CMPSC 360 Discrete Mathematics for Computer Science
Fall 2024 Mahfuza Farooque

Extra Credit

1 RSA Implementation

In this assignment, you will implement the RSA algorithm in Python. You will learn how modular arithmetic can be leveraged to enforce security in a PKI system. Unlike the shift cipher you've encountered in class, RSA employs an asymmetric key encryption technique. This means that the encryption key used to secure a message is different from the decryption key, which is kept confidential and known only to the intended recipient. The computational hardness of this algorithm is prime factorization. To decrypt the messages, an adversary needs to recover the prime numbers p and q from the common modulus $n = p * q$. While this is far more secure than a symmetric key encryption technique like the shift cipher, there are many known vulnerabilities in RSA. However, they are beyond the scope of this course. You can learn more about the algorithm here.

In class, you learned how to encrypt integers using modular exponentiation, however, in this programming assignment, you will be encrypting strings. For example, in the string "hello world!", we need to encrypt the space and special character "!" in addition to the characters. Specifically,

- Your encryption/decryption should support all characters corresponding to ASCII values in the range **32-128 (inclusive)**.
- Your encryption/decryption should support "chunking" i.e., instead of encrypting/decrypting each character of a string using RSA, you will split the string into chunks (substrings) and then encrypt/decrypt those chunks. For example, with a chunk size of 2, the string "hello" is split as "he", "ll", " o". Observe that we apply the necessary padding to generate chunks of equal size. Without chunking, an individual character would be mapped to a unique value modulo n (for a given public modulus n). You need to map "chunks" to unique values according to the following strategy:
  Treat each chunk as a base-n number and convert it to decimal.
  If your chunk has k characters, a chunk could have a total of n^k values (more secure!). Based on this strategy, what would you do after decrypting the chunks in order to recover the plaintext message? You need to figure out the answer to this question to complete the decryption function.

1.1 Function Descriptions

You need to implement the following functions:

- *generate_prime*: This function accepts a value n and generates a random $n-$bit prime number. We will test your function and verify that it can generate large prime numbers (e.g., 1024-bit prime numbers) within a reasonable amount of time. If you try to brute force the prime number generation, you will not receive points for this. Consider using a primality test to implement the prime number generation process (e.g., Fermat's primality test).

- *generate_keypair*: This function accepts two distinct prime numbers and generates the public and private key pairs. The format **MUST** be of the form $((n,e),(n,d))$ where n is the common modulus, e is the public exponent, and d is the private exponent. If the input integers p and q are invalid, you **MUST** raise a Value Error exception. Please refer to the class slides and textbook about how to compute these values.
- *chunk_to_num*: This function maps a message chunk (substring) to a unique value based on the base conversion strategy from section 1. The only parameter is the chunk (substring).
- *num_to_chunk*: Maps a number back to a chunk (substring) using the base conversion strategy from section 1. You will need this to implement the RSA decryption process. The parameters are the number and the chunk size.
- *rsa_encrypt*: The RSA encryption function that encrypts a message using a public key and specified chunk size.
- *rsa_decrypt*: The RSA decryption function that decrypts a message using a private key and specified chunk size.

You will need to write helper functions to complete the functions provided in the starter code. Note that the first two functions are fairly straightforward in the sense that they directly operate on integers, but you'll need to implement the code to break strings according to the provided chunking scheme. Here are the functionalities that you will need to implement as helper functions:

- *Modular Exponentiation*: Recall that encryption and decryption are implemented by computing modular exponentiation problems (see lecture slides for more details). You should implement your own modular exponentiation computation that operates on integers.
- *Modular Inverse*: You need to implement a helper function (or incorporate this code in the starter code) to find the private key
- *is_prime function*: This can optionally be implemented as a helper function or something you directly write in the generate_prime function as part of a primality test.
- *gcd*: You will need to write your own gcd function given two numbers. (*Hint*: Which part of the RSA encryption process needs this?)

Note: When choosing the value of your public exponent e, please choose the smallest value of $e > 2$ for a given k. If you do not follow this selection policy, you will fail the test cases and lose points.

1.2 Libraries

You are not allowed to use any external libraries including RSA, cryptography, or similar options. You are explicitly forbidden from using math.gcd() for computing the gcd, pow/math.pow for modular exponentiation (using pow for computing normal n^{th} powers for things other than modular exponentiation is fine), and pow/math.pow for modular inverse, and any external prime number library (e.g., gmpy2, primepy, etc.) for prime number generation. You cannot use built-in base conversion functions (or from external libraries) for the chunking algorithm. You are allowed to use the random.randint() function if you want to generate a random integer. If you are unsure about whether you can use a built-in function, please raise a question on Campuswire. Failure to follow the guidelines will result in heavy point deductions.

1.3 Testing

Your code will be tested on several test cases, many of which will be hidden test cases. You need to thoroughly test your code before submitting the assignment. Please make sure to perform input validation for **ALL** functions. We will randomly test individual functions for input validation. For example, the *generate_keypair* function cannot accept composite values for p and q, so you need to validate the input and throw a value error if necessary.

It is a good idea to validate your code by encrypting some text and then decrypting the ciphertext you obtained. You should get the same string that you started with.

2 Grading

The maximum score of the assignment is 100 points. You will be graded based on the number of test cases your code passes. As mentioned earlier, some test cases will be visible, but passing them doesn't guarantee a full score since there are hidden test cases as well. The point distribution is as follows:

1. *generate_prime* - 15 points
2. *generate_keypair* - 15 points
3. *chunk_to_num* - 10 points
4. *num_to_chunk* - 10 points
5. *rsa_encrypt* - 25 points
6. *rsa_decrypt* - 25 points

3 Submission

Download the starter code (please download the updated starter code.) from Canvas and submit the rsa.py file on gradescope. **Do not modify the file name or function names in your submission**. The due date for this assignment is December 3, 2024, 11:59 P.M.