**Cryptography and 19115045**

**Network Security Lab Kunal Sachdeva**

**CS106402CS 6th Sem CSE**

**LAB EXAM**

**1. Write a program to implement the concept of SHA-1 algorithm.**

**Code:**

#include <stdint.h>

#include <string.h>

#include <stdio.h>

#include <stdlib.h>

#include <vector>

#define SHA1\_HEX\_SIZE (40 + 1)

#define SHA1\_BASE64\_SIZE (28 + 1)

class sha1 {

private:

void add\_byte\_dont\_count\_bits(uint8\_t x){

buf[i++] = x;

if (i >= sizeof(buf)){

i = 0;

process\_block(buf);

}

}

static uint32\_t rol32(uint32\_t x, uint32\_t n){

return (x << n) | (x >> (32 - n));

}

static uint32\_t make\_word(const uint8\_t \*p){

return

((uint32\_t)p[0] << 3\*8) |

((uint32\_t)p[1] << 2\*8) |

((uint32\_t)p[2] << 1\*8) |

((uint32\_t)p[3] << 0\*8);

}

void process\_block(const uint8\_t \*ptr){

const uint32\_t c0 = 0x5a827999;

const uint32\_t c1 = 0x6ed9eba1;

const uint32\_t c2 = 0x8f1bbcdc;

const uint32\_t c3 = 0xca62c1d6;

uint32\_t a = state[0];

uint32\_t b = state[1];

uint32\_t c = state[2];

uint32\_t d = state[3];

uint32\_t e = state[4];

uint32\_t w[16];

for (int i = 0; i < 16; i++) w[i] = make\_word(ptr + i\*4);

#define SHA1\_LOAD(i) w[i&15] = rol32(w[(i+13)&15] ^ w[(i+8)&15] ^ w[(i+2)&15] ^ w[i&15], 1);

#define SHA1\_ROUND\_0(v,u,x,y,z,i) z += ((u & (x ^ y)) ^ y) + w[i&15] + c0 + rol32(v, 5); u = rol32(u, 30);

#define SHA1\_ROUND\_1(v,u,x,y,z,i) SHA1\_LOAD(i) z += ((u & (x ^ y)) ^ y) + w[i&15] + c0 + rol32(v, 5); u = rol32(u, 30);

#define SHA1\_ROUND\_2(v,u,x,y,z,i) SHA1\_LOAD(i) z += (u ^ x ^ y) + w[i&15] + c1 + rol32(v, 5); u = rol32(u, 30);

#define SHA1\_ROUND\_3(v,u,x,y,z,i) SHA1\_LOAD(i) z += (((u | x) & y) | (u & x)) + w[i&15] + c2 + rol32(v, 5); u = rol32(u, 30);

#define SHA1\_ROUND\_4(v,u,x,y,z,i) SHA1\_LOAD(i) z += (u ^ x ^ y) + w[i&15] + c3 + rol32(v, 5); u = rol32(u, 30);

SHA1\_ROUND\_0(a, b, c, d, e, 0);

SHA1\_ROUND\_0(e, a, b, c, d, 1);

SHA1\_ROUND\_0(d, e, a, b, c, 2);

SHA1\_ROUND\_0(c, d, e, a, b, 3);

SHA1\_ROUND\_0(b, c, d, e, a, 4);

SHA1\_ROUND\_0(a, b, c, d, e, 5);

SHA1\_ROUND\_0(e, a, b, c, d, 6);

SHA1\_ROUND\_0(d, e, a, b, c, 7);

SHA1\_ROUND\_0(c, d, e, a, b, 8);

SHA1\_ROUND\_0(b, c, d, e, a, 9);

SHA1\_ROUND\_0(a, b, c, d, e, 10);

SHA1\_ROUND\_0(e, a, b, c, d, 11);

SHA1\_ROUND\_0(d, e, a, b, c, 12);

SHA1\_ROUND\_0(c, d, e, a, b, 13);

SHA1\_ROUND\_0(b, c, d, e, a, 14);

SHA1\_ROUND\_0(a, b, c, d, e, 15);

SHA1\_ROUND\_1(e, a, b, c, d, 16);

SHA1\_ROUND\_1(d, e, a, b, c, 17);

SHA1\_ROUND\_1(c, d, e, a, b, 18);

SHA1\_ROUND\_1(b, c, d, e, a, 19);

SHA1\_ROUND\_2(a, b, c, d, e, 20);

SHA1\_ROUND\_2(e, a, b, c, d, 21);

SHA1\_ROUND\_2(d, e, a, b, c, 22);

SHA1\_ROUND\_2(c, d, e, a, b, 23);

SHA1\_ROUND\_2(b, c, d, e, a, 24);

SHA1\_ROUND\_2(a, b, c, d, e, 25);

SHA1\_ROUND\_2(e, a, b, c, d, 26);

SHA1\_ROUND\_2(d, e, a, b, c, 27);

SHA1\_ROUND\_2(c, d, e, a, b, 28);

SHA1\_ROUND\_2(b, c, d, e, a, 29);

SHA1\_ROUND\_2(a, b, c, d, e, 30);

SHA1\_ROUND\_2(e, a, b, c, d, 31);

SHA1\_ROUND\_2(d, e, a, b, c, 32);

SHA1\_ROUND\_2(c, d, e, a, b, 33);

SHA1\_ROUND\_2(b, c, d, e, a, 34);

SHA1\_ROUND\_2(a, b, c, d, e, 35);

SHA1\_ROUND\_2(e, a, b, c, d, 36);

SHA1\_ROUND\_2(d, e, a, b, c, 37);

SHA1\_ROUND\_2(c, d, e, a, b, 38);

SHA1\_ROUND\_2(b, c, d, e, a, 39);

SHA1\_ROUND\_3(a, b, c, d, e, 40);

SHA1\_ROUND\_3(e, a, b, c, d, 41);

SHA1\_ROUND\_3(d, e, a, b, c, 42);

SHA1\_ROUND\_3(c, d, e, a, b, 43);

SHA1\_ROUND\_3(b, c, d, e, a, 44);

SHA1\_ROUND\_3(a, b, c, d, e, 45);

SHA1\_ROUND\_3(e, a, b, c, d, 46);

SHA1\_ROUND\_3(d, e, a, b, c, 47);

SHA1\_ROUND\_3(c, d, e, a, b, 48);

SHA1\_ROUND\_3(b, c, d, e, a, 49);

SHA1\_ROUND\_3(a, b, c, d, e, 50);

SHA1\_ROUND\_3(e, a, b, c, d, 51);

SHA1\_ROUND\_3(d, e, a, b, c, 52);

SHA1\_ROUND\_3(c, d, e, a, b, 53);

SHA1\_ROUND\_3(b, c, d, e, a, 54);

SHA1\_ROUND\_3(a, b, c, d, e, 55);

SHA1\_ROUND\_3(e, a, b, c, d, 56);

SHA1\_ROUND\_3(d, e, a, b, c, 57);

SHA1\_ROUND\_3(c, d, e, a, b, 58);

SHA1\_ROUND\_3(b, c, d, e, a, 59);

SHA1\_ROUND\_4(a, b, c, d, e, 60);

SHA1\_ROUND\_4(e, a, b, c, d, 61);

SHA1\_ROUND\_4(d, e, a, b, c, 62);

SHA1\_ROUND\_4(c, d, e, a, b, 63);

SHA1\_ROUND\_4(b, c, d, e, a, 64);

SHA1\_ROUND\_4(a, b, c, d, e, 65);

SHA1\_ROUND\_4(e, a, b, c, d, 66);

SHA1\_ROUND\_4(d, e, a, b, c, 67);

SHA1\_ROUND\_4(c, d, e, a, b, 68);

SHA1\_ROUND\_4(b, c, d, e, a, 69);

SHA1\_ROUND\_4(a, b, c, d, e, 70);

SHA1\_ROUND\_4(e, a, b, c, d, 71);

SHA1\_ROUND\_4(d, e, a, b, c, 72);

SHA1\_ROUND\_4(c, d, e, a, b, 73);

SHA1\_ROUND\_4(b, c, d, e, a, 74);

SHA1\_ROUND\_4(a, b, c, d, e, 75);

SHA1\_ROUND\_4(e, a, b, c, d, 76);

SHA1\_ROUND\_4(d, e, a, b, c, 77);

SHA1\_ROUND\_4(c, d, e, a, b, 78);

SHA1\_ROUND\_4(b, c, d, e, a, 79);

#undef SHA1\_LOAD

#undef SHA1\_ROUND\_0

#undef SHA1\_ROUND\_1

#undef SHA1\_ROUND\_2

#undef SHA1\_ROUND\_3

#undef SHA1\_ROUND\_4

state[0] += a;

state[1] += b;

state[2] += c;

state[3] += d;

state[4] += e;

}

public:

uint32\_t state[5];

uint8\_t buf[64];

uint32\_t i;

uint64\_t n\_bits;

sha1(const char \*text = NULL): i(0), n\_bits(0){

state[0] = 0x67452301;

state[1] = 0xEFCDAB89;

state[2] = 0x98BADCFE;

state[3] = 0x10325476;

state[4] = 0xC3D2E1F0;

if (text) add(text);

}

sha1& add(uint8\_t x){

add\_byte\_dont\_count\_bits(x);

n\_bits += 8;

return \*this;

}

sha1& add(char c){

return add(\*(uint8\_t\*)&c);

}

sha1& add(const void \*data, uint32\_t n){

if (!data) return \*this;

const uint8\_t \*ptr = (const uint8\_t\*)data;

// fill up block if not full

for (; n && i % sizeof(buf); n--) add(\*ptr++);

// process full blocks

for (; n >= sizeof(buf); n -= sizeof(buf)){

process\_block(ptr);

ptr += sizeof(buf);

n\_bits += sizeof(buf) \* 8;

}

// process remaining part of block

for (; n; n--) add(\*ptr++);

return \*this;

}

sha1& add(const char \*text){

if (!text) return \*this;

return add(text, strlen(text));

}

sha1& finalize(){

// hashed text ends with 0x80, some padding 0x00 and the length in bits

add\_byte\_dont\_count\_bits(0x80);

while (i % 64 != 56) add\_byte\_dont\_count\_bits(0x00);

for (int j = 7; j >= 0; j--) add\_byte\_dont\_count\_bits(n\_bits >> j \* 8);

return \*this;

}

const sha1& print\_hex(

char \*hex,

bool zero\_terminate = true,

const char \*alphabet = "0123456789abcdef"

) const {

// print hex

int k = 0;

for (int i = 0; i < 5; i++){

for (int j = 7; j >= 0; j--){

hex[k++] = alphabet[(state[i] >> j \* 4) & 0xf];

}

}

if (zero\_terminate) hex[k] = '\0';

return \*this;

}

const sha1& print\_base64(char \*base64, bool zero\_terminate = true) const {

static const uint8\_t \*table = (const uint8\_t\*)

"ABCDEFGHIJKLMNOPQRSTUVWXYZ"

"abcdefghijklmnopqrstuvwxyz"

"0123456789"

"+/";

uint32\_t triples[7] = {

((state[0] & 0xffffff00) >> 1\*8),

((state[0] & 0x000000ff) << 2\*8) | ((state[1] & 0xffff0000) >> 2\*8),

((state[1] & 0x0000ffff) << 1\*8) | ((state[2] & 0xff000000) >> 3\*8),

((state[2] & 0x00ffffff) << 0\*8),

((state[3] & 0xffffff00) >> 1\*8),

((state[3] & 0x000000ff) << 2\*8) | ((state[4] & 0xffff0000) >> 2\*8),

((state[4] & 0x0000ffff) << 1\*8),

};

for (int i = 0; i < 7; i++){

uint32\_t x = triples[i];

base64[i\*4 + 0] = table[(x >> 3\*6) % 64];

base64[i\*4 + 1] = table[(x >> 2\*6) % 64];

base64[i\*4 + 2] = table[(x >> 1\*6) % 64];

base64[i\*4 + 3] = table[(x >> 0\*6) % 64];

}

base64[SHA1\_BASE64\_SIZE - 2] = '=';

if (zero\_terminate) base64[SHA1\_BASE64\_SIZE - 1] = '\0';

return \*this;

}

};

void example(){

const char \*text = "This is an SHA-1 encryption algorithm!";

char hex[SHA1\_HEX\_SIZE];

char base64[SHA1\_BASE64\_SIZE];

// constructor can be empty or take a const char\*

sha1("")

// can be chained

// can add single chars

.add(text[0])

// number of bytes

.add(&text[1], 4)

// 0-terminated const char\*

.add(&text[5])

// finalize must be called, otherwise the hash is not valid

// after that, no more bytes should be added

.finalize()

// print the hash in hexadecimal, 0-terminated

.print\_hex(hex)

// print the hash in base64, 0-terminated

.print\_base64(base64);

printf("SHA1(%s)\n",text);

printf("\n");

printf("hexadecimal\n");

printf("calculated: %s\n", hex);

printf("\n");

printf("base64 encoded\n");

printf("calculated: %s\n", base64);

}

void test(const char \*expected, const char \*text){

char hex[SHA1\_HEX\_SIZE];

sha1(text).finalize().print\_hex(hex);

if (strcmp(expected, hex) != 0){

printf("hash of : %s\n", text);

printf("wrong hash : %s\n", hex);

printf("expected hash: %s\n", expected);

}

}

int main(){

example();

const char \*expected = "c6e7d00fedc0ca6f41e7c96ca5ed6221486f947b";

// initialize with a-z one million times

std::vector<char> buf(26\*1000\*1000 + 1);

for (size\_t i = 0; i < buf.size(); i++) buf[i] = 'a' + (i%26);

buf.back() = '\0';

sha1 s;

// chop up buf and feed it bite by bite

size\_t offset = 0;

while (1){

size\_t remaining = buf.size() - 1 - offset;

if (remaining == 0) break;

size\_t n = rand() % 128;

if (n > remaining) n = remaining;

s.add(&buf[offset], n);

offset += n;

}

s.finalize();

char hex[SHA1\_HEX\_SIZE];

s.print\_hex(hex);

if (strcmp(expected, hex) != 0){

printf("hash of a to z one million times\n");

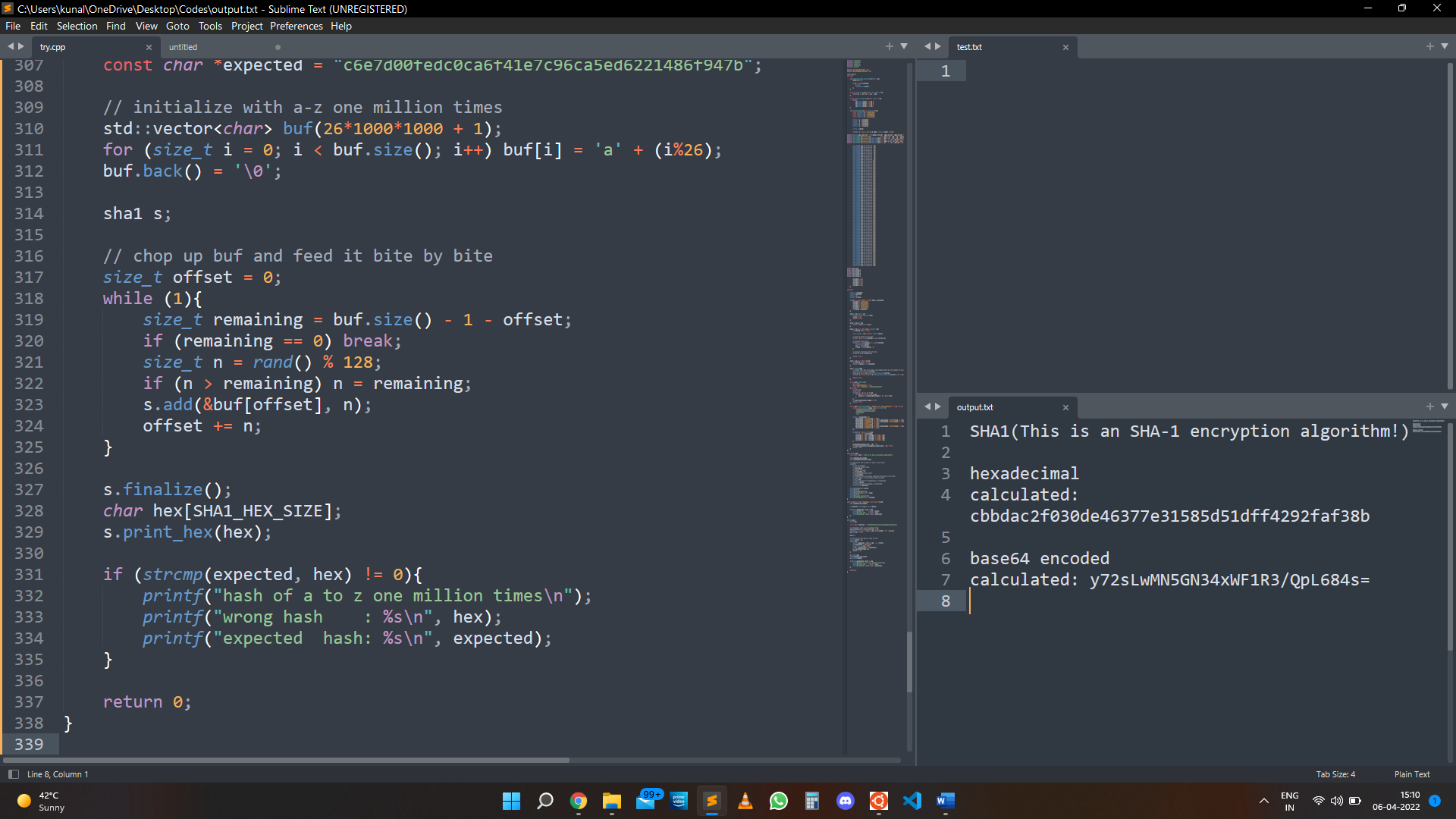
printf("wrong hash : %s\n", hex);

printf("expected hash: %s\n", expected);

}

return 0;

}

**Output:**

**2. Write a program to implement the concept of Elliptic Curve Cryptosystem.**

**Code:**

#include <iostream>

#include <math.h>

#include <cstdlib>

#include <vector>

using namespace std;

int main()

{

int n, p;

cout << "Elliptic Curve General Form \t y^2 mod p = (x^3 + A\*x + B) mod p \n";

cout << "Enter the value of P: \n";

cin >> p;

n = p;

int LHS[2][n], RHS[2][n], a, b, i, j;

cout << "\nEnter the Value of a: \n";

cin >> a;

cout << "\nEnter the Value of b: \n";

cin >> b;

cout << "\nCurrent Elliptic Curve \t\t ---> y^2 mod " << p << " = (x^3 + " << a << "\*x + " << b << ") mod p\n\n\n";

vector<int> arr\_x;

vector<int> arr\_y;

// Equating LHS and RHS as per arbitrary index to generate range of values.

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

{

LHS[0][i] = i;

RHS[0][i] = i;

LHS[1][i] = ((i \* i \* i) + a \* i + b) % p;

RHS[1][i] = (i \* i) % p;

}

// Generating Base Points

int in\_c = 0;

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

{

for (j = 0; j < n; j++)

{

if (LHS[1][i] == RHS[1][j])

{

in\_c++;

arr\_x.push\_back(LHS[0][i]);

arr\_y.push\_back(RHS[0][j]);

}

}

}

cout << endl

<< "Generated Points are:" << endl;

for (i = 0; i < in\_c; i++)

{

cout << i + 1 << "\t( " << arr\_x[i] << " , " << arr\_y[i] << " )" << endl;

}

cout << "Base Point: (" << arr\_x[0] << "," << arr\_y[0] << ")"

<< "\n";

int k, d, M;

cout << "Enter the random number 'd' i.e. Private key of Sender (d<n)\n";

cin >> d;

int Qx = d \* arr\_x[0];

int Qy = d \* arr\_y[0];

// Q is the public key of sende

// Encryption

cout << "Enter the random number 'k' (k<n)\n";

cin >> k;

cout << "Enter the message to be sent:\n";

cin >> M;

cout << "The message to be sent is:\n"

<< M << "\n";

int c1x = k \* arr\_x[0];

int c1y = k \* arr\_y[0];

cout << "Value of C1: (" << c1x << "," << c1y << ")"

<< "\n";

int c2x = k \* Qx + M;

int c2y = k \* Qy + M;

cout << "Value of C2: (" << c2x << "," << c2y << ")"

<< "\n";

// Decryption

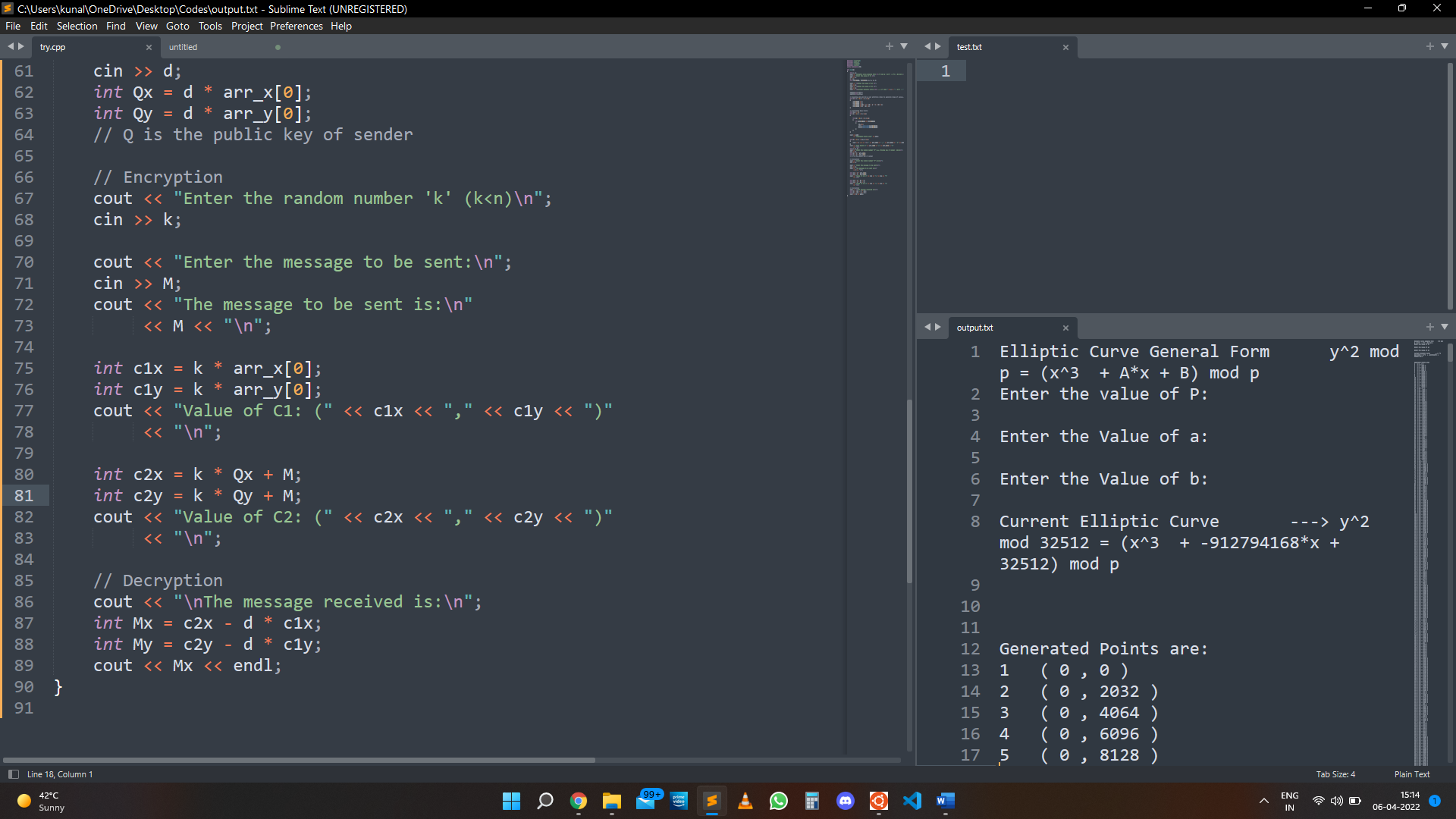
cout << "\nThe message received is:\n";

int Mx = c2x - d \* c1x;

int My = c2y - d \* c1y;

cout << Mx << endl;

}

**Output:**

