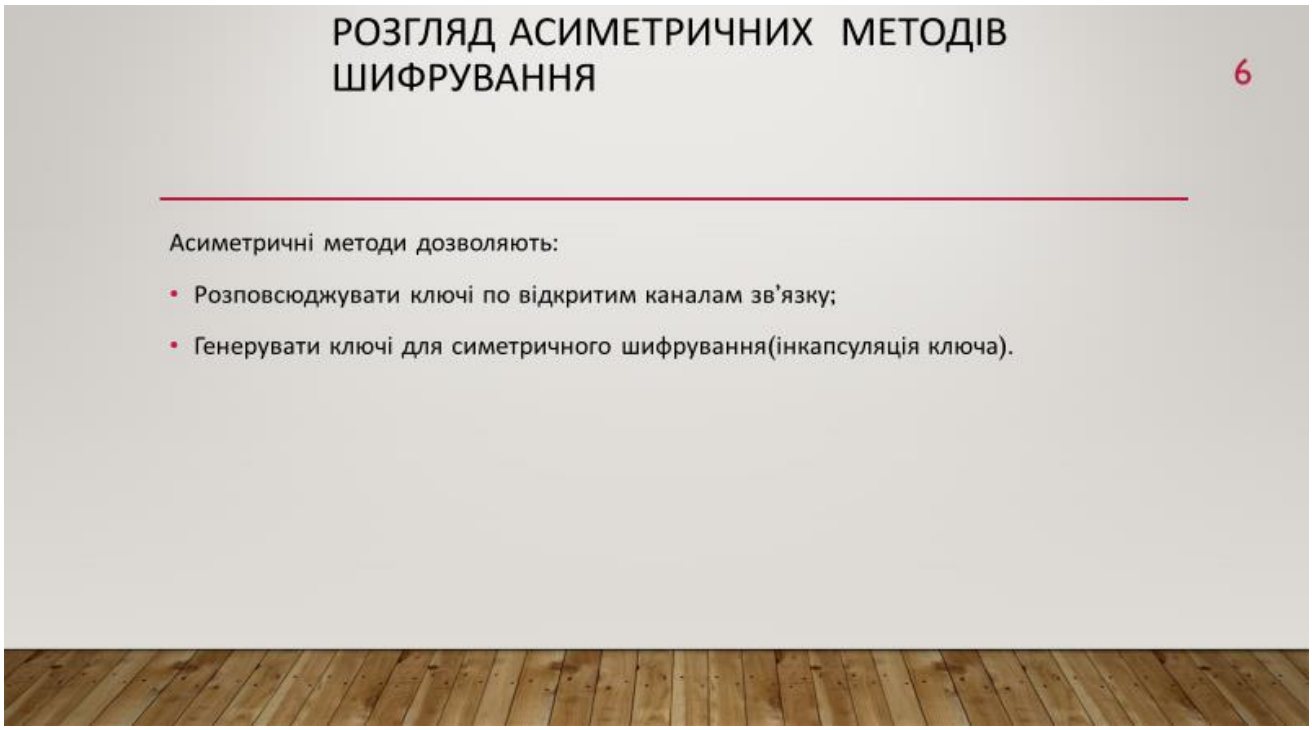

Рисунок Б.2 – Слайд 6

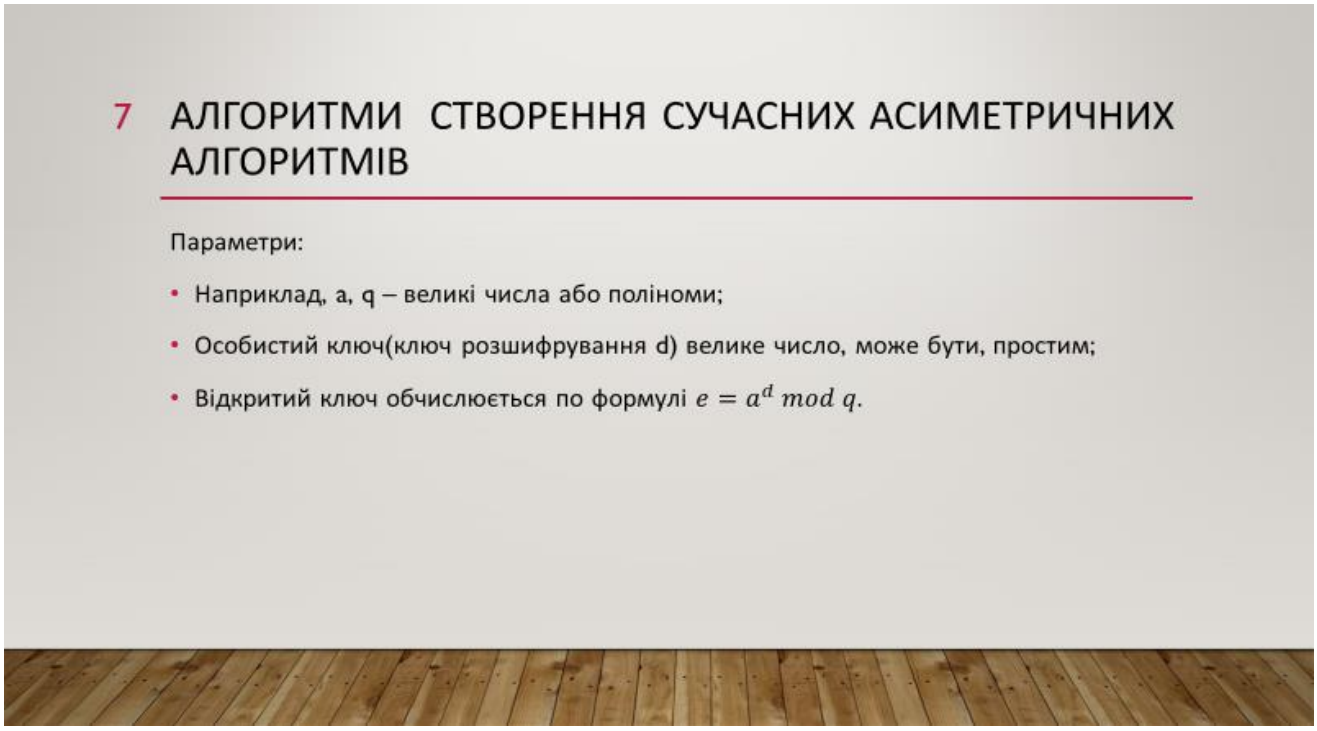

Рисунок Б.2 – Слайд 7

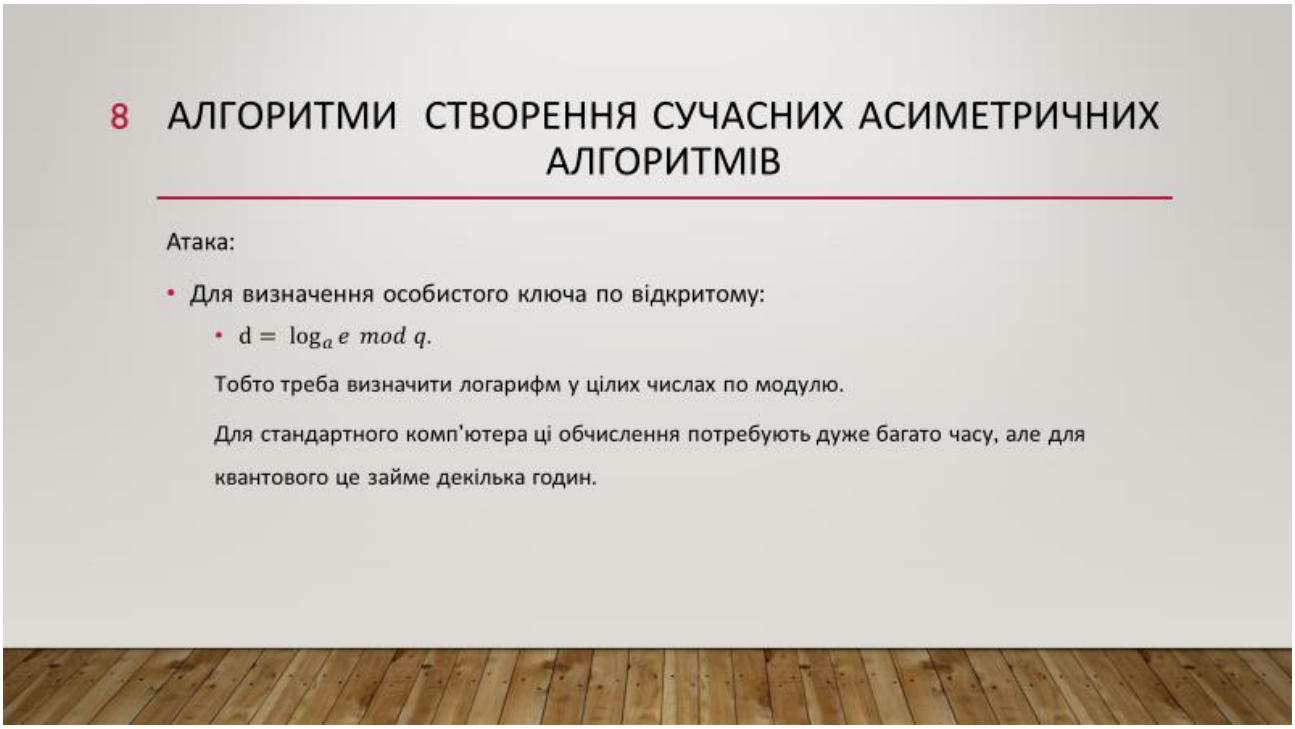

Рисунок Б.2 – Слайд 8

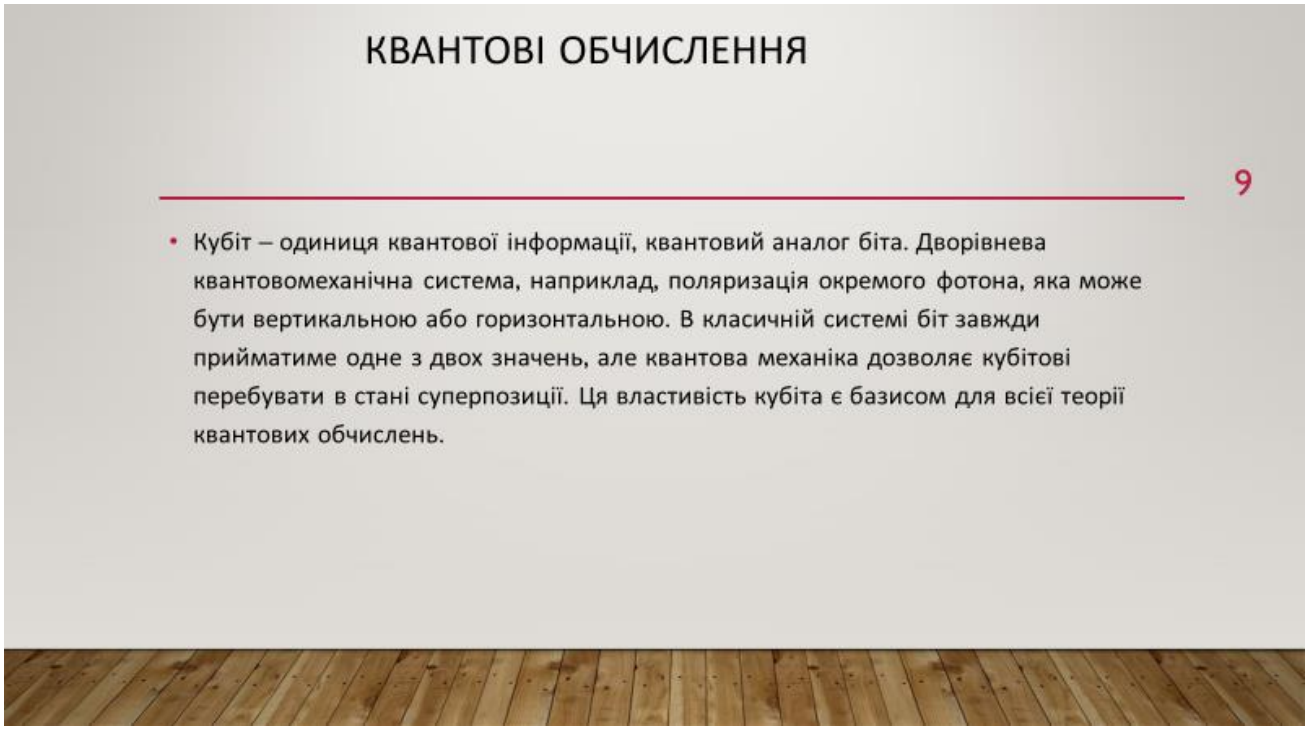

Рисунок Б.2 – Слайд 9

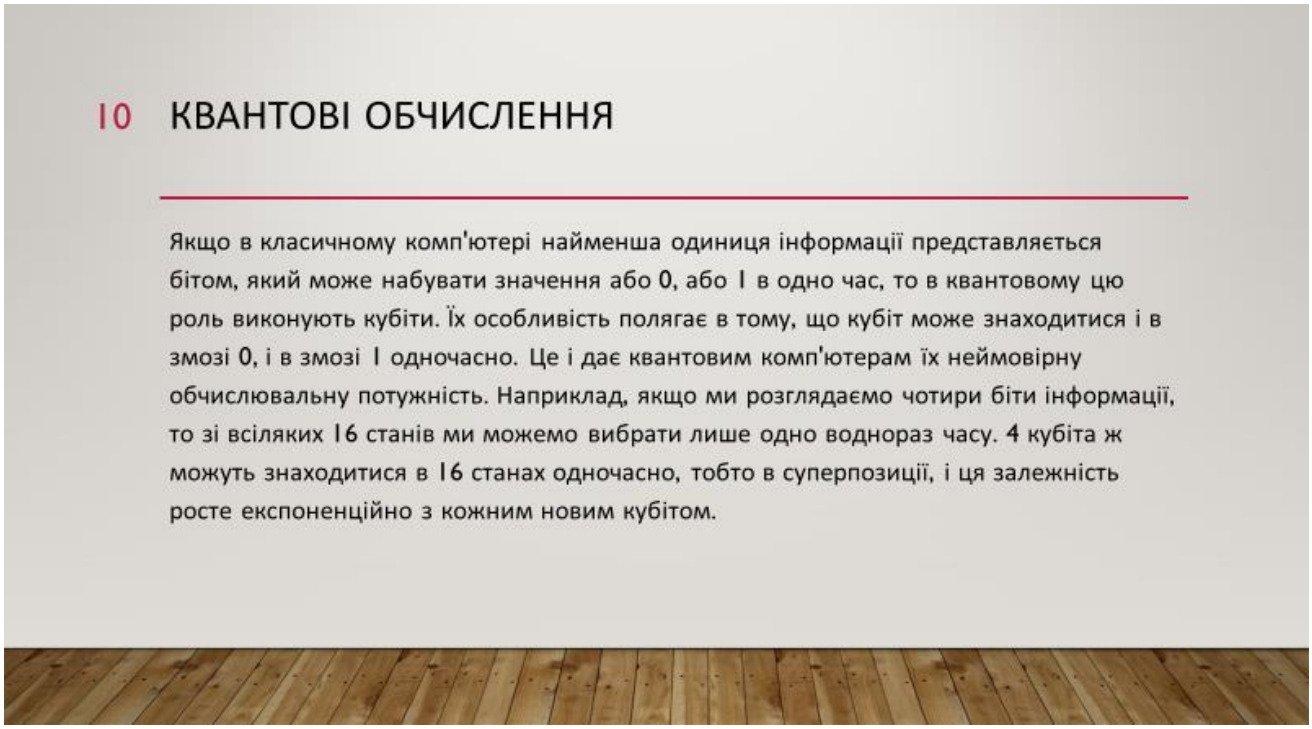

Рисунок Б.2 – Слайд 10

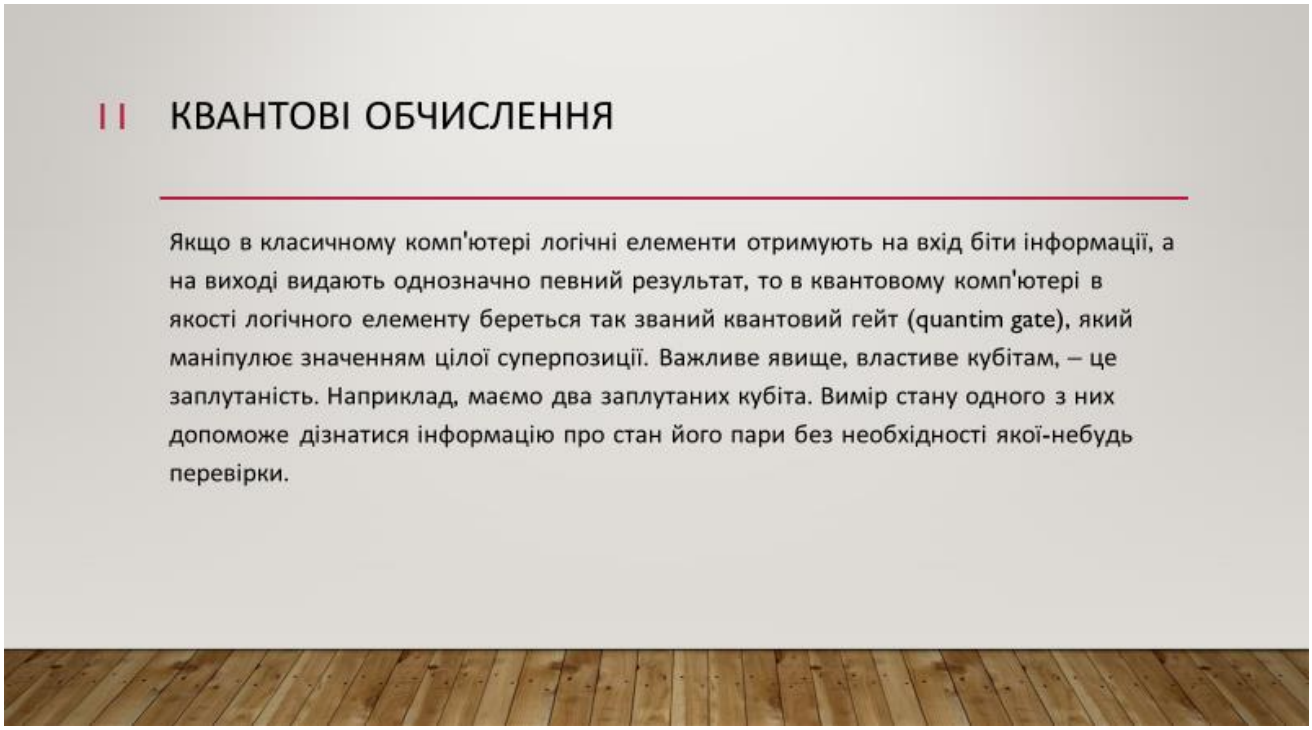

Рисунок Б.2 – Слайд 11

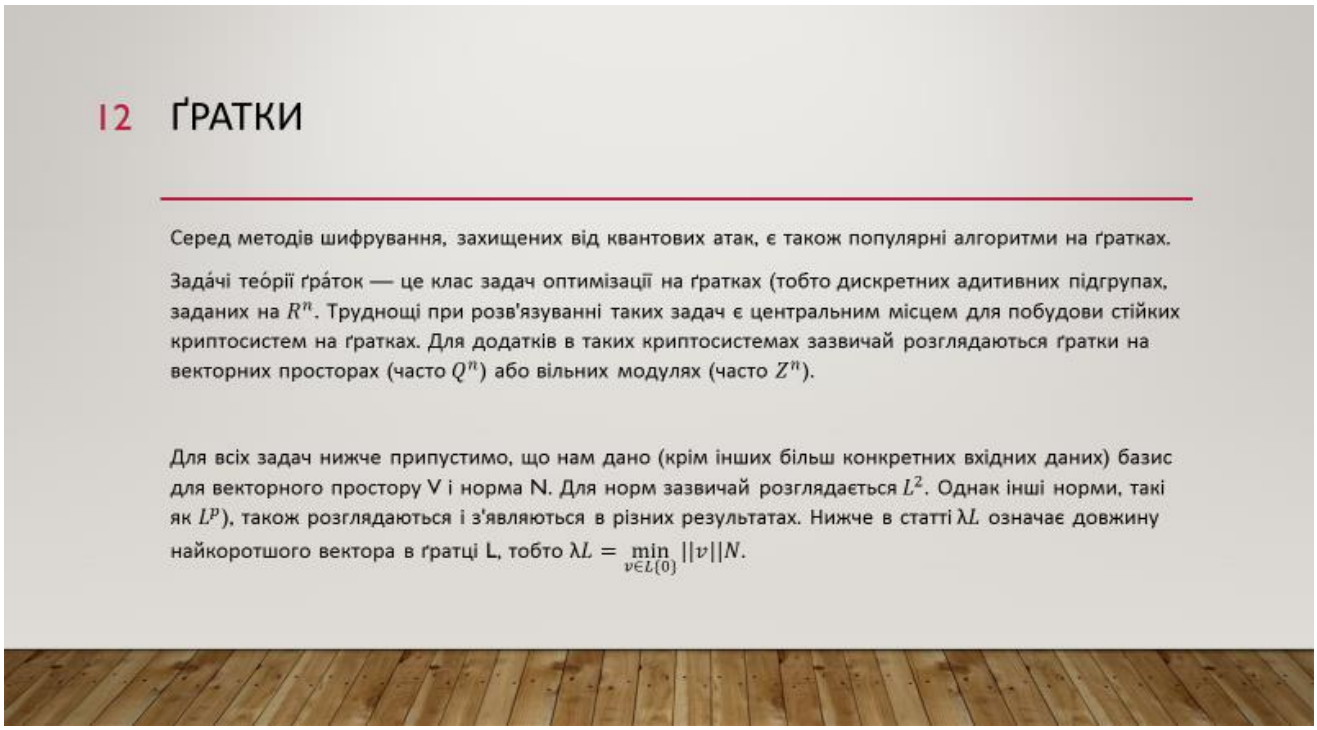

Рисунок Б.2 – Слайд 12

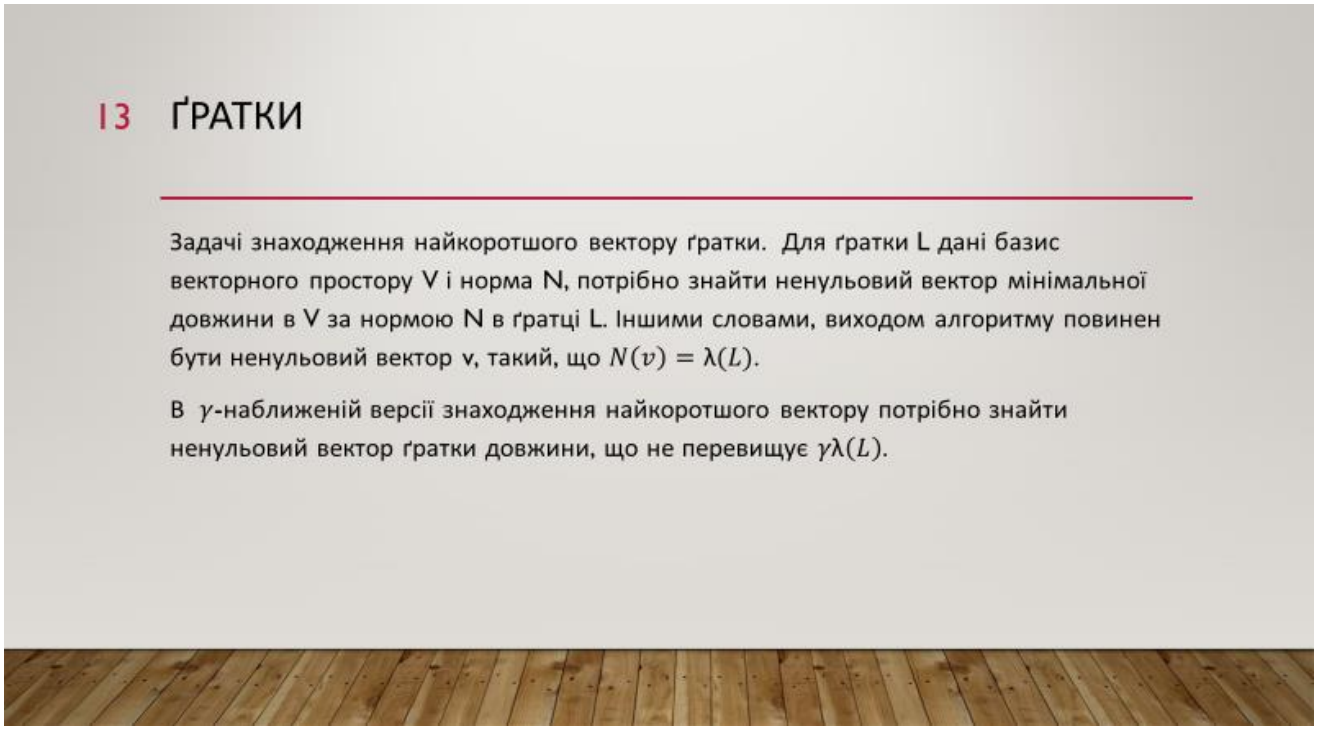

Рисунок Б.2 – Слайд 13

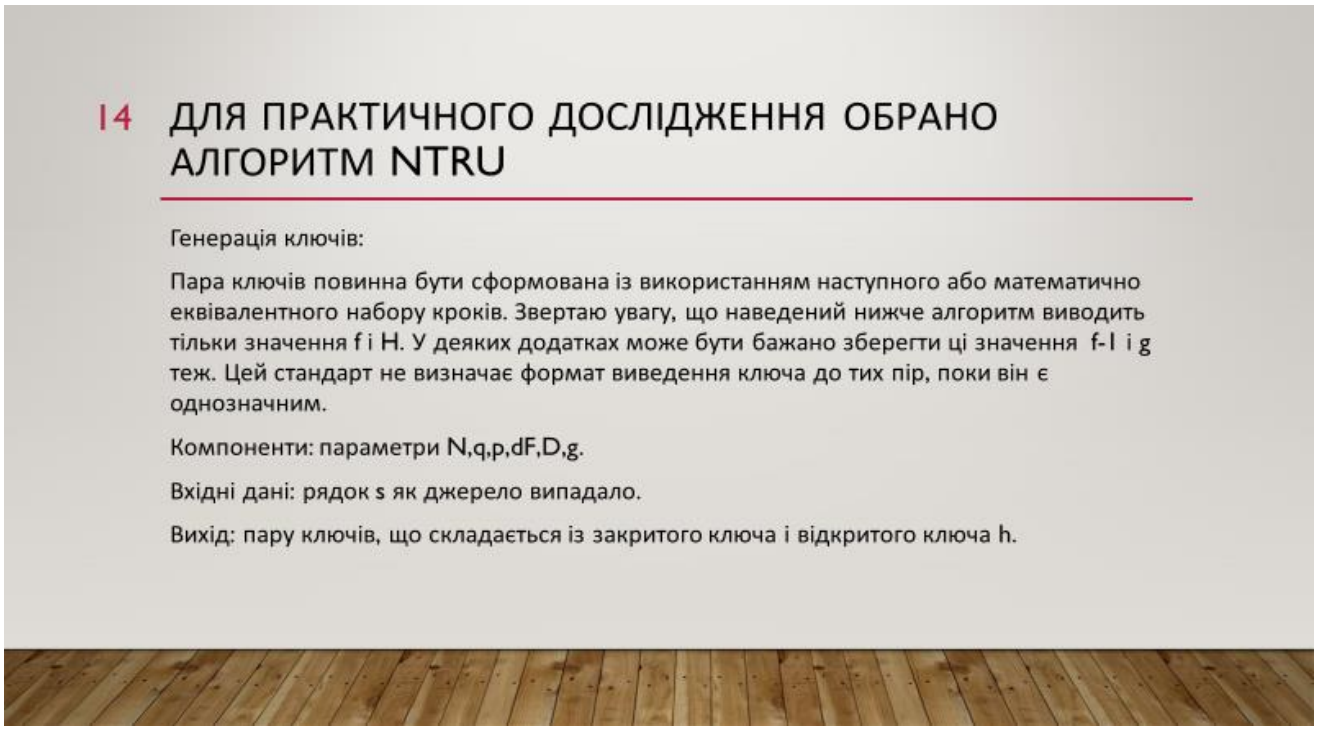

Рисунок Б.2 – Слайд 14

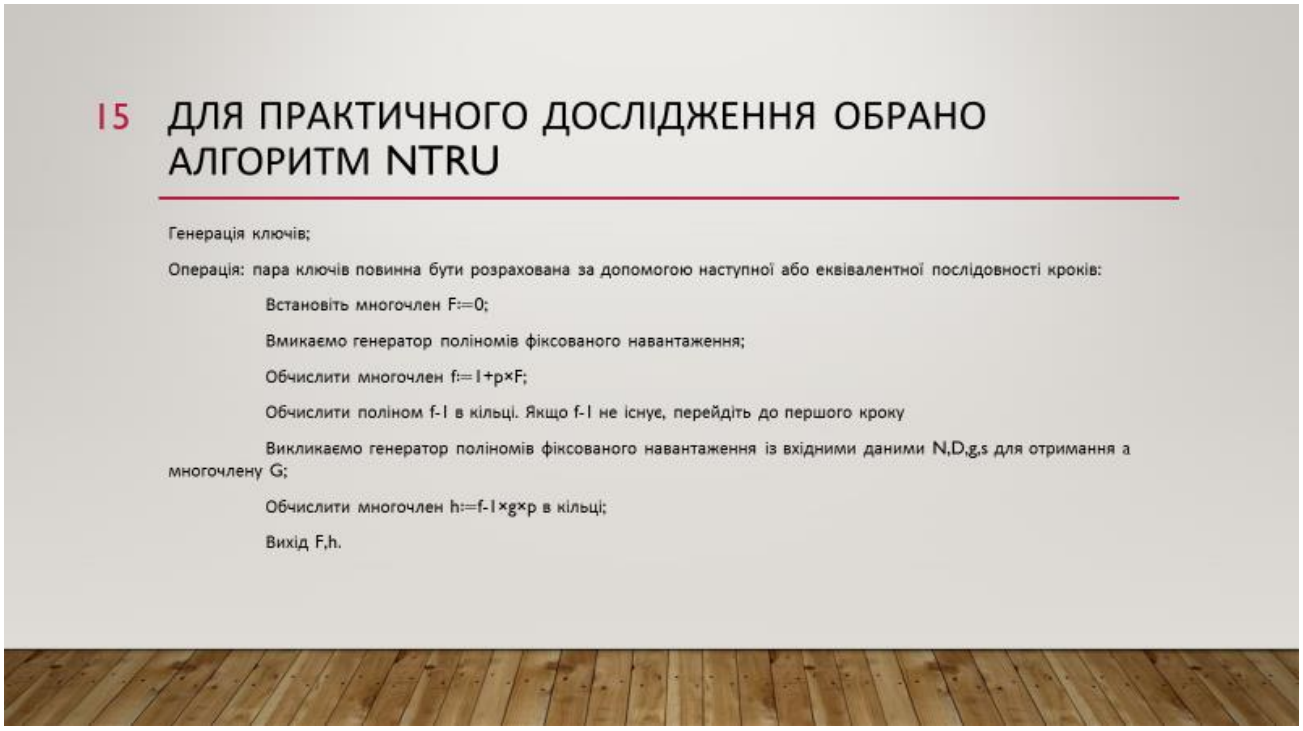

Рисунок Б.2 – Слайд 15

Рисунок Б.2 – Слайд 16

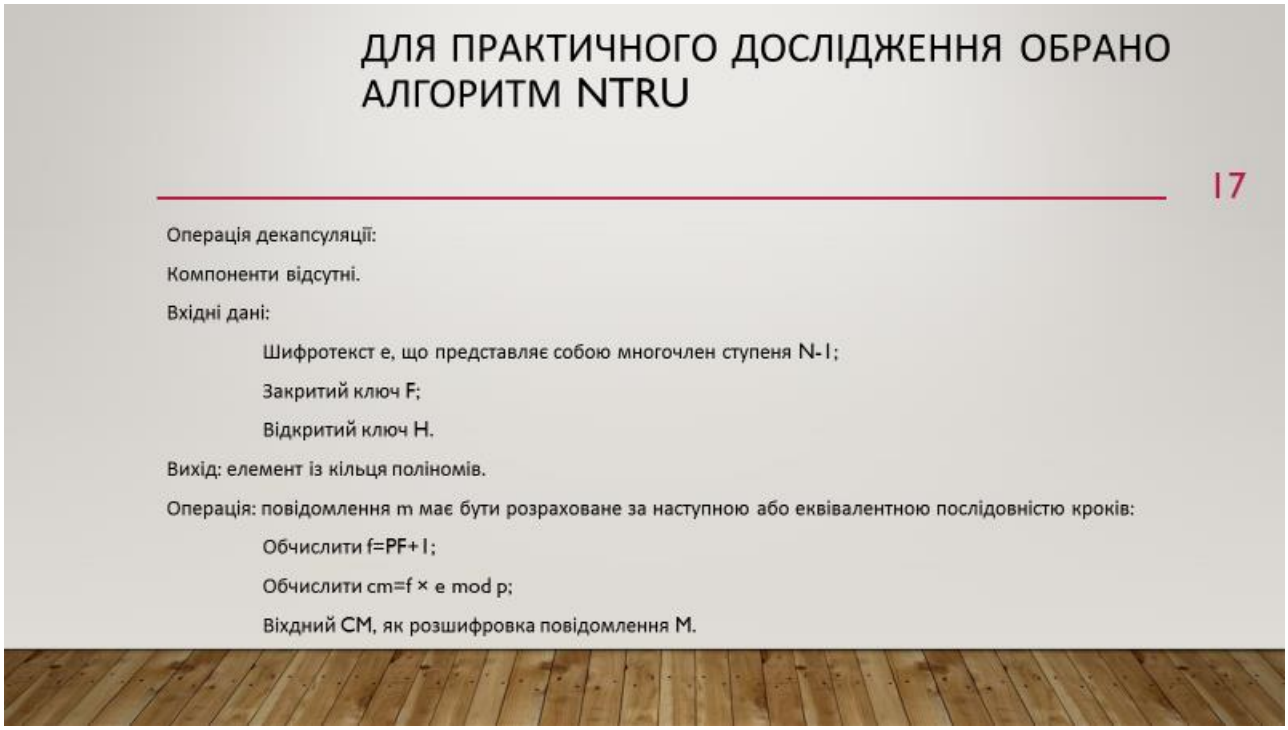

Рисунок Б.2 – Слайд 17

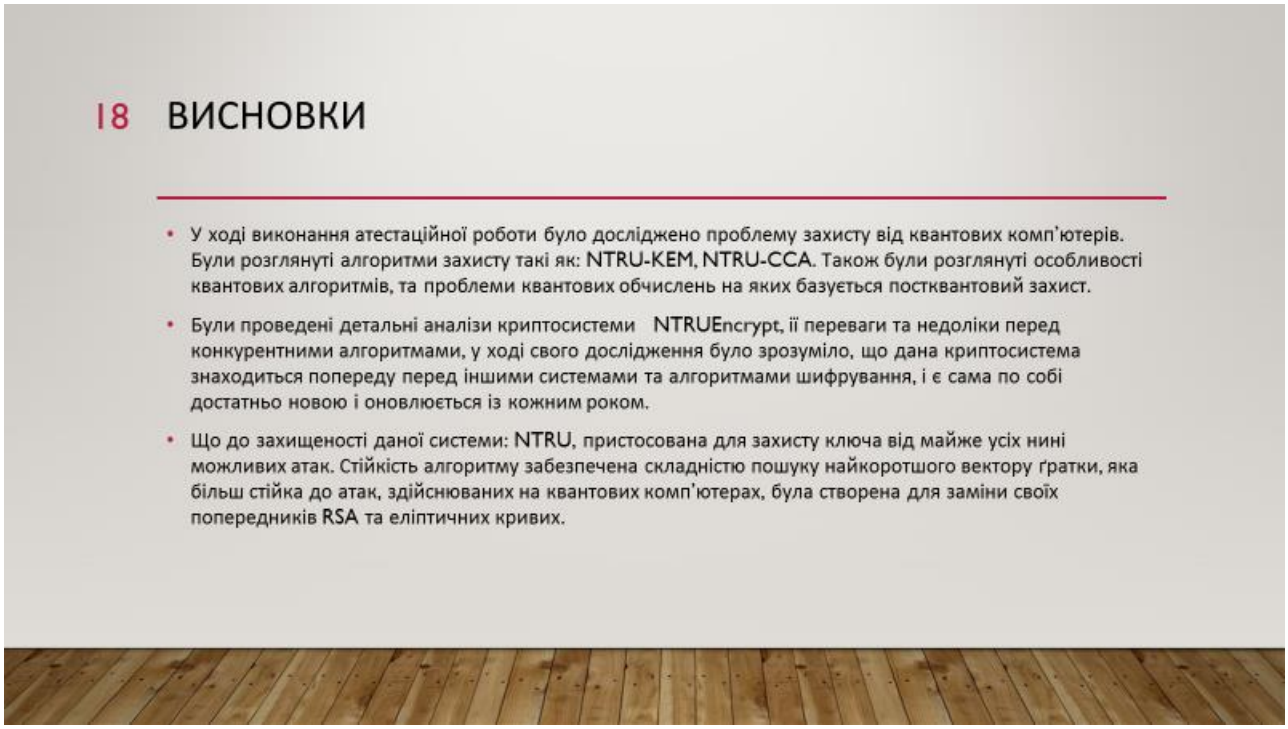

Рисунок Б.2 – Слайд 18

Рисунок Б.2 – Слайд 19