Project #1 (Taken from the SEED lab project): Due March 13th.

1) Task 1: Frequency Analysis -- It is well-known that monoalphabetic substitution cipher (also known as monoalphabetic cipher) is not secure, because it can be subjected to frequency analysis. In this project, you are given a cipher text that is encrypted using a monoalphabetic cipher; namely, each letter in the original text is replaced by another letter, where the replacement does not vary (i.e., a letter is always replaced by the same letter during the encryption). Your job is to find out the original text using frequency analysis. It is known that the original text is an English article.

   In the following, we describe how we encrypt the original article, and what simplification we have made. We use the tr command to do the encryption. We only encrypt letters, while leaving the space and return characters alone.

   ```
   $ tr 'abcdefghijklmnopqrstuvwxyz' 'sxtrwinqbedpvgkfmalhyuojzc' ₩
       < plaintext.txt > ciphertext.txt
   ```

   A ciphertext is created using a different encryption key (not the one described above). It is provided as a file. Your job is to use the frequency analysis to figure out the encryption key and the original plaintext.

   There are many online resources that you can use to do the task. We list four useful links in the following:

   a) https://www.dcode.fr/frequency-analysis : This website provides a frequency analysis.

   b) https://en.wikipedia.org/wiki/Frequency_analysis: This Wikipedia page provides frequencies for a typical English plaintext.

   c) https://en.wikipedia.org/wiki/Bigram: Bigram frequency.

   d) https://en.wikipedia.org/wiki/Trigram: Trigram frequency

2) Task 2: Make a program and find the private key for the given p, q, and e. Let p, q, and e be three prime numbers. Let n = p*q. We will use (e, n) as the public key. Please calculate the private key d. The hexadecimal values of p, q, and e are listed in the following. It should be noted that although p and q used in this task are quite large numbers, they are not large enough to be secure. We intentionally make them small for the sake of simplicity. In practice, these numbers should be at least 512 bits long (the one used here are only 128 bits).

p = F7E75FDC469067FFDC4E847C51F452DF

q = E85CED54AF57E53E092113E62F436F4F

e = 0D88C3

To be able to do this task, you may need big number library.

There are several libraries that can perform arithmetic operations on integers of arbitrary size. In this project, we will use the Big Number library provided by openssl. To use this library, we will define each big number as a BIGNUM type, and then use the APIs provided by the library for various operations, such as addition, multiplication, exponentiation, modular operations, etc.

All the big number APIs can be found from https://linux.die.net/man/3/bn. In the following, we describe some of the APIs that are needed for this project.

a) Some of the library functions requires temporary variables. Since dynamic memory allocation to create BIGNUMs is quite expensive when used in conjunction with repeated subroutine calls, a BN CTX structure is created to holds BIGNUM temporary variables used by library functions. We need to create such a structure, and pass it to the functions that requires it.

   ```
   BN_CTX *ctx = BN_CTX_new()
   ```

b) Initialize a BIGNUM variable.

   ```
   BIGNUM *a = BN_new()
   ```

c) There are a number of ways to assign a value to a BIGNUM variable.

   ```
   // Assign a value from a decimal number string
   BN_dec2bn(&a, "12345678901112231223");
   // Assign a value from a hex number string
   BN_hex2bn(&a, "2A3B4C55FF77889AED3F");
   // Generate a random number of 128 bits
   BN_rand (a, 128, 0, 0);
   // Generate a random prime number of 128 bits
   BN_generate_prime_ex (a, 128, 1, NULL, NULL, NULL);
   ```

d) Print out a big number.

```
void printBN (char *msg, BIGNUM * a)
{
    // Convert the BIGNUM to number string
    char * number_str = BN_bn2dec(a);
    // Print out the number string
    printf("%s %s₩n", msg, number_str);
    // Free the dynamically allocated memory
    OPENSSL_free (number_str);
}
```

e) Compute res = a − b and res = a + b:

```
BN_sub (res, a, b);
BN_add (res, a, b);
```

f) Compute res = a ∗ b. It should be noted that a BN CTX structure is need in this API.

```
BN_mul (res, a, b, ctx)
```

g) Compute res = a ∗ b mod n:

```
BN_mod_mul (res, a, b, n, ctx)
```

h) Compute res = a c mod n:

```
BN_mod_exp (res, a, c, n, ctx)
```

i) Compute modular inverse, i.e., given a, find b, such that a ∗ b mod n = 1. The value b is called the inverse of a, with respect to modular n.

```
BN_mod_inverse (b, a, n, ctx);
```

j) A Complete Example: In this example, we initialize three BIGNUM variables, a, b, and n; we then compute a ∗ b and (a b mod n).

```
/* bn_sample.c */
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256
void printBN (char *msg, BIGNUM * a)
{
```

```
    /* Use BN_bn2hex(a) for hex string
    * Use BN_bn2dec(a) for decimal string */
    char * number_str = BN_bn2hex(a);
    printf("%s %s₩n", msg, number_str);
    OPENSSL_free(number_str);
}

int main ()
{
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *a = BN_new();
    BIGNUM *b = BN_new();
    BIGNUM *n = BN_new();
    BIGNUM *res = BN_new();
    // Initialize a, b, n
    BN_generate_prime_ex(a, NBITS, 1, NULL, NULL, NULL);
    BN_dec2bn(&b, "273489463796838501848592769467194369268");
    BN_rand(n, NBITS, 0, 0);
    // res = a*b
    BN_mul(res, a, b, ctx);
    printBN("a * b = ", res);
    // res = aˆb mod n
    BN_mod_exp(res, a, b, n, ctx);
    printBN("aˆc mod n = ", res);
    return 0;
}
```

k) Compilation. We can use the following command to compile bn sample.c (the character after - is the letter l, not the number 1; it tells the compiler to use the crypto library).

```
$ gcc bn_sample.c -lcrypto
```