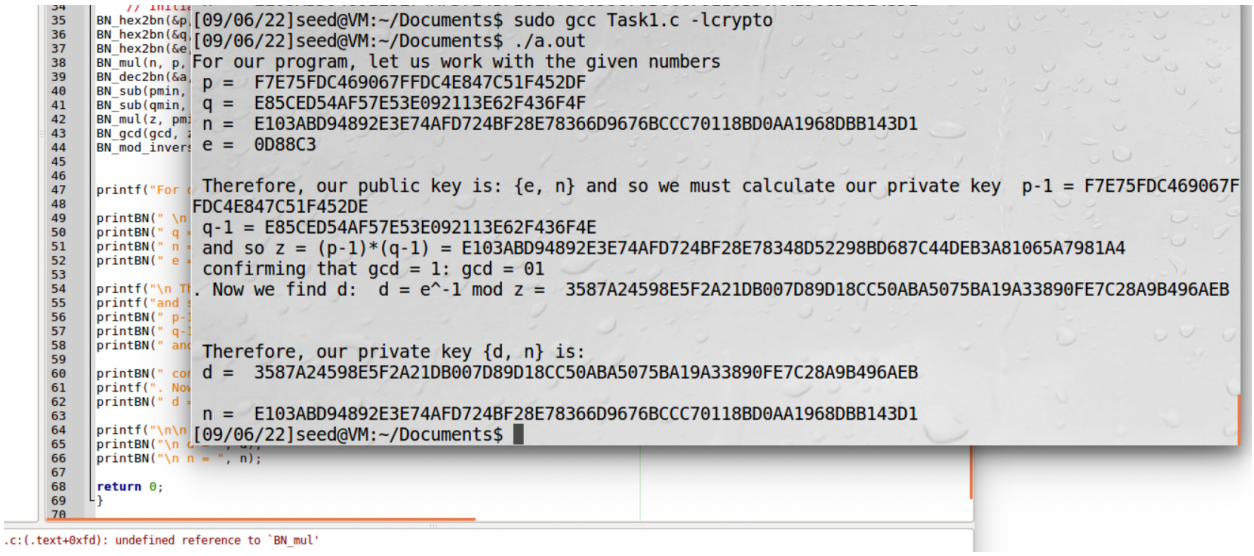
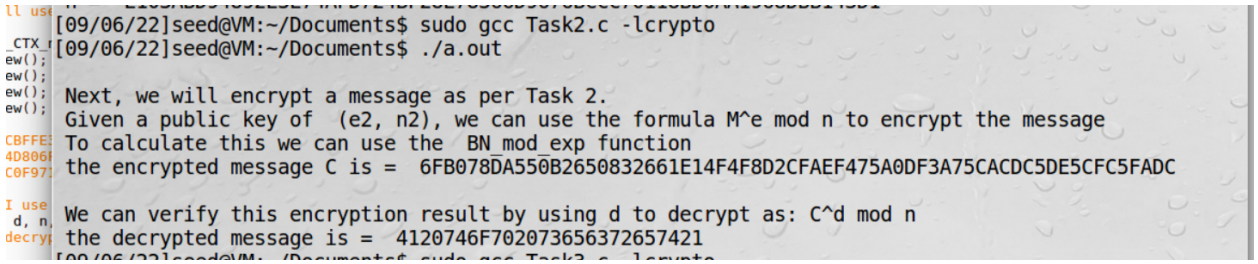
Task 1: Finding private key.

First, we used the BIGNum library to create the variables to store our ‘p’, ‘q’ and ‘e’ values, and BN_hex to initialize those variables with the given values from the assignment. Next, we used BN_mul to calculate and assign the value of p*q to a new variable ‘n’.
BN_dec2bn was used to create a BIGNUM type of value 1, which we then assigned to a variable ‘a’. With this value, we were able to call BN_sub to find the value of p-1 and q-1. This was necessary in order to get the value of (p-1)*(q-1), which we assigned to the variable z.
BN_gcd was used to confirm that the gcd of ‘z’ and ‘e’ was, in fact, 1.
BN_mod_inverse was then used to find the modular inverse d = e^-1 mod z, the value of which got assigned to the variable ‘d’.
With this information, we now have the public key {e, n}, and the private key {d, n}.



Task 2: Encrypting a message.

Firstly, we verified the hex value of the message “A top secret!” to be the value provided on the assignment instructions by using the python command given in the assignment.
We then declared new BIGNUM variables to hold the contents of the newly given ‘n’, ‘e’ , ‘m’ and ‘d’ values, and initialized them using hex2bn in order to convert the hex values provided to BIGNUM types.
Next, we used the public key information provided to encrypt the message, which we will call ‘c’, using the formula C=M^e mod n . In order to calculate this with our program, we used the BIGNUM API BN_mod_exp. We use a new variable, ‘c’, to hold the value of this calculation. In BN_mod_exp, the arguments provided are (c, m, e, n, ctx), where ‘m’ is the message, ‘e’ is the exponent e, and ‘n’ is the mod.
The result is calculated and held in the ‘c’ variable. We tested the encryption by using the provided ‘d’ value, and the formula c^d mod n to hopefully get back the original ‘m’ value. If this is the returned value, then the encryption is verified to be correct.
In order to test this in our program, we use a new variable, ‘res’ to hold the value of the calculation, and use BN_mod_exp again, this time with the arguments (res, c, d, n, ctx). The variable ‘c’, was calculated in the previous step, and is used again here as the base to the ‘d’ exponent.
The result of this calculation gives us the same value as the original message, therefore the encryption is verified.



Task 3: Decrypting a message.

In this task, we must decrypt a message,
The first thing we did is declare two variables: c, which holds the new ciphertext as provided in the assignment, and p, which will holds the decrypted message (in hex) after the calculation.
We initialized the value of c with the hex2bn function, then used the same formula of c^d mod n as in Task 2. For this we called BN_mod_exp, with the arguments of ‘p’ to hold the value of the calculation, the ciphertext ‘c’ (in hex form), and our previously known values of ‘d’ and ‘n’, and the ‘ctx’ value.
Once the function is run, we use printBN to show the hex value now held within c.
Using this newly-found hex value, we can run the python decoding method shown in the assignment instructions on the terminal to get the plaintext value of the ciphertext.
The resulting plaintext is “Password is dees”.

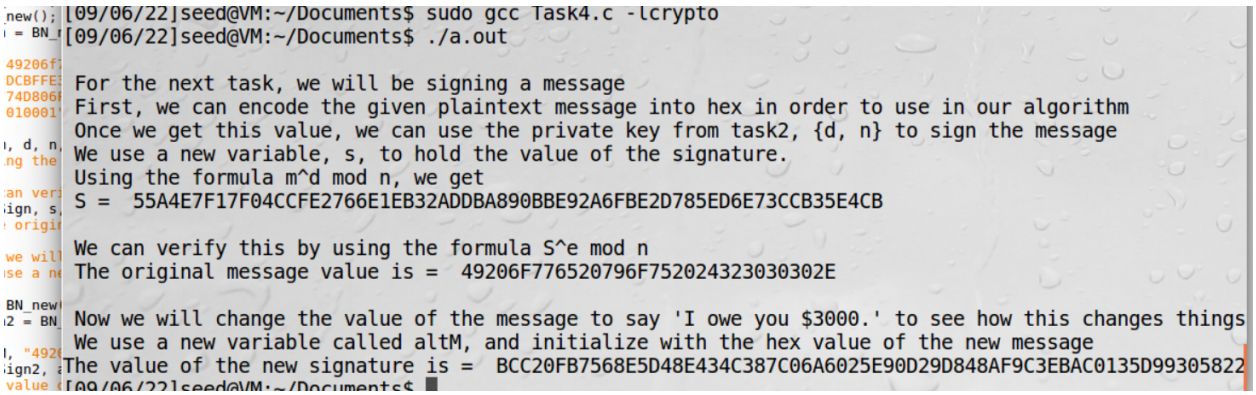
```
[09/06/22]seed@VM:~/Documents$ sudo gcc Task3.c -lcrypto
[09/06/22]seed@VM:~/Documents$ ./a.out

 Next, we will decrypt a message as per Task 3.
 We will use a new variable, called c, which holds the ciphertext, and p, which holds the decrypted m
essage
 Now, I use the same formula as the encryption verification in Task2 to decrypt the message
 The decrypted message =  50617373776F72642069732064656573
[09/06/22]seed@VM:~/Documents$
```

Task 4: Signing a message.

First we can run the given message, M, through the python hex encoder method on our terminal to get a hex value for the plaintext.
This gives us a hex value for the message of 49206f776520796f752024323030302e , which we can now use in our program, along with the private key from Task 2.
We can use the formula S = M^d mod n to get the signature (where S stands for signature).
We can verify this signature by using the formula s^e mod n, and a variable we call verSign, which should give us the same value (in hex) of our original M. The two values match in our program output, and therefore we can conclude that the signature is verified.

We tried changing the value of the original message to , I owe you $3000.
and ran the program using this new value.
The encoded hex number is almost identical, save for a single change of a digit of 3 in a place where a 2 was previously:
For “I owe you $3000” : 49206f776520796f7520243 3 030302e
For “I owe you $2000” : 49206f776520796f7520243 2 030302e
We can use the same formulas as before to get the signature value (S = M^d mod n). We used a new variable called verSign2 to hold the calculated result of the new signature, as well as altM to hold the hex value of the new message. The resulting signature value of BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822 is much different than the original value (for message “I owe you $2000) of 55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB.



Task 5: Verifying a signature.

For this task, Bob must verify a signature given a new message and public key information for Alice.
First, we declared new variables for the hex value of the new message, for the value of the signature, and for the new ‘e’ and ‘n’ values.
We then initialized these values by using hex2bn to convert the hex values to BIGNUM types.
For the signature to be verified, we must calculate M’ by using the formula M’ = S^e mod n.
If M’ is equal to the hex value of our original message (“Launch a missile.”), then the signature is verified.
We ran the formula using BN_mod_exp, with a new variable to hold our result, then printed that result.
The value calculated from the formula, 4C61756E63682061206D697373696C652E , matched the hex value of our original message, and therefore we can say that M’ = M, and so Alice’s signature is verified.
Next, we ran the same formula, but with the signature value ending in 3F instead of 2F. Even though there is only one bit of difference between the alternate S and this one, the resulting calculation of M’ is completely different: 91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294. And so we would be able to see that M’ and M did not match, and so the signature would not be verified.

```
[09/06/22]seed@VM:~/Documents$ sudo gcc Task5.c -lcrypto
[09/06/22]seed@VM:~/Documents$ ./a.out
Now for the final task, we will be  decrypting a message
 We can use the formula s^e mod n , along with the new e, s, and n values given
 The newly calculated M' should equal the hex value of our provided M in order for signature to be ve
rified
 This is the verification =  4C61756E63682061206D697373696C652E

 Therefore, since  the new M' matches the hex value of the provided M, our signature is verified
 This is the alternate verification 91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
```