

tyte8393

November 5, 2024

0.1 RSA - The Project

Name: Tyler Teichmann

Moodle Email: tyte8393@colorado.edu

Name your file “cuidentikey”.ipynb This means YOUR identikey.

Do NOT delete instructions or pre-existing comments in the code. Removal will trigger significant point reductions

Required: Add your own comments as if the instructions and comments are not visible - this allows you to customize your project after the course

The Prompt:

Your boss has been long intrigued by the RSA algorithm and has asked you to create a tutorial RSA walk through and exploration project to help the boss and others in the company to understand the basic structures and mathematics of an RSA implementation.

Your project will:

1. Implement the algorithm for breaking codes with a given basic structure.
2. Explain the steps in an “Interview Grading Style” with Jupyter Lab Text blocks demonstrating and explaining the function as you go.
3. These demonstrations, explanations, and code comments must demonstrate that you understand the code and what it is doing.
4. After the basic implementation you will rewrite the code completely with your own improvements, or copy the basic code package and add new features. This will be your custom feature.
5. IMPORTANT - in the first implementation you will keep all instructions, given comments in the code, and use the variable names and set up as given. For your custom feature you may remove these and make the code “your own.” ALL CHANGE must be explained in text - why did you change/make improvements? We should be able to see how you adapted the original code.

1 Table of Contents/Grading Structure

0. Bonus +/- Overall structure, code comments, and following directions. (Score may go up or down if overall structure or format is excellent or incomplete). Not extra credit.

1.1 1. Introduction to your RSA Project (5 points)

1.2 2. Code Package: FME (10 points)

- * Convert_Binary_String
- * FME

1.3 3. Code Package: Key Generation using Euclidean Algorithms (10 points)

- * Euclidean Algorithm
- * Extended EA
- * Find_Public_Key_e
- * Find_Private_Key_d

1.4 4. Code Package: En/Decode Your Messages (10 points)

- * Convert_Text
- * Convert_Num
- * Encode
- * Decode

1.5 5. Create a Code Demo(10 points)

1.6 6. Code Exchange Results (5 points)

1.7 7. Write a Narrative (25 points)

1.8 8. Code Breaking: (10 points)

1.9 9. Code Breaking: Complete Examples (5 points)

1.10 10. Custom code feature / Advanced options (10 points)

1.11 1. Introduction to your RSA Project (5 points)

Write an introduction/overview here. This should be something someone who has never seen your project could read to understand what is coming. Give a broad summarized overview of RSA components, discuss your custom feature a bit, and include some preliminary narrative regarding your process. Much of this will be fleshed out in your narrative below, so remember to simply summarize details. This should be a robust paragraph or two of 200 - 300 words.

Every day the average person sends information all across the internet, information that contains some of their most sensitive data. Private messages, financial account details, and personally identifiable information all travel along networks that touch thousands of computers, computers that users might not know and might not trust. Through the use of complex encryption methods, this information can all be obfuscated in a way that nefarious actors will have no way to view a person's most important data. One such method is the RSA public-key cryptosystem. Invented by Ron Rivest, Adi Shamir and Leonard Adleman, RSA relies on the numerical properties of prime numbers and remainders to encrypt data in a way that makes it extremely computationally demanding to ever decrypt without the message recipient's private key. To demonstrate its functionality, this project walks through the major algorithms required to implement such a system and explains their use within the greater implementation. Starting with fast modular exponentiation the program leverages squares and modulus to compute the remainder for very large numbers quickly and efficiently. Next it implements a version of the extended Euclidean Algorithm to find the greatest common divisor and modular inverses of two numbers. Finally, it puts everything together to generate keys, encode messages, and decode cyphers. By the end, it will be perfectly clear why this simple technique is so powerful.

1.12 2. Code Package FME (10 points)

FME is an essential function for RSA to pre-process the messages before the messages are encoded and decoded by the RSA algorithm. You have 3 choices of algorithms for this: * The Rosen Book FME - which will need `Convert_Binary_String` below. * Sriram's FME from the video. * Your own "Square and Mod" function defined with a look up table.

Do not copy code from the internet - it is recognized immediately by our software. In particular, use of bitshifting is forbidden in the first part of the project but may be explored in the custom feature.

```
[1]: def Convert_Binary_String(_int):  
    """  
    Here, you need to define a function that converts an integer to  
    a string of its binary expansion. You may or may not need this function.  
  
    For example:  
    _int = 345  
    bits = 101011001  
    """  
  
    return bin(_int)[2:]
```

```
[2]: def FME(b, n, m):  
    """  
    1. Using the fast modular exponentiation algorithm,  
    the below function should return  $b^n \bmod m$ .  
    As described on page 254. (however, you may input the exponent and  
    then convert it to a string - the book version imports the binary_  
    ↪expansion)  
    2. You should use the function defined above Convert_Binary_String() if_  
    ↪using the book algorithm.  
    3. For this block you MUST use one of the 3 methods above.  
    4. Any method using bit-shifting or copied from the internet (even changing_  
    ↪variable names) will result in a 0.  
  
    **If you are completely stuck, you may use pow() with a 10pt penalty.**  
  
    You may use the function you developed in your Mastery Workbook - but be_  
    ↪sure it is your own  
    work, commented, etc. and change inputs as needed.  
    """  
  
    # Steps in initial video explanation  
    # Step 1: Write n in binary  
    # Step 2: Take all the bits that are 1  
    # Step 3: Multiply them together mod b  
  
    # Sriram's algorithm in video  
    # Result will store the accumulation of each modulo
```

```

result = 1
# Square will repeatedly square a to represent each bit position
square = b

# Loop converts n to binary
# The while loop will occur the number of bits n has
while (n > 0):
    # Extracts least sig bit because all odd numbers have the least sig bit
    ↪set to 1
    k = n % 2

    # Repeatedly square a mod b
    # If that bit position is set
    if k == 1:
        # Use the result variable to increase the total
        result = (result * square) % m

    # square the current square everytime to increase it with the bit
    ↪position
    square = (square * square) % m

    # divide n by two to move the next bit into the lsb position
    # need to use floor division or n will turn into a float
    n = n // 2

return result

```

1.13 3. Code Package: Key Generation using Euclidean Algorithms (10 points)

The functions below will generate the public and private key pairs which will then be used to create a ciphertext using the public key and then decode the same using the private key.

```

[3]: def Euclidean_Alg(a, b):
    """
    1. Calculate the Greatest Common Divisor of a and b.

    2. This version should have only positive inputs and outputs.

    3. The function must return a single integer ('x') which is
    the gcd of a and b.

    """
    # continue execution until b (holds last remainder) is less than or equal
    ↪to 0
    while b > 0:
        # move b into a temporary value for later

```

```

x = b
# set b equal to the remainder of a divided by b
b = a % b
# set a equal to the old value of b
a = x

# once the remainder is zero, the value stored in a and x will be the GCD.
return x

```

```

[4]: def EEA(a, b):
    """
    This is a helper function utilizing Bezout's theorem as discussed in your_
    ↪MW.
    You will follow these same steps closely to construct this function.

    This version will return both:
    1. the GCD of a, b
    2. Bezout's coefficients in any form you wish. We recommend returning your_
    ↪coefficients as a list or a tuple.
    HINT: return GCD, (s1, t1)

    * Ensure that your inputs are positive integers. Implement these kinds of_
    ↪checks.
    * It might also behoove you to consider reassigning a, b to new_
    ↪coefficients depending on which is greater.

    """
    # Ensure inputs are positive
    if a < 0 or b < 0:
        print("Arguments must be positive integers")
        return

    # Originally I implemented code to switch a and b if a < b. However, this
    # created issues with my s1, t1 and s2, t2 values. After running some tests
    # I determined that this implementation works regardless of the a b relation

    # Start with our original values of s and t
    # the original "recipe" for m and n
    # s1 and t1 are the values that represent the current value of m
    s1, t1 = 1, 0
    # s2 and t2 are the values that represent the current value of n
    s2, t2 = 0, 1

    # while loop to run if there is a remainder
    # m and n values are expanded with bezout coefficients for conceptual_
    ↪reasons
    # understanding that they can allways be written as such and that updating

```

```

# the coefficients updates m and n appropriately
while (s2 * a + t2 * b) > 0:
    # find the quotient of current m divided by current n
    q = (s1 * a + t1 * b) // (s2 * a + t2 * b)

    # store s1 and t1 in a temporary variable for use later
    s_hat, t_hat = s1, t1
    # update s1 and t1 (i.e. update the recipe for m with the recipe for n)
    s1, t1 = s2, t2
    # update s2 and t2 (i.e. update the recipe for n with the remainder)
    s2, t2 = (s_hat - q * s2), (t_hat - q * t2)

# return m (GCD), and its recipe (Bezout Coefficient)
gcd = s1 * a + t1 * b

return gcd, (s1, t1)

```

```

[5]: def Find_Public_Key_e(p, q):
    """
    Implement this function such that
    it takes 2 primes p and q.

    Use the gcd function that you have
    defined before.

    The function should return 2 elements as follows:
    public key: n
    public key: e

    HINT: this function will run a loop to find e such
    that e is relatively prime to (p - 1) (q - 1)
    and not equal to p or q.

    NOTE: There are a number of ways to implement this key feature.
    You, as the coder, can choose to how to achieve this goal.

    """
    # Calculate n as the product of p and q
    n = p * q

    # Calculate (p-1)(q-1) and store as x to avoid calculating repeatedly
    x = (p - 1) * (q - 1)

    # Set e one less than the smaller of p and q
    e = min(p, q) - 1

    # find the largest relatively prime number less than (p-1)(q-1)

```

```

# starting from the top ensures e will have the largest possible value
# included while e > 1 to avoid infinite loop if there are errors
while (EEA(e, x)[0] != 1) and (e > 1):
    # decrement e
    e -= 1

# return the public key as a tuple
return (n, e)

```

```

[6]: def Find_Private_Key_d(e, p, q):
    """
    Implement this function to find the decryption exponent d,
    such that d is the modular inverse of e.

    This will use the Extended Euclidean Algorithm

    This function should return the following:
    d: the decryption component.

    This is not a single action, and there are multiple methods to create this.

    You may create a helper function or have all code within this function.

    Plan ahead before coding this.

    """
    # calculate (p-1)(q-1) and store as x
    # this value is the modulus and is used
    # to find the modular inverse of e
    x = (p - 1) * (q - 1)

    # call function EEA to get the bezout coefficients of e and x.
    # s1 by definition, is the modular inverse of e
    d = EEA(e, x)[1][0]

    # ensure d is positive
    # if not, increase it by the modulus until it is
    while d < 0:
        d += x

    # return the modular inverse of e
    return d

```

1.14 4. Code Package: En/Decode Your Messages (10 points)

1. In this part, you will define two functions `Encode` and `Decode` which will use the public and private keys that you calculated above.

2. Using the public key, the `Encode` function will encode a message and generate the corresponding `cipher_text`.
3. Using the private key, the `Decode` function will decode a `cipher_text` and recover the original message.
4. We are proving `Convert_Text` and `Convert_Num` for you.

```
[7]: def Convert_Text(_string):  
    """  
    Define this function such that it takes in a simple  
    string such as "hello" and outputs the corresponding  
    standard list of integers (ascii) for each letter in the word hello.  
    For example:  
    _string = hello  
    integer_list = [104, 101, 108, 108, 111]  
  
    You may use "ord()"  
    """  
    # Using list comprehension iterate across every character in the string  
    # return that character's ascii value using ord()  
    integer_list = [ord(c) for c in _string]  
    return integer_list
```

```
[8]: def Convert_Num(_list):  
    """  
    Do the opposite of what you did in the Convert_Text  
    function defined above.  
  
    Define this function such that it takes in a list of integers  
    and outputs the corresponding string (ascii).  
  
    For example:  
    _list = [104, 101, 108, 108, 111]  
    _string = hello  
    """  
    # No changes made to provided function, comments explain what is happening  
    # initialize an empty string to store the message  
    _string = ''  
    # iterate over every item in the list  
    for i in _list:  
        # concatenate the string with the provided value symbol based on ascii  
        _string += chr(i)  
    return _string
```

```
[9]: def Encode(n, e, message):  
    """  
    Here, the message will be a string of characters.  
    Use the function Convert_Text from
```


the basic tool set and get a list of numbers.

*Encode each of these numbers using n and e and
return the encoded cipher_text.*

"""

First, convert the input message to a list of it's ascii values
message = Convert_Text(message)

*# Next, using list comprehension, for each ascii value in the list
perform the mathematical function $M^e \bmod n$ using Fast Modular*

↪ *Exponentiation*

cipher_text = [FME(M, e, n) for M in message]

Return the now encrypted message
return cipher_text

[10]: def Decode(n, d, cipher_text):

"""

*Here, the cipher_text will be a list of integers.
First, you will decrypt each of those integers using
n and d.*

*Later, you will need to use the function Convert_Num from the
basic toolset to recover the original message as a string.*

"""

*# First, using list comprehension, for each value in the list
perform the mathematical function $C^d \bmod n$ using Fast Modular*

↪ *Exponentiation*

message = [FME(C, d, n) for C in cipher_text]

Next, conver the message of ascii values to their character equivilants ↪

↪

message = Convert_Num(message)

Return the now decrypted message
return message

1.14.1 5. Create a Code Demo (10 points)

Construct a demonstration of your functions above. This will be a **step-by-step** guide to using your code with a specific example that we can follow using the functions you have created above **using a mix of code and Markdown blocks**. Choose your own “Hello World” type message or a favorite quote of your choosing to code and decode yourself.

This is essentially a test that demonstrates your code with a small walkthrough of how it works. You are simply calling the functions above. You are not using a main function or custom feature here. Imagine this as a code interview scenario, and your interviewer/future

boss wants a preliminary walkthrough of your work. Therefore, some text blocks are essential and helpful for your boss to follow your process.

Note: You can change and add the type of cell block you are editing with the drop down menu above and go between Code and Markdown.

This is a mix of code blocks and text blocks and must discuss all three parts for full credit. You must **show and explain your method and demonstrate (by calling your function as an example)** for generating keys and how your encoding and decoding functions work.

For full credit:

- When you call FME, explain briefly why we can't take the "mod" of our number using "%".
- When you call the EEA, explain why this is so essential and really the key to RSA process.
- Demonstrate and explain how you Generate keys with code and text blocks.

To Begin, we will first need to generate a public and private key that will allow us to encrypt and decrypt messages respectively. The public key is used to generate encrypted versions of messages and should be given to the sender by the recipient. Since this key only allows an individual to encrypt a message, there is no risk of sending this information over networks. To generate these keys, a recipient must first select two (preferably large) prime numbers. For this example, to match the textbook (Rosen, 300-301), the algorithm will use 43 and 59.

```
[11]: p = 43  
      q = 59
```

Next, the algorithm generates the public key, passing the two values into the "Find_Public_Key_e" function. This function takes two prime numbers as input and returns the public key as a tuple. The public key consists of (n, e) where n is the product of p and q , and e is a relatively prime number to the product of $(p - 1)$ and $(q - 1)$

```
[12]: # Generate the public key (n, e)  
      public_key = Find_Public_Key_e(p, q)  
      print(public_key)
```

```
(2537, 41)
```

Under the hood, this function computes n by multiplying p and q , and computes e using the Extend Euclidean Algorithm. Use of EEA to compute the GCD of e and $(p - 1)(q - 1)$ will return 1, if the two numbers are relatively prime. We must find an e that is relatively prime in order to ensure e has a modular inverse, essential for finding the private key.

After the algorithm determines the public key, it must generate the private key. The private key is used to generate the plain text versions of a message that was encrypted with the corresponding public key. This key should be kept private and not sent over networks, as the recipient uses this key to decrypt data. The function "Find_Private_Key_d" takes the previously generated value for e as well as p and q and returns the value d such that d is the modular inverse of $e \bmod (p - 1)(q - 1)$. The private key then consists of (n, d) .

```
[13]: # Generate d, the modular inverse of e mod (p-1)(q-1)
d = Find_Private_Key_d(public_key[1], p, q)
# Create the private key (n, d)
private_key = (public_key[0], d)
print(private_key)
```

(2537, 713)

To find the inverse, the function leverages the Extended Euclidean Algorithm to find the Bezout coefficients of e and $(p-1)(q-1)$ such that $se + t((p-1)(q-1)) = 1$. In doing so, it can be sure that the value of s is one possible modular inverse of e .

Now that both keys have been generated, it is possible to encode a message.

```
[14]: # Zero p and q for security
p = 0
q = 0
```

- Demonstrate and explain how you encode a message with code and text blocks.

To encode a message, the algorithm calls the “Encode” function. This function takes as input the public key value n , the public key value e , and the message in the form of a string. It then returns the encrypted message as a python list of integers, where each list item corresponds to a character in the original message. For example, we will use the message “Hello World!” with the previously generated public key.

```
[15]: # Plain text message
pt_message = "Hello World!"

# Encode plain text message using elements of the public key
ct_message = Encode(public_key[0], public_key[1], pt_message)
print(ct_message)
```

[347, 367, 1851, 1851, 1407, 2533, 1377, 1407, 1912, 1851, 2018, 2008]

Under the hood, the function leverages Fast Modular Exponentiation to compute the value of $M^e \bmod n$ where M is the ascii value of each character in the message, and n and e are the public key values. It is important to note that in real world implementations of the RSA algorithm, the original p and q values, and subsequently the public and private keys, are primes with hundreds or thousands of digits. In those cases, Fast Modular Exponentiation is essential because the value of M^e will be larger than normal processors can accommodate, and the normal ‘%’ operator would require eons to compute.

Finally, using the private key generated earlier, we can decode the encrypted message to ensure it is valid.

- Demonstrate and explain how you decode the same message with code and text blocks.

To decode a message, the algorithm calls the “Decode” function. This function takes the private key values n and d and the encoded message as input. Once complete, it returns the decrypted version of the input message.

```
[16]: # Decode the cypher text message using elements of the private key
new_pt_message = Decode(private_key[0], private_key[1], ct_message)
print(new_pt_message)
```

Hello World!

As demonstrated, the deciphered message is equal to the original plain text message from before. Just as with the “Encode” function, the “Decode” function leverages Fast Modular Exponentiation to compute the value of $C^d \bmod n$ where C is the encrypted message, d is a modular inverse of $e \bmod (p-1)(q-1)$, and n is the product of p and q .

1.14.2 6. Code Exchange Results (5 points)

Now that your code is working and you are able read and write messages, include 3 complete examples of exchanging code (and exchange is both reading a message and responding) from Piazza here. Both call your code and output code exchange results here. We are just looking to see which 3 codes you have exchanged in Piazza.

Example 1: Professor Stade’s Message

```
[17]: # Encrypted message
ct_message1 = [
    2128, 1150, 4250, 1349, 1262, 3336, 2371, 2497, 519, 1262, 1263, 1105,
    ↪3336, 1349, 1262, 2310, 1105, 3336, 4115,
    762, 2405, 1263, 1105, 3336, 1262, 1349, 1150, 1105, 1262, 506, 1105, 1105,
    ↪4723, 2405, 2497, 519, 1262, 1974,
    2371, 58, 1262, 519, 1105, 1349, 1262, 4839, 1150, 1105, 2497, 1262, 1974,
    ↪2371, 58, 762, 1262, 13, 4679, 1573,
    1262, 4115, 2371, 2310, 1105, 1262, 2405, 3336, 1262, 4839, 2371, 762,
    ↪1560, 2405, 2497, 519, 3250
]

# Corresponding keys
public_key = (5251, 3)
private_key = (5251, 3403)

# Call decode function with private key
pt_message1 = Decode(private_key[0], private_key[1], ct_message1)
print(pt_message1)
```

What song best describes the feeling you get when your RSA code is working?

```
[18]: # Plain text reply
pt_reply1 = "Sweet Victory from SpongeBob, because finishing RSA is 'sweet
    ↪sweet sweet victory!'"

# Call encode function with public key
ct_reply1 = Encode(public_key[0], public_key[1], pt_reply1)
```

```
print(ct_reply1)
```

```
[4679, 4839, 1105, 1105, 1349, 1262, 685, 2405, 4115, 1349, 2371, 762, 1974,
1262, 506, 762, 2371, 3283, 1262, 4679, 2911, 2371, 2497, 519, 1105, 3942, 2371,
1263, 1168, 1262, 1263, 1105, 4115, 4250, 58, 3336, 1105, 1262, 506, 2405, 2497,
2405, 3336, 1150, 2405, 2497, 519, 1262, 13, 4679, 1573, 1262, 2405, 3336, 1262,
1558, 3336, 4839, 1105, 1105, 1349, 1262, 3336, 4839, 1105, 1105, 1349, 1262,
3336, 4839, 1105, 1105, 1349, 1262, 4720, 2405, 4115, 1349, 2371, 762, 1974,
4431, 1558]
```

```
[19]: # Verify correct encoding
test = Decode(private_key[0], private_key[1], ct_reply1)
print(test)
```

Sweet Victory from SpongeBob, because finishing RSA is 'sweet sweet sweet victory!'

Example 2: Student Reply

```
[20]: # Encrypted message
ct_message2 = [
    2615, 861, 5887, 8183, 3506, 5706, 854, 2048, 88, 487, 3506, 6656, 854,
    ↪3357, 4161, 4925, 3506, 5911, 854, 3357,
    3506, 653, 5887, 8183, 6656, 861, 3506, 854, 2048, 487, 4711, 3506, 5887,
    ↪1461, 4925, 3506, 854, 2048, 487, 4711,
    3506, 5887, 7617, 5887, 88, 1461, 3506, 5887, 1461, 4925, 3506, 1461, 487,
    ↪2048, 487, 4711, 3506, 7617, 487, 8183,
    3506, 8570, 88, 6656, 1072, 3506, 854, 8004, 1414, 3506, 313, 8570, 3506,
    ↪88, 8183, 3506, 5911, 854, 3357, 4711,
    3506, 8004, 5887, 2048, 854, 4711, 88, 8183, 487, 3506, 5706, 854, 2048,
    ↪88, 487, 3506, 854, 8004, 3506, 5887,
    4161, 4161, 3506, 8183, 88, 5706, 487, 1414
]

# Corresponding keys
public_key = (9071, 8545)
private_key = (9071, 7201)

# Call decode function with private key
pt_message2 = Decode(private_key[0], private_key[1], ct_message2)
print(pt_message2)
```

What movie could you watch over and over again and never get sick of? Is it your favorite movie of all time?

```
[21]: # Plain text reply
pt_reply2 = "Office Space, that movie never gets old!"
```

```
# Call encode function with public key
ct_reply2 = Encode(public_key[0], public_key[1], pt_reply2)
print(ct_reply2)
```

```
[2237, 8004, 8004, 88, 6656, 487, 3506, 7996, 2042, 5887, 6656, 487, 8448, 3506,
8183, 861, 5887, 8183, 3506, 5706, 854, 2048, 88, 487, 3506, 1461, 487, 2048,
487, 4711, 3506, 7617, 487, 8183, 8570, 3506, 854, 4161, 4925, 2090]
```

```
[22]: # Verify correct encoding
test = Decode(private_key[0], private_key[1], ct_reply2)
print(test)
```

Office Space, that movie never gets old!

Example 3: New Post

```
[23]: # Plain text post
pt_post = "What TV show are you currently into? I can't stop watching Bob's_
↳Burgers right now!"

# small p and q values
p = 73
q = 83

# Find the public key
public_key = Find_Public_Key_e(p, q)
print(f"The public key is {public_key}")

# Find the private key
private_key = (public_key[0], Find_Private_Key_d(public_key[1], p, q))
print(f"The private key is {private_key}")
```

The public key is (6059, 71)

The private key is (6059, 4823)

```
[24]: # Encrypt the post using the public key
ct_post = Encode(public_key[0], public_key[1], pt_post)
print(ct_post)
```

```
[1434, 2223, 2479, 4178, 1914, 4400, 3330, 1914, 3325, 2223, 25, 2217, 1914,
2479, 3415, 5462, 1914, 1130, 25, 2925, 1914, 2687, 2925, 3415, 3415, 5462,
2411, 4178, 4574, 1130, 1914, 3374, 2411, 4178, 25, 3555, 1914, 876, 1914, 2687,
2479, 2411, 5563, 4178, 1914, 3325, 4178, 25, 1986, 1914, 2217, 2479, 4178,
2687, 2223, 3374, 2411, 3998, 1914, 125, 25, 2958, 5563, 3325, 1914, 125, 2925,
3415, 3998, 5462, 3415, 3325, 1914, 3415, 3374, 3998, 2223, 4178, 1914, 2411,
25, 2217, 5506]
```

```
[25]: # Verify the correct encoding
test = Decode(private_key[0], private_key[1], ct_post)
print(test)
```

What TV show are you currently into? I can't stop watching Bob's Burgers right now!

1.14.3 7. Write a Narrative (25 points)

TWO pages of meaningful narrative about your process - not including the examples - (800 words) here in this block. Include specific examples of debugging and challenges referencing your individual process. Responses that are vague, lacking in specific and individual details, or sound like CHatGPT will not receive full credit. Include the following:

- Describe your process. Did you plan ahead? Did you wing it? What have you learned about your own process?
- Describe the results of exchanging keys and codes with classmates. Did it work the first time or did you need to make adjustments? Use the word fortitude
- What was most challenging? Use a cooking metaphor.
- Who helped you with your project? Which resources were most helpful?
- What was your Best Mistake? (funniest? most frustrating? the one you learned the most from?)

From this demonstration, it is clear to see just how powerful the RSA algorithm can be. Leveraging Fast Modular Exponentiation and the Extended Euclidean Algorithm, RSA can be used to create complex encryptions that require a great deal of computer power to crack, even at lower values. With the demonstration complete, it is now valuable to discuss some of the techniques and processes used in the project to employ this relatively simple algorithm. I will start by discussing my overall strategy for putting together this project and how the Mastery Workbooks contributed to its development. Following that, I will expand on my specific experiences when developing the Fast Modular Exponentiation and Euclidean Algorithm implementations, the core of the project. Finally, I will explain the coding methods I used to complete the remaining functions required to fully implement the project.

For the overall process, I found that the easiest way to tackle this problem was to follow the outline of the Mastery Workbooks. In doing so, I was able to gather a solid understanding of the two important algorithms required to implement RSA. The Mastery Workbooks did an excellent job of demonstrating the fundamental procedures, and core theoretical concepts that are essential for understanding how Fast Modular Exponentiation and the Extended Euclidean Algorithm work. Then, only after building up a strong foundation, the Master Workbooks guide you through developing the implementation of the two algorithms. Had I instead tried to design each algorithm without working through the workbooks, I would not have had as strong of an understanding of how they work, and most certainly would have struggled implementing them. By the time I had completed each Mastery Workbook, I was able to adapt the algorithm for the project with ease, fully understanding each step. The other resource that I leveraged most were the lecture videos on each of the algorithms. It took me a few times watching them to internalize them, but by the time I completed the code in the Mastery Workbooks, I felt very confident in their implementation.

Starting with Fast Modular Exponentiation, I generally followed the pseudocode that was discussed by Sriram in the lecture videos. For the majority of the algorithm, the code did not require much adjustment and I was able to adjust the required variables with relative ease. The only major problem that needed to be solved from the pseudo code was the risk of an infinite loop. Because the loop operates until n is less than 1, there needs to be a way to update n to reach the exit condition. Additionally, since in each iteration, the algorithm examines the least significant bit of n , there also needs to be a way to either evaluate the next bit or move the next least significant bit into that position to be evaluated. Under normal circumstances, I would have implemented a right shift to accomplish both requirements in one operation. However, because bit shifting was forbidden for the project, I instead used integer division to divide n by 2, effectively accomplishing the same operation. Moving into the code feature portion of the lab, I intend to modify this function by using bit shifting and bitwise comparisons to slightly optimize operation count and memory space.

For the next major implementation, the Extended Euclidean Algorithm, I again followed the pseudocode by Sriram in the lecture videos. However for this implementation, I had to make a few more adjustments to employ it within the project. First, I included an additional if statement at the beginning of the function to ensure that the two integers passed into the function are positive. Next, I considered flipping the two integers to ensure that the larger of the two was first, but this proved to cause more issues than fix. I found that I did not encounter any anomalous return values when the first integer was less than the second, but I did have issues ensuring the Bezout coefficients corresponded to the correct integer. Instead of attempting to correct the latter, I neglected to implement the swapping and left the code as is. Finally, unlike the pseudocode, I only employed one pair of temporary variables to update s_1, t_1 and s_2, t_2 .

After employing the two core functions above, all that remained were the functions to generate keys, encode messages, and decode messages. Since there was not any direct lecture with pseudo code for these functions, they required more work, but proved to be significantly less complex algorithms than the previous.

Starting with the public key generation, I began the code by calculating n as the product of p and q . Then, to find e , I first found the product of $(p - 1)$ and $(q - 1)$, then iterated through all values less than the smaller of p and q until I found a value that was relatively prime. I chose to start with the largest value e could be and decrement to identify the largest possible value of e , instead of smallest.

For the private key generation, I initially assumed that this would be more difficult but found it to be slightly easier. With the understanding that the Bezout coefficients are modular inverses, the only operation I had to perform was to pass e and the product of $(p - 1)$ and $(q - 1)$ into the Extended Euclidean Algorithm, and then set $d = s$. To ensure $d \geq 0$ I then implemented a while loop that continued to add the product of $(p - 1)$ and $(q - 1)$ to d until $d \geq 0$.

To implement the Encode and Decode functions, I relied heavily on example 8 and 9 in the textbook pages 300 and 301. Using that resource, and list comprehension in python, I was easily able to implement a few lines of code to encode and decode messages using Fast Modular Exponentiation.

After completing all code elements, I was eager to test it. Trying my code on the initial encrypted message I ran into an issue. I had at first neglected to adjust the variables of my code implementations to the required ones in the project. After making the appropriate changes, I was pleasantly surprised to get a full English sentence on my first attempt to decode. Following the initial attempt, I only again ran into issues when my code execution was out of order in the Jupyter Lab code blocks.

Overall the most challenging part of this project was understanding the core algorithms. Leveraging the Mastery Workbooks and the video lectures I was able to gain a full in depth understanding and employ them successfully.

1.14.4 8. How to Break Code. (10 points)

This section (and section 9 and 10) of the project is only available to students exchanging codes by the Friday (midnight) before the project is due.

Implementing a factoring algorithm:

Begin by coding the basic brute force factorization algorithm given in pseudocode below.

Brute Force Factoring

```
[26]: def factorize(n):
      factors = [1]
      # Iterate through all values from 2 to n - 1
      for i in range(2, n):
          # if i divides n
          if n % i == 0:
              # prints all factors of n
              factors.append(i)

      factors.append(n)

      return factors
```

After you have tested your RSA package yourself, and tested it with classmates by publishing both the private and public keys on Piazza, post 2 messages with just the public keys for people to break on the “Just Public Keys” thread - use both very small n ’s (under 1000) for practice. Feel free to come up with n ’s that create a challenge.

- Explain in detail how breaking works in this section using markdown blocks. Demo a short example with a small n . What are the complete steps? What tools do you need? Use a sports metaphor.
- Try brute forcing a larger value of n , say 15-18 digits long. What do you notice? (If this is taking a while, just ride it out, or Ctrl+C will KeyboardInterrupt the cell block). Provide a sample large n that will take ~3-5 min to crack.
 - Optional Challenge: Consider illustrating this with python math and time modules. matplotlib and time are good examples. Documentation is available.
- How secure is RSA? Why is it difficult to break codes in RSA?
- Is our implementation of RSA secure? What are it’s flaws?

Code Breaking Process

To implement an algorithm that can decrypt a message without using a private key, it must be able to generate a modular inverse of $e \bmod (p-1)(q-1)$ with only knowing the value of n . Since n and e are part of the public key, it is assumed that in this case those two values are known. To originally

generate the public key, the algorithm computes the modular inverse of $e \bmod (p-1)(q-1)$ using e, p, q . Therefore, to generate a private key using the public key, there needs to be a way to find the value of p and q . Since $n = pq$, the most straightforward approach would be to find all the factors of n and repeatedly test them as inputs to the decode function until a readable English sentence is produced.

Testing Factorize

To do this, the factorize function takes an integer n as input, and prints out all possible factors. With a small number, this is relatively simple.

```
[27]: factorize(78)
```

```
[27]: [1, 2, 3, 6, 13, 26, 39, 78]
```

However, as n gets larger, the amount of time it takes to factorize will increase linearly. To illustrate this, I will first set up some code that will allow us to time how long it takes to factor every number from 1 to some sample size, and then plot those times on a graph.

I am currently taking Introduction to Data Science Algorithms, so I will import matplotlib.pyplot to display the function's processing times.

To measure the time that this takes, I will import the time module and implement a timer using this guide from Geeks for Geeks.

<https://www.geeksforgeeks.org/python-measure-time-taken-by-program-to-execute/>

```
[28]: import time
import matplotlib.pyplot as plt
import numpy as np
import math
```

```
[29]: # Function to run factorize on all numbers 1 to some sample size, and return
      ↪ the time to factor each number
def test_factorize(samples):
    # Create a numpy array filled with zeros the size of samples to test
    times = np.zeros(samples)

    # itterate through all values of n from 1 to sample size
    for i in range(samples):
        # Start timer
        begin = time.time()

        # Call facorize(n)
        factorize(i+1)

        #Stop timer
        end = time.time()
```

```

        # Add time to times array
        times[i] = end - begin

    return times

```

```

[30]: # set a sample size
samples = 10000

# execute the test_factorize funtion
times = test_factorize(samples)
# create the x axis from 1 to sample size
n_value = [i+1 for i in range(samples)]

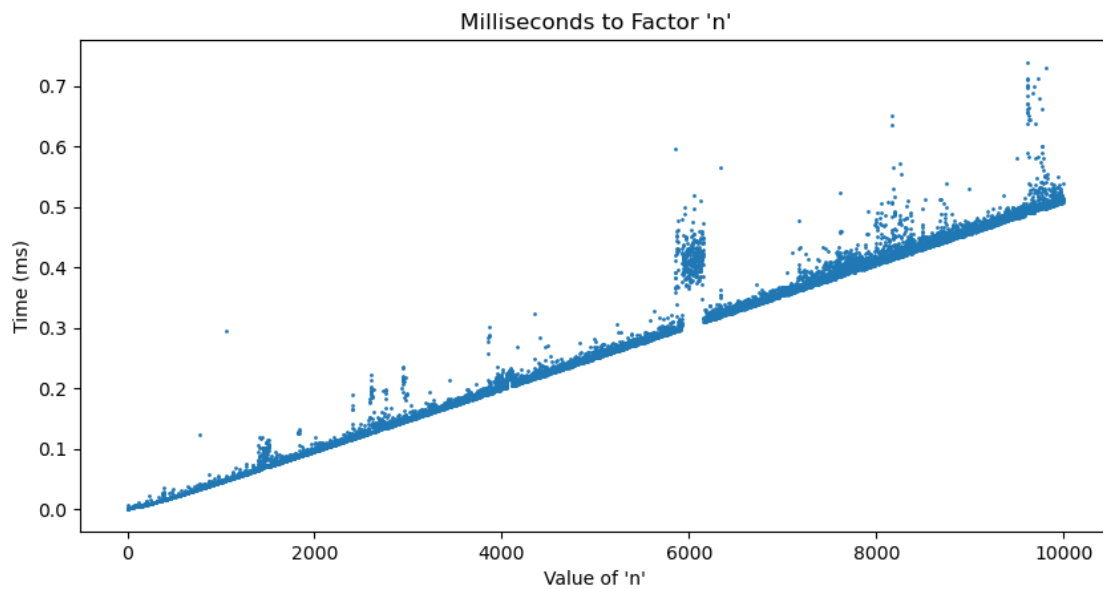
# make the graph slightly longer
f = plt.figure()
f.set_figwidth(10)

# Plot all times for each value of n on a scatterplot
plt.scatter(n_value, times*1000, s=1)

# Add title and axis labels
plt.title("Milliseconds to Factor 'n'")
plt.ylabel("Time (ms)")
plt.xlabel("Value of 'n'")

plt.show()

```



As demonstrated by our tests, a value of n approaches 5 digits, the time it takes to factor n increases linearly. This is because in all cases the function needs to check every value up to n with a few extra operations to append that value. Regardless of how many factors n has, the execution time for the function will be dominated by the size of n . This will be important to understand later on.

Breaking code

Now that we have discussed how to obtain factors of n in the public key, I will demonstrate how they can be used to generate a private key and decode a message.

Using the original encoded message posted by Professor Stade, along with the associated public key, we can start the code breaking process by factorizing n .

```
[31]: # Encrypted message
ct_message_stade = [
    2128, 1150, 4250, 1349, 1262, 3336, 2371, 2497, 519, 1262, 1263, 1105,
    ↪3336, 1349, 1262, 2310, 1105, 3336, 4115,
    762, 2405, 1263, 1105, 3336, 1262, 1349, 1150, 1105, 1262, 506, 1105, 1105,
    ↪4723, 2405, 2497, 519, 1262, 1974,
    2371, 58, 1262, 519, 1105, 1349, 1262, 4839, 1150, 1105, 2497, 1262, 1974,
    ↪2371, 58, 762, 1262, 13, 4679, 1573,
    1262, 4115, 2371, 2310, 1105, 1262, 2405, 3336, 1262, 4839, 2371, 762,
    ↪1560, 2405, 2497, 519, 3250
]

# Corresponding keys
public_key_stade = (5251, 3)
```

```
[32]: factorize(public_key_stade[0])
```

```
[32]: [1, 59, 89, 5251]
```

By definition, one of these factor pairs will be the p and q values that were used to generate the encryption. To generate all possible private keys we need to systematically input these pairs into the Find_Public_Key_d function and attempt to decode with the returned private key.

The function below first generates a list of all factors of n . Then, the for loop iterates over the half of the list of factors, generating a d for each pair and using that value as the private key in the decode function.

```
[33]: def break_code(public_key, message):
    factors = factorize(public_key[0])

    # iterate over half the list of factors since the list is pairs
    # in the event that n is a perfect square, there will be an odd number of
    ↪factors
    # in this case perform one more iteration
```

```

l = len(factors)
if l % 2 != 0:
    l += 1

for i in range(l//2):
    # First factor
    p = factors[i]
    # Corresponding factor at the other end of list.
    q = factors[-i - 1]

    # Generated private key from current set of factors p and q
    private_key = (public_key[0], Find_Private_Key_d(public_key[1], p, q))

    # Decoded message using generated private key.
    try:
        pt_message_hack = Decode(private_key[0], private_key[1], message)
    except ValueError:
        pt_message_hack = "Values out of ascii range"
    print(f"{pt_message_hack}, Private Key = {private_key}")

```

```
[34]: break_code(public_key_stade, ct_message_stade)
```

```

      e, Private Key = (5251, 1) :
What song best describes the feeling you get when your RSA code is working?,
Private Key = (5251, 3403)

```

Since the provided public key has a very low amount of factors, we only need to try two different private keys to break the cypher text. This demonstrates why having large prime values for p and q will provide the most security. When p and q are large, n is large, and when n is large, the time it takes to generate all factors is large.

To show this fact further, we can attempt the same decode with a more difficult public key.

```

[35]: # Select a large p and q so that n is roughly 15-18 digits
      # To get within the 3-5 min range, I had to reduce n to 10 digits
      #
      # To find a p and q, I used an online larger prime number generator
      # https://bigprimes.org/
      p = 29453
      q = 93113

      # Generate a hard public key
      hard_public_key = Find_Public_Key_e(p, q)

      hard_public_key

```

```
[35]: (2742457189, 29451)
```

```
[36]: # Encode with hard public key
ct_message_hard = Encode(hard_public_key[0], hard_public_key[1], "This is a
↳difficult message to crack")
```

```
[37]: # Start timer
begin = time.time()

break_code(hard_public_key, ct_message_hard)

# Stop timer
end = time.time()

print(f"Total time: {end - begin} seconds")
```

Values out of ascii range, Private Key = (2742457189, 1)

This is a difficult message to crack, Private Key = (2742457189, 950798915)

Total time: 286.064261674881 seconds

Voila! We cracked the code!

Notice though that it took almost 5 minutes for the algorithm to run. Notice further that we ended up only needing to test two private keys to find the working one. This means that the vast majority of time spent decoding the message was spent factorizing. Luckily, there are a few optimizations that can be made to our function.

Optimization 1

To improve the code breaking function we must look at improving our factorization algorithm. As discussed earlier the algorithm currently runs at $O(n)$ (ignoring the append function). However, it can be improved in a few ways. Since p and q must be prime we can immediately rule out all even factors of n . To do this, we can adjust the factorize function below.

```
[38]: def factorize(n):
    # remove 1 (not prime)
    factors = []

    if n % 2 == 0:
        factors.append(2)

    # Iterate through all values from 3 to n - 1
    # step by a factor of two to ignore all evens
    for i in range(3, n, 2):
        # if i divides n
        if n % i == 0:
            # prints all factors of n
            factors.append(i)
```

```

# By definition n cannot be prime
# factors.append(n)

return factors

```

To analyze our new improvement we can modify our test function and plot the time it takes

```

[39]: times2 = test_factorize(samples)

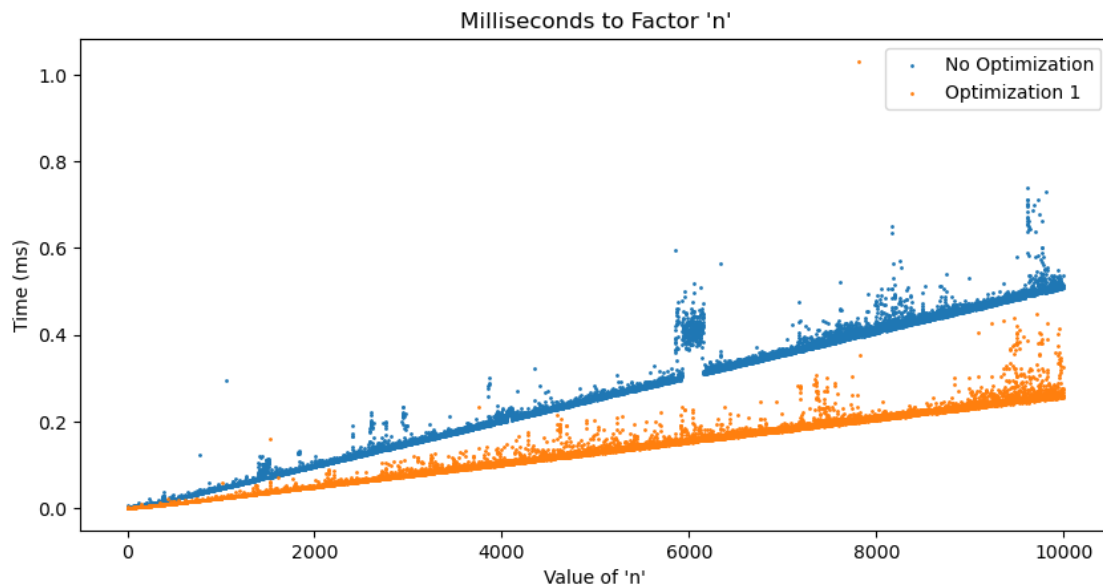
f = plt.figure()
f.set_figwidth(10)

plt.scatter(n_value, times*1000, s=1, label="No Optimization")
plt.scatter(n_value, times2*1000, s=1, label="Optimization 1")

plt.title("Milliseconds to Factor 'n'")
plt.ylabel("Time (ms)")
plt.xlabel("Value of 'n'")
plt.legend()

plt.show()

```



Since the algorithm now skips every even value of i , we have managed to cut the number of loop iterations in half. As well as the runtime.

```

[40]: # Start timer
begin = time.time()

```

```

break_code(hard_public_key, ct_message_hard)

# Stop timer
end = time.time()

print(f"Total time: {end - begin} seconds")

```

This is a difficult message to crack, Private Key = (2742457189, 950798915)
Total time: 141.7469208240509 seconds

When again implimenting our break_code function with the optimizations, we can see that it takes about half the time as before.

Optimization 2

To take this even further, we can modify the factorize algorithm once more, to simultaneously produce the factors as a pair, rather than individually. Additionally since we are finding all of the factor pairs, once the corresponding pair is larger than our current value we can safely exit the loop.

```

[41]: def factorize(n):
    # remove 1 (not prime)
    factors = []

    if n % 2 == 0:
        factors.append(2)
        factors.append(n//2)

    # Iterate through all values from 3 to n - 1
    # step by a factor of two to ignore all evens
    for i in range(3, n, 2):
        # if i divides n
        if n % i == 0:
            # find corresponding factor of i
            f = n//i
            # if it has already been found, end the loop
            if f < i:
                break
            # append i
            factors.append(i)
            # append the corresponding factor of i
            factors.append(f)

    # By definition n cannot be prime
    # factors.append(n)

```



```
return factors
```

```
[42]: factorize(783)
```

```
[42]: [3, 261, 9, 87, 27, 29]
```

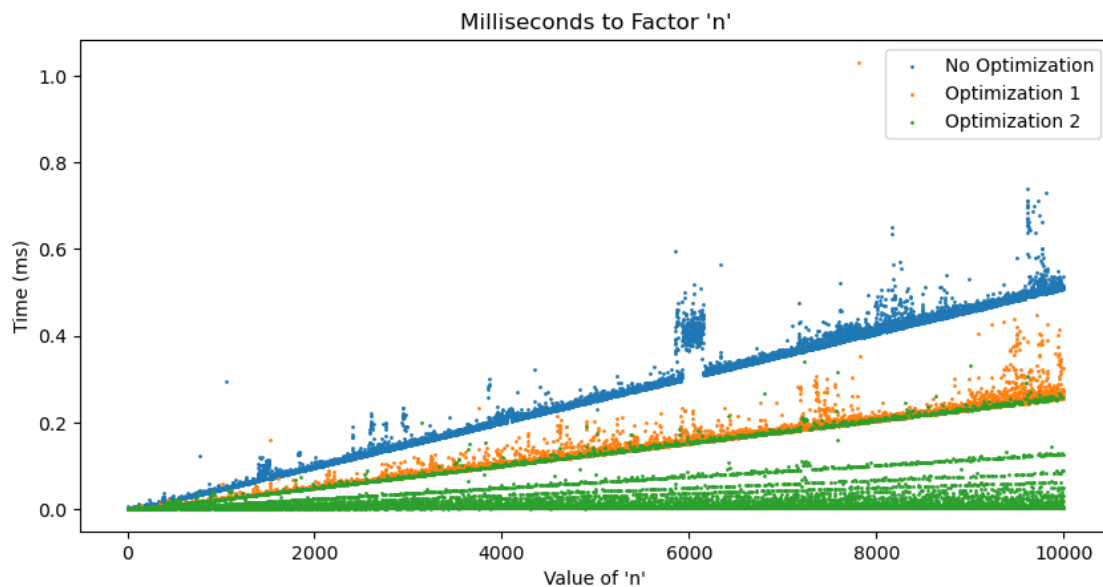
```
[43]: times3 = test_factorize(samples)
```

```
f = plt.figure()
f.set_figwidth(10)

plt.scatter(n_value, times*1000, s=1, label="No Optimization")
plt.scatter(n_value, times2*1000, s=1, label="Optimization 1")
plt.scatter(n_value, times3*1000, s=1, label="Optimization 2")

plt.title("Milliseconds to Factor 'n'")
plt.ylabel("Time (ms)")
plt.xlabel("Value of 'n'")
plt.legend()

plt.show()
```



As is apparent, we do not have all factors of n , nor do we have only prime factors. Notice further, how some numbers still require the full loop to find the factors

```
[44]: def break_code(public_key, message):
    factors = factorize(public_key[0])

    for i in range(0, len(factors), 2):
        # First factor
        p = factors[i]
        # Corresponding factor at the other end of list.
        q = factors[i+1]

        # Generated private key from current set of factors p and q
        private_key = (public_key[0], Find_Private_Key_d(public_key[1], p, q))

        # Decoded message using generated private key.
        try:
            pt_message_hack = Decode(private_key[0], private_key[1], message)
        except ValueError:
            pt_message_hack = "Values out of ascii range"
        print(f"{pt_message_hack}, Private Key = {private_key}")
```

```
[45]: # Start timer
begin = time.time()

break_code(hard_public_key, ct_message_hard)

# Stop timer
end = time.time()

print(f"Total time: {end - begin} seconds")
```

This is a difficult message to crack, Private Key = (2742457189, 950798915)
 Total time: 0.004868268966674805 seconds

Final Optimization

At this point, there is a pattern that is beginning to appear. After factoring, the brake_code function only ever seems to test one private key before finding the decoded message. This is not a fluke.

According to the fundamental law of arithmetic, any number can be written uniquely as a prime or the product of primes. Since we know p and q are both prime numbers, it is true that the unique product of p and q can only have 4 factors, p , q , 1 and itself. Therefore, to optimize our algorithm further, we can iterate through all integers up to n until one factor is found. That factor and $\frac{n}{factor}$ must then be p and q .

Additionally, I added a few other optimizations. Most notably, since p and q are more likely to be large values with many digits, it is significantly faster to start at a larger digit number and work down. As we proved in the Mastery Workbook, any composit number n must have a prime factor less than its squareroot. Leveraging that, the new factorize function loops through all odd values

less than the square root of n starting with the largest value. To ensure values stay odd even when the truncated square root is even, it adds the inverse of the least significant bit of that value.

```
[46]: def factorize(n):
    # remove 1 (not prime)
    # function will now only return one value
    # factors = []

    # if n is divisible by two, that is the first prime factor
    if not n & 1:
        return 2

    # Iterate through all values from 3 to n - 1
    # we have proven n will have a prime factor less than sqrt(n) so use this
    ↪ as a start
    start = int(math.sqrt(n))
    # If even, add 1
    start += not start & 1

    # step by a factor of two to ignore all evens
    # because p and q are likely to be large, start at the top and move down
    for i in range(start, 2, -2):
        # if i divides n
        if not n % i:
            return i
            # find corresponding factor of i
            # f = n//i
            # if it has already been found, end the loop
            # if f < i:
            #     break
            # append i
            # factors.append(i)
            # append the corresponding factor of i
            # factors.append(f)

    # By definition n cannot be prime
    # factors.append(n)

    return -1
```

```
[47]: factorize(5251)
```

```
[47]: 59
```

```
[48]: times4 = test_factorize(samples)
```

```

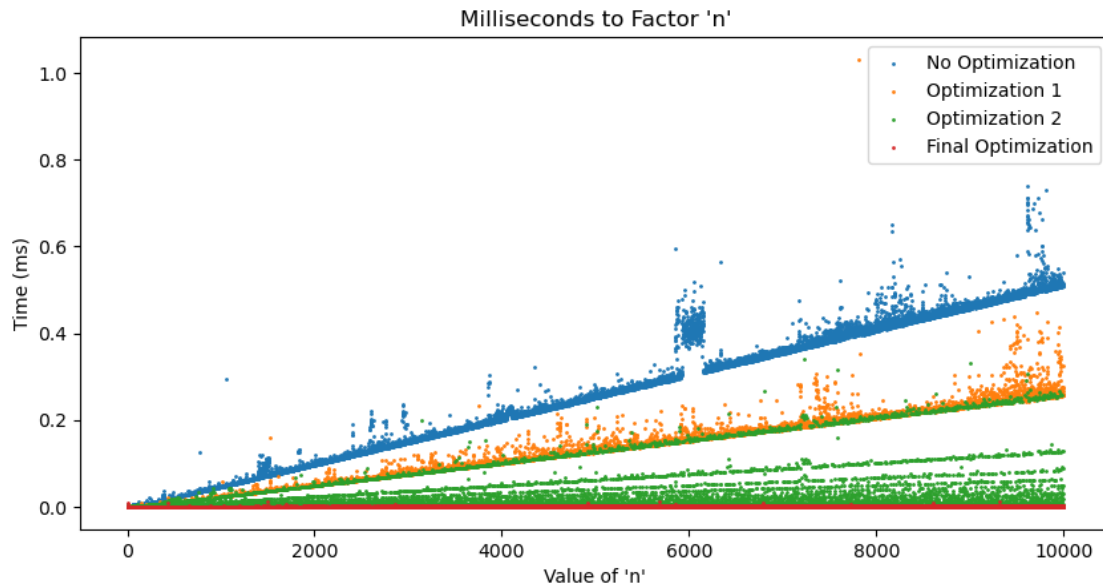
f = plt.figure()
f.set_figwidth(10)

plt.scatter(n_value, times*1000, s=1, label="No Optimization")
plt.scatter(n_value, times2*1000, s=1, label="Optimization 1")
plt.scatter(n_value, times3*1000, s=1, label="Optimization 2")
plt.scatter(n_value, times4*1000, s=1, label="Final Optimization")

plt.title("Milliseconds to Factor 'n'")
plt.ylabel("Time (ms)")
plt.xlabel("Value of 'n'")
plt.legend()

plt.show()

```



```

[49]: def break_code(public_key, message):
    # factors = factorize(public_key[0])

    # factors is no longer a list
    # for i in range(0, len(factors), 2):

    # First factor
    p = factorize(public_key[0])
    # Corresponding factor.
    q = public_key[0]//p

    # Generated private key from current set of factors p and q

```

```

private_key = (public_key[0], Find_Private_Key_d(public_key[1], p, q))

# Decoded message using generated private key.
# try:
pt_message_hack = Decode(private_key[0], private_key[1], message)
# except ValueError:
#     pt_message_hack = "Values out of ascii range"

print(f"{pt_message_hack}, Private Key = {private_key}")

```

```

[50]: # Select a large p and q so that n is roughly 15-18 digits
# To get within the 3-5 min range, I had to reduce n to 10 digits
#
# To find a p and q, I used an online larger prime number generator
# https://bigprimes.org/
p = 2142996269
q = 1194404623

# Generate a very hard public key
very_hard_public_key = Find_Public_Key_e(p, q)

# Encode with very hard public key
ct_message_very_hard = Encode(very_hard_public_key[0], very_hard_public_key[1],
↵    ↵"This is an extremely difficult message to crack")
very_hard_public_key

```

```

[50]: (2559604650765351587, 1194404621)

```

```

[51]: # Start timer
begin = time.time()

break_code(very_hard_public_key, ct_message_very_hard)

# Stop timer
end = time.time()

print(f"Total time: {end - begin} seconds")

```

This is an extremely difficult message to crack, Private Key =
 (2559604650765351587, 1148944288546474373)
 Total time: 23.979650497436523 seconds

1.14.5 9. Code Breaking: Complete Examples (5 points)

- Decode one example from Piazza in detail showing all the steps. Use a combination of code and markdown blocks.
- Decode and Respond to 3 “just codes with public keys” in Piazza with different sizes of n. Include complete results here - you may just show the results here.

Piazza Example To demonstrate this function on the piazza posts I will break the `break_code` function into its different parts, explain them, and work through decoding the first post on the code breaking thread.

```
[52]: # Encrypted message
ct_message_craven = [
    544, 472, 233, 242, 250, 555, 591, 550, 376, 364, 210, 165, 526, 47, 555,
    ↪95, 476, 210, 242, 165, 95, 233, 526,
    210, 376, 95, 233, 242, 364, 210, 242, 224, 376, 95, 364, 233, 47, 242,
    ↪210, 555, 376, 165, 472, 242, 555, 410,
    242, 250, 95, 233, 233, 47, 145]

# Corresponding keys
public_key_craven = (703, 113)
```

Step 1: Factor

The first step we need to do, is to factor n in the public key using the `factorize` function.

```
[53]: p = factorize(public_key_craven[0])
p
```

```
[53]: 19
```

Step 2: Generate d

Now that we have one of the two prime factors, we can generate our private key. As discussed earlier, because n is the product of two prime numbers, we know that it's only other factors are 1, q , and n . Therefore the other prime factor will be equal to $\frac{n}{p}$ or $\frac{703}{19} = 37$.

Using these two numbers we can generate d using the `Find_Private_Key_d` function.

```
[54]: d = Find_Private_Key_d(public_key_craven[1], p, 37)
d
```

```
[54]: 281
```

Step 3: Decode the Message

The final step is to decode the message. Since we have all the peices required for the private key, we can pass them strait into the `Decode` function with the message.

```
[55]: pt_message_craven = Decode(public_key_craven[0], d, ct_message_craven)
pt_message_craven
```

```
[55]: "The Conquistador's treasure is buried south of Creed."
```

```
[56]: print(Encode(public_key_craven[0],public_key_craven[1], "South? I thought it_
↳was North!"))
```

```
[182, 555, 376, 165, 472, 195, 242, 517, 242, 165, 472, 555, 376, 88, 472, 165,
242, 364, 165, 242, 541, 526, 210, 242, 469, 555, 95, 165, 472, 86]
```

Response 1

```
[57]: ct_message_craven2 = [
      85308, 48594, 15927, 71285, 61037, 15927, 767, 23406, 71285, 23406, 48594,↳
↳12676, 37298, 100459, 71285, 90170, 61037,
      38946, 38783, 23406, 71285, 90170, 71285, 113492, 38946, 38946, 86792,↳
↳15927, 90170, 37298, 71285, 12676, 767, 71285,
      15927, 121493, 15927, 37298, 71285, 12676, 84628, 71285, 29228, 38946,↳
↳38783, 71285, 90170, 110238, 15927, 71285, 81798,
      110238, 38946, 37298, 100459, 126494, 71285, 29228, 38946, 38783, 71285,↳
↳90170, 110238, 15927, 71285, 38946, 37298, 86792,
      29228, 71285, 38946, 84628, 84628, 71285, 61037, 29228, 71285, 90170,↳
↳71285, 61037, 12676, 23406, 10510]

public_key2 = (130177, 13)

break_code(public_key2, ct_message_craven2)
```

The best thing about a Boolean is even if you are wrong, you are only off by a bit., Private Key = (130177, 59749)

```
[58]: print(Encode(public_key2[0], public_key2[1], "Because they have no windows Matt!
↳"))
```

```
[113492, 15927, 5530, 90170, 38783, 767, 15927, 71285, 23406, 48594, 15927,
29228, 71285, 48594, 90170, 121493, 15927, 71285, 37298, 38946, 71285, 81798,
12676, 37298, 38831, 38946, 81798, 767, 71285, 10715, 90170, 23406, 23406,
124788]
```

Response 2

```
[59]: ct_message_brauer = [
      36414, 3399, 32768, 8546, 309822, 191201, 32768, 67055, 207467, 294794,↳
↳32768, 150484, 67055, 325095, 32768, 150484, 99816,
      237220, 325095, 103823, 237220, 309822, 273191, 325095, 8546, 85184, 32768,↳
↳274747, 67055, 207467, 150484, 32768, 67055,
```

```

    309822, 235986, 235986, 8546, 32768, 274747, 309822, 191201, 201903,
    ↪294794, 32768, 8546, 309822, 191201, 32768, 347119,
    99816, 265093, 167234, 32768, 191201, 347119, 250047, 32768, 36414, 32768,
    ↪150484, 67055, 99816, 273191, 167234, 32768,
    36414, 32768, 274747, 309822, 191201, 201903, 294794, 32768, 201903, 99816,
    ↪167234, 325095, 32768, 150484, 309822, 32768,
    110463, 265093, 191201, 235986, 207467, 32768, 294794, 99816, 232620,
    ↪325095, 32768, 237220, 309822, 71132, 325095, 32768,
    309822, 3399, 150484, 325095, 273191, 85184, 32768, 235986, 191201, 150484,
    ↪32768, 150484, 67055, 207467, 150484, 32768,
    99816, 273191, 232620, 309822, 201903, 232620, 325095, 110463, 32768,
    ↪207467, 32768, 201903, 309822, 150484, 32768, 309822,
    3399, 32768, 150484, 71132, 207467, 232620, 325095, 201903, 32768, 3399,
    ↪71132, 309822, 237220, 32768, 274747, 67055,
    325095, 71132, 325095, 32768, 36414, 32768, 201903, 99816, 232620, 325095,
    ↪35937
]

public_key_brauer = (352603, 3)

break_code(public_key_brauer, ct_message_brauer)

```

If you had the time/money, what hobby would you pick up? I think I would like to scuba dive more often, but that involves a lot of travel from where I live!,
Private Key = (352603, 234267)

```
[60]: print(Encode(public_key_brauer[0], public_key_brauer[1], "Mountain Climbing!"))
```

```
[103930, 309822, 191201, 273191, 150484, 207467, 99816, 273191, 32768, 300763,
201903, 99816, 237220, 235986, 99816, 273191, 34918, 35937]
```

Resposne 3

```
[61]: ct_message_huyer = [
    61010840592233335, 50965008984786859, 70937473165940755, 52883360311310107,
    ↪22978321700500060, 52883360311310107, 46958956358091443,
    6648963000258614, 42704356524869523, 52883360311310107, 53005202974701704,
    ↪50965008984786859, 42704356524869523, 3013150194457141,
    52883360311310107, 50965008984786859, 16338696893836718, 50965008984786859,
    ↪6648963000258614, 22978321700500060, 53005202974701704,
    50965008984786859, 39980344173847984, 50965008984786859, 8829315279589188,
    ↪21494489470651665, 52883360311310107, 53005202974701704,
    50965008984786859, 42704356524869523, 1773092908267621, 50965008984786859,
    ↪70937473165940755, 52883360311310107, 42704356524869523,

```


50965008984786859, 42704356524869523, 3013150194457141, 22101886973127150, [↩](#)
[↩](#)21494489470651665, 50965008984786859, 22978321700500060,
50965008984786859, 16231593379261352, 22101886973127150, 42704356524869523, [↩](#)
[↩](#)3013150194457141, 50965008984786859, 6648963000258614,
50965008984786859, 16338696893836718, 46958956358091443, 22101886973127150, [↩](#)
[↩](#)57742504562933385, 52883360311310107, 3214130891713985,
70937473165940755, 52883360311310107, 22978321700500060, 52883360311310107, [↩](#)
[↩](#)46958956358091443, 6648963000258614, 42704356524869523,
1773092908267621, 46958956358091443, 50965008984786859, 61010840592233335, [↩](#)
[↩](#)50965008984786859, 70758040515363712, 1773092908267621,
53005202974701704, 52883360311310107, 53005202974701704, 67079269891775927, [↩](#)
[↩](#)50965008984786859, 61010840592233335, 50965008984786859,
70758040515363712, 6648963000258614, 22978321700500060, 5895841574733797, [↩](#)
[↩](#)42704356524869523, 50965008984786859, 36332622461712651,
52883360311310107, 19253444273529194, 22101886973127150, 52883360311310107, [↩](#)
[↩](#)34258710235675678, 52883360311310107, 50965008984786859,
22101886973127150, 42704356524869523, 5895841574733797, 21494489470651665, [↩](#)
[↩](#)50965008984786859, 21494489470651665, 42704356524869523,
22101886973127150, 19253444273529194, 19253444273529194, 50965008984786859, [↩](#)
[↩](#)19253444273529194, 52883360311310107, 21494489470651665,
21494489470651665, 50965008984786859, 42704356524869523, 3013150194457141, [↩](#)
[↩](#)6648963000258614, 22978321700500060, 50965008984786859,
40911409245201015, 4486982623767293, 50965008984786859, 53005202974701704, [↩](#)
[↩](#)22101886973127150, 70937473165940755, 22101886973127150,
42704356524869523, 21494489470651665, 67079269891775927, 50965008984786859, [↩](#)
[↩](#)61010840592233335, 50965008984786859, 70758040515363712,
6648963000258614, 22978321700500060, 5895841574733797, 42704356524869523, [↩](#)
[↩](#)50965008984786859, 22101886973127150, 57742504562933385,
6648963000258614, 70937473165940755, 22101886973127150, 22978321700500060, [↩](#)
[↩](#)52883360311310107, 50965008984786859, 70937473165940755,
52883360311310107, 22978321700500060, 52883360311310107, 46958956358091443, [↩](#)
[↩](#)6648963000258614, 42704356524869523, 22101886973127150,
22978321700500060, 70937473165940755, 50965008984786859, 6648963000258614, [↩](#)
[↩](#)50965008984786859, 16338696893836718, 46958956358091443,
22101886973127150, 57742504562933385, 52883360311310107, 21494489470651665, [↩](#)
[↩](#)50965008984786859, 22101886973127150, 22978321700500060,
50965008984786859, 42704356524869523, 3013150194457141, 52883360311310107, [↩](#)
[↩](#)50965008984786859, 40911409245201015, 4486982623767293,
4486982623767293, 50965008984786859, 53005202974701704, 22101886973127150, [↩](#)
[↩](#)70937473165940755, 22101886973127150, 42704356524869523,
50965008984786859, 46958956358091443, 6648963000258614, 22978321700500060, [↩](#)
[↩](#)70937473165940755, 52883360311310107, 54892929530128079,
50965008984786859, 67471479331645926, 3013150194457141, 6648963000258614, [↩](#)
[↩](#)42704356524869523, 5895841574733797, 21494489470651665,

```

50965008984786859, 42704356524869523, 3013150194457141, 52883360311310107,
↪50965008984786859, 6648963000258614, 22978321700500060,
21494489470651665, 16231593379261352, 52883360311310107, 46958956358091443,
↪50965008984786859, 42704356524869523, 1773092908267621,
50965008984786859, 42704356524869523, 3013150194457141, 52883360311310107,
↪50965008984786859, 8829315279589188, 22978321700500060,
22101886973127150, 34258710235675678, 52883360311310107, 46958956358091443,
↪21494489470651665, 52883360311310107, 674596580030732
]

public_key_huyer = (72057545182682069, 65537)

break_code(public_key_huyer, ct_message_huyer)

```

I generated the p and q used to get this n with a prime_generator I coded. I can't believe it's still less than 20 digits. I can't imagine generating a primes in the 200 digit range! What's the answer to the universe?, Private Key = (72057545182682069, 3582151768559033)

```
[62]: print(Encode(public_key_huyer[0], public_key_huyer[1], "42!"))
```

```
[1035253653162361, 40911409245201015, 54892929530128079]
```

Additional Post

```
[63]: pt_post_hack = "Whats the fastest your code can break this key? I got down to 0.
↪56 milliseconds!"
```

```

p = 3697
q = 1201

public_key_post_hack = Find_Public_Key_e(p, q)
print(public_key_post_hack)

ct_post_hack = Encode(public_key_post_hack[0], public_key_post_hack[1],
↪pt_post_hack)
print(ct_post_hack)

```

```

(4440097, 1193)
[2308405, 79162, 4244727, 3355035, 2035913, 1723627, 3355035, 79162, 2914488,
1723627, 1054412, 4244727, 2035913, 3355035, 2914488, 2035913, 3355035, 1723627,
3134308, 2337190, 765334, 3083536, 1723627, 307890, 2337190, 454005, 2914488,
1723627, 307890, 4244727, 3282085, 1723627, 3159822, 3083536, 2914488, 4244727,
4046808, 1723627, 3355035, 79162, 3220943, 2035913, 1723627, 4046808, 2914488,
3134308, 2164157, 1723627, 3215733, 1723627, 843279, 2337190, 3355035, 1723627,
454005, 2337190, 3502570, 3282085, 1723627, 3355035, 2337190, 1723627, 1867558,

```

1347399, 3436266, 4358917, 1723627, 2948016, 3220943, 2671253, 2671253, 3220943, 2035913, 2914488, 307890, 2337190, 3282085, 454005, 2035913, 4089773]

```
[64]: # Start timer
begin = time.time()

break_code(public_key_post_hack, ct_post_hack)

# Stop timer
end = time.time()

print(f"Total time: {(end - begin) * 1000} milliseconds")
```

Whats the fastest your code can break this key? I got down to 0.56 milliseconds!, Private Key = (4440097, 936857)
Total time: 0.5373954772949219 milliseconds

1.14.6 10. Custom CODE Feature (10 points)

- The custom feature should be a “Stand Alone” feature at the end of the notebook. - If you need to use code from above, please copy the entire package and functions again as part of your feature so it can be tested independently. - You may remove the give code comments in this section. - Be sure to provide a demonstration showing that it works. See RSA Guide for more information and suggestions. - Include a narrative describing your project. - All custom features must include a narrative explaining the features and all code must be commented.

Do not forget the self-grading element.

Custom Code Feature: Decode Piazza *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

1.15 ##### Narrative

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 BeautifulSoup 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 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 BeautifulSoup 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, then 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 BeautifulSoup 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 verbosity, 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.

UPDATE To ensure the code operates entirely in Jupyter Lab environment, I have implemented the previously spoken of login feature. I have verified operation with my own login data (password temporarily changed).

1.16 ##### Demonstration

```
[65]: # Code requires selenium 4.25.0
      # Install required dependencies
```

```
[66]: pip install selenium
```

```
Requirement already satisfied: selenium in /opt/conda/lib/python3.10/site-
packages (4.26.1)
Requirement already satisfied: typing_extensions~=4.9 in
/opt/conda/lib/python3.10/site-packages (from selenium) (4.12.2)
Requirement already satisfied: trio~=0.17 in /opt/conda/lib/python3.10/site-
packages (from selenium) (0.27.0)
Requirement already satisfied: certifi>=2021.10.8 in
/opt/conda/lib/python3.10/site-packages (from selenium) (2022.12.7)
Requirement already satisfied: trio-websocket~=0.9 in
/opt/conda/lib/python3.10/site-packages (from selenium) (0.11.1)
Requirement already satisfied: websocket-client~=1.8 in
/opt/conda/lib/python3.10/site-packages (from selenium) (1.8.0)
Requirement already satisfied: urllib3[socks]<3,>=1.26 in
/opt/conda/lib/python3.10/site-packages (from selenium) (1.26.13)
Requirement already satisfied: sniffio>=1.3.0 in /opt/conda/lib/python3.10/site-
packages (from trio~=0.17->selenium) (1.3.0)
Requirement already satisfied: exceptiongroup in /opt/conda/lib/python3.10/site-
packages (from trio~=0.17->selenium) (1.2.2)
Requirement already satisfied: idna in /opt/conda/lib/python3.10/site-packages
(from trio~=0.17->selenium) (3.4)
```

Requirement already satisfied: outcome in /opt/conda/lib/python3.10/site-packages (from trio~=0.17->selenium) (1.3.0.post0)
 Requirement already satisfied: sortedcontainers in /opt/conda/lib/python3.10/site-packages (from trio~=0.17->selenium) (2.4.0)
 Requirement already satisfied: attrs>=23.2.0 in /opt/conda/lib/python3.10/site-packages (from trio~=0.17->selenium) (24.2.0)
 Requirement already satisfied: wsproto>=0.14 in /opt/conda/lib/python3.10/site-packages (from trio-websocket~=0.9->selenium) (1.2.0)
 Requirement already satisfied: PySocks!=1.5.7,<2.0,>=1.5.6 in /opt/conda/lib/python3.10/site-packages (from urllib3[socks]<3,>=1.26->selenium) (1.7.1)
 Requirement already satisfied: h11<1,>=0.9.0 in /opt/conda/lib/python3.10/site-packages (from wsproto>=0.14->trio-websocket~=0.9->selenium) (0.14.0)
 Note: you may need to restart the kernel to use updated packages.

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.

```
[67]: #####
#
# Author: Tyler Teichmann
# Date: 2024-10-19
# Purpose: These are the dependency functions from RSA that are required for
# the main function.
# Usage: no standalone usage
#
#####

# No change to function from initial implementation
def Decode(private_key, cipher_text):
    n = private_key[0]
    d = private_key[1]
    message = [FME(C, d, n) for C in cipher_text]
    message = Convert_Num(message)
    return message

# Optimized for bit shifting
def FME(b, n, m):
    result = 1

    while (n > 0):
        # use bitwise comparison instead of mod 2 to find LSB
        if n & 1:
            result = (result * b) % m
```

```

    b = (b * b) % m

    # Use arithmetic right shift instead of interger division
    n = n >> 1

    return result

# No chage to function from initial implementation
def EEA(a, b):
    s1, t1 = 1, 0
    s2, t2 = 0, 1

    while (s2 * a + t2 * b) > 0:
        q = (s1 * a + t1 * b) // (s2 * a + t2 * b)

        s_hat, t_hat = s1, t1
        s1, t1 = s2, t2
        s2, t2 = (s_hat - q * s2), (t_hat - q * t2)

    gcd = s1 * a + t1 * b

    return gcd, (s1, t1)

# No chage to function from initial implementation
def Find_Private_Key_d(e, p, q):
    x = (p - 1) * (q - 1)

    d = EEA(e, x)[1][0]

    while d < 0:
        d += x

    return d

# Modified to print an error if an integer is not in the range of char values
def Convert_Num(_list):
    _string = ''
    for i in _list:
        try:
            _string += chr(i)
        except ValueError:
            print(f"CharError converting {i} to char")
            _string += str(i)
    return _string

```

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.

```
[68]: #####  
#  
# Author: Tyler Teichmann  
# Date: 2024-10-19  
# Purpose: These are the dependency functions from RSA that are required for  
# the main function.  
# Usage: no standalone usage  
#  
#####  
  
# No change to function from code breaking implementation  
def factorize(n):  
  
    if not n & 1:  
        return 2  
  
    upper_bound = int(math.sqrt(n))  
    upper_bound += not upper_bound & 1  
  
    for i in range(upper_bound, 2, -2):  
        if not n % i:  
            return i  
  
    return -1  
  
# Included error checking if there are no prime factors of n  
# Some posted public keys are invalid numbers  
# raises an exception if no factors are found.  
def break_key(public_key):  
    p = factorize(public_key[0])  
  
    if p < 0:  
        raise Exception  
  
    q = public_key[0]//p  
  
    private_key = (public_key[0], Find_Private_Key_d(public_key[1], p, q))  
  
    return private_key
```


The following block of code begins new implementations. The first function gets login information from the user and returns the data as a list.

```
[69]: # This function gets login information from the user
def get_login():
    email = input("Email: ").strip()
    password = input("Password: ").strip()

    login = [email, password]

    return login
```

```
[70]: login_info = get_login()
login_info
```

```
Email: tyte8393@colorado.edu
Password: 1qaz@WSX3edc
```

```
[70]: ['tyte8393@colorado.edu', '1qaz@WSX3edc']
```

The next function leverages the python requests module to create a login session and extract the required cookies for use later on.

```
[71]: import requests

# This function creates a login session using the requests module
# Then extracts the cookie information for browser automation
def get_cookies(login_info):
    token_page = "https://piazza.com/main/csrf_token"
    login_page = "https://piazza.com/class"

    # login information
    payload = {
        "email": login_info[0],
        "password": login_info[1]
    }

    # create new request session
    session = requests.Session()

    # get request to CSRF token url and parse data
    token = session.get(token_page)
    payload["csrf_token"] = token.text.split('=')[1][1:-2]

    # Extract the required cookies from a login session
    response = session.post(login_page, data=payload)

    if response.status_code == 200:
```

```

        print("Successful Login")
    else:
        session.close()
        raise Exception("Error logging in. Response code: "
                        + response.status_code)

    cookies = response.cookies.get_dict()

    # close the session
    session.close()

    return cookies

```

```

[72]: cookies = get_cookies(login_info)
      cookies

```

Successful Login

```

[72]: {'AWSALB': '/K5aaLiH63qDwurLx8shCZJekfJTQJs2KELR3iswyNvZdtJUBe3lsAjATZCTKu5dmjAL
kA4GgaEx1gFv8tFQp5np4ZqWKfzuYPi8m4j9198qSZ4oxLmigInPIshx',
      'AWSALBCORS': '/K5aaLiH63qDwurLx8shCZJekfJTQJs2KELR3iswyNvZdtJUBe3lsAjATZCTKu5d
mjALkA4GgaEx1gFv8tFQp5np4ZqWKfzuYPi8m4j9198qSZ4oxLmigInPIshx',
      'piazza_session': '2.eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzU4NCJ9.eyJkYXRhIjp7IndoZW4iOi
jE3MzA3Nzg2NjAsInVzZXIiOiJtMGt2NDI2NDI2NDI2NDI2NDI2NDI2NDI2NDI2NDI2NDI2NDI2NDI2NDI2
2x60Xlkn2Z0NjM5MnlnOjA7bHo4eGtnMTM2cXo1ZXM6MDtsemswa2pocmJrNWE10jAiLCJ0YWciOiIiL
CJwZXJtIjowLCJlbWFpbCI6InR5dGU4MzZkZG9yYWRvLmVkdSI6ImhvbWU0IiwibG9nZ2luZ
19pbiI6dHJ1ZSwic2Vzc2l2b190b2t1b1I6InN0X2dtQm5leUxlb1FzZ1Q4OUxQZlpxIiwiaXhwaXJlc
yI6MTczMTk4ODI2MCwiY3JlYXRlZlZ9hdCI6MTczMDc3ODY2MH0sIm5iZiI6MTczMDc3ODY2MCwiZXhwa
XJlcYI6MTczMTk4ODI2MCwiaXNzIjoicGlnenphLmNvbSJ9.4FGVdLcactOP0oRpxXiDn1BCqXXGv7_h
f7njv86voSpk3B1Ngxx4QLE9ST1TYg93D9h0ESxYfuLE1xTf9K0Y1Wf4NZkNM6bdkmSZDvklb3Fd8nZl
oJ3NhRCaRQ-MxnVZ',
      'session_id': 'e44d79207dbc2b32f66da1c5e1d32d0666f350b14b786594'}

```

This block contains the function that requests the web page and downloads the HTML. Leveraging Selenium, this function takes the post number and cookies as input, requests the webpage, and downloads the html data for use by the scrape data function.

```

[73]: #####
#
# Author: Tyler Teichmann
# Date: 2024-10-19
# Purpose: This is a support package for the main application. This package is
# responsible for downloading the html using browser automation selenium.
# Ensure all required cookies are located in the cookies.txt file in the format
# name=cookie
#
# Referenced Documentation for Selenium at:

```

```

# https://www.selenium.dev/documentation/
#
#####

from selenium import webdriver

def download_data(post, cookies):
    # Set the homepage url
    home_page = "https://piazza.com"
    post_page = "https://piazza.com/class/lz9yd7ft6392yg/post/" + post

    options = webdriver.ChromeOptions()
    options.add_argument('--headless')
    # Start the web browser, load the cookies, the download the required page
    browser = webdriver.Chrome(options=options)

    browser.get(home_page)

    for key in cookies:
        browser.add_cookie({"name":key, "value":cookies[key]})

    browser.get(post_page)
    html = browser.page_source

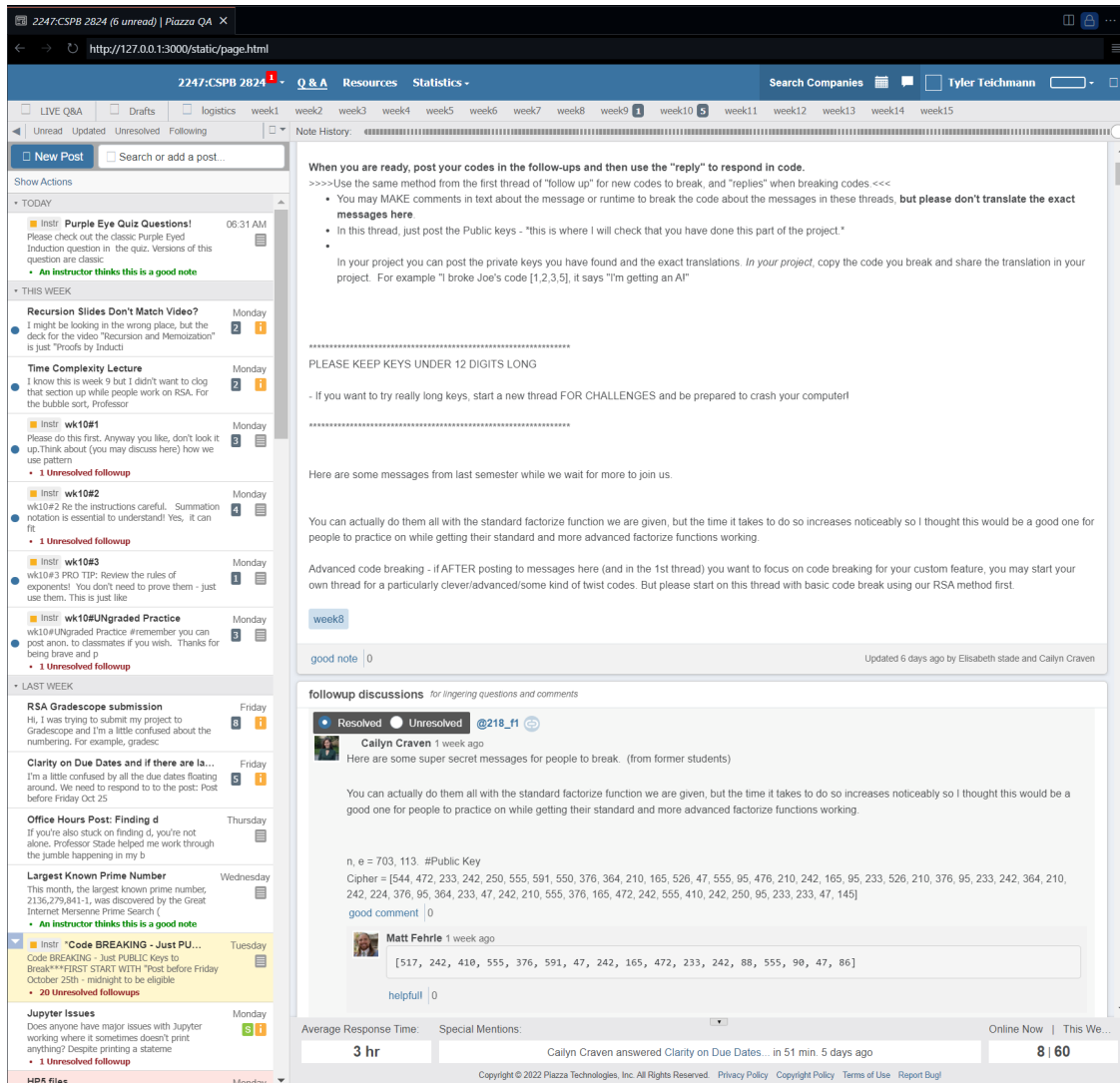
    # Close browser
    browser.quit()

    return html

```

```
[74]: html = download_data("218", cookies)
```

Downloaded HTML:



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.

```
[75]: # This function scrapes a string that contains the public key
# and returns that meassage as an intger list.
def scrape_public_key(key_string):
```

```
    # For loop setup
    length = len(key_string)
    public_key = []
    value = ""
```

```

on_number = False

# Iterate over every charactre in the string to find two integer values
# then add them to the public_key as integers
for i, c in enumerate(key_string):

    if c.isnumeric():
        on_number = True
        value += c
    elif on_number:
        public_key.append(int(value))
        value = ""
        on_number = False

    # if the loop ends on an integer add that integer to the key
    if (i == length - 1) and on_number:
        public_key.append(int(value))
        break

    # Ensure to break if public key is filled, some messages contain
    # many intger values due to sporatic html structure
    if len(public_key) > 1:
        break

    if len(public_key) != 2:
        raise Exception

return public_key

```

```

[76]: test_string = "n, e = 703, 113. #Public Key"

print(scrape_public_key(test_string))

```

[703, 113]

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.

```

[77]: # This function scrapes a string that contains the cypher text message
    # and returns that meassage as an intger list.
    def scrape_ct_message(message_string):

        # For loop setup
        ct_message = ""

```

```

in_message = False

# Iterate over all characters to find the start of the message
# then add it to the string until the message is closed
for i, c in enumerate(message_string):

    if c == '[':
        in_message = True
    elif c == ']':
        in_message = False
        break
    elif in_message:
        ct_message += c

# try to convert the message to integers
# any non-numeric chars will cause a ValueError
try:
    ct_message = [int(letter) for letter in ct_message.split(',')]
except ValueError:
    ct_message = str(ct_message)

return ct_message

```

```

[78]: test_string = """
    Cipher = [544, 472, 233, 242, 250, 555, 591, 550, 376, 364, 210,
    165, 526, 47, 555, 95, 476, 210, 242, 165, 95, 233, 526, 210, 376,
    95, 233, 242, 364, 210, 242, 224, 376, 95, 364, 233, 47, 242, 210,
    555, 376, 165, 472, 242, 555, 410, 242, 250, 95, 233, 233, 47, 145]
    """

print(scrape_ct_message(test_string))

```

```

[544, 472, 233, 242, 250, 555, 591, 550, 376, 364, 210, 165, 526, 47, 555, 95,
476, 210, 242, 165, 95, 233, 526, 210, 376, 95, 233, 242, 364, 210, 242, 224,
376, 95, 364, 233, 47, 242, 210, 555, 376, 165, 472, 242, 555, 410, 242, 250,
95, 233, 233, 47, 145]

```

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.

```
[79]: #####
#
# Author: Tyler Teichmann
# Date: 2024-10-19
# Purpose: This is a support package for the main application. This package is
# responsible for scraping the webpage for the cipher text and replacing it
# with the plain text message using BeautifulSoup 4.
# Error checking limited at this time.
#
# Referenced Documentation for BeautifulSoup at:
# https://beautiful-soup-4.readthedocs.io/en/latest/#
#####

import re
from bs4 import BeautifulSoup
# import rsa

def scrape_data(html):
    # Create initial html parser on passed input
    soup = BeautifulSoup(html, 'html.parser')

    # Select all elements with the followup and row class
    # ignoring those with the comment class
    posts = soup.select(".followup.row:not(.comment)")

    for post in posts:

        # These are all the main threads under the intial posting
        # May be redudant
        response = post.find(
            class_="render-html-content overflow-hidden latex_process"
        )

        # This uses regular expressions to narrow down any content in the
        # intial post to the element that contains the public key removing
        # any white space and converting to a string,
        # i.e. n, e ####, ##### # Public key ...
        #
        # This regular expression is highly spesific and will likely cause
        # issues when posts do not follow the required format.
        key_string = str(
            response.find(
                string=re.compile("^[NnEe] *=? * [0-9]* *[,.]? * [NnEe]")
            )
        )
```

```

    )
).strip()

try:
    public_key = scrape_public_key(key_string)
except Exception:
    post_number = post.find(class_="post_number_copy_link")
    print(f"""
        FormattingError: Post {post_number.get_text()} no key found.
        """)
    continue

# This uses regular expressions to narrow down any content in the
# initial post to the element that contains the public key removing
# any white space and converting to a string,
# i.e. n, e ####, ##### # Public key ...
#
# This regular expression is highly specific and will likely cause
# issues when posts do not follow the required format.
message_string = str(
    response.find(
        string=re.compile(re.escape '[') + "[0-9, ]+" + re.escape(']'))
    )
).strip()

ct_message = scrape_ct_message(message_string)

# For time purposes, only decode messages with n less than 20 digits
if public_key[0] <= 100000000000000000000:
    try:
        private_key = break_key(public_key)
    except Exception:
        post_number = post.find(class_="post_number_copy_link")
        print(f"""
            KeyError: Post {post_number.get_text()}
            no factors of n found.
            """)
        continue
    pt_message = Decode(private_key, ct_message)
    response.string = pt_message

# These are all the replies to the main threads
replies = post.select(
    ".followup_reply .render-html-content.overflow-hidden.latex_process"
)

# Iterate over all the replies attempting to decode each

```



```

for reply in replies:

    # Each reply contains only the cypertext so it is much easier to
    # extract the message
    ct_reply_message = (reply
                        .get_text()
                        .strip()
                        .replace("[", "")
                        .replace("]", ""))

    # Try to convert the message to integers
    # any non-numeric chars will cause a ValueError
    try:
        ct_reply_message = [
            int(letter) for letter in ct_reply_message.split(',')
        ]
    except ValueError:
        ct_reply_message = str(ct_reply_message)
        reply.string = ct_reply_message
        continue

    # Decode the messages using the public key from that thread
    pt_reply = Decode(private_key, ct_reply_message)
    reply.string = pt_reply

# Write all changes to the html document.
with open("code_breaking.html", "wb") as html:
    html.write(soup.prettify("utf-8"))

```

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

```

[80]: #####
#
# Author: Tyler Teichmann
# Date: 2024-10-19
# Purpose: The purpose of this application is to download the Piazza post
# and decode all the encrypted posts.
# Usage: python app.py
#
#####

# from download_html import download_data
# from scrape_html import scrape_data

```

```

def main():
    post = "218"

    # get login information from user
    login_info = get_login()

    try:
        cookies = get_cookies(login_info)
    except Exception:
        return 201

    # Download data from the piazza web page
    html = download_data(post, cookies)

    # Scrape the Data and reload the page
    scrape_data(html)

if __name__ == "__main__":
    main()

```

Email: tyte8393@colorado.edu

Password: 1qaz@WSX3edc

Successful Login

```

CharError converting 2245617 to char
CharError converting 4030207 to char
CharError converting 2985957 to char
CharError converting 2088034 to char
CharError converting 3795099 to char
CharError converting 2495247 to char
CharError converting 3795099 to char
CharError converting 2088034 to char
CharError converting 2762567 to char
CharError converting 4030207 to char
CharError converting 3011876 to char
CharError converting 2088034 to char
CharError converting 1734377 to char
CharError converting 4388074 to char
CharError converting 3740747 to char
CharError converting 2413427 to char
CharError converting 3011876 to char
CharError converting 2088034 to char
CharError converting 3550777 to char
CharError converting 2567462 to char
CharError converting 2495247 to char
CharError converting 2762567 to char

```

CharError converting 2088034 to char
CharError converting 4030207 to char
CharError converting 2495247 to char
CharError converting 3345887 to char
CharError converting 2088034 to char
CharError converting 1816867 to char
CharError converting 4388074 to char
CharError converting 2079429 to char
CharError converting 1255630 to char
CharError converting 2088034 to char
CharError converting 2061773 to char
CharError converting 3011876 to char
CharError converting 2040788 to char
CharError converting 2567462 to char
CharError converting 2413427 to char
CharError converting 3345887 to char
CharError converting 3011876 to char
CharError converting 2088034 to char
CharError converting 4030207 to char
CharError converting 3011876 to char
CharError converting 2088034 to char
CharError converting 3795099 to char
CharError converting 2495247 to char
CharError converting 3795099 to char
CharError converting 2762567 to char
CharError converting 2088034 to char
CharError converting 3740747 to char
CharError converting 3011876 to char
CharError converting 2762567 to char
CharError converting 2088034 to char
CharError converting 2413427 to char
CharError converting 2413427 to char
CharError converting 2985957 to char
CharError converting 3345887 to char
CharError converting 2952917 to char

KeyError: Post @218_f54
no factors of n found.

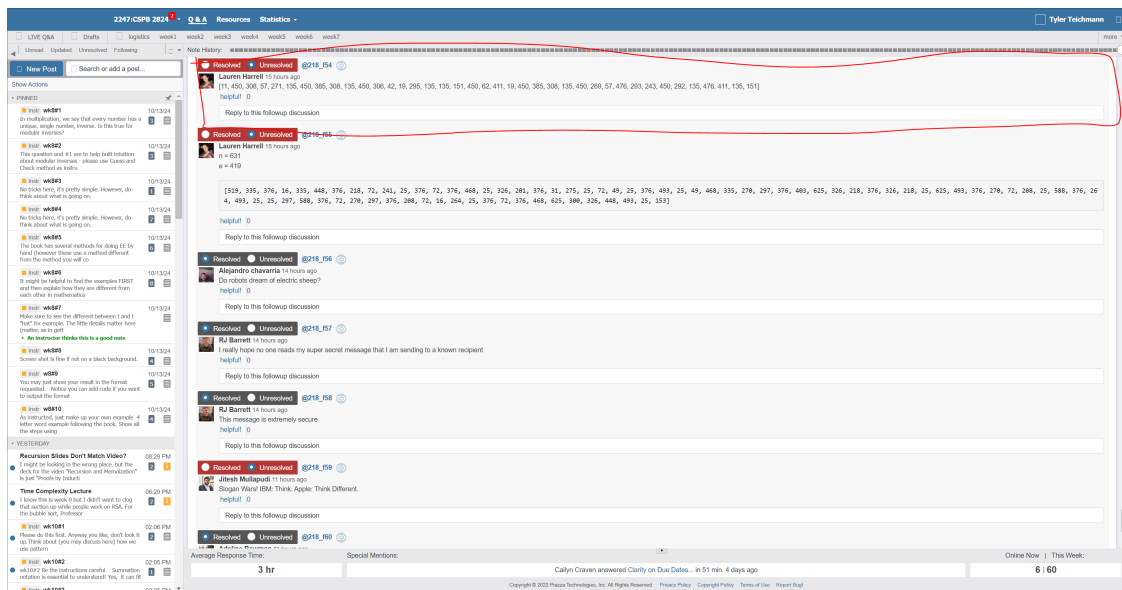
FormattingError: Post @218_f55 no key found.

FormattingError: Post @218_f63 no key found.

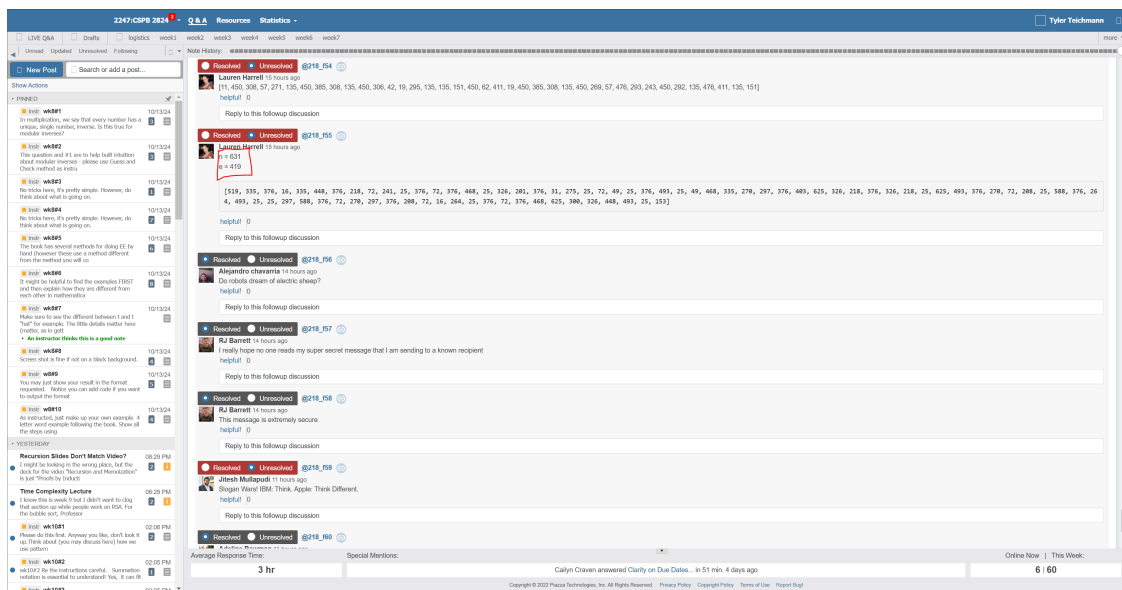
Example page:

CharError:

FactorError: (public key n is prime)



KeyError: (cannot extract public key due to formatting)



Please use the grading guide below to self grade your Custom Feature.

GRADING FOR CUSTOM FEATURE:

10 pts Wow. It is amazing. Shows initiative and originality. You did something extra special and pushed yourself. You went beyond all expectations You broke the rules in a creative way. Your coding and commenting is exceptional. Excellent

7-9 points - It is GOOD! You were a “self starter.” You did everything requested. All expectations met. You did a very good job. Good use of commenting. Shows mastery of skills.

5-6 points - Okay. Minimum requirements. Commenting weak.

0-4 points - It is not finished. Does meet objectives. Did not follow directions. Chose not to do this part (which is a totally fine choice)

1.17 Self grading of custom feature

Here in this block, please rate what score out of 8 your custom feature merits given the criteria above:

- For example, if you are brand new to programming, using a main function could be an amazing feature - tell me how this helped you move to a new level.
- In what ways did you push yourself? Did you try something you've never done before?
- Alternatively, if you just did not have the time to do the custom feature, this is actually just fine and a reasonable choice. No judgment.

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.