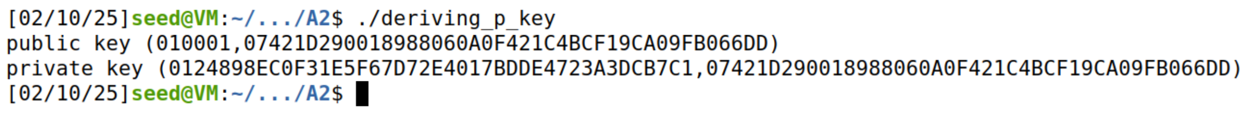
Task 1:Deriving_p_key.c

```c
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 128

void printBN(char *msg, BIGNUM *a, BIGNUM *b)
{
    char *number_str_a = BN_bn2hex(a);
    char *number_str_b = BN_bn2hex(b);
    printf("%s (%s,%s)\n", msg, number_str_a, number_str_b);
    OPENSSL_free(number_str_a);
    OPENSSL_free(number_str_b);
}

int main()
{
    // init
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *p = BN_new();
    BIGNUM *q = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *phi = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *res = BN_new();
    BIGNUM *p_minus_1 = BN_new();
    BIGNUM *q_minus_1 = BN_new();

    // assign value
    BN_hex2bn(&p, "5adb09bdbee704d048da9");
    BN_hex2bn(&q, "14738741a1260adf78815");
    BN_hex2bn(&e, "10001");
    // n = pq
    BN_mul(n, p, q, ctx);
    printBN("public key", e, n);

    // phi(n) = (p-1)*(q-1)
    BN_sub(p_minus_1, p, BN_value_one());
    BN_sub(q_minus_1, q, BN_value_one());
    BN_mul(phi, p_minus_1, q_minus_1, ctx);

    // check if e and phi(n) is relatively prime
    BN_gcd(res, phi, e, ctx);
    if (!BN_is_one(res))
    {
        printf("Error: e and phi(n) is not relatively prime \n ");
        exit(0);
    }

    BN_mod_inverse(d, e, phi, ctx);
    printBN("private key", d, n);

    BN_clear_free(p);
    BN_clear_free(q);
    BN_clear_free(n);
    BN_clear_free(res);
    BN_clear_free(phi);
    BN_clear_free(e);
    BN_clear_free(d);
    BN_clear_free(p_minus_1);
    BN_clear_free(q_minus_1);

    return 0;
}
```

Bit Length of modulus n:

1. Convert values to decimal
    a. P = 686485635739882925094O329
    b. Q = 1545255223585965958727701
2. Multiply decimal version of p and q to get n
    a. 1.06079551* 10^49
3. log_2^ x + 1 = bit length
    a. The above equation gives 163 which means bit length of modulus n = 163

Result after `gcc -o deriving_p_key deriving_p_key.c -lcrypto`

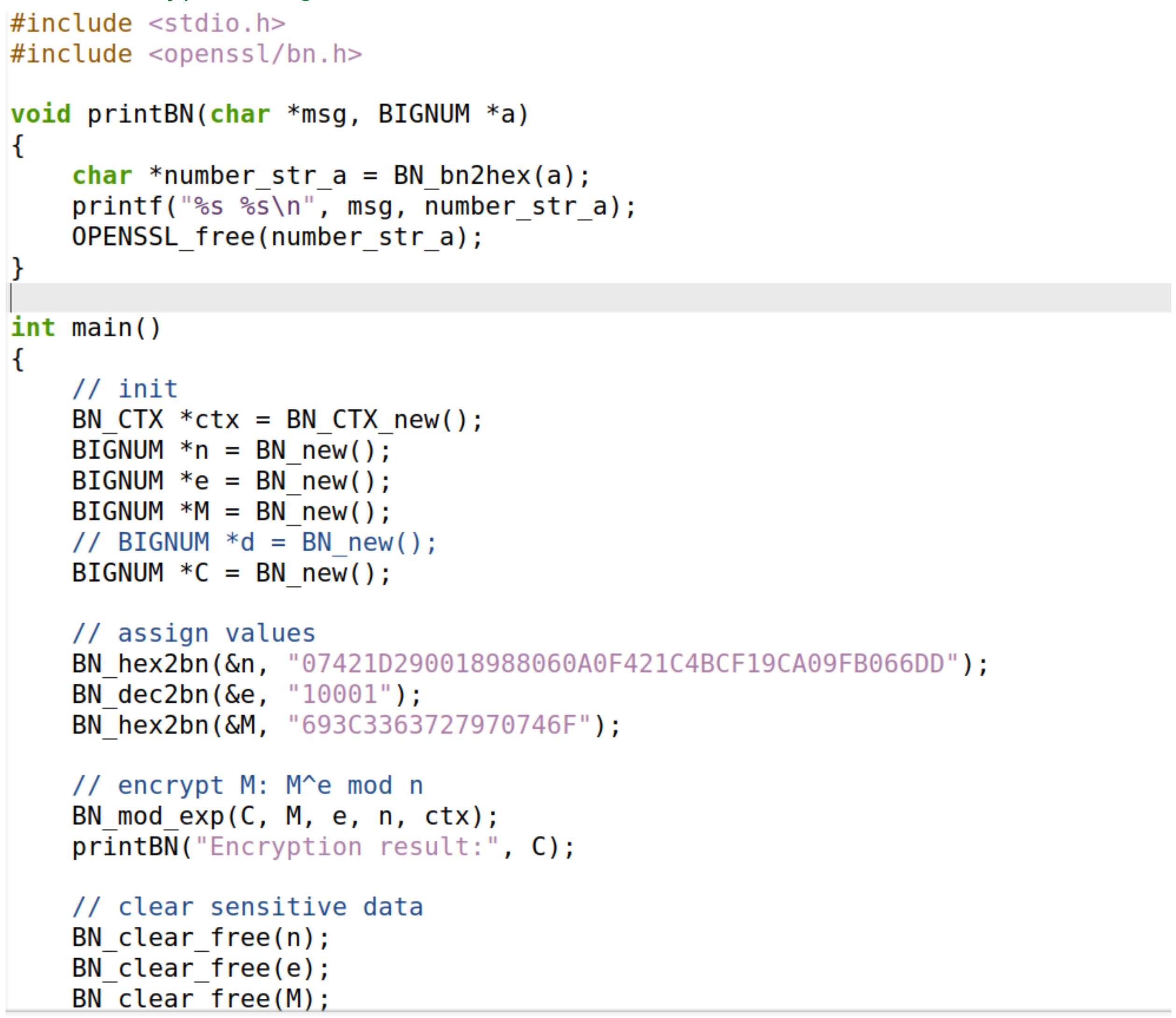


```
[02/10/25]seed@VM:~/.../A2$ ./deriving_p_key
public key (010001,07421D290018988060A0F421C4BCF19CA09FB066DD)
private key (0124898EC0F31E5F67D72E4017BDDE4723A3DCB7C1,07421D290018988060A0F421C4BCF19CA09FB066DD)
[02/10/25]seed@VM:~/.../A2$ 
```

Private Key is:
0124898EC0F31E5F67D72E4017BDDE4723A3DCB7C1,07421D290018988060A0F421C4BC
F19CA09FB066DD

Task 2:
Convert “i<3crypto” to hexadecimal: 693C3363727970746F

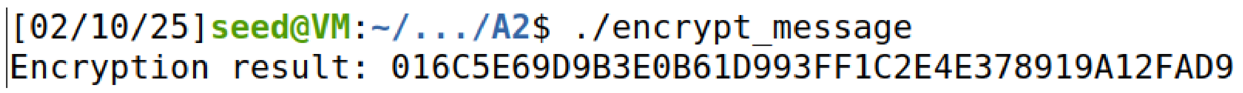
```
touch encrypt_mesage.c
```



```c
#include <stdio.h>
#include <openssl/bn.h>

void printBN(char *msg, BIGNUM *a)
{
    char *number_str_a = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str_a);
    OPENSSL_free(number_str_a);
}

int main()
{
    // init
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *M = BN_new();
    // BIGNUM *d = BN_new();
    BIGNUM *C = BN_new();

    // assign values
    BN_hex2bn(&n, "07421D290018988060A0F421C4BCF19CA09FB066DD");
    BN_dec2bn(&e, "10001");
    BN_hex2bn(&M, "693C3363727970746F");

    // encrypt M: M^e mod n
    BN_mod_exp(C, M, e, n, ctx);
    printBN("Encryption result:", C);

    // clear sensitive data
    BN_clear_free(n);
    BN_clear_free(e);
    BN_clear_free(M);
```

Missing lines from screenshot: BN_clear_free(C) & return 0

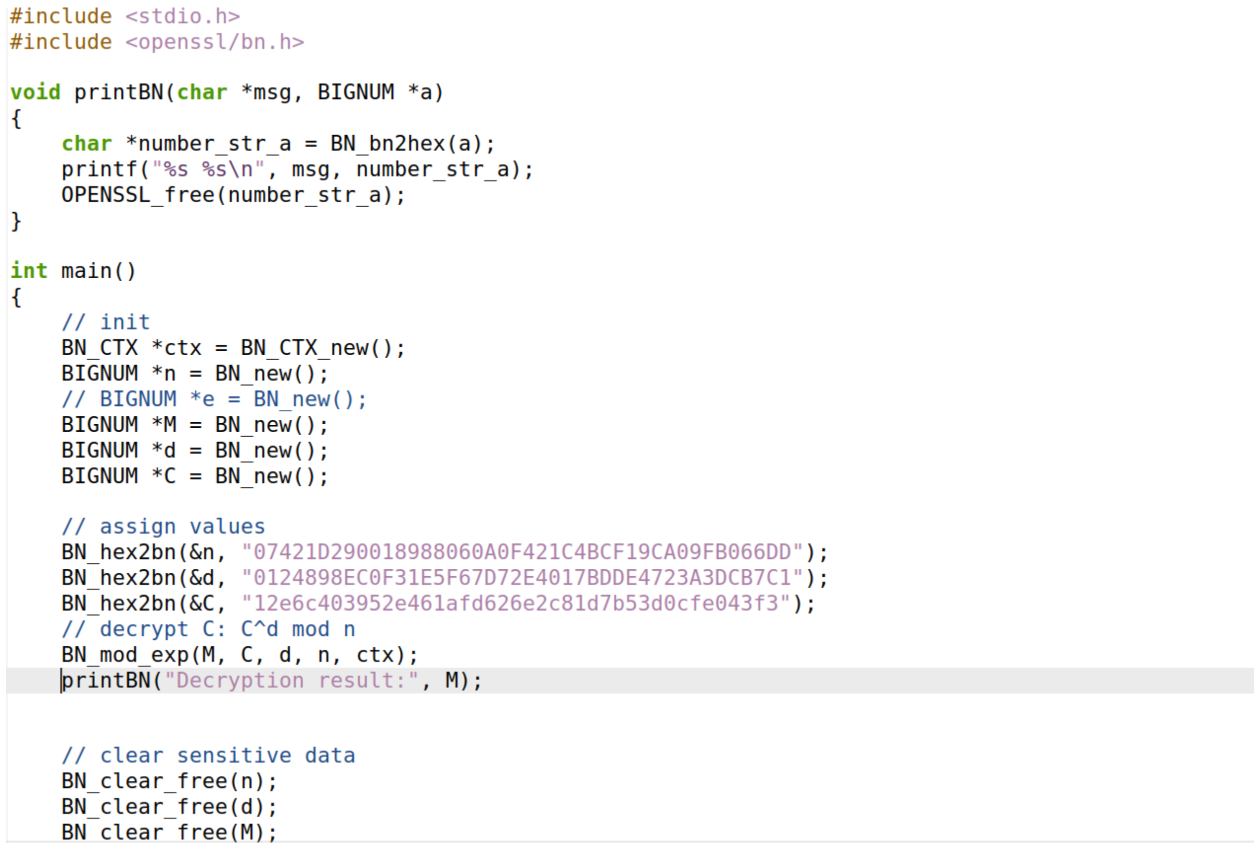
Output when compiled and run:



```
[02/10/25]seed@VM:~/.../A2$ ./encrypt_message
Encryption result: 016C5E69D9B3E0B61D993FF1C2E4E378919A12FAD9
```

Task 3:

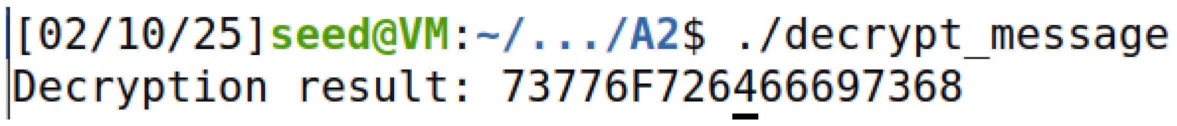
`touch decrypt_message.c`



```c
#include <stdio.h>
#include <openssl/bn.h>

void printBN(char *msg, BIGNUM *a)
{
    char *number_str_a = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str_a);
    OPENSSL_free(number_str_a);
}

int main()
{
    // init
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *n = BN_new();
    // BIGNUM *e = BN_new();
    BIGNUM *M = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *C = BN_new();

    // assign values
    BN_hex2bn(&n, "07421D290018988060A0F421C4BCF19CA09FB066DD");
    BN_hex2bn(&d, "0124898EC0F31E5F67D72E4017BDDE4723A3DCB7C1");
    BN_hex2bn(&C, "12e6c403952e461afd626e2c81d7b53d0cfe043f3");
    // decrypt C: C^d mod n
    BN_mod_exp(M, C, d, n, ctx);
    printBN("Decryption result:", M);


    // clear sensitive data
    BN_clear_free(n);
    BN_clear_free(d);
    BN_clear_free(M);
```

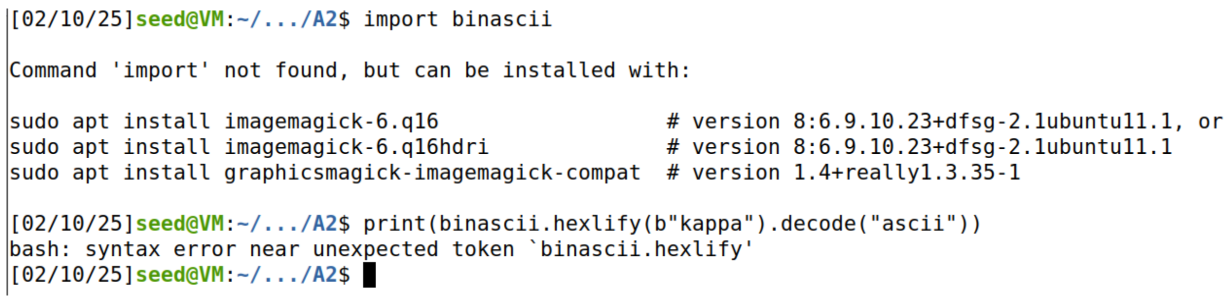
Missing lines from screenshot: BN_clear_free(C) & return 0

After compiling and running `./decrypt_message`:



```
[02/10/25]seed@VM:~/.../A2$ ./decrypt_message
Decryption result: 73776F726466697368
```

Converting HEX to ASCII using import binascii provided by document outputs

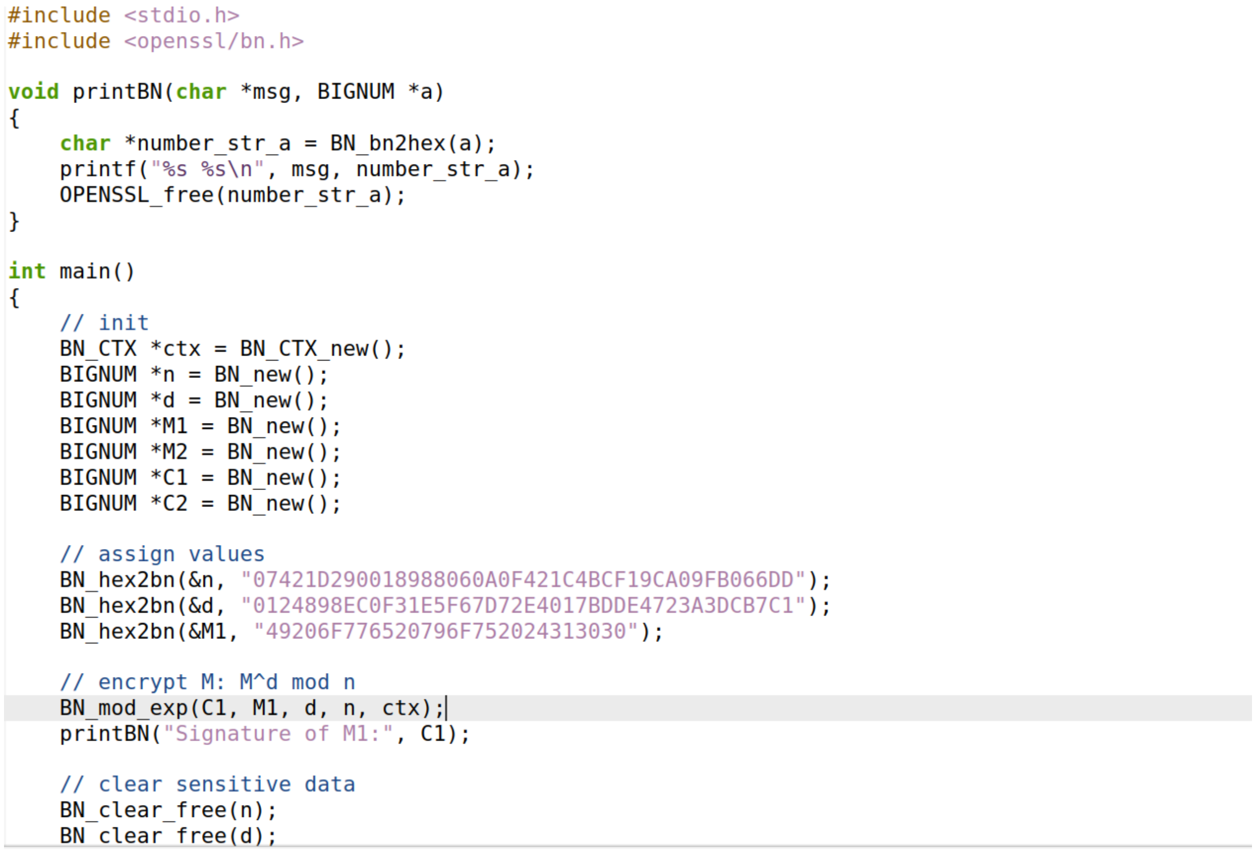


```
[02/10/25]seed@VM:~/.../A2$ import binascii

Command 'import' not found, but can be installed with:

sudo apt install imagemagick-6.q16                  # version 8:6.9.10.23+dfsg-2.1ubuntu11.1, or
sudo apt install imagemagick-6.q16hdri              # version 8:6.9.10.23+dfsg-2.1ubuntu11.1
sudo apt install graphicsmagick-imagemagick-compat  # version 1.4+really1.3.35-1

[02/10/25]seed@VM:~/.../A2$ print(binascii.hexlify(b"kappa").decode("ascii"))
bash: syntax error near unexpected token `binascii.hexlify'
[02/10/25]seed@VM:~/.../A2$ 
```

Just used internet HEX to ASCII

73776F726466697368 to ASCII gives: swordfish

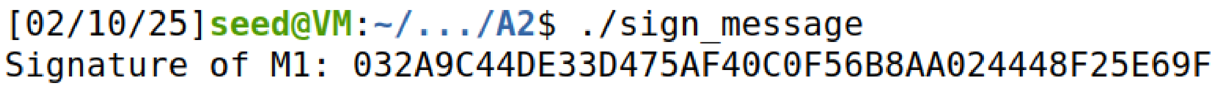
Task 4:
I owe you $100 → hex: 49206F776520796F752024313030



```
#include <stdio.h>
#include <openssl/bn.h>

void printBN(char *msg, BIGNUM *a)
{
    char *number_str_a = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str_a);
    OPENSSL_free(number_str_a);
}

int main()
{
    // init
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *n = BN_new();
    BIGNUM *d = BN_new();
    BIGNUM *M1 = BN_new();
    BIGNUM *M2 = BN_new();
    BIGNUM *C1 = BN_new();
    BIGNUM *C2 = BN_new();

    // assign values
    BN_hex2bn(&n, "07421D290018988060A0F421C4BCF19CA09FB066DD");
    BN_hex2bn(&d, "0124898EC0F31E5F67D72E4017BDDE4723A3DCB7C1");
    BN_hex2bn(&M1, "49206F776520796F752024313030");

    // encrypt M: M^d mod n
    BN_mod_exp(C1, M1, d, n, ctx);
    printBN("Signature of M1:", C1);

    // clear sensitive data
    BN_clear_free(n);
    BN_clear_free(d);
```

Lines missing from sign_message.c:
BN_clear_free(M1);
BN_clear_free(C1);
return 0;
*note BIGNUM *C2 variable created for last step -

Initial signature of “I owe you $100” → 032A9C44DE33D475AF40C0F56B8AA024448F25E69F



```
[02/10/25]seed@VM:~/.../A2$ ./sign_message
Signature of M1: 032A9C44DE33D475AF40C0F56B8AA024448F25E69F
```

The private exponent d is used for signing given that

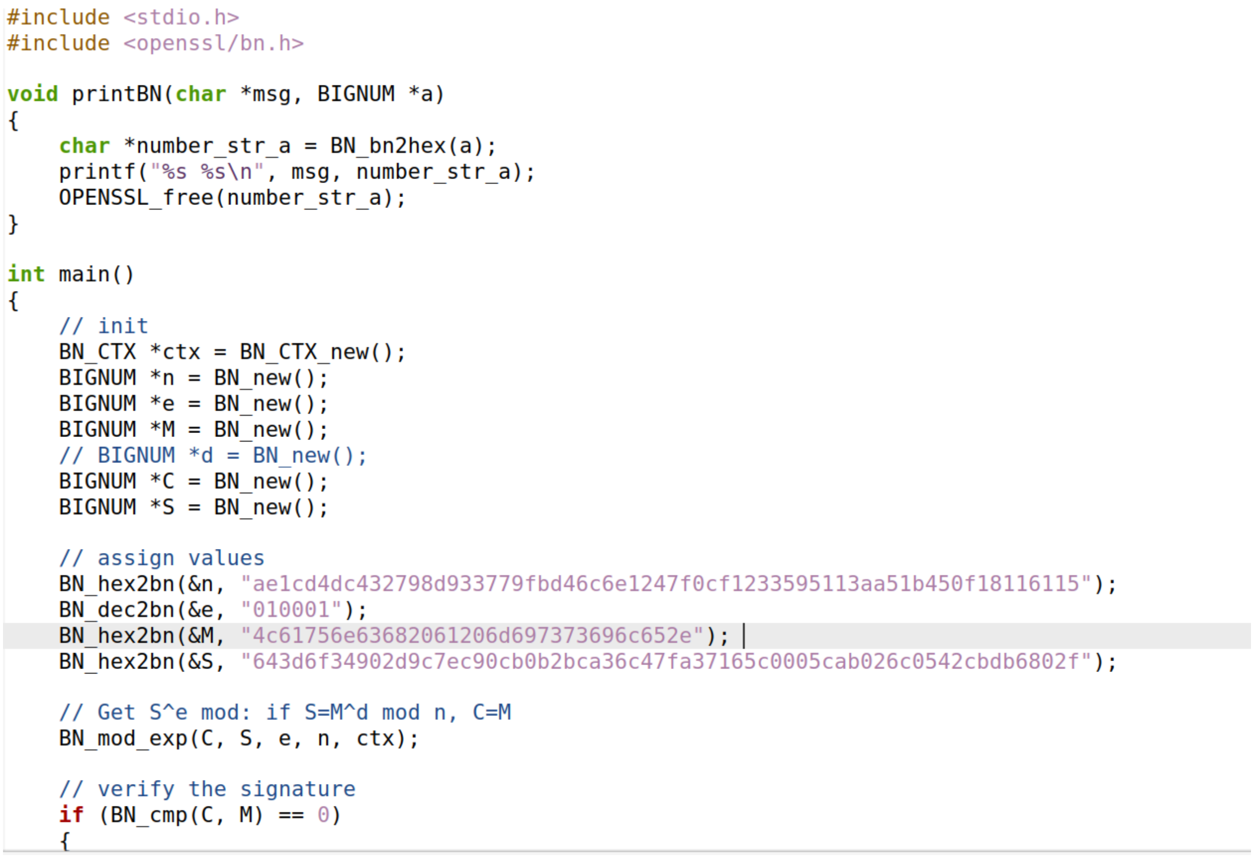
- S = M^d mod n

After changing the ASCII message to “I owe you $200”
Signature M2 → 02FF9B900B701412DE3A6620CB96E7F6216C589157
vs
Signature M1 → 032A9C44DE33D475AF40C0F56B8AA024448F25E69F

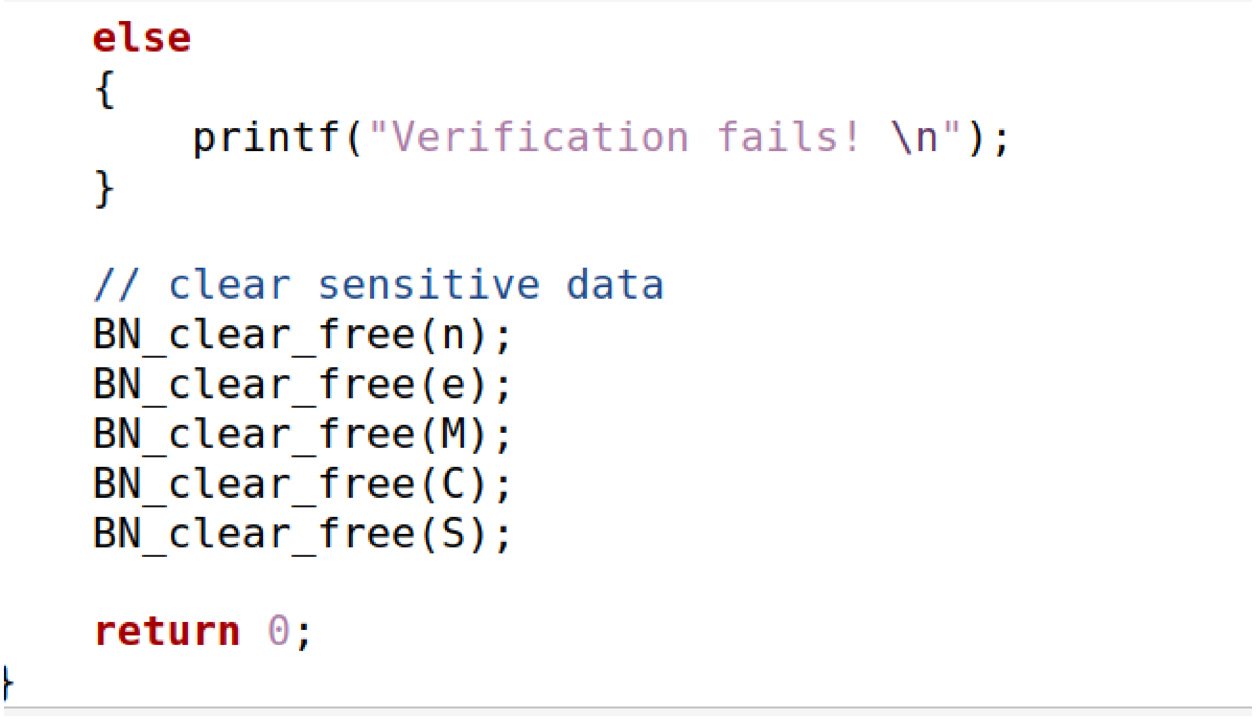
Only one value in the hex code changed but due to modular exponentiation, large changes occur in the signature. This is due to the non-linear properties of S = M^d mod n.

Task 5:

Get hex value for “Launch a missile.” → 4c61756e63682061206d697373696c652e

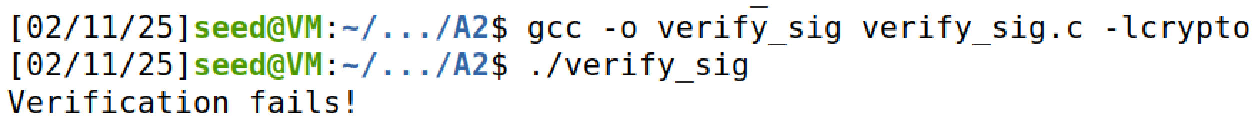


```c
#include <stdio.h>
#include <openssl/bn.h>

void printBN(char *msg, BIGNUM *a)
{
    char *number_str_a = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str_a);
    OPENSSL_free(number_str_a);
}

int main()
{
    // init
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *n = BN_new();
    BIGNUM *e = BN_new();
    BIGNUM *M = BN_new();
    // BIGNUM *d = BN_new();
    BIGNUM *C = BN_new();
    BIGNUM *S = BN_new();

    // assign values
    BN_hex2bn(&n, "ae1cd4dc432798d933779fbd46c6e1247f0cf1233595113aa51b450f18116115");
    BN_dec2bn(&e, "010001");
    BN_hex2bn(&M, "4c61756e63682061206d697373696c652e");
    BN_hex2bn(&S, "643d6f34902d9c7ec90cb0b2bca36c47fa37165c0005cab026c0542cbdb6802f");

    // Get S^e mod: if S=M^d mod n, C=M
    BN_mod_exp(C, S, e, n, ctx);

    // verify the signature
    if (BN_cmp(C, M) == 0)
    {
```



```c
    else
    {
        printf("Verification fails! \n");
    }

    // clear sensitive data
    BN_clear_free(n);
    BN_clear_free(e);
    BN_clear_free(M);
    BN_clear_free(C);
    BN_clear_free(S);

    return 0;
}
```

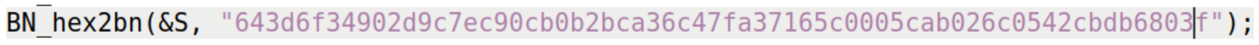
Compile verify_sig.c and run



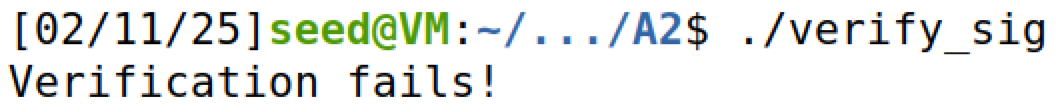
```
[02/11/25]seed@VM:~/.../A2$ gcc -o verify_sig verify_sig.c -lcrypto
[02/11/25]seed@VM:~/.../A2$ ./verify_sig
Verification fails!
```

This is not Alice's Signature

If we change a value in the signature, invalidates the C = S^e mod n because if S changes even only slightly, the outputted C will be vastly different from the original message M



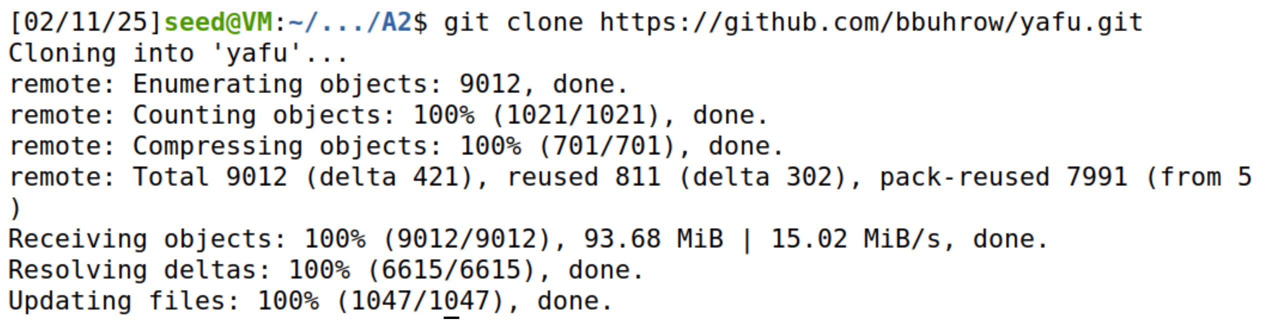
```
BN_hex2bn(&S, "643d6f34902d9c7ec90cb0b2bca36c47fa37165c0005cab026c0542cbdb6803f");
```



```
[02/11/25]seed@VM:~/.../A2$ ./verify_sig
Verification fails!
```

Task 6:

Using yafu to factorize



```
[02/11/25]seed@VM:~/.../A2$ git clone https://github.com/bbuhrow/yafu.git
Cloning into 'yafu'...
remote: Enumerating objects: 9012, done.
remote: Counting objects: 100% (1021/1021), done.
remote: Compressing objects: 100% (701/701), done.
remote: Total 9012 (delta 421), reused 811 (delta 302), pack-reused 7991 (from 5
)
Receiving objects: 100% (9012/9012), 93.68 MiB | 15.02 MiB/s, done.
Resolving deltas: 100% (6615/6615), done.
Updating files: 100% (1047/1047), done.
```

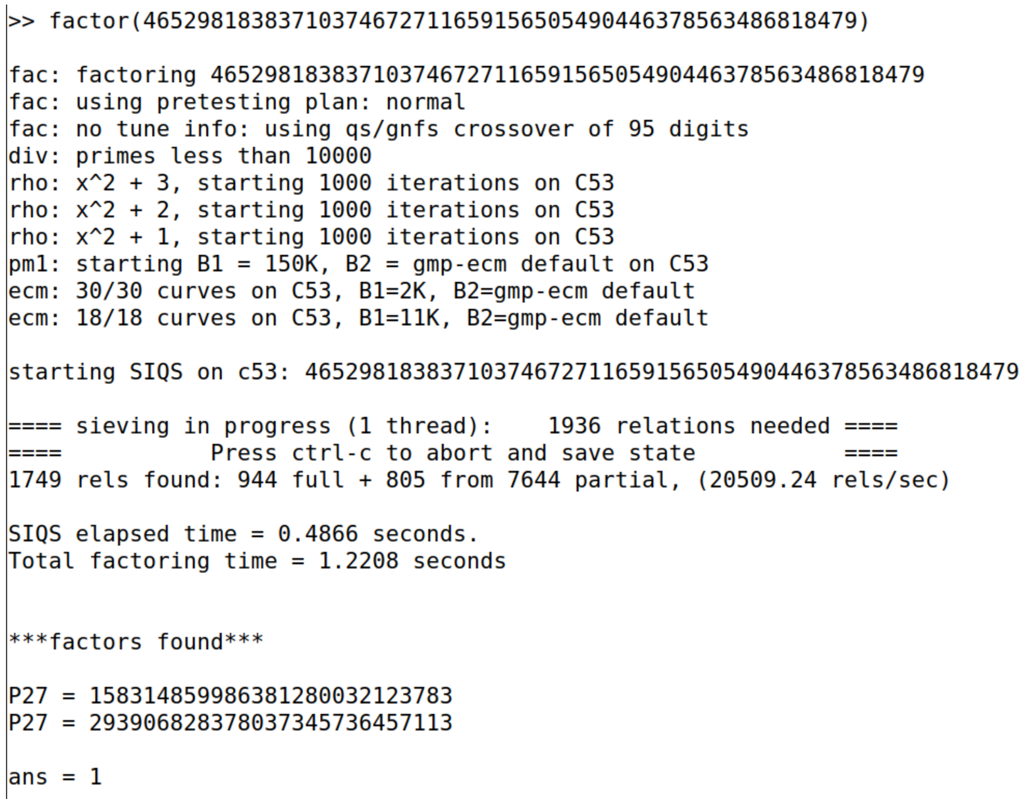
For

e = 0x10001

n = 0x7c5cfe617c286a27ffc10ecf88a8d35ebbf1e30320af

Prepare input for yafu by changing n to decimal

- 46529818383710374672711659156505490446378563486818479

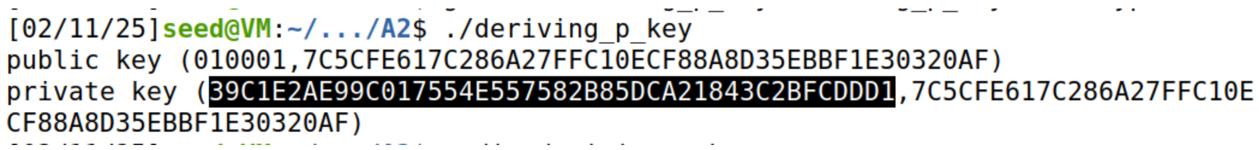


```
>> factor(46529818383710374672711659156505490446378563486818479)

fac: factoring 46529818383710374672711659156505490446378563486818479
fac: using pretesting plan: normal
fac: no tune info: using qs/gnfs crossover of 95 digits
div: primes less than 10000
rho: x^2 + 3, starting 1000 iterations on C53
rho: x^2 + 2, starting 1000 iterations on C53
rho: x^2 + 1, starting 1000 iterations on C53
pm1: starting B1 = 150K, B2 = gmp-ecm default on C53
ecm: 30/30 curves on C53, B1=2K, B2=gmp-ecm default
ecm: 18/18 curves on C53, B1=11K, B2=gmp-ecm default

starting SIQS on c53: 46529818383710374672711659156505490446378563486818479

==== sieving in progress (1 thread):    1936 relations needed ====
====           Press ctrl-c to abort and save state           ====
1749 rels found: 944 full + 805 from 7644 partial, (20509.24 rels/sec)

SIQS elapsed time = 0.4866 seconds.
Total factoring time = 1.2208 seconds


***factors found***

P27 = 158314859986381280032123783
P27 = 293906828378037345736457113

ans = 1
```

Factors of n:
P = 158314859986381280032123783
Q = 293906828378037345736457113

Convert back p and q back into hex and then plug p and q back into our script from task 1
deriving_p_key.c

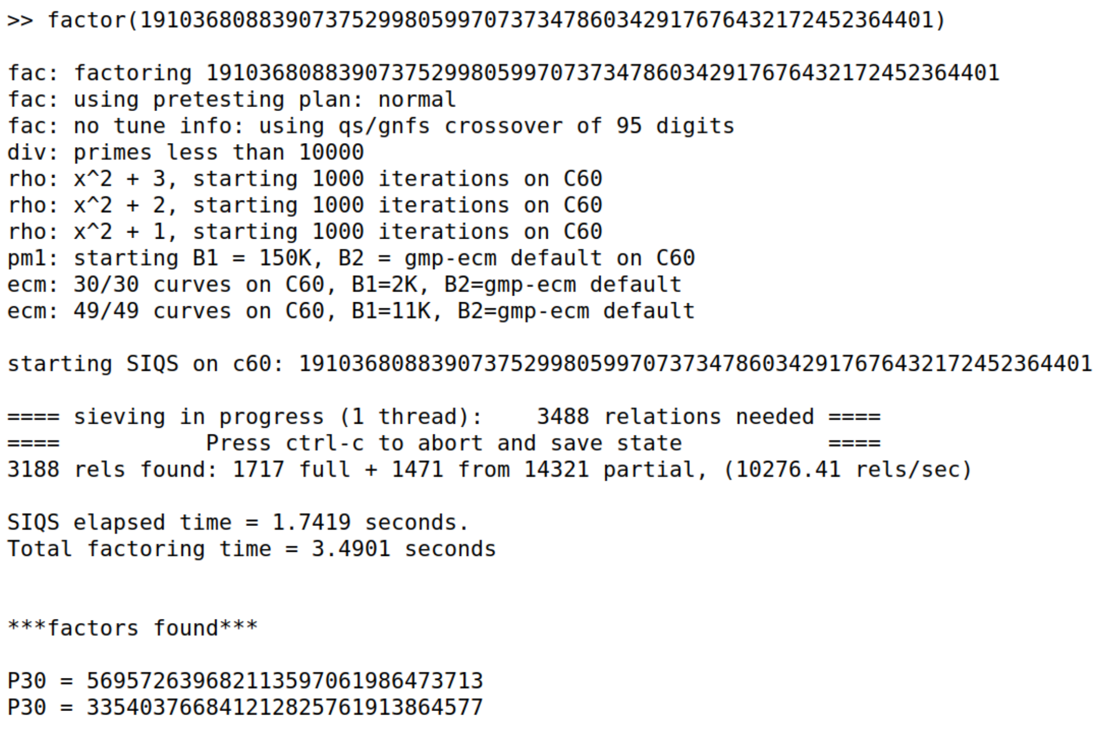
Get back
D = 39C1E2AE99C017554E557582B85DCA21843C2BFCDDD1
N = 7C5CFE617C286A27FFC10ECF88A8D35EBBF1E30320AF



```
[02/11/25]seed@VM:~/.../A2$ ./deriving_p_key
public key (010001,7C5CFE617C286A27FFC10ECF88A8D35EBBF1E30320AF)
private key (39C1E2AE99C017554E557582B85DCA21843C2BFCDDD1,7C5CFE617C286A27FFC10E
CF88A8D35EBBF1E30320AF)
```

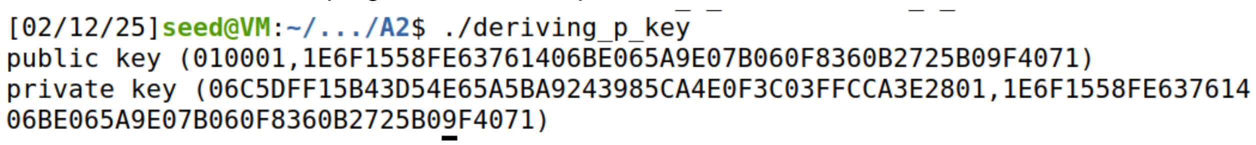
For
e = 0x10001
N = 0x1e6f1558fe63761406be065a9e07b060f8360b2725b09f4071

Prepare input for yafu by changing n to a decimal
- 191036808839073752998059970737347860342917676432172452364401



```
>> factor(191036808839073752998059970737347860342917676432172452364401)

fac: factoring 191036808839073752998059970737347860342917676432172452364401
fac: using pretesting plan: normal
fac: no tune info: using qs/gnfs crossover of 95 digits
div: primes less than 10000
rho: x^2 + 3, starting 1000 iterations on C60
rho: x^2 + 2, starting 1000 iterations on C60
rho: x^2 + 1, starting 1000 iterations on C60
pm1: starting B1 = 150K, B2 = gmp-ecm default on C60
ecm: 30/30 curves on C60, B1=2K, B2=gmp-ecm default
ecm: 49/49 curves on C60, B1=11K, B2=gmp-ecm default

starting SIQS on c60: 191036808839073752998059970737347860342917676432172452364401

==== sieving in progress (1 thread):    3488 relations needed ====
====           Press ctrl-c to abort and save state           ====
3188 rels found: 1717 full + 1471 from 14321 partial, (10276.41 rels/sec)

SIQS elapsed time = 1.7419 seconds.
Total factoring time = 3.4901 seconds


***factors found***

P30 = 569572639682113597061986473713
P30 = 335403766841212825761913864577
```

Factors of n:
P = 569572639682113597061986473713
Q = 335403766841212825761913864577
Convert back into hex to plug into task 1 script



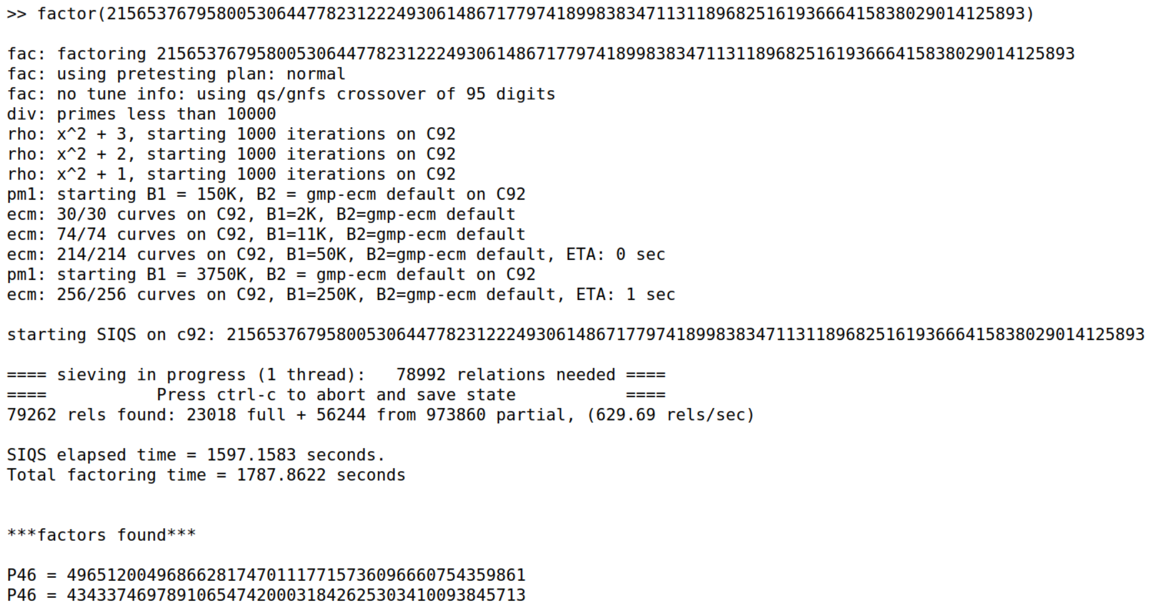
```
[02/12/25]seed@VM:~/.../A2$ ./deriving_p_key
public key (010001,1E6F1558FE63761406BE065A9E07B060F8360B2725B09F4071)
private key (06C5DFF15B43D54E65A5BA9243985CA4E0F3C03FFCCA3E2801,1E6F1558FE637614
06BE065A9E07B060F8360B2725B09F4071)
```

D = 06C5DFF15B43D54E65A5BA9243985CA4E0F3C03FFCCA3E2801

For
e = 0x10001
N =
a962e5e6455feec804c6e6faa62b66a50675a824e5e73ac03b009a7d36abb557c14418444d45

Prepare input for yafu by changing n to a decimal

- 21565376795800530644778231222493061486717797418998383471131189682516193666415838029014125893



```
>> factor(21565376795800530644778231222493061486717797418998383471131189682516193666415838029014125893)

fac: factoring 21565376795800530644778231222493061486717797418998383471131189682516193666415838029014125893
fac: using pretesting plan: normal
fac: no tune info: using qs/gnfs crossover of 95 digits
div: primes less than 10000
rho: x^2 + 3, starting 1000 iterations on C92
rho: x^2 + 2, starting 1000 iterations on C92
rho: x^2 + 1, starting 1000 iterations on C92
pm1: starting B1 = 150K, B2 = gmp-ecm default on C92
ecm: 30/30 curves on C92, B1=2K, B2=gmp-ecm default
ecm: 74/74 curves on C92, B1=11K, B2=gmp-ecm default
ecm: 214/214 curves on C92, B1=50K, B2=gmp-ecm default, ETA: 0 sec
pm1: starting B1 = 3750K, B2 = gmp-ecm default on C92
ecm: 256/256 curves on C92, B1=250K, B2=gmp-ecm default, ETA: 1 sec

starting SIQS on c92: 21565376795800530644778231222493061486717797418998383471131189682516193666415838029014125893

==== sieving in progress (1 thread):   78992 relations needed ====
====           Press ctrl-c to abort and save state           ====
79262 rels found: 23018 full + 56244 from 973860 partial, (629.69 rels/sec)

SIQS elapsed time = 1597.1583 seconds.
Total factoring time = 1787.8622 seconds


***factors found***

P46 = 4965120049686628174701117715736096660754359861
P46 = 4343374697891065474200031842625303410093845713
```

P = 4343374697891065474200031842625303410093845713
Q = 4965120049686628174701117715736096660754359861

Convert P & Q back into hex to plug back into task 1 script

D =
A55CB36C36560D5A9FAF3E577C59EA7F38EB2A803BE7F5F6DA9C40A8D87298E726F028
15F301

The first two keys factorized relatively quickly at
1.22 seconds and 3.5 seconds respectively, while the last key took 1788 seconds or 29.8 minutes

For keys around 50-60 digits, the factoring time is extremely fast. But for a Key of 93 digits, the factoring time was much larger.
Given these facts, cracking RSA keys is definitely non-linear, and the time it takes to factor bigger and bigger numbers grows exponentially. Additionally, I believe hardware plays a big role. More RAM and faster processor = more calculations per second.

CPU - intel i7-7700HQ 2.80GHz
RAM - 16 GB