Aristotle University of Thessaloniki

Faculty of Sciences

SCHOOL OF INFORMATICS

Cryptography Project

Panagiotis Chatzipanagiotou AEM: 63

January 29, 2021

Contents

Summary 3

Subject 1 4

Subject 2 4

Subject 3 6

Subject 4 9

Subject 5 9

Subject 6 11

Subject 7 11

Subject 8 12

Subject 9 15

Subject 10 17

Subject 11 18

Subject 12 18

Subject 13 20

Subject 14 22

Subject 15 24

Subject 16 25

Subject 17 **25**

Subject 18 **26**

Subject 19 **26**

Subject 20 **29**

Excercise 3.1 **34**

Excercise 3.4 **35**

Excercise 3.6 **35**

Excercise 3.17 **35**

Excercise 3.24 **37**

Excercise 3.25 **37**

Excercise 3.26 **37**

Excercise 3.36 **38**

Excercise 3.49 **38**

Excercise 3.99 **39**

Summary

This paper contains the collected documentation of the Cryptography Project that was part of the final examination for CNSS106 Cryptography for the 20-21 period. The first part, up to Subject 20, is the main project and the second part, up to 3.99, contains the answers to ten number theory exercises. When it was announced, after a cursory examination, it seemed impossible and even after a few subjects where solved it still seemed unrealistic. The same feelings came about again when it was time to figure out LaTeX. In the end, though, everything came together. The result is definitely not perfect but the feeling of achievement is thoroughly satisfying.

It is important to mention that I had ZERO knowledge of python before starting this project. This will definitely be apparent to someone who is experienced in that language especially in the early answers. As i started tackling the project from the beginning to the end, my skill with Python shows an apparent evolution where i start using functions and more "pythonic" ways of solving things. I feel it is only fitting, as the cryptographic themes are getting more complex that the quality of the code follows close.

Special thanks are in order for my father who helped me with some editing tips, my friend Georgia for helping out with the English syntax and Mr. Draziotis for the template and of course the lessons.

Subject 1

For this exercise we had to create two(2) different programs, each one pertaining to one of the two sub-queries. Bellow we are going to quickly highlight some parts of it to show the methodology we followed.

i) This program initializes the variables, then asks for the phrase that we want to encrypt and finally the encryption key. The key itself is repeated internally by the program as many times as needed so as to match the size of the message that we want to encrypt. Afterwards the program encrypts and prints out the encrypted message and then decrypts it and prints out the original. This is done in order to make sure the procedure worked correctly both ways.
[See: rc4_final.py]

ii) This program is a bit more complex in the sense that there are more things we must take care off to make sure it functions properly. First the variables are initialized and then we get prompted to input the text that we want encrypted. It is important to only use upper case letters and to not use any spaces. Afterwards the program splits the text into letters and changes those letters into bits using the provided table. Then a random OTP is created and used to encrypt the bits. Using the same table we used before, the encrypted bits are turned into characters, merged together and then printed out. Finally, the process is reversed instead of just showing the original message, in order to make sure our algorithm is working correctly. This means that the encrypted text is broken into characters, "translated" to bits using the tables, decrypted using the OTP, "translated" again to characters, merged together and finally the program prints the original message.
[See: otp_final.py]

Subject 2

This is a very interesting exercise in the nature of cryptographic algorithms, specifically in the effectiveness of the encryption. The metric we are using is the Avalanche Effect which, in an effective cryptographic algorithm, is the difference between the original text and the cipher-text. More specifically, the bigger the difference, the more effective the algorithm. In this case we have created four(4) separate programs in order to highlight any possible differences we might notice while testing. In every case we perform the same test, which is to create two messages each different by only one(1) bit, and then encrypt them using the same parameters and comparing the bit difference of the results. We do that for as many pairs as we want(>30) and print out the median difference between them. The results are discussed below:

i) We have created two very simple programs using the "pycryptodome" library to implement AES. The program only asks how many pairs we want to try out. After we input a number the "for" loop in the main part of the program creates a number, then shifts one(1) bit, makes sure that the difference between them is one(1) bit using the "bincount" function we have included, then encrypts those two messages using the AES ECB implementation of pycryptodome, counts the difference in bits between the encrypted messages and finally presents us with an average difference between all the repetitions. The typical difference is around 118 bits.
[See: AES_ECB_final.py]

The second program is pretty much the same except it includes the use of an initialization vector which is necessary for the implementation of AES CBC. In this case we occasionally notice a slightly higher difference averaging around 128 bits.
[See: AES_CBC_final.py]

ii) For the second part of the exercise we chose the Blowfish algorithm which also has different modes of operation like AES, so again we made one program for ECB and one for CBC.
We are going to be using the "pycryptodome" library which includes both implementations of Blowfish and the results we get are a bit less divergent this time around. The Blowfish ECB implementation averages 73 bits of difference.
[See: Blowfish_ECB_final.py]

Once again the CBC shows a stronger Avalanche Effect than its counterpart with an average of 96 bits of difference.
[See: Blowfish_CBC_final.py]

It is clear that the Avalanche Effect is apparent in all implementations but there is a strong indication that AES is much better suited as a cryptographic algorithm as far as that effect is concerned.

Subject 3

This was one of the hardest but personally one of the most enjoyable exercises. The program attempts to be as verbose as possible in order to make the cryptanalysis much more clear than it can be described here. The method we used was the Kasiski attack. The solution is not elegant but it gets results.
[See: Vigenere_final.py]
Kasiski had noticed that sometimes there were bigrams and trigrams that were separated by exact multiples of the key-length which means that they would be encrypted in the same way. So firstly our program separates the text in bigrams and trigrams and then calculates their occurrence and distance. We check the numbers we have and try to find multiples of one specific number which would be "k", our key-length.

This is one of the hardest parts and it requires a bit of trial and error. In this case we took the highest occurring bigrams and trigrams and decided it is probably number 7 because it is the most typical divisor of the distances. In the program this is the occurrence of y after estimating key probability.
Having possibly found our key we can now use the letter occurrence chance in the English language and try to approximate which letter of the key was subbed for the letter "E". This is done thanks to the fact that the text was big enough to contain statistically significant occurrences of every letter.
Every step of the procedure is quite clear in the program up to the point that we get a final suggestion for a key , the word "EMPEROR".
Using that word we verify that it is actually correct and we get rewarded with the captivating monologue in the end of Charlie Chaplin's "The Great Dictator" which, with the addition of spaces and periods goes as follows:

"I'M SORRY, BUT I DON'T WANT TO BE AN EMPEROR. THAT'S NOT MY BUSINESS. I DON'T WANT TO RULE OR CONQUE ANYONE. I SHOULD LIKE TO HELP EVERYONE, IF POSSIBLE, JEW, GENTILE, BLACK MAN, WHITE. WE ALL WANT TO HELP ONE ANOTHER. HUMAN BEINGS ARE LIKE THAT. WE WANT TO LIVE BY EACH OTHER'S HAPPINESS NOT BY EACH OTHER'S MISERY. WE DON'T WANT TO HATE AND DESPISE ONE ANOTHER. IN THIS WORLD THERE IS ROOM FOR EVERYONE. AND THE GOOD EARTH IS RICH AND CAN PROVIDE FOR EVERYONE. THE WAY OF LIFE CAN BE FREE AND BEAUTIFUL, BUT WE HAVE LOST THE WAY. GREED HAS POISONED MEN'S SOULS, HAS BARRICADED THE WORLD WITH HATE, HAS GOOSE STEPPED US INTO MISERY AND BLOODSHED.WE HAVE DEVELOPED SPEED, BU WE HAVE SHUT OURSELVES IN. MACHINERY THAT GIVES ABUNDANCE HAS LEFT US IN WANT. OUR KNOWLEDGE HAS MADE US CYNICAL. OUR CLEVERNESS HARD AND UNKIND. WE THINK TOO MUCH AND FEEL TOO LITTLE. MORE THAN MACHINERY WE NEED HUMANITY. MORE THAN CLEVERNESS WE NEED KINDNESS AND GENTLENESS. WITHOUT THESE QUALITIES LIFE WILL BE VIOLENT AND ALL WILL BE LOST. THE AEROPLANE AND THE RADIO HAVE BROUGHT US CLOSER TOGETHER. THE VERY NATURE OF THESE INVENTIONS CRIES OUT FOR THE GOODNESS IN MEN, CRIES OUT FOR UNIVERSAL BROTHERHOOD, FOR THE UNITY OF US ALL. EVEN NOW MY VOICE IS REACHING MILLIONS THROUGHOUT THE WORLD, MILLIONS OF DESPAIRING MEN WOMEN AND LITTLE CHILDREN, VICTIMS OF A SYSTEM THAT MAKES

MEN TORTURE AND IMPRISON INNOCENT PEOPLE. TO THOSE WHO CAN HEAR ME I SAY DO NOT DESPAIR. THE MISERY THAT IS NOW UPON US IS BUT THE PASSING OF GREED, THE BITTERNESS OF MEN WHO FEAR THE WAY OF HUMAN PROGRESS. THE HATE OF MEN WILL PASS AND DICTATORS DIE AND THE POWER THEY TOOK FROM THE PEOPLE WILL RETURN TO THE PEOPLE. AND SO LONG AS MEN DIE LIBERTY WILL NEVER PERISH. SOLDIERS, DON'T GIVE YOURSELVES TO BRUTES, MEN WHO DESPISE YOU, ENSLAVE YOU, WHO REGIMENT YOUR LIVES, TELL YOU WHAT TO DO, WHAT TO THINK AND WHAT TO FEEL. WHO DRILL YOU, DIET YOU, TREAT YOU LIKE CATTLE, USE YOU AS CANNON FODDER. DON'T GIVE YOURSELVES TO THESE UNNATURAL MEN, MACHINE MEN WITH MACHINE MINDS AND MACHINE HEARTS! YOU ARE NOT MACHINES! YOU ARE NOT CATTLE! YOU ARE MEN! YOU HAVE THE LOVE OF HUMANITY IN YOUR HEARTS! YOU DON'T HATE! ONLY THE UNLOVED HATE, THE UNLOVED AND THE UNNATURAL! SOLDIERS DON'T FIGHT FOR SLAVERY! FIGHT FOR LIBERTY! IN THE SEVENTH CHAPTER OF ST. LUKE IT IS WRITTEN: "THE KINGDOM OF GOD IS WITHIN MAN". NOT ONE MAN NOR A GROUP OF MEN BUT IN ALL MEN! IN YOU! YOU, THE PEOPLE HAVE THE POWER. THE POWER TO CREATE MACHINES. THE POWER TO CREATE HAPPINESS. YOU, THE PEOPLE HAVE THE POWER TO MAKE THIS LIFE FREE AND BEAUTIFUL, TO MAKE THIS LIFE A WONDERFUL ADVENTURE. THEN, IN THE NAME OF DEMOCRACY LET US USE THAT POWER, LET US ALL UNITE LET US FIGHT FOR A NEW WORLD, A DECENT WORLD THAT WILL GIVE MEN A CHANCE TO WORK, THAT WILL GIVE YOUTH A FUTURE AND OLD AGE A SECURITY. BY THE PROMISE OF THESE THINGS BRUTES HAVE RISEN TO POWER. BUT THEY LIE. THEY DO NOT FULFIL THAT PROMISE. THEY NEVER WILL. DICTATORS FREE THEMSELVES BUT THEY ENSLAVE THE PEOPLE. NOW LET US FIGHT TO FULFIL THAT PROMISE. LET US FIGHT TO FREE THE WORLD, TO DO AWAY WITH NATIONAL BARRIERS, TO DO AWAY WITH GREED, WITH HATE AND INTOLERANCE. LET US FIGHT FOR A WORLD OF REASON, A WORLD WHERE SCIENCE AND PROGRESS WILL LEAD TO ALL MEN'S HAPPINESS SOLDIERS IN THE NAME OF DEMOCRACY, LET US ALL UNITE!

Subject 4

In order to decrypt the message of this exercise we created a table containing each of its letters encrypted with all three letters of the key:

Decryption

CIPHERTEXT	K	E	Y
A	**P**	V	B
J	Y	**E**	K
Z	O	U	**A**
B	Q	W	**C**
P	**E**	K	Q
M	**B**	H	N
D	S	Y	**E**
L	A	**G**	M
H	W	C	**I**
Y	**N**	T	Z
D	**S**	Y	E
B	Q	**W**	C
T	**I**	O	U
S	H	N	**T**
M	B	**H**	N
F	U	**A**	G
D	**S**	Y	E
X	**M**	S	Y
T	**I**	O	U
Q	F	**L**	R
J	Y	**E**	K

PEACE	BEGINS	WITH	A	SMILE
KEYYK	KYEYKK	EKYE	E	KKKEE

Subject 5

This is another very interesting and fun exercise. Our objective is to brute-force a locked compressed file. The program is pretty straightforward if only a bit inelegant. When we run the program, the zip file and the dictionary are loaded. The program then attempts to extract the zip using a "for" loop until it is able to do so. It usually takes about one minute and the answer pops up on the screen.

The password is "**secret**"

[See: BruteForce_final.py]

```
import zipfile
import sys

wordlist = "english.txt"
zip_file = "test_zip.zip"

zip_file = zipfile.ZipFile(zip_file)

with open(wordlist, "rb") as wordlist:
    for line in wordlist.readlines():
        try:
            zip_file.extractall(pwd=line.strip())
        except:
            continue
        else:
            print("The password is: ",line.decode().strip())
            sys.exit()
print("The password is not on this wordlist.")
```

Subject 6

This exercise is an implementation of the LFSR. It has many variables and many distinct steps that are thoroughly documented inside the code. In the first part of the exercise we have to implement a way to translate to and from the table that was provided so we initialize a table of letters and symbols and we create a list of bits from 00000 to 11111.

From the theory on LFSR we know that since we have the feedback function and n consecutive bits we can easily get our seed [1]. Our first step is to break the ciphertext into characters and translate them into bits. Then, by using our known encryption, ENC(ab) = .s, we translate "a" and "b" to bits and xOr them with "." and "s" in bits. This gives us a 10 bit stream but because in LFSR the input bit is a linear function of its previous state and this is the first state since we only encrypted 10 bits, this is also the key but in reverse.

Now that we have our key which is **1100101011**, we can produce the keystream by using the feedback function until we have enough bits for every character of the ciphertext. We separate the keystream in 5bit lengths since our characters are 5 bits long and we xOr them so we finally have our decrypted message. There are many intricacies that we had to take care for this to happen which are described in comments in the code.

Our final message is "**simplecanbeharderthancomplex**".

[See: LFSR_final.py]

Subject 7

For this exercise the biggest difficulty we met was finding a way to create a random number of specific bit length with which we would be able to perform bitwise oper-

[1] https://elearning.auth.gr/pluginfile.php/1282462/mod_resource/content/0/cource-2.pdf - P.20

ations and the method that was chosen was using a BitArray.
Through trial and error it was not that complex to realize that the difference in plaintext and ciphertext happens on the 4 left-most bits (Most Significant Bits). So in order to reverse the xOr operations of the encryption algorithm, another operation had to be included which made the final decryption algorithm:

$$m = c \oplus (c << 6 \oplus c << 12) \oplus (c << 10)$$

The program is fast so we can easily make sure the decryption algorithm is correct. **[See: decryptxor_final.py]**

Subject 8

We begin by solving the mathematical part of this exercise
i)

$$GCD(126048, 5050) \Rightarrow$$
$$126048 = 5050 \cdot x + 4 \Rightarrow$$
$$126048 = 5050 \cdot 24 + 48484 \Rightarrow$$
$$5050 = 4848 \cdot x + r \Rightarrow$$
$$5050 = 4848 \cdot 1 + 202 \Rightarrow$$
$$4848 = 202 \cdot x + r \Rightarrow$$
$$4848 = 202 \cdot 24 + 0$$

So our GCD is 202 and in order to calculate our Bezout Coefficients:

$$202 = 5050 - 4848 \Rightarrow$$

$$202 = 5050 - (126048 - 5050 \cdot 24) \Rightarrow$$

$$202 = 25 \cdot 5050 + (-1) \cdot 126048$$

So our Bezout Coefficients are 25 and −1

ii) First we have to check if the number is reversible in Z_{1001} so we must calculate the following:

$$GCD(1001, 809)$$

$$1001 = 809 \cdot x + r \Rightarrow$$

$$1001 = 809 \cdot 1 + 192 \Rightarrow$$

$$809 = 192 \cdot 4 + 41 \Rightarrow$$

$$192 = 41 \cdot 4 + 28 \Rightarrow$$

$$41 = 28 \cdot 1 + 13 \Rightarrow$$

$$28 = 13 \cdot 2 + 2 \Rightarrow$$

$$13 = 2 \cdot 6 + 1$$

$$GCD(1001, 809) = 1$$

So our number 809 is reversible in Z_{1001} and we can use the following:

$$13 + (-6) \cdot 2 = 1$$

$$28 + (-2) \cdot 13 = 2$$

$$41 + (-1) \cdot 28 = 13$$

$$192 + (-4) \cdot 41 = 28$$

$$809 + (-4) \cdot 192 = 41$$

$$1001 + (-1) \cdot 809 = 192$$

To exponentially build our calculation like this:

$$13 + (-6) \cdot 2 = 1 \Rightarrow$$

$$13 + (-6) \cdot [28 + (-2) \cdot 13] = 1 \Rightarrow$$

$$2 \cdot 13 + (-6) \cdot 28 = 1 \Rightarrow$$

$$2 \cdot [41 + (-1) \cdot 28] + (-6) \cdot [192 + (-4) \cdot 41] = 1 \Rightarrow$$

$$13 \cdot 41 + (-19) \cdot [192 + (-4) \cdot 41] = 1 \Rightarrow$$

$$89 \cdot 41 + (-19) \cdot 192 = 1 \Rightarrow$$

$$89 \cdot [809 + (-4) \cdot 192] + (-19) \cdot 192 = 1 \Rightarrow$$

$$89 \cdot 809 + (-375) \cdot 192 = 1 \Rightarrow$$

$$89 \cdot 809 + (-375)[1001 + (-1) \cdot 809] = 1 \Rightarrow$$

$$464 \cdot 809 + (-376) \cdot 1001 = 1$$

And finally we have our reverse which is 464

iii) For that calculation we are going to need the following numbers:

$$e = 100 \; m = 2 \; n = 101$$

$$100 \text{ in binary is: } 1100100$$

That number expressed in decimal is:

$$100 = 2^2 + 2^5 + 2^6$$

$$100 = 4 + 32 + 64$$

Which means that the following are true:

$$2^{100} \bmod 101 = (2^4 \cdot 2^{32} \cdot 264) \bmod 101$$

$$2^{100} \bmod 101 = (2^4 \bmod 101 \cdot 2^{32} \bmod 101 \cdot 264 \bmod 101) \bmod 101 \quad (1)$$

In the next step we get:

$$2^{32} \bmod 101 = (2^4 \cdot 2^4 \cdot 2^4 \cdot 2^4 \cdot 2^4 \cdot 2^4 \cdot 2^4 \cdot 2^4 \cdot) \bmod 32 = \ldots$$

$$\ldots = [2^4 \bmod 101 \cdot \ldots (6\ more) \cdot 2^4 \bmod 101] \bmod 101 = \ldots$$

$$\ldots = (16 \cdot 16 \cdot 16 \cdot 16 \cdot 16 \cdot 16 \cdot 16 \cdot 16) \bmod 101 = 4294967296 \bmod 101 = 68$$

Then, for the last one we have:

$$2^{64} \bmod 10 = (2^{32} \cdot 2^{32} \cdot) \bmod 101 = 2^{32} \bmod 101 \cdot 2^{32} \bmod 101 = 4624 \bmod 101 = 79$$

Now replacing the previous results to (1) we have the siple calculation:

$$(16 \cdot 27 \cdot 54) \bmod 101 = 85952 mod 101 = 1$$

iv) The final part of this exercise is basic yet quite important since it is used in many of the following exercises. A simple yet effective algorithm for calculating modulo exponent. In this iteration we have included some functions to make sure that the numbers we input cannot break the function. And it is fully functional if you need to try it. The answers are **8648** and **319**.

[See: Exponent_modulo_final.py]

Subject 9

i) First we need to calculate the chances of every possible combination:

$$\sum P(X, Y)(0, Y) = 3/7$$

$$\sum P(X, Y)(1, Y) = 2/7$$

$$\sum P(X, Y)(2, Y) = 2/7$$

$$\sum P(X, Y)(X, 0) = 3/7$$

$$\sum P(X, Y)(X, 1) = 2/7$$

$$\sum P(X, Y)(X, 2) = 2/7$$

The formula we have to use to calculate the H is:

$$H(x) = -\sum_{i=0}^{2} p_x(i) \cdot \log_2[p_x(i)]$$

Which in our case, for 3 different values becomes:

$$-(p_x(0) \cdot \log_2[P_x(0)] + (p_x(1) \cdot \log_2[P_x(1)] + (p_x(2) \cdot \log_2[P_x(2)])$$

Using the chances we calculated above we have:

$$H(X) = -3/7 \log 3/7 - 2/7 \log 2/7 - 2/7 \log 2/7 = 1.5566$$

$$H(Y) = -3/7 \log 3/7 - 2/7 \log 2/7 - 2/7 \log 2/7 = 1.5566$$

ii) Using the above results and applying them to our next formula we can calculate:

$$H(X, Y) = -\sum_{X}\sum_{Y} p(X, Y)\log_2 p(X, Y)$$

$$H(X, Y) = -\sum_{x\in X}(p(x, 0)\cdot\log_2[p(x, 0)] + p(x, 1)\cdot\log_2[p(x, 1)] + p(x, 2)\cdot\log_2[p(x, 2)])$$

$$H(X, Y) = -p(0, 0)\log_2 p(0, 0)\ldots - p(2, 2)\log_2 p(2, 2) = 2.52164$$

iii) We already calculated that $H(X, Y) = H(Y, X) = 2.52164$ so we can easily calculate the conditional entropies:

$$H(X|Y) = H(Y, X) - H(Y) = 2.52164 - 1.5566 = 0.965$$

$$H(Y|X) = H(X, Y) - H(X) = 2.52164 - 1.5566 = 0.965$$

iv) We have all we need to calculate:

$$I(X, Y) = H(X) + H(Y) - H(X, Y) = 1.5566 + 1.5566 - 2.52164 = 0.59156$$

v) Again we have all we need to calculate:

$$\rho = 1 - \frac{H(X|Y)}{H(x)} = 1 - \frac{0.965}{1.5566} = 0.38$$

vi) This was an interesting program that required some advanced mathematics to implement but was much easier after first practising the necessary functions. We have pre-loaded the table and when you run it , the necessary functions are calculated and the results are printed in the same way they were asked in the description.
[See: Entropy_final.py]

Subject 10

Precisely as mentioned in the description of this exercise, this is a "textbook" implementation of the RSA protocol. The program that was created has very good documentation in the comments so most of it's intricacies can be understood from there. The fast() algorithm from Subject 8 is used in this solution. The program initializes the ciphertext, asks for N and e and then proceeds to decrypt the message which reads: **"welcowe to the real world"**.
[See: textbookRSA_final.py]

Subject 11

This exercise was a more advanced version of the previous one. In this case we must use a specific attack vector. Once again, the program makes use of the fast() algorithm from Subject 8 and the documentation in the comments is very descriptive so we are just going to highlight a few spots. First of all, the cipher text required one more step to be able to be used because it had base64 encoding. After decoding it we initialize it in our program and then, like in the last exercise, we input N and e and our program proceeds to give us the decrypted message which reads: **"Just because you are a character doesn't mean that you have character"**.
[See: RSA_Wiener_final.py]

Subject 12

This exercise is pretty straightforward. The only issue that we met was during the third part when trying to create a safe prime. Using the mathematical formulas of Fermat's and Miller-Rabin's method we have created three different functions in the same program. Firstly the user is asked to input a number "n" representing the bits

of the number we want to generate. Then the functions produce a prime number by generating random numbers of "n" bits and applying each Primality Test to them until one passes. The comments in the code are quite descriptive so not a lot of documentation is required.

i)A 2048 bit number we have generated that passes Fermat's test is:

31255774634374650782003084310973725522303957113296431335450036769263353779226850898115316821637311666213823293452034471947963725178435705195642054908645535800434212962064763822995516352826196683231325618927944335983197906477507174564348542353577541610542932429668537150172922622427809234516993536702121285293280457008787672273827128817549133533628393469959357461943595212947351768339938737987220411209777011849021954742069304677300007239126048889968726510174001827816082567953962849044641016428410259385565259936760910466214570676845604941983820507870880589422964843698441258453903438739280657052443069674166863240999

ii) A 1300 bit number we have generated that passes Miller Rabin's test for 5 repetitions is:

13051917864901354546421869072016238362351577509747244047110417520775552964769374217179091732202383974875367488223695538563503637922233225506351356416828989582517357468477457251306421357985302033805376640794654810375545064710463227375068101299812103769649619423860226543630927233498576799561440975135864069271769187594032452009575429892507015113206645018192463029130454228192395529954058304 79

iii)In this case, while we were able to easily generate safe primes with less bits using our algorithm, the time it took to create longer-bit numbers was prohibitive for an older CPU. We suggest you take caution if you try doing that as your pc might get stuck. For reference our code took around 3 days to finally give us an answer.

We have to keep in mind that a safe prime is not just the number p that satisfies the two primality tests ($p =$ *prime* and $2 \cdot p + 1 =$ prime) but the number $2 \cdot p + 1$

An example of a 3000 bit safe prime is:

10166489032524783698659502818484232874651689204456190253910866286813815486593178543166848138419310248847143110086051940138550659254551916322941095196557802497753486815055444816814740227647318506221364289143707485877453811879215359038531842951199406859004786910343313347386524293842856334945598882330300646544582061819807524241462724332078703482572688307033806928894061696008044214359412587730044154486312387631775727362096488481371749831073882194208296511710926367632342342933205981884936383051431223656801453234257642719548742940116301502194146387154355672022479686326004541096163906027455642754689217698375965815197329301697953941031493402882849090875688303369127863195301312205853250440714168861754553865885137217097329069817370150774201278074004511775881102077914445722345698006550972755024530745447146635189345435850917325169898471400492329527803601127563320718592744844747019520481105352524701742 39

[See: Primality_final.py]

Subject 13

We have the following system of congruences:

$$x \equiv 9 (\mathrm{mod} 17)$$
$$x \equiv 9 (\mathrm{mod} 12)$$
$$x \equiv 13 (\mathrm{mod} 19)$$

All of which are of the general formula:

$$x \equiv b_i (\mathrm{mod} n_i)$$

In order to solve it we have to calculate the following:

$$N = n_1 \cdot n_2 \cdot n_3 = 17 \cdot 12 \cdot 19 = 3876$$
$$x = n_2 \cdot n_3 + n_1 \cdot n_3 + n_1 \cdot n_2$$
$$x = 12 \cdot 19 + 17 \cdot 19 + 17 \cdot 12$$
$$x = 228 + 323 + 204$$

So for the next step we have to calculate the following values of i:

$$\text{For: } x = 228 \bmod 17 = 7 \neq 9 \iff 228 \cdot i \bmod 17 = 9 \iff i = 11$$
$$\text{For: } x = 323 \bmod 12 = 11 \neq 9 \iff 323 \cdot i \bmod 12 = 9 \iff i = 3$$
$$\text{For: } x = 204 \bmod 19 = 14 \neq 13 \iff 204 \cdot i \bmod 19 = 13 \iff i = 5$$

So recalculating our equivalencies from above we have:

$$x = 228 \cdot 11 + 323 \cdot 3 + 204 \cdot 5 = 4497$$

$$x \equiv 621 (\mathrm{mod} 3876)$$

And finally: **x = 621**

Subject 14

This was a very interesting exercise introducing us to the concept of prime formulas also known as prime generating polynomials. We know that prime numbers are one of the most important concepts in cryptography so it does not surprise us that there has always been a very high demand for ways to consistently create big ones. Let's move on to the one presented in the description of this exercise.

a) It was easy to create a program that picks as many random integers from the range that was set and solve the function for them. Afterwards they were checked for primality with the Miller-Rabin method and by comparing the successful tests with the total number of tests we realized that this specific function produces prime numbers with around 40% success, which is quite impressive.
[See: RSApolynomial_final.py]

b) Before we begin we should mention that all prime numbers above 3 can be expressed as $6k \pm 1$ but that is not a polynomial so we move on with our caclulations.

Let's assume that a polynomial $Z(x)$ that produces prime numbers for every $n \in N$ actually exists. Then $Z(x)$ can be writen as:

$$Z(x) = \sum_{i=0}^{d} a_i x^i, a_i \in Z, 0 \leq i \leq d, a_d \neq 0 \text{ where d, the degree of } Z(x)$$

If our assumption is correct then there exists an $f(1) = q$ which is prime.
There also exists a $g(x) = Z(x) - q$ which is a polynomial of d degree therefore it can have at most d roots. (1)
Our q can be written as:

$$q = \sum_{i=0}^{d} a_i, a_i \in Z, 0 \leq i \leq d, a_d \neq 0 \ (2)$$

There also exists a $k \in N$ for which we have:

$$Z(1 + k \cdot q) = \sum_{i=0}^{d} a_i (1 + k \cdot q)^i \ (3)$$

Which is also prime for every $k \in N$
From (3) we have:

$$(3) \Longleftrightarrow \sum_{i=0}^{d} a_i \cdot \sum_{j=0}^{i} \binom{i}{j} (k \cdot q)^j$$

$$Z(1 + k \cdot q) = \sum_{i=0}^{d} a_i + \sum_{i=0}^{d} a_i \cdot \sum_{j=1}^{i} \binom{i}{j} (k \cdot q)^j$$

From (2) this becomes:

$$Z(1 + k \cdot q) = q + q \cdot \sum_{j=1}^{i} \binom{i}{j} (k \cdot q)^j$$

Which would mean that $Z(1 + k \cdot q)$ is a multiple of q.

So if it is a prime number then that means the only way for that to be true is if $Z(1 + k \cdot q) = q$ for all $k \in N$.

But that would mean that our (1) for $x = 1 + k \cdot q$ becomes $g(1 + k \cdot q) = 0$ for all $k \in N$.

So our g(x) must have infinite roots when it should have at most d, thus our initial assumption is a fallacy so there cannot be such a polynomial in the first place.

Subject 15

The article mentioned in the footnote was a lot of help in understanding the Rabin Cryptosystem as well as approaching the exercise [2]. We have $p = 5$, $q = 11$, $c = 14$ so $N = 5 \cdot 11 = 55$

For m_p:

$$m_p^2 \equiv 14 \bmod 5 \Rightarrow m_p^2 \equiv 4 \bmod 5 \Rightarrow m_p \equiv 3 \bmod 5 \rightarrow m_p = 3$$

For m_q:

$$m_q = c^{\frac{q+1}{4}} \bmod q \Rightarrow m_q = 5$$

By applying the Extended Euclidean Algorithm we can calculate y_p and y_q for which:

$$y_p \cdot p + y_q \cdot y = 1 \rightarrow y_p \cdot 5 + y_q \cdot 11 = 1$$

and we get:

$$y_p = -2 \, , \; y_q = 1$$

[2] https://cryptography.fandom.com/wiki/Rabin_cryptosystem

Then using the Chinese Remainder Algorithm and all the known variables on the following equations:

$$r = (y_p \cdot p \cdot m_q + y_q \cdot q \cdot m_p) \bmod n = 32$$

$$-r = n - r = 17$$

$$s = (y_p \cdot p \cdot m_q - y_q \cdot q \dot{m}_p) \bmod n = 27$$

$$-s = n - s = 28$$

And since we know that $m < 20$ we finally get that $m = 17$

Subject 16

This exercise had two steps. First we had to create a PGP - Certificate and send an unencrypted email with our public key to mr. Draziotis. Then, using his own public key, we had to send him a second email, only this time it had to be encrypted. The following is the **date** and the title of the email:

11/26/2020 - My Public Key - Panagiotis Chatzipanagiotou

And the answer we recieved was the following **hash**:

hF4DajrBo6DnsB0SAQdAOy3pnE0QQ0A8HDZgdtjbg2WggzHykWEL5KhN0
nFttA0ws/AGuElwwEQtizFnIRmjTUcSbVNr1XSpad8foHJwbustJUUI36zCnedeNNl/
Ws9a0m4Bt0jHCnOUWzhOBV9dxV1E6ZQKbY2wM56z9e6cxIVZxxtgx3WHWoW7
rd6/iBfjzg202YrsMe/SKpb247y/79/MCTVPfCmx7SxRX9fP82RuvwWDq4FS7gRonsq
EStnG7juZxqSisz3BRoIdx1Bvyg===TAC7

Subject 17

This exercise is a change of pace from the typical RSA encryption and decryption methods we have been using so far. This time we are going use the Chinese Remainder Theorem(CRT) in the decryption process as well as generate a few more numbers that are necessary for that method. The program is pretty straightforward and the documentation in the code is very thorough. When executed, the program first asks for the number of bits we want to use for the encryption. That number is used by the first function to generate the necessary numbers. Afterwards, we have to input a number that we would like to encrypt. The program then uses this number along with the numbers produced by the first function to encrypt the message using the second function which is similar to the way we encrypted with RSA in the previous exercises. Our final function decrypts the message using CRT and prints out our original number.

[See: RSA_CRT_final.py]

Subject 18

A unique exercise among traditional ones, more akin to a Capture the flag. After some introspection we finally thought about opening the provided .pdf with a text editor and hidden somewhere inside was a URL that did not fit in [3]. After visiting the challenge page and some creative thinking we caught the CHEMIST reference and tried to use the Periodic Table to "translate" our riddle. After getting He-Re-Ti-C it we knew we were on the right track and soon we unlocked the zip using #heretic!. [4]

The provided hash is: **9e94b15ed312fa42232fd87a55db0d39**

[3] https://cryptology.csd.auth.gr:8080/home/pub/15/

[4] When studying for the final exam we realised that the URL is also the title of the pdf (FoxitReader)

Subject 19

The first bash command we are going to use in openssl is the following:

```
$ echo | openssl s_client -servername cryptology.csd.auth.gr -connect \
cryptology.csd.auth.gr:8080 | \
sed -ne '/-BEGIN CERTIFICATE-/,/-END CERTIFICATE-/p' > certificate.crt
```

This command will export the public key from the given URL in a file named: **certificate.crt**.

We will then use the following command to extrapolate the modulus from our certificate:

```
openssl x509 -in certificate.crt -noout -modulus,
```

The resulting Modulus is:

AE59555CF9F1DDD29E6AEFF8646805FC8423C476184B5A895268B304911
211D27F66A1AB018700FFA526CF5BE05E8562EA967A572618A2CCBA91B632E
067EE3B5AEF05216DEA5B155D00257A6FD08992F8E67D0A130FABC51D659E
F7731A265D76567844A2E520E199715E9EDBBB80E0C1D3E619E7EFDA65C1AD
380348C4856A05E267AE961FB314AB77996D1488A4E597A0DB352A8A9118811
8DF5E9A7C1D10BA3B5901358AFB080110F70BF5B43931DD3532AAEBF0A44A
4881E671750F64F5C089F28273F8CE3DD0653FD47104346FDAEA1627EC3DE7A
18F262EE35BDE3647DC2E7A0D60339CF0EC41D14E438EBA94C9AC8FD5D154
E87258A10C6BEE8527D3

We are going to use this modulus in our final bash command which will translate it into an integer. The syntax of the command is:

```
echo 'ibase=16;AE59555CF9F1DDD29E6AEFF8646805FC8423C476184B5A89526
8B304911211D27F66A1AB018700FFA526CF5BE05E8562EA967A572618A2CCBA91B
632E067EE3B5AEF05216DEA5B155D00257A6FD08992F8E67D0A130FABC51D659EF
7731A265D76567844A2E520E199715E9EDBBB80E0C1D3E619E7EFDA65C1AD38034
8C4856A05E267AE961FB314AB77996D1488A4E597A0DB352A8A91188118DF5E9A7
C1D10BA3B5901358AFB080110F70BF5B43931DD3532AAEBF0A44A4881E671750F6
4F5C089F28273F8CE3DD0653FD47104346FDAEA1627EC3DE7A18F262EE35BDE364
7DC2E7A0D60339CF0EC41D14E438EBA94C9AC8FD5D154E87258A10C6BEE8527D3'
| bc.
```

Which finally produces our integer which is:

22009517030106990383763465105609899090941157494196382925533444156480589292450396529663642850138695190588855805260835175270086166355060564256148792439233592472232233056950905840539154677121486094877597472487300649250039699499067065703656740364634411636945242292939908153976592139191106268327154757211809894086104466154946604045715191337979340318964774322733014276440202233403512948667806285436385349767243073268946721052707444588507283401697641243190019700333343119537943616295162029028870293037484015297179234631401220439538481352300280727764959004584544520437625051512885404930600391580169572809550082972986630285267.

This integer is **2048 bits**. We know the exponent which is e = 65537 since it is given in the certificate, as well as our N which is the integer we found above. Using a very practical library called "owiener" we have created a tiny program to quickly check if the wiener attack works on this pair and we quickly find out that it does not produce any keys.

[See: Wienerkey_final.py]

Subject 20

i) A One-Time Pad (OTP) is a cryptographic system in which we generate a truly random key that is the same size or longer than the message we want to encrypt, we share it secretly and it is completely discarded after use. The encryption usually takes place by performing an XOR between each character of the plaintext and each character of the pad. The OTP is considered uncrackable if all four variables mentioned above are adhered to. Let us discuss what would happen if one of them, the one from which it gets it's name (One - Time), is ignored.

In cryptographic circles when discussing what happens if the OTP is used more than once, it is often colloquially called "Two-Time Pad"[1] or "Many-Time Pad"[2]. It has been thoroughly analyzed and there are many ways in which it can be dangerous. Here we are going to discuss the main way that one can exploit the opportunity provided by reusing an OTP or Keystream.

Historically one of the best known cases about the dangers of reusing a keystream is the VENONA project by the Signal Intelligence Agency(SIA), which was later absorbed by the National Intelligence Agency(NSA). In this well documented counterintelligence project, American cryptanalists of the SIA noticed that a lot of encrypted telegraphs sent by their Soviet counterparts, were encrypted using the same keystreams[3]. This operation, that lasted from 1943 to 1980 and was declassified partially in 1995, managed to partially or fully decrypt over 3000 messages. Sadly, only the decrypted messages were released and neither the cryptanalytic methods or the ciphertexts.

The typical method that repeated use of an OTP or keystream can be exploited is quite simple. It requires one to have two ciphertexts created by the same pad which we will call C_1 and C_2 and the language in witch the original message was written. Firstly, one has to pick a common word that they believe is included in the messages,

for example, the word **"the"** (W). Then one must create a "crib" or cheat-sheet by performing $C = C_1 \oplus C_2$. Finally, by performing: W $\oplus$ C, for each 3 letter part of C, we check each result and wherever we get a result that "looks like" English, there is a strong chance that one of the messages had the word "the" in that three letter part. If that proccess is repeated enough times portions of the plaintexts can be recreated and eventually, if the messages are long enough, both the whole messages can be recreated.

This method was further refined after 1996 when Dawson and Nielsen added specific use cases for common characters, for example if two characters XORed to 0 they were equal to space which is a pretty common occurrence in plaintexts, as well as using dictionaries of common words of various lengths to speed up the procedure[4]. In later years even more refined methods have been designed that have almost total success, using more complex methods and what is known as a *language model*[1] where it is definitively demonstrated that one should never reuse keystreams.

ii) A very important part of some cryptographic functions is the concept of padding. It is the practice of adding data in some specific part of the original message prior to encrypting. It may be necessary for many reasons from hiding the size of the plaintext from the attacker, to more effectively encrypting a smaller message.

Optimal Asymetric Encryption Padding or OAEP is a padding scheme introduced by Bellare and Rogaway[5] in 1995 that is mainly used with RSA in order to transform it from a deterministic into a probabilistic encryption scheme. It works using the concept of a trapdoor function which is a function that is easy to calculate one-way but very hard to calculate the other way.

In OAEP we use two functions:
G which is a hash function that outputs **g** bits
H which is a hash function that outputs **h** bits
We also use a one-use random string which in cryptography is called a nonce:
r randomly generated nonce of **g** bits

The way that OAEP pads our message before the encryption is the following:

m is our initial message and m_t is a temporary version of the message
We add 0's to our message so the length of our temporary message is equal to **g**

$$m_t = m + [0] * (g - len(m))$$

Then we produce the two parts of the final message that we will encrypt with RSA. The first part p_1 is our temporary message XORed with the output of our **G** hash of our nonce **r**

$$p_1 = m_t \oplus G(r)$$

And the second part is our nonce, **r** XORed with the output of our **H** hash of our first part

$$p_2 = r \oplus H(p_1) = r \oplus H(m_t \oplus G(r)$$

Finally we concatenate the two parts and we get the final message **M** which we can encrypt with RSA

$$M = p_1||p_2 = m_t \oplus G(r)||r \oplus H(m_t \oplus G(r)$$

The way that OAEP unpads our message after the decryption is the following:

In order to recover the nonce, **r** we hash the first part p_1 and XOR it with the second part p_2

$$r = H(x) \oplus p_2$$

And finally we hash the nonce **r** and then XOR it with the first part p_1 to get the original message

$$m = G(r) \oplus p_1$$

iii) This is a very common yet interesting discussion in cryptographic circles. There are different dangers with each philosophy so each one should be observed in the context of it is more vulnerable. Let's take a look at the two systems and try to reach a conclusion.

In our examples we are going to use an attempted communication between Alice (A) and Bob (B) and a third party called Eve(E) that is trying to subvert the communication in some way. We are going to signify signing a message with a lower case exponent and an encryption with an uppercase exponent for example:
A message sent by Alice to Bob that is first signed by Alice and then encrypted with Bob's public key is

$$A \rightarrow B \ : \ [(message)^a]^B$$

First we are going to focus on the *first encrypt then sign* approach. In this case the biggest danger is the possibility of an attacker claiming the message's contents as their own[6]. If Eve is able to block Bob from receiving Alice's message, she can

simply remove Alice's signature, add her own and claim Alice's message as hers.

$$A \rightarrow B \ : \ [(\textit{this is mine})^{B}]^{a}$$

$$E \rightarrow B \ : \ [(\textit{this is mine})^{B}]^{e}$$

This is obviously not intended and a major risk in any context.

After this use-case one might think that using a *sign and encrypt* approach is the only way but let us observe a vulnerability that might arise in this case. If Eve is watching the communication between Alice and Bob she might, at some point, intercept a message that can be used again as is to subvert another flow of communication. For example in one case Alice might ask Bob the question "Do you like pizza?" and Bob might answer "Yes". Both messages are signed first and encrypted with each other's public key afterwards.

$$A \rightarrow B \ : \ [(\textit{Do you like pizza})^{a}]^{B}$$

$$B \rightarrow A \ : \ [(\textit{Yes})^{b}]^{A}$$

If Eve knew the contents of that communication or, in more advanced attack schemes, could decrypt Bob's answer we might face the following case. Alice asks Bob "Can I trust Eve?". Eve intercepting that communication could simply forward Bob's answer from before andgain her trust.[6]. This is clearly not intended.

$$A \rightarrow B \ : \ [(\textit{Can I trust Eve})^{a}]^{B}$$

$$E \rightarrow A \ : \ [(\textit{Yes})^{b}]^{A}$$

Another way this could be exploited is by forwarding a signed message to a third party pretending you are the first party. For example Alice sends Eve the message "I hate you" signed and encrypted with Eve's public key. Eve decrypts it, encrypts it with Bob's public key and forwards it to him making him think that Alice hates him[7].

$$A \rightarrow E \ : \ [(I\ hate\ you)^a]^E$$

$$E \rightarrow B \ : \ [(I\ hate\ you)^a]^B$$

The discussion around an approach that would circumvent these and the many more attack vectors is ongoing. Here we will see some suggestions.

The first one is the concept of triple wrapping(ESE)[7] which suggests an Encrypt - Sgin - Encrypt model *e.g* $A \rightarrow B \ : \ [[(example)^B]^a]^B$. This protocol is CPU intensive and has the added issue of requiring a primary decryption to take place before making sure that the message is signed, leaving you open to attacks hiding under the first encryption.

Another way to implement triple wrapping is the Sign-Encrypt-Sign[7][6] model in which the typical Sign-Encrypt approach adds another signature in the end to make sure that it cannot be reused as we saw in our example above *e.g* $A \rightarrow B \ : \ [[(example)^a]^B]^a$. It has many benefits compared to the previous suggestion, being much less CPU intensive and including an outer layer signature.

In this writer's opinion, the security needs of any given situation should be taken into account before choosing a method, depending of the possible costs of message misuse. Both ESE and SES have their merits but if we do not limit ourselves and include other security measures like Assymetric Encryption[8] we are able to use equally or even more secure means of communication, at lower CPU cost and intensiveness. After all, in the words of cryptographer Bruce Schneier, "Security is a process, not a product.".

Excercise 3.1

For numbers that are even: $\sqrt{n} = 2 \cdot m \longrightarrow n = (2m)^2 = 4 \cdot m^2 \equiv 0 \bmod 4$.

For numbers that are odd: $\sqrt{n} = 2 \cdot m + 1 \longrightarrow n = (2m+1)^2 = 4 \cdot m^2 + 4 \cdot m + 1 \equiv 1 \bmod 4$.

We see that we can never get a result of 3, so $4 \cdot k + 3$ is never a perfect square.

The numbers in the range: $11, 111, 111\ldots\ldots, 111\ldots$ are factors of $4 \cdot k + 3$, so, as we just proved, none of them can be a perfect square.

Excercise 3.4

The sum of any two consecutive integers is always odd. The number 2^m , $m \geq 2$ is always an even number thus it can never be the sum of two consecutive integers.

Excercise 3.6

The factorization of $x^{10} + 1$ is :

$$(x^4 + x^3 + x^2 + x + 1) \cdot (x^4 - x^3 + x^2 - x + 1) \cdot (x + 1) \cdot (x - 1)$$

The factorization of $x^5 + 1$ is:

$$(x^4 - x^3 + x^2 - x + 1) \cdot (x + 1)$$

Which are also factors of $x^{10} + 1$

This means that $x^5 + 1 \mid x^{10} + 1$ is always true.

Excercise 3.17

For $m \geq 1$ we construct the following **Binomial Coefficient**

$$M = \binom{2m+1}{m+1}$$

For which we know that

$$\binom{a}{b} = \frac{a \cdot (a-1) \cdot \ldots \cdot (a-b+1)}{b!}$$

So we can do the following calculations and get

$$M = \binom{2m+1}{m+1} = \frac{(2m+1)2m \cdot \ldots \cdot [(2m+1) - (m+1) + 1)]}{1 \cdot 2 \cdot \ldots \cdot m \cdot \cancel{(m+1)}} = \binom{2m+1}{m}$$

From the above we can say that

$$2M = \binom{2m+1}{m} + \binom{2m+1}{m+1}$$

$$2M < \sum_{x=0}^{2m+1} \binom{2m+1}{x} = 2^{2m+1}$$

$$2M < 2^{2m+1} \leftrightarrow M < 2^m$$

If a number **p** that is prime that satisfies $m + 1 \leq p \leq 2m + 1$ then P divides the following product:

$$P \mid (2m+1) \cdot m \cdot (2m-1) \cdot \ldots \cdot (m-2) = M$$

but it does not divide m! So $\mathbf{p}$ divides $\mathbf{M}$ and the following is true

$$\prod_{m+1 \leq p \leq 2m+1} p \leq M < 4^m$$

which means that

$$\prod_{p \leq x} p \leq 4^{x-1}$$

Excercise 3.24

We know that $\gcd(n, a) = 1 \geq n \cdot a$. Because $n|a$, b must be the product of n, so $n|b$ is true.

Excercise 3.25

We did not manage to solve this exercise.

Excercise 3.26

Using the Euclidian Algorithm we get:

$$540 = 1 \cdot 315 + 225$$

$$315 = 1 \cdot 225 + 90$$

$$225 = 2 \cdot 90 + 45$$

$$90 = 2 \cdot 45 + 0$$

So our **GCD is 45**. Now to calculate our α and β we do the following:

$$225 = 540 - 315$$

$$90 = 315 - 225$$

$$45 = 225 - 2 \cdot 90$$

$$45 = 255 - 2 \cdot (315 - 225)$$

$$45 = 3 \cdot 255 - 2 \cdot 315$$

$$45 = 3 \cdot (540 - 315) - 2 \cdot 315$$

$$45 = 3 \cdot 540 - 5 \cdot 315$$

So our α and β are **3** and **-5**

Excercise 3.36

We will try to prove that there is a general rule that:

$$a \equiv b(mod n) \Rightarrow a^n \equiv b^n(mod n^2)$$

If $a \equiv b(mod n)$ there exists an integer i that $a = b + n \cdot i$

Raising both sides to the power of n and using the Binomial Theorem we get:

$$a^n = (b + n \cdot i)^n \Rightarrow a^n = b^n + n(n \cdot i)b^{n-1} + (n \cdot i)^2(\ldots) \Rightarrow a^n = b^n + n^2(\ldots) \Rightarrow a^n \equiv b^n(mod n^2)$$

So our assumption is valid and if we use n = 3 we get:

$$a^3 \equiv b^3(mod 3^2)$$

Excercise 3.49

If **n** is a composite number then we know that there are two integers **a** and **b** that:

$$n = ab\ ,\ 1 < a \leq b < n$$

If we assume that

$$a > \sqrt{n}$$

then since

$$\sqrt{n} < a \leq b$$

We can multiply $a > \sqrt{n}$ with $b > \sqrt{n}$ and we get $ab > \sqrt{n}\sqrt{n} \Rightarrow ab > n$ which is a fallacy.

So $a \leq \sqrt{n}$ and n must have a prime divisor a_1 that is also a prime divisor of n so we get:

$$a_1 \leq a \leq \sqrt{n}$$

Excercise 3.99

The hardest part of this exercise was to find exactly what a fermat pseudoprime[5] and a strong pseudoprime[6] is and then differentiate them to find the correct way to calculate it. Combining some of the resources found in the footnotes we managed to formulate a simple and elegant solution to the problem using a short python program. The result is **25**.

[See: Fermat_pseudoprime.py]

[5] https://mathworld.wolfram.com/FermatPseudoprime.html
[6] https://mathworld.wolfram.com/StrongPseudoprime.html

References

[1] Joshua Mason, Kathryn Watkins, Jason Eisner and Adam Stubblefield. *A Natural Language Approach to Automated Cryptanalysisof Two-time Pads*. Johns Hopkins University, 2006.

[2] D. E. Denning. *The Many-Time Pad: Theme and Variations*. 1983 IEEE Symposium on Security and Privacy, pp. 23-23, doi: 10.1109/SP.1983.10010, Oakland, CA, USA, 1983.

[3] R. L. Benson and M. Warner. *Venona: Soviet Espionage and the American Response*. 1939-1957.Central Intelligence Agency, Washington, D.C., 1996.

[4] E. Dawson and L. Nielsen. *Automated cryptanalysis of xor plaintext strings*. Cryptologia, 20(2):165–181, April 1996.

[5] M. Bellare and P. Rogaway. *Optimal Asymmetric Encryption – How to encrypt with RSA*. Springer-Verlag, 1995.

[6] D. Davis. *Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML*. 65-78, 2001

[7] Pooja Lal Mundaniya and Naveen Choudhary *High Security by using Triple Wrapping Feature and their Comparison*. ISSN: 0975-4172& Print ISSN: 0975-4350. College of Technology and Engineering, India 2015.

[8] H. Krawczyk *The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)*. 2139. 10.1007/3-540-44647-8_19, 2001.