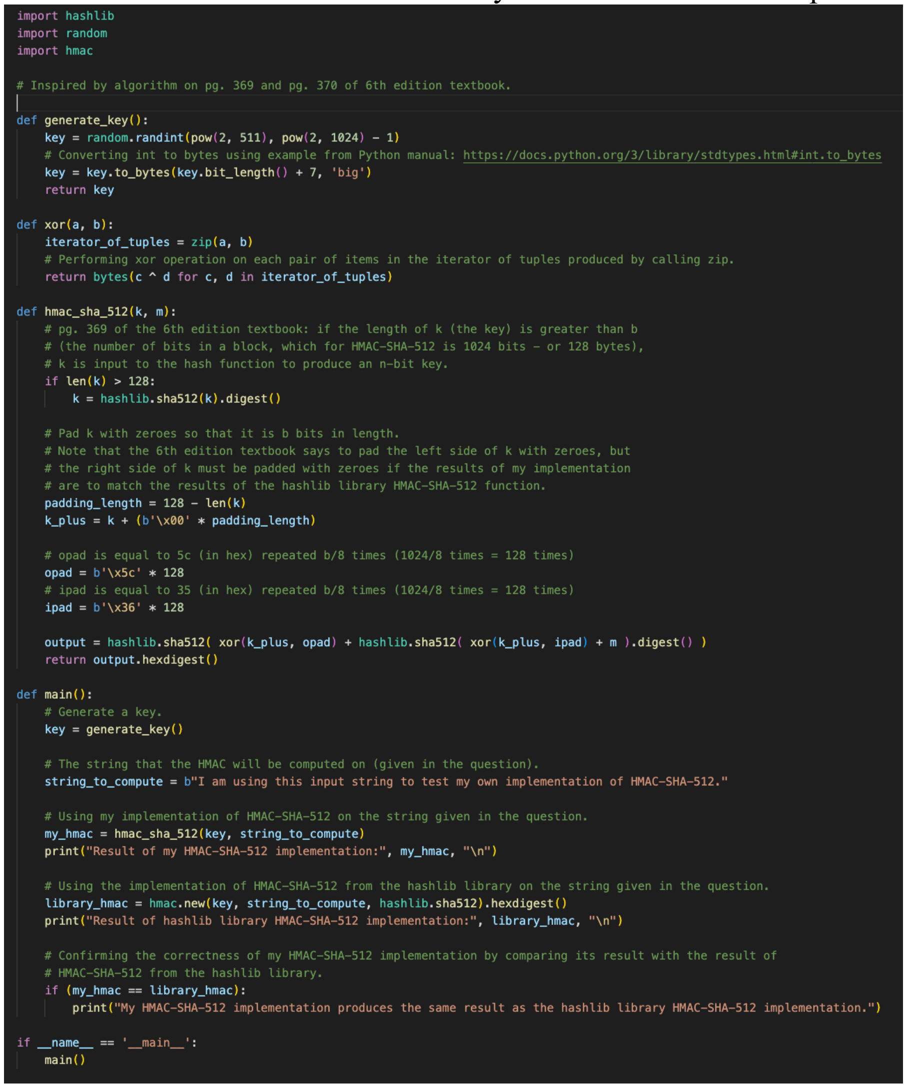
# CSI4108 - Assignment 4
**Nicholas Gin (300107597)**

1. **[2.5 marks]** Implement HMAC-SHA-512. You may use any library or toolkit to call SHA-512, but implement the rest of HMAC yourself. Compute the HMAC of the following string: "I am using this input string to test my own implementation of HMAC-SHA-512." Use any library or toolkit to call HMAC-SHA-512 on this string to confirm that your implementation is correct.

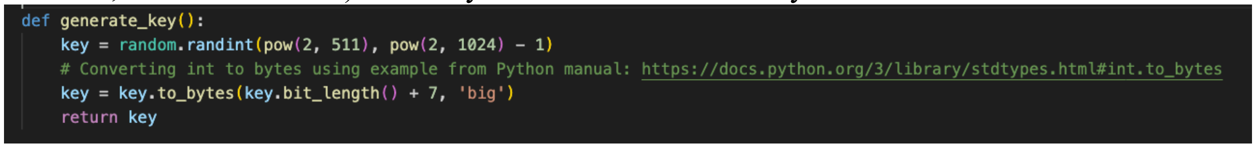
**My code for Question 1 can be found in q1.py.**

I followed the algorithm detailed on pg. 369-370 of the 6th edition textbook to implement HMAC-SHA-512 and compute the HMAC of the string: "I am using this input string to test my own implementation of HMAC-SHA-512." I compared the results of my HMAC-SHA-512 implementation with the results of the hashlib library's HMAC-SHA-512 implementation.



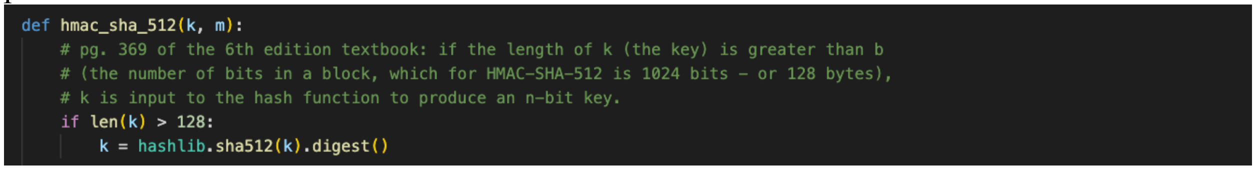
```python
import hashlib
import random
import hmac

# Inspired by algorithm on pg. 369 and pg. 370 of 6th edition textbook.

def generate_key():
    key = random.randint(pow(2, 511), pow(2, 1024) - 1)
    # Converting int to bytes using example from Python manual: https://docs.python.org/3/library/stdtypes.html#int.to_bytes
    key = key.to_bytes(key.bit_length() + 7, 'big')
    return key

def xor(a, b):
    iterator_of_tuples = zip(a, b)
    # Performing xor operation on each pair of items in the iterator of tuples produced by calling zip.
    return bytes(c ^ d for c, d in iterator_of_tuples)

def hmac_sha_512(k, m):
    # pg. 369 of the 6th edition textbook: if the length of k (the key) is greater than b
    # (the number of bits in a block, which for HMAC-SHA-512 is 1024 bits - or 128 bytes),
    # k is input to the hash function to produce an n-bit key.
    if len(k) > 128:
        k = hashlib.sha512(k).digest()

    # Pad k with zeroes so that it is b bits in length.
    # Note that the 6th edition textbook says to pad the left side of k with zeroes, but
    # the right side of k must be padded with zeroes if the results of my implementation
    # are to match the results of the hashlib library HMAC-SHA-512 function.
    padding_length = 128 - len(k)
    k_plus = k + (b'\x00' * padding_length)

    # opad is equal to 5c (in hex) repeated b/8 times (1024/8 times = 128 times)
    opad = b'\x5c' * 128
    # ipad is equal to 35 (in hex) repeated b/8 times (1024/8 times = 128 times)
    ipad = b'\x36' * 128

    output = hashlib.sha512( xor(k_plus, opad) + hashlib.sha512( xor(k_plus, ipad) + m ).digest() )
    return output.hexdigest()

def main():
    # Generate a key.
    key = generate_key()

    # The string that the HMAC will be computed on (given in the question).
    string_to_compute = b"I am using this input string to test my own implementation of HMAC-SHA-512."

    # Using my implementation of HMAC-SHA-512 on the string given in the question.
    my_hmac = hmac_sha_512(key, string_to_compute)
    print("Result of my HMAC-SHA-512 implementation:", my_hmac, "\n")

    # Using the implementation of HMAC-SHA-512 from the hashlib library on the string given in the question.
    library_hmac = hmac.new(key, string_to_compute, hashlib.sha512).hexdigest()
    print("Result of hashlib library HMAC-SHA-512 implementation:", library_hmac, "\n")

    # Confirming the correctness of my HMAC-SHA-512 implementation by comparing its result with the result of
    # HMAC-SHA-512 from the hashlib library.
    if (my_hmac == library_hmac):
        print("My HMAC-SHA-512 implementation produces the same result as the hashlib library HMAC-SHA-512 implementation.")

if __name__ == '__main__':
    main()
```

The generate_key function generates a secret key of a length between 512 bits and 1024 bits long, since it is recommended that the length of the key for HMAC should be greater than or equal to n. Please note that n is the length of the hash code produced by the SHA-512 hash function, which is 512 bits). This key is then converted into bytes.



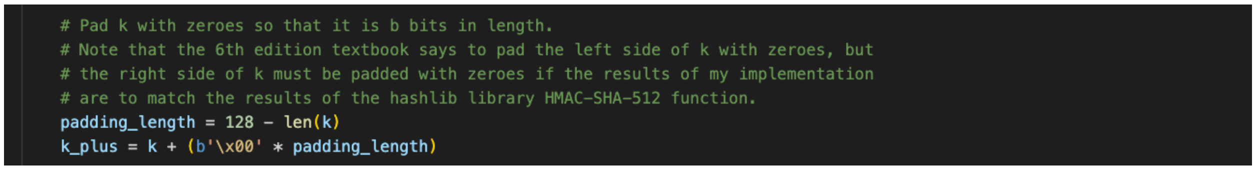
```python
def generate_key():
    key = random.randint(pow(2, 511), pow(2, 1024) - 1)
    # Converting int to bytes using example from Python manual: https://docs.python.org/3/library/stdtypes.html#int.to_bytes
    key = key.to_bytes(key.bit_length() + 7, 'big')
    return key
```

In terms of the actual function to compute HMAC-SHA-512 on a message (the hmac_sha_512 function), this function accepts a key, k, and a message, m, as parameters. I start by checking if the length of k is greater than b (the number of bits in a block, which for HMAC-SHA-512 is 1024 bits - or 128 bytes). If the length of k is greater than b, k is input to the hash function to produce an n-bit key (as indicated on pg. 369 of the 6th edition textbook). Since the key I generated using generate_key() is between 512 bits and 1024 bits long, this hashing will not take place.



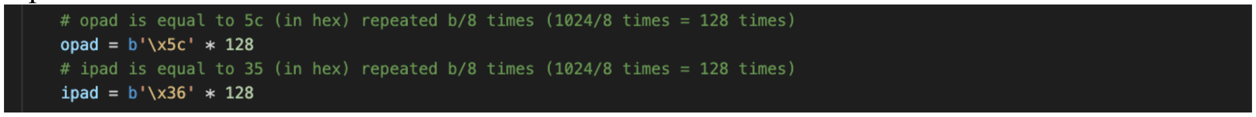
```python
def hmac_sha_512(k, m):
    # pg. 369 of the 6th edition textbook: if the length of k (the key) is greater than b
    # (the number of bits in a block, which for HMAC-SHA-512 is 1024 bits - or 128 bytes),
    # k is input to the hash function to produce an n-bit key.
    if len(k) > 128:
        k = hashlib.sha512(k).digest()
```

I then create k+ (k_plus) by padding k with zeroes so that it is b bits in length. Note that the 6th edition textbook says to pad the left side of k with zeroes, but the right side of k must be padded with zeroes if the results of my implementation are to match the results of the hashlib library HMAC-SHA-512 function.



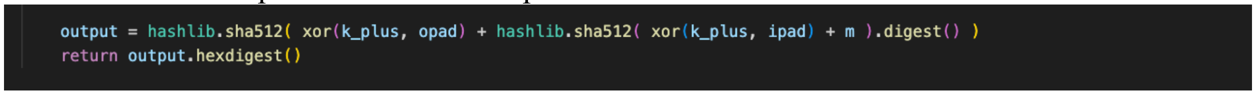
```python
    # Pad k with zeroes so that it is b bits in length.
    # Note that the 6th edition textbook says to pad the left side of k with zeroes, but
    # the right side of k must be padded with zeroes if the results of my implementation
    # are to match the results of the hashlib library HMAC-SHA-512 function.
    padding_length = 128 - len(k)
    k_plus = k + (b'\x00' * padding_length)
```

I set opad to 5c (in hex) repeated b/8 times (1024/8 times = 128 times) and ipad to 36 (in hex) repeated b/8 times.



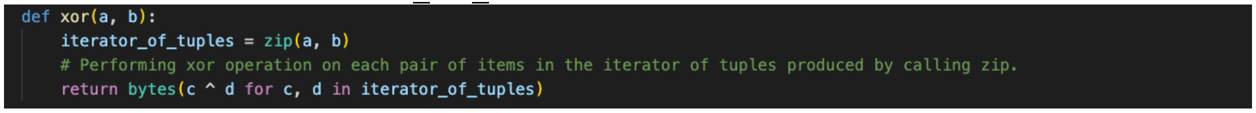
```python
    # opad is equal to 5c (in hex) repeated b/8 times (1024/8 times = 128 times)
    opad = b'\x5c' * 128
    # ipad is equal to 35 (in hex) repeated b/8 times (1024/8 times = 128 times)
    ipad = b'\x36' * 128
```

I then return the output of the HMAC expression.



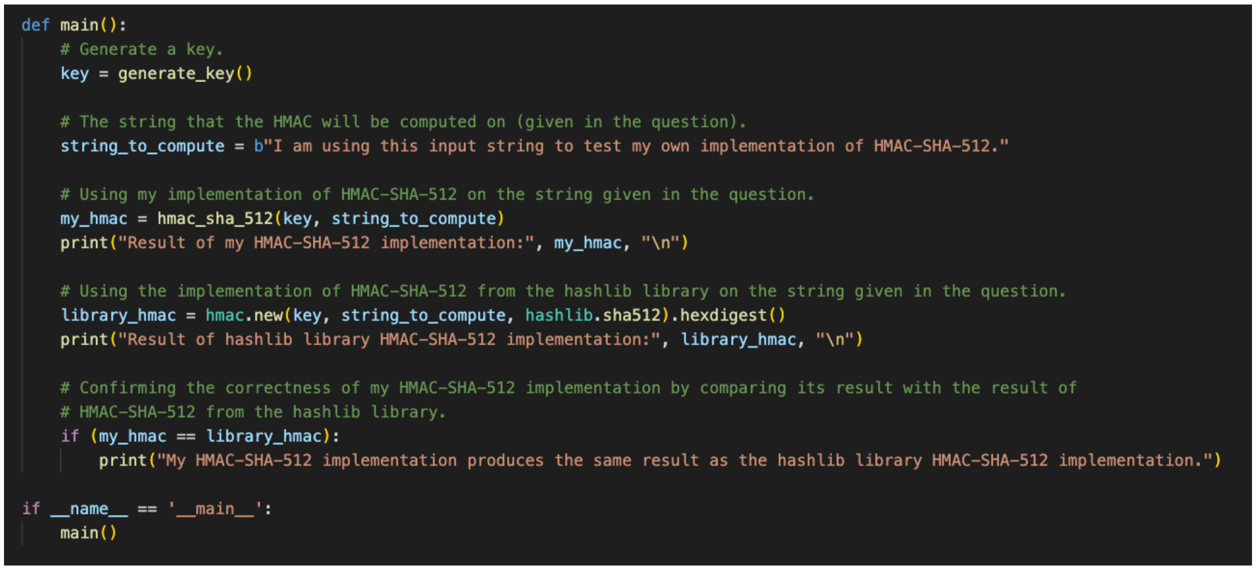
```python
    output = hashlib.sha512( xor(k_plus, opad) + hashlib.sha512( xor(k_plus, ipad) + m ).digest() )
    return output.hexdigest()
```

The xor function called in hmac_sha_512:



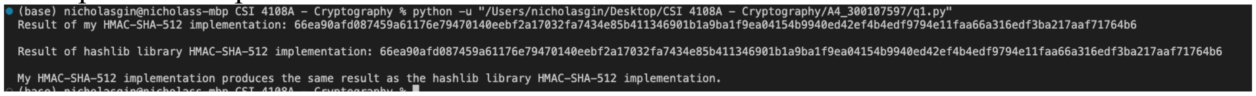
```python
def xor(a, b):
    iterator_of_tuples = zip(a, b)
    # Performing xor operation on each pair of items in the iterator of tuples produced by calling zip.
    return bytes(c ^ d for c, d in iterator_of_tuples)
```

From the main, I call all necessary functions in order. I first generate a key. I then use the key to compute the HMAC-SHA-512 of the message given in the question using two implementations of this algorithm: my implementation and the hashlib library's HMAC-SHA-512 implementation. I compared the results of these implementations; the result of my implementation matches that of hashlib library's implementation, so my implementation should be correct.



```python
def main():
    # Generate a key.
    key = generate_key()

    # The string that the HMAC will be computed on (given in the question).
    string_to_compute = b"I am using this input string to test my own implementation of HMAC-SHA-512."

    # Using my implementation of HMAC-SHA-512 on the string given in the question.
    my_hmac = hmac_sha_512(key, string_to_compute)
    print("Result of my HMAC-SHA-512 implementation:", my_hmac, "\n")

    # Using the implementation of HMAC-SHA-512 from the hashlib library on the string given in the question.
    library_hmac = hmac.new(key, string_to_compute, hashlib.sha512).hexdigest()
    print("Result of hashlib library HMAC-SHA-512 implementation:", library_hmac, "\n")

    # Confirming the correctness of my HMAC-SHA-512 implementation by comparing its result with the result of
    # HMAC-SHA-512 from the hashlib library.
    if (my_hmac == library_hmac):
        print("My HMAC-SHA-512 implementation produces the same result as the hashlib library HMAC-SHA-512 implementation.")

if __name__ == '__main__':
    main()
```

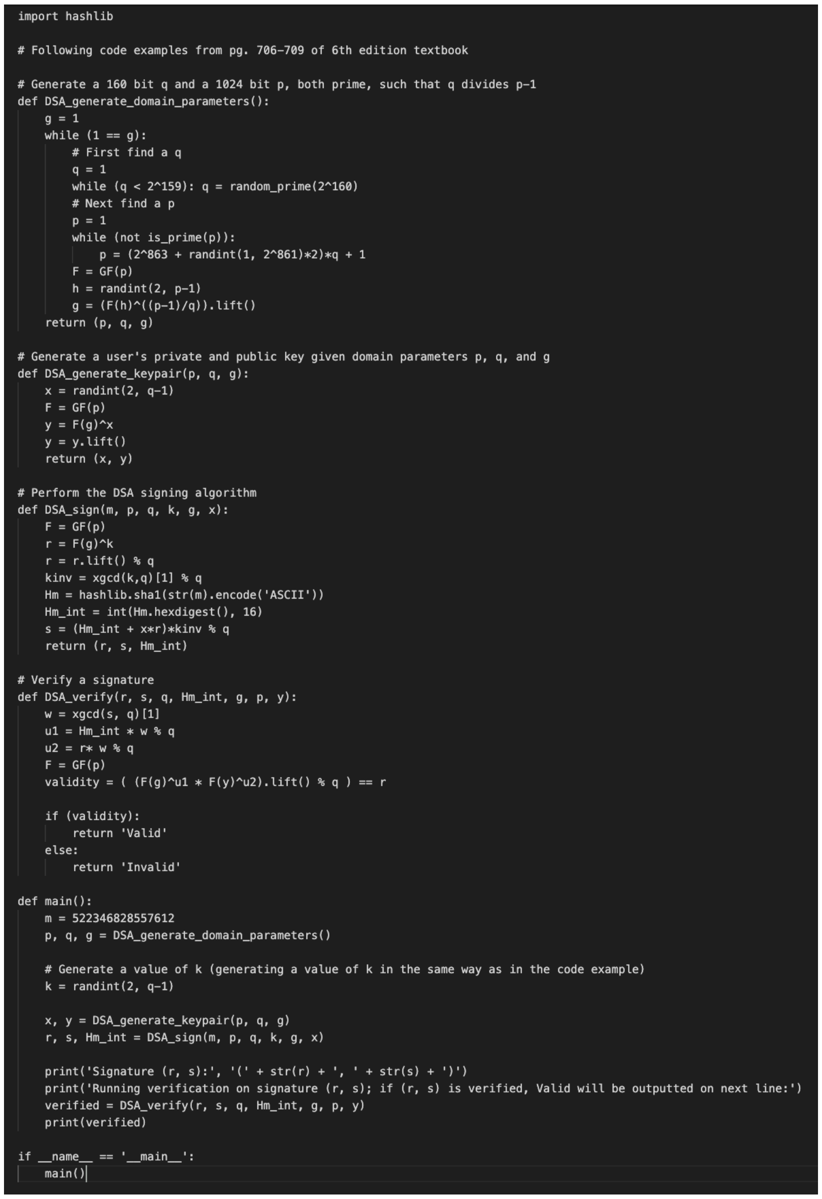
Example code output:



```
(base) nicholasgin@nicholass-mbp CSI 4108A – Cryptography % python -u "/Users/nicholasgin/Desktop/CSI 4108A – Cryptography/A4_300107597/q1.py"
Result of my HMAC-SHA-512 implementation: 66ea90afd087459a61176e79470140eebf2a17032fa7434e85b411346901b1a9ba1f9ea04154b9940ed42ef4b4edf9794e11faa66a316edf3ba217aaf71764b6

Result of hashlib library HMAC-SHA-512 implementation: 66ea90afd087459a61176e79470140eebf2a17032fa7434e85b411346901b1a9ba1f9ea04154b9940ed42ef4b4edf9794e11faa66a316edf3ba217aaf71764b6

My HMAC-SHA-512 implementation produces the same result as the hashlib library HMAC-SHA-512 implementation.
(base) nicholasgin@nicholass-mbp CSI 4108A – Cryptography %
```

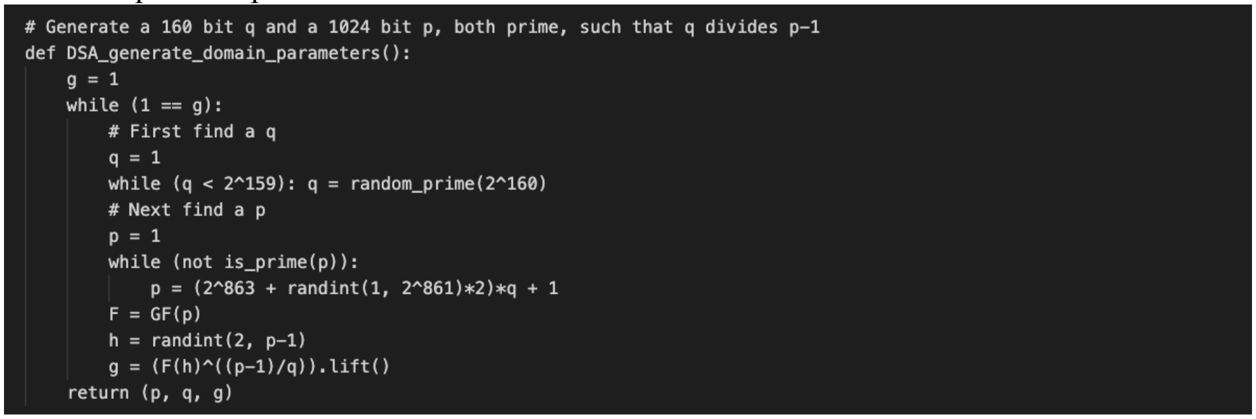
2. **[2.5 marks]** Implement DSA. Using a 1024-bit prime $p$ and an appropriate 160-bit prime $q$ with generator $g$ of the $q$-order subgroup of $Z_p$*, choose a signature key pair $(x, y)$, an appropriate value $k$, and the hash function SHA-1. You may use any library or toolkit to find $p$ and $q$ and to call SHA-1, but implement the rest of DSA yourself. Sign the message $m = 522346828557612$ using the privacy key $x$. Verify the signature using the public key $y$.

**My code for Question 2 can be found in q2.sage.** I used the Sagemath toolkit to answer this question; this is a toolkit recommended and used in the 6th edition textbook.
To implement DSA using a 1024-bit prime p and an appropriate 160-bit prime q with generator g of the q-order subgroup of $Z_p$* and the hash function SHA-1, I followed the code examples from pg. 706-709 of the 6th edition textbook.



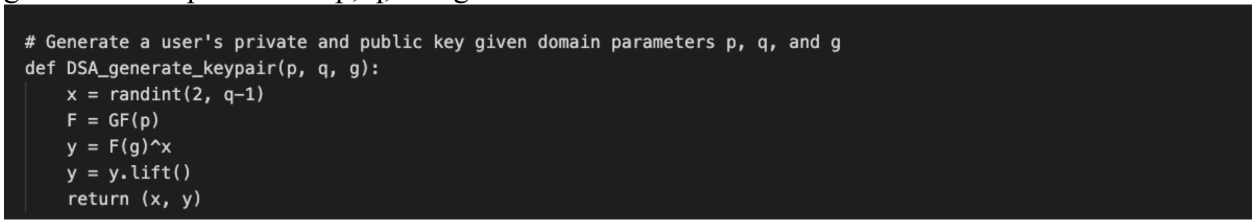
```
import hashlib

# Following code examples from pg. 706-709 of 6th edition textbook

# Generate a 160 bit q and a 1024 bit p, both prime, such that q divides p-1
def DSA_generate_domain_parameters():
    g = 1
    while (1 == g):
        # First find a q
        q = 1
        while (q < 2^159): q = random_prime(2^160)
        # Next find a p
        p = 1
        while (not is_prime(p)):
            p = (2^863 + randint(1, 2^861)*2)*q + 1
        F = GF(p)
        h = randint(2, p-1)
        g = (F(h)^((p-1)/q)).lift()
    return (p, q, g)

# Generate a user's private and public key given domain parameters p, q, and g
def DSA_generate_keypair(p, q, g):
    x = randint(2, q-1)
    F = GF(p)
    y = F(g)^x
    y = y.lift()
    return (x, y)

# Perform the DSA signing algorithm
def DSA_sign(m, p, q, k, g, x):
    F = GF(p)
    r = F(g)^k
    r = r.lift() % q
    kinv = xgcd(k,q)[1] % q
    Hm = hashlib.sha1(str(m).encode('ASCII'))
    Hm_int = int(Hm.hexdigest(), 16)
    s = (Hm_int + x*r)*kinv % q
    return (r, s, Hm_int)

# Verify a signature
def DSA_verify(r, s, q, Hm_int, g, p, y):
    w = xgcd(s, q)[1]
    u1 = Hm_int * w % q
    u2 = r* w % q
    F = GF(p)
    validity = ( (F(g)^u1 * F(y)^u2).lift() % q ) == r

    if (validity):
        return 'Valid'
    else:
        return 'Invalid'

def main():
    m = 522346828557612
    p, q, g = DSA_generate_domain_parameters()

    # Generate a value of k (generating a value of k in the same way as in the code example)
    k = randint(2, q-1)

    x, y = DSA_generate_keypair(p, q, g)
    r, s, Hm_int = DSA_sign(m, p, q, k, g, x)

    print('Signature (r, s):', '(' + str(r) + ', ' + str(s) + ')')
    print('Running verification on signature (r, s); if (r, s) is verified, Valid will be outputted on next line:')
    verified = DSA_verify(r, s, q, Hm_int, g, p, y)
    print(verified)

if __name__ == '__main__':
    main()
```

Following the code examples, I generate the domain parameters for DSA using the DSA_generate_domain_parameters function; I generate a 160 bit q and a 1024 bit p, both prime, such that q divides p-1.



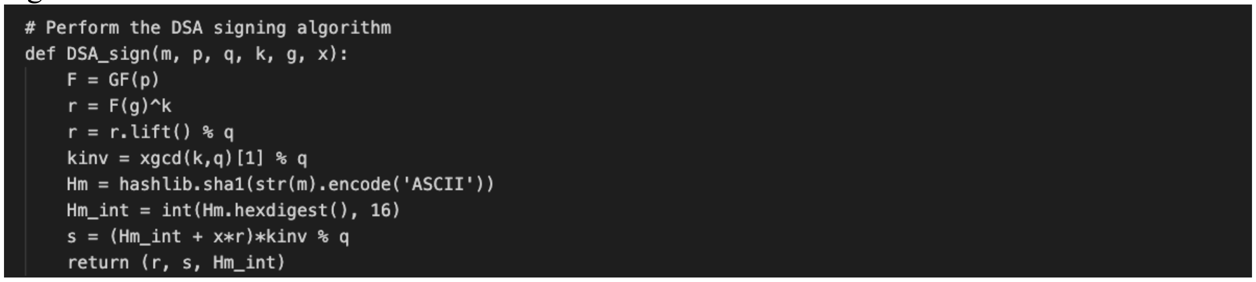
```
# Generate a 160 bit q and a 1024 bit p, both prime, such that q divides p-1
def DSA_generate_domain_parameters():
    g = 1
    while (1 == g):
        # First find a q
        q = 1
        while (q < 2^159): q = random_prime(2^160)
        # Next find a p
        p = 1
        while (not is_prime(p)):
            p = (2^863 + randint(1, 2^861)*2)*q + 1
        F = GF(p)
        h = randint(2, p-1)
        g = (F(h)^((p-1)/q)).lift()
    return (p, q, g)
```

In the DSA_generate_keypair function, I generate a user's private key (x) and public key (y), given domain parameters p, q, and g.



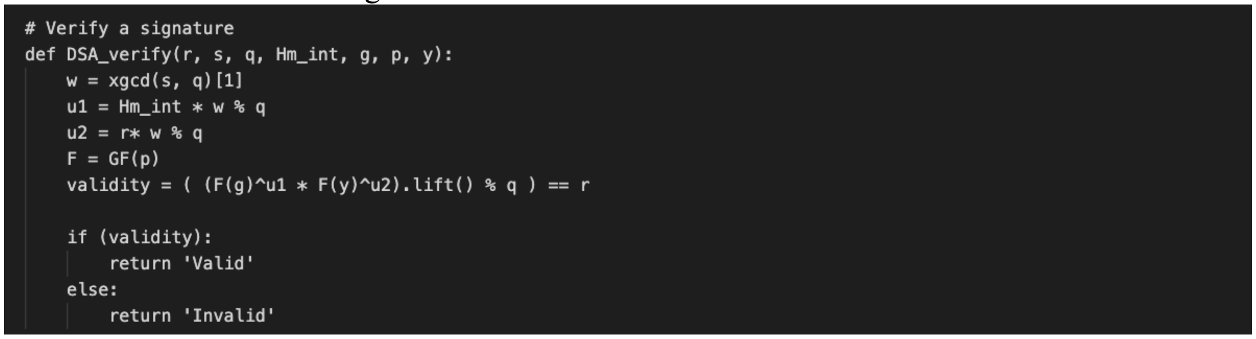
```
# Generate a user's private and public key given domain parameters p, q, and g
def DSA_generate_keypair(p, q, g):
    x = randint(2, q-1)
    F = GF(p)
    y = F(g)^x
    y = y.lift()
    return (x, y)
```

I use the builtin SHA-1 hash function from the hashlib library and perform the DSA signing algorithm.



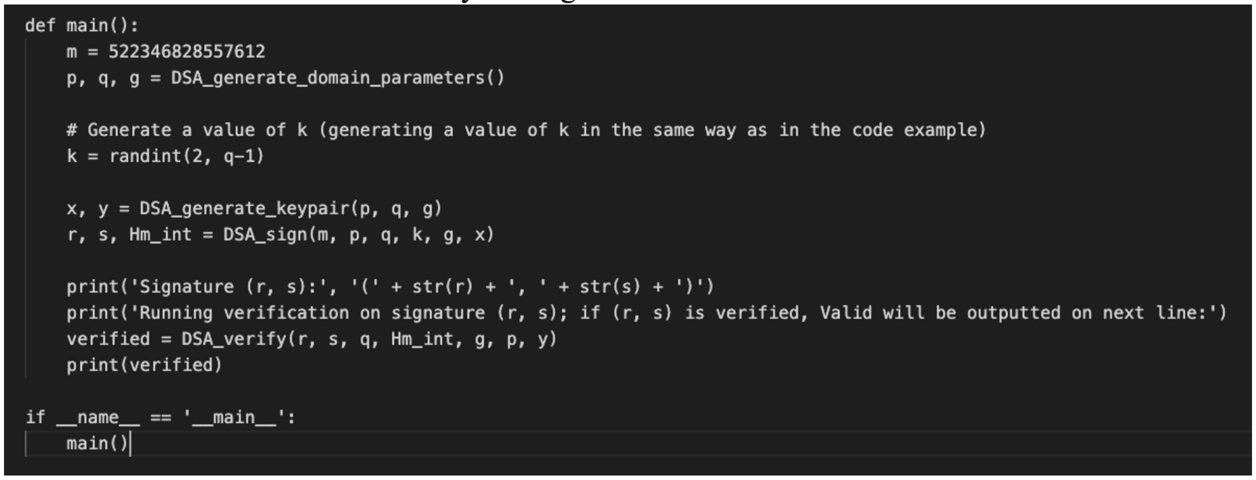
```
# Perform the DSA signing algorithm
def DSA_sign(m, p, q, k, g, x):
    F = GF(p)
    r = F(g)^k
    r = r.lift() % q
    kinv = xgcd(k,q)[1] % q
    Hm = hashlib.sha1(str(m).encode('ASCII'))
    Hm_int = int(Hm.hexdigest(), 16)
    s = (Hm_int + x*r)*kinv % q
    return (r, s, Hm_int)
```

I verify the signature produced by the DSA_sign function in the DSA_verify function. If the signature is valid, this function will return the string 'Valid,' and if the signature is invalid, this function will return the string 'Invalid.'



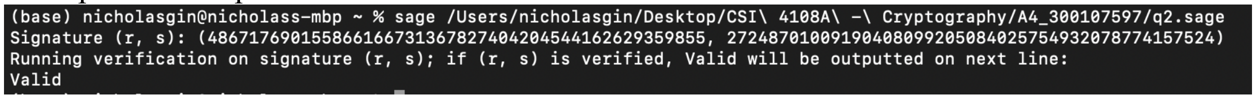
```
# Verify a signature
def DSA_verify(r, s, q, Hm_int, g, p, y):
    w = xgcd(s, q)[1]
    u1 = Hm_int * w % q
    u2 = r* w % q
    F = GF(p)
    validity = ( (F(g)^u1 * F(y)^u2).lift() % q ) == r

    if (validity):
        return 'Valid'
    else:
        return 'Invalid'
```

From the main, I call all necessary functions in order to produce a signature (r, s) on the message m = 522346828557612. I then verify this signature.



```
def main():
    m = 522346828557612
    p, q, g = DSA_generate_domain_parameters()

    # Generate a value of k (generating a value of k in the same way as in the code example)
    k = randint(2, q-1)

    x, y = DSA_generate_keypair(p, q, g)
    r, s, Hm_int = DSA_sign(m, p, q, k, g, x)

    print('Signature (r, s):', '(' + str(r) + ', ' + str(s) + ')')
    print('Running verification on signature (r, s); if (r, s) is verified, Valid will be outputted on next line:')
    verified = DSA_verify(r, s, q, Hm_int, g, p, y)
    print(verified)

if __name__ == '__main__':
    main()
```

Example code output:



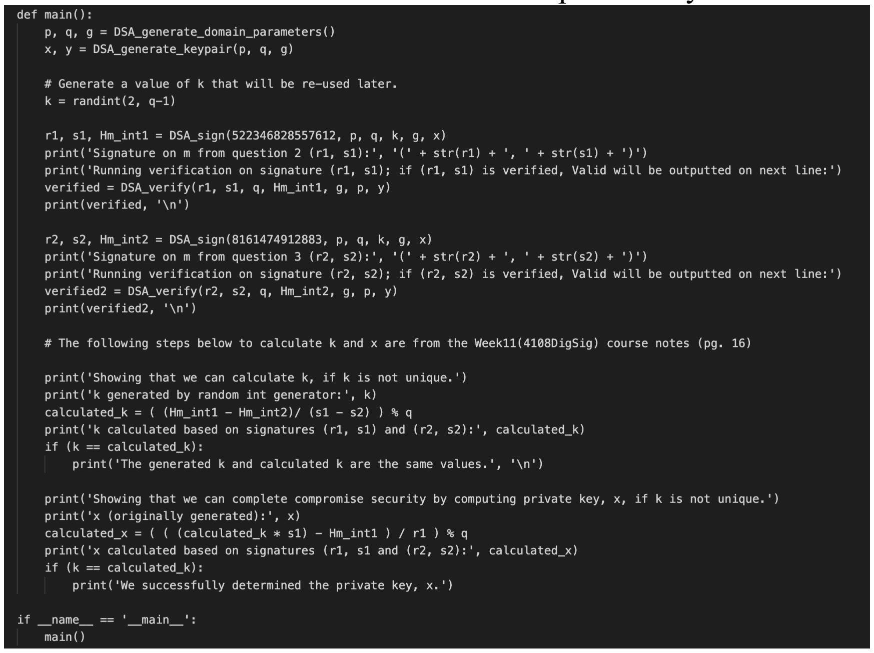
```
(base) nicholasgin@nicholass-mbp ~ % sage /Users/nicholasgin/Desktop/CSI\ 4108A\ -\ Cryptography/A4_300107597/q2.sage
Signature (r, s): (486717690155866166731367827404204544162629359855, 27248701009190408099205084025754932078774157524)
Running verification on signature (r, s); if (r, s) is verified, Valid will be outputted on next line:
Valid
```

3. **[1.5 marks]** With your implementation from question #2, sign the message *m* = 8161474912883 using the same value of *k*. Show that an observer of the two signatures will be able to completely compromise security.

**My code for Question 3 can be found in q3.sage.** I used the Sagemath toolkit to answer this question; this is a toolkit recommended and used in the 6th edition textbook. For Question 3, I re-used my exact implementation of the DSA signing and verifying process
from Question 2 (with the exception of the main function).

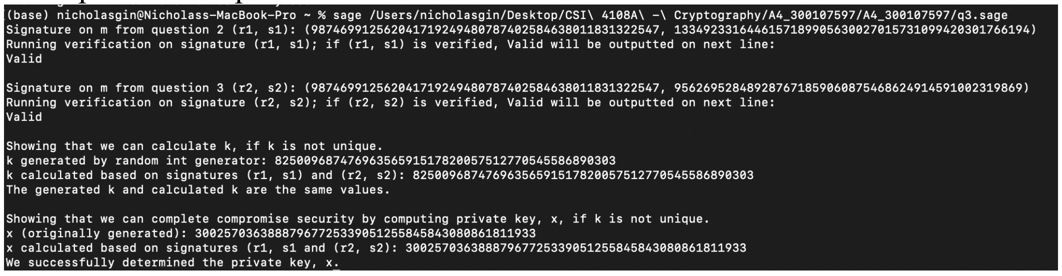
In the main, I generate a value of k and use this value of k when signing both m = 522346828557612 and m = 8161474912883.

To demonstrate that an observer of the two signatures will be able to completely compromise security, I followed the steps from the Week11(4108DigSig) course notes (pg. 16) to calculate the value of k from two signatures that used the same k values. I can then use this value of k to calculate the private key (the value of x); with the private key, an attacker could commit a forgery, completely compromising security. I check to see if the k and the private key I calculated are indeed the same k value and private key that were used to sign the messages.



```
def main():
    p, q, g = DSA_generate_domain_parameters()
    x, y = DSA_generate_keypair(p, q, g)

    # Generate a value of k that will be re-used later.
    k = randint(2, q-1)

    r1, s1, Hm_int1 = DSA_sign(522346828557612, p, q, k, g, x)
    print('Signature on m from question 2 (r1, s1):', '(' + str(r1) + ', ' + str(s1) + ')')
    print('Running verification on signature (r1, s1); if (r1, s1) is verified, Valid will be outputted on next line:')
    verified = DSA_verify(r1, s1, q, Hm_int1, g, p, y)
    print(verified, '\n')

    r2, s2, Hm_int2 = DSA_sign(8161474912883, p, q, k, g, x)
    print('Signature on m from question 3 (r2, s2):', '(' + str(r2) + ', ' + str(s2) + ')')
    print('Running verification on signature (r2, s2); if (r2, s2) is verified, Valid will be outputted on next line:')
    verified2 = DSA_verify(r2, s2, q, Hm_int2, g, p, y)
    print(verified2, '\n')

    # The following steps below to calculate k and x are from the Week11(4108DigSig) course notes (pg. 16)

    print('Showing that we can calculate k, if k is not unique.')
    print('k generated by random int generator:', k)
    calculated_k = ( (Hm_int1 - Hm_int2)/ (s1 - s2) ) % q
    print('k calculated based on signatures (r1, s1) and (r2, s2):', calculated_k)
    if (k == calculated_k):
        print('The generated k and calculated k are the same values.', '\n')

    print('Showing that we can complete compromise security by computing private key, x, if k is not unique.')
    print('x (originally generated):', x)
    calculated_x = ( ( (calculated_k * s1) - Hm_int1 ) / r1 ) % q
    print('x calculated based on signatures (r1, s1 and (r2, s2):', calculated_x)
    if (k == calculated_k):
        print('We successfully determined the private key, x.')

if __name__ == '__main__':
    main()
```

Example code output:



```
(base) nicholasgin@Nicholass-MacBook-Pro ~ % sage /Users/nicholasgin/Desktop/CSI\ 4108A\ -\ Cryptography/A4_300107597/A4_300107597/q3.sage
Signature on m from question 2 (r1, s1): (98746991256204171924948078740258463801183132 2547, 13349233164461571899056300270157310994203017661 94)
Running verification on signature (r1, s1); if (r1, s1) is verified, Valid will be outputted on next line:
Valid

Signature on m from question 3 (r2, s2): (987469912562041719249480787402584638011831322547, 956269528489287671859060875468624914591002319869)
Running verification on signature (r2, s2); if (r2, s2) is verified, Valid will be outputted on next line:
Valid

Showing that we can calculate k, if k is not unique.
k generated by random int generator: 825009687476963565915178200575127705455868903 03
k calculated based on signatures (r1, s1) and (r2, s2): 82500968747696356591517820057512770545586890303
The generated k and calculated k are the same values.

Showing that we can complete compromise security by computing private key, x, if k is not unique.
x (originally generated): 300257036388879677253390512558458430808618119 33
x calculated based on signatures (r1, s1 and (r2, s2): 30025703638887967725339051255845843080861811933
We successfully determined the private key, x.
```

4. **[1 mark]** It is possible to use a hash function to construct a block cipher with a structure similar to DES. Since a hash function is one-way and a block cipher must be reversible (in order to decrypt), how is it possible? Explain in your own words.

The structure of DES is built upon the structure of a Feistel network. In a Feistel network, there are round functions, which are used during the encryption and decryption processes. Regardless of if we are encrypting or decrypting, the round functions can remain the same (we do not invert the round function). If you are encrypting, you move through the network from its top to its bottom, and if you are decrypting, you move through the network from its bottom to its top; but, you do not invert or change the round functions themselves. In this way, round functions are one-way functions. Hash functions are also one-way functions, as indicated in the question. So, although a hash function is one-way and a block cipher must be reversible (in order to decrypt), it would be possible to use a hash function to construct a block cipher with a structure similar to DES. What we would have to do is construct the block cipher such that it follows the Feistel network structure, but in place of the round functions, we implement the hash function. Since round and hash functions are both one-way, structuring the cipher in this manner would be possible.