\*\*\*CODE HAS BEEN MODIFIED FROM MY ORIGINAL IMPLEMENTATION TO FUNCTION IN JUPYTER HUB\*\*\*

Please see the full implementation on my public GitHub repository at: <https://github.com/tylerteichmann/RSA_Coding_Project>

##### Custom Code Feature: Decode Piazza

Tired of calling multiple functions and adjusting variables, I decided that for the custom code feature I would develop an all-in-one application to implement all the project’s functions. I started by developing a simple command-line-based application with flexible options, but as I proceeded with the project, I noticed that there was a better solution, decode the entire Piazza page. Accomplishing this was difficult. Since the page dynamically generates content, a traditional download and find replace wouldn’t work. Instead, I needed to dive into some unfamiliar techniques to complete my goal. I began by implementing some optimizations within the RSA functions to create a seamless code breaking function and improve factorization time. Next, I researched and leveraged the Selenium framework to download the Piazza webpage for scraping. Finally, I used the Beautiful Soup framework to find and replace the encrypted posts with the decrypted messages. The final product is not a perfect implementation, there is minimal Error checking, process validation, and modularity. However, with limited time and learning two new frameworks I feel accomplished with the product I produced.

Before beginning the difficult portion of the feature, I began by optimizing a few of the required functions. The first of these optimizations was with the fast modular exponentiation function where I replaced the division operation with bit shifting and used bitwise comparisons to determine parity. The next optimization I made was to the factorize function. As demonstrated in the code breaking section of my project, this version uses the square root of $n$ and works backwards, skipping even numbers, to bet that the smaller prime factor is a large number. With this optimization, the code succeeds in finding a factor for an n of 20 digits in a few seconds. Reflecting on this portion, if I were to further refine my code, I would look into an implementing the Pollard Rho algorithm to increase factoring speed even further. Finally, I reworked the break code function into a break key function that extracts only the public key and raises exceptions for invalid public keys or keys that failed to factor. In doing so, the program can break the public key of a post and reuse it multiple times to decode the replies without refactoring.

With the RSA functions optimized and ready to go, the main challenge in creating this feature was downloading and editing the source HTML. Attempting to download the page source from a browser resulted in an HTML document filled with links to JavaScript files. Furthermore, since the web page is constantly updated, I wanted to download a fresh version each time the program ran. I attempted at first to use the python built in request module to send a GET request to the host and write the response to a file. However each time I attempted to do that the host redirected the program request to the login screen, and it could not access the post. At first, I thought that sending my request with login credentials would avoid this. To test my theory, I used the Postman application to try different request headers to achieve desired result. Eventually I noticed that the request header during a normal session contained five cookies. I copied those cookies into the current version of my python program and successfully received the logged-in session. However, it was not a complete success. Since the request module downloaded the html response, the rendering of the page only said, ‘loading Piazza’. Now, I needed a way to force the session to wait until after the page rendered completely to download html. Enter the Selenium framework. Combing through the documentation, I adjusted my program to open a chromium browser with no window to the Piazza homepage. Once there, all required cookies were set, and a request to the desired post URL was sent to the host. At last, the full Piazza webpage was downloaded with all content included in my user’s session. Unfortunately, I did not implement any error checking at the current time so all verification still needs to be done visually once the page is downloaded. Furthermore, since Selenium uses browser automation, future improvements of the program could prompt the user to log into the webpage using their own credentials without using session cookies that refresh and give access to my personal account. (No sensitive information is currently on my Piazza account!)

Now that the webpage has been successfully loaded into python, I needed to parse the HTML and replace the encrypted messages with decoded versions. I started by using JavaScript in the browser console to identify which selectors would return information for each thread and its replies. Using those selectors, I then leveraged the Beautiful Soup framework to extract the text that contained public keys and the messages written by other students. Because the content varies in formatting, some posts contained lots of divs and generated HTML that isn’t easy to parse. To get around this, I used regular expressions that Isolated elements with strings that contained public keys as well as encrypted messages. Next, I developed two functions to extract the keys and messages from a variety of different strings within those tags. The first function extracts public keys by finding the first numeric character and continuing to concatenate characters to a temporary string until it encounters a non-numeric character. Then it converts the temporary string to an integer while appending it to a public key list. It repeats this process until it two integers are appended to the public key. However, there is potential for errors if user’s posts are not properly formatted, and if the public key is not the first two integers within that element. The second function extracts the encoded messages. Similar to the extraction process for public keys it uses regular expressions to isolate the element, the iterates over the string to find an open bracket and reads all characters until the next closed bracket. This function is also prone to formatting errors. Once the public key and messages are extracted. The program runs the break key function from earlier to find the private key and then decodes that message along with subsequent replies. After decoding each message, the program replaces the entire element with the plaintext version, since each message is not contained within their own tag. This is fine for replies where the whole element is the message, but content, like the public key, is lost for the original threads. Future improvements could append this data or do more sophisticated replacements to avoid loss.

To develop this feature, I needed to learn a lot of new tools in the Selenium and Beautiful Soup frameworks. There was a great deal of time spent reading their documentation along with many google searches to gain full understanding. There are still a lot of improvements that can be made to areas of portability, error checking, and verboseness, and in the future could contain web based, login, and multiple post features. However, ultimately it was a rewarding experience that I was unsure was possible, but I kept trying and eventually accomplished what I set out to do.

The first block of code contains the dependent functions from the initial RSA implementation. This includes the Decode, Fast Modular Exponentiation, Extended Euclidean Algorithm, Find Private Key, and Convert Number functions. The only changes made were to the Fast Modular Exponentiation function and the Convert Number function.

The next block of code contains the factorize and break key functions. I previously demonstrated these functions in the code breaking section. The only modification made is to the break key function that is a version of the break code function to reduce overhead when decoding replies additionally it checks that factorize returns a factor of n. If not, the function raises an exception handled by the caller.

The following block of code begins new implementations. This block contains the function that requests the web page and downloads the HTML. Leveraging Selenium, this function takes a URL as input, loads required cookies, requests the webpage, and downloads the html data for use by the scrape data function. In the original implementation, the cookies are loaded from a text file for ease of use. However, to keep it contained in the Jupyter Notebook, they have been hard coded as a list.

The next block contains the scrape public key function. Used in the scrape data function later on, this function takes a string that contains a public key and returns that key as an integer list of length two. The overall execution of the function searches a string for an numeric character, then concatenates that character as an integer to a value string until the run of numeric characters ends. The function does this twice to the first to runs of numeric characters found and raises an exception if a full public key was not extracted. Usage requires two integer runs separated by a non-numeric character with those integers being first in the string.

To compliment the previous code, the below function completes the same requirement but for the message. It iterates over a string to find the cypher message located between brackets and returns that message as an integer list. For cases where there is text within brackets that contain values that cannot convert to integers, the function returns the original message as a string to be handled later. Input string requirements include comma separated values with one set of brackets. There is room for improvement later on to allow for broader string formats.

The final function for HTML parsing is the scrape data function. This function takes an HTML page as input and iterates over all posts and replies to that post identifying the public key and decoding all messages in that thread with the respective public key. The function saves the HTML in a new file and returns nothing. The current form of the code leverages regular expressions to separate public keys from messages to improve and format the input strings for the previous two functions. One major difficulty with implementing this part of the code was selecting the correct text. Since the majority of the page is generated through scripts, tags and formats used to the content varies between posts. Additionally, posts that were not formatted correctly required the regular expressions to be more complicated to allow for more flexible key and message identification.

Finally, the main function executes the two support functions to download the html file, decode all encrypted posts, and then save the decoded webpage.

8 - Originally, I only planned to create an application that would package the project into a usable command line interface application. However, as I continued on, I got tired of constantly decoding messages on the Piazza page, so I decided to try and create an application that would replace the cypher text   
in the html with the decoded messages for each post. In doing so, I had to teach myself web scraping techniques with two frameworks I have never used before. I feel I really pushed out of my comfort level with this and made something I am proud of.