Aristotle University of Thessaloniki

Faculty of Sciences

SCHOOL OF INFORMATICS

Project in Cryptography

Melina Zikou

3357

June 10, 2021

Contents

Abstract

This paper reports and documents the solutions of the exercises given in the course of Fundamentals of Cryptography. Many exercises require coding scripts, and so for this purpose Python code was developed. Programming files, after the submission deadline, will be available on GitHub at https://github.com/melinazik/crypto.

Exercise 1

(i) RC4 stream cipher is implemented in the file "ex1a.py".

According to RC4 algorithm, initially an array S is indexed from 0 to 255 consisting of the integers 0,..., 255, which is then permuted. For encryption and decryption a keystream of bytes K is produced which is XORed with the plain text byte by byte.

Key Scheduling Algorithm (KSA): the permutation of the array S is initialized. In the code segment below, first, the array S is initialized to the identity permutation. S is then processed 256 times, mixing the bytes of the key at the same time.

```python
def KSA(key):
    keyLen = len(key)
    key = [ord(x) for x in key]
    S = [x for x in range(256)]
    j = 0
    for i in range(256):
        j = (j + S[i] + key[i % keyLen]) % 256
        S[i], S[j] = S[j], S[i]
    return S
```

Pseudo random generation algorithm (PRGA): Once the vector S is set by the KSA, we iterate through array S, swiping an element in S at index i, with another element in S at index j. Then, the next element in the encrypted text is the element of S at the index calculated by (S[i] + S[j]) mod 256. At each iteration, i is recalculated as (i + 1) mod 256, and j is recalculated as (j + S[i]) mod 256.

Encryption: For the encryption of the plain text, each character of the plain text is XORed with the value S[(S[i] + S[j]) mod 256] as explained above.

This operation appears in the PRGAEncryption method, which is used by the encrypt method as shown below.

```python
def encrypt(key, plainText):
    S = KSA(key)
    cipherText = PRGAEncrypt(S, plainText)
    return listToString(cipherText)
```

Decryption: The decryption is performed the same way as the encryption, since XOR with given data is an involution. Thus, each character of the cipher text is XORed with the value S[(S[i] + S[j]) mod 256] as explained above.

This operation appears in the PRGADecryption method, which is used by the decrypt method as shown below.

```python
def decrypt(key, cipherText):
    S = KSA(key)
    cipherList=cipherText.split(' ')
    plainText = PRGADecrypt(S, cipherList)
    return plainText
```

The results of the RC4 encryption and decryption algorithm are the following:

PLAIN TEXT: MISTAKES ARE AS SERIOUS AS THE RESULTS THEY CAUSE
KEY: HOUSE

ENCRYPTED: 15DDA667FE6ADD1A49167A12A40CF54B42D4CD
7014E2EF5C0A87A3010B47EB7919777DDB5786
14557294AEAFD9621C8B19

DECRYPTED: MISTAKES ARE AS SERIOUS AS THE RESULTS THEY CAUSE

(ii) One - Time Pad (OTP) encryption algorithm is implemented in the file "ex1b.py".

First, letters of the alphabet and characters where matched to their binary representation as given by the table 1.

```
table = {'A' :0b00000, 'B' :0b00001, 'C' :0b00010, 'D' :0b00011, 'E' :0b00100,
         'F' :0b00101, 'G' :0b00110, 'H' :0b00111, 'I' :0b01000, 'J' :0b01001,
         'K' :0b01010, 'L' :0b01011, 'M' :0b01100, 'N' :0b01101, 'O' :0b01110,
         'P' :0b01111, 'Q' :0b10000, 'R' :0b10001, 'S' :0b10010, 'T' :0b10011,
         'U' :0b10100, 'V' :0b10101, 'W' :0b10110, 'X' :0b10111, 'Y' :0b11000,
         'Z' :0b11001, '.' :0b11010, '!' :0b11011, '?' :0b11100, '(' :0b11101,
         ')' :0b11110, '-' :0b11111}
```

Once the plain text is read from input, each character is converted to it's binary representation. We iterate through the plain text, and in order to isolate the required bit sequence, an initially binary zero valued parameter, is shifted by 5 bits to the left, and then ORed with the bit representation of the character.

```
def convertToBits(plainText, table):
    bitStream = 0b0
    n = 1
    for c in plainText.upper():
        n = n + 1
        bitStream = bitStream << 5
        bitStream = (bitStream | table[c])
    return bitStream
```

Once the random One - Time Pad key is generated with the use of otpKey function, the key and the now converted plain text are XORed, and a new sequence of bits is generated.

Next on, the bits of the cipher text are transalated to their character representation according to the table, with the use of binaryToString method. In this function, a mask (binary 11111) is used, to isolate the last 5 bits of the stream, and then the stream is shifted 5 bits to the right to find the next 5 - bit sequence. During this process, all the now converted characters, one by one are added to an array, which is then reversed, because the accessing of the values happened from the last one to the first one.

```python
def binaryToString(binary, length, table):
    mask = 0b11111
    charList = []
    for i in range(length):
        last = binary & mask
        binary = binary >> 5
        for element in table:
            if (table[element] == last):
                charList.append(element)
    result = charList[::-1]
    return result
```

The decryption process is the same as the encryption, since XOR with given data is an involution.

The results of the OTP encryption and decryption algorithm are the following:

<u>PLAIN TEXT:</u> **YOU HAVE PERFORMED AN ACT OF TREASON AGAINST THE KING OF BEES!!**

<u>ENCRYPTED:</u> **EQ-HWMMHWDM-TYVNVCC-DDUL.QWCBXTMXXUXU(K(FCPB-SXIVESA**

<u>DECRYPTED:</u> **YOUHAVEPERFORMEDANACTOFTREASONAGAINSTTHE KINGOFBEES!!**

Each time OTP uses another random key. As a result the encrypted text is changed when the same message is encrypted more than one times, as shown below.

<u>PLAIN TEXT:</u> **YOU HAVE PERFORMED AN ACT OF TREASON AGAINST THE KING OF BEES!!**

<u>ENCRYPTED:</u> **S?BQEJLB?BH?V)MX!QBI-NFPPFHW!LA.-TYQ-UMH MJAPQMADYJZS**

<u>DECRYPTED:</u> **YOUHAVEPERFORMEDANACTOFTREASONAGAINSTTHE KINGOFBEES!!**

Exercise 2

The code implementation of this exercise can be found in the file "ex2.py".

Both **AES** and **Blowfish** algorithms belong to the category of Block Ciphers. Block ciphers operate on blocks of plaintext one at a time to produce blocks of ciphertext. The main difference between a block cipher and a stream cipher (Ex. 1) is that block ciphers are stateless, while stream ciphers maintain an internal state which is needed to determine which part of the keystream should be generated next.

The **Avalanche effect**, which is tested in AES and Blowfish algorithms in two separate modes of operation, is a property of cryptographic algorithms. More specifically, if an input is changed slightly, in this situation, changed by one bit, the output changes significantly.

Modes of operation of block ciphers, describe how to repeatedly apply a cipher's single - block operation to securely transform amounts of data larger than a block. Some modes require a unique binary sequence, called an initialization vector (IV), which has to be random and non - repeating. The modes of operation used for the scope of this project are:

Electronic codebook (ECB): The message is divided into blocks, and each block is encrypted separately.

Cipher block chaining (CBC): Each block of plaintext is XORed with the previous ciphertext block before being encrypted. This way, each ciphertext block depends on all plaintext blocks processed up to that point. To make each message unique, an initialization vector (IV) is used in the first block.

To test the avalanche effect, we first have to create a number of pairs of messages, and randomly change one of their bits. As seen in the code segment below, first 'x' message is created of length 64 * 2 bits, and is then copied to 'y' message, which is changed by one bit, at a random index. Then the messages are converted to their byte representation and a 16 - byte random key is created.

```python
messages = 35
for j in range(messages):
    x = []
    for i in range(0, 64*2):
        x.append(random.randint(0, 1))
    y = x.copy()
    randIndex = random.randint(0, 64*2 - 1)
    changeBit(y, randIndex)
```

Next on, AES and Blowfish algorithm are implemented with the use of Crypto.Cipher python library. Two methods were created in a similar way to each other, called 'aes' and 'blowfish', executing all the necessary steps for encryption and decryption.

```python
def aes(key, xBytes, yBytes):
    cipher = AES.new(key, AES.MODE_ECB)
    # ECB Mode
    countECB = ECBMode(key, cipher, xBytes, yBytes)

    # CBC Mode
    blockSize = AES.block_size
    iv = Random.new().read(blockSize)
    cipher = AES.new(key, Blowfish.MODE_CBC, iv)
    countCBC = CBCMode(key, blockSize, cipher, iv, xBytes, yBytes)
    return countECB, countCBC
```

The methods 'countECB' and 'countCBC' are the same for both AES and Blowfish.Both messages of each message - pair are encrypted using ECB or CBC mode of operation accordingly, and then the difference of bits between those messages is counted.

```python
def ECBMode(key, cipher, xBytes, yBytes):
    xEnc = cipher.encrypt(xBytes)
    yEnc = cipher.encrypt(yBytes)
    return differentBits(xEnc, yEnc)

def CBCMode (key, blockSize, cipher, iv, xBytes, yBytes):
    xEnc = iv + cipher.encrypt(xBytes)
    yEnc = iv + cipher.encrypt(yBytes)
    return differentBits(xEnc, yEnc)
```

There were used 35 pairs of messages with 128 bits of length. The average bit difference as generated by the program can be seen below.

AES - ECB MODE: 65.71428571428571
AES - CBC MODE: 61.82857142857143

Blowfish - ECB MODE: 31.742857142857144
Blowfish - CBC MODE: 63.714285714285715

Exercise 3

The code implementation of this exercise can be found in the file "ex3.py".

Vigenère Cipher is an encryption method, that uses a simple form of polyalphabetic substitution. A polyalphabetic cipher is based on substitution, using multiple substitution alphabets. The encryption of the plain text is done with the use of Vigenère table. The table consists of 26 rows and 26 columns. The alphabet is written in each row shifted one letter to the left compared to the previous alphabet. The use of each alphabet - row depends on a repeating key.

In order to break Vigenère Cipher, without knowing the key (or the length of the key) used to encrypt the message, we first have to determine the length of the key.

We iterate through the length of the cipher text, placing the cipher text at the top, and the plain text shifted by one position at the bottom. In each iteration, the cipher text is shifted by one position. We then calculate the number of coincidences of letters in each index with the following condition:

```python
for j in range(textLen):
    if(text[j] == text[(j + i) % textLen]):
        coincidences[i] += 1
```

The number of coincidences goes up sharply when the bottom message is shifted by a multiple of the key length, because then the adjacent letters are in the same language using the same alphabet. So, the key length equals the index of the maximum coincidences found.

In the table below there is a visualization of the shifting of the message. The coincidences found in the 19 - letter long sample text, are marked with red. When the text is larger in length, the coincidences of the letters are much more different numbers, and the difference between the keylenght - coincidences and the non - keylength - coincidences (as described above) are clearer.

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**
M	Y	H	S	I	F	P	**F**	G	**I**	**M**	U	C	E	X	**I**	**P**	R	K
	M	Y	H	S	I	F	P	F	G	I	M	U	C	E	X	I	P	R
		M	Y	H	S	I	**F**	P	F	G	I	M	U	C	E	X	I	P
			M	Y	H	S	I	F	P	F	G	I	M	U	C	E	X	I
				M	Y	H	S	I	F	P	F	G	I	M	U	C	E	X
					M	Y	H	S	**I**	F	P	F	G	I	M	U	C	E
						M	Y	H	S	I	F	P	F	G	**I**	M	U	C
							M	Y	H	S	I	F	P	F	G	I	M	U
								M	Y	H	S	I	F	P	F	G	I	M
									M	Y	H	S	I	F	P	F	G	I
										M	Y	H	S	I	F	**P**	F	G
											M	Y	H	S	**I**	F	P	F
												M	Y	H	S	I	F	P
													M	Y	H	S	I	F

Once the key length is found, we can determine the characters of the key. This procedure is based on frequency analysis. A table with frequencies of all the letters of the English alphabet is used to conduct frequency analysis.

For each letter in the cipher text we have to find the maximum frequency of the letter in accordance to another letter (the letter of the key). The method that implements the search of the key characters can be seen below.

One character of the key at a time is selected to be found. A frequency value of a letter is calculated by multiplying the frequency of the letter in the English Alphabet, and the frequency of that letter in the cipher text. For each letter of the alphabet (for the character of the key we are searching) the maximum frequency value wins, and the corresponding letter is the missing character. We loop through all the characters of the key, according to the key length.

```python
def getKey(text, keyLen, frequency):
    textLen = float(len(text))
    key = ''
    for i in range(0, keyLen):

        textFrequency = {} # array with frequency of each letter in the cipher
 text
        maxVal= -1
        maxPosition = 0
        rolledArray = frequency

        # for each letter in the alphabet we have to find the
        # max frequency of the letter in accordance to another letter
        # [letter of key - letter of cipher text]
        for j in range(0, 26):
            if(j > 0):
                rolledArray = np.roll(rolledArray, 1)

            for k in range(0, 26):
                # set initial frequency of each letter to 0

                textFrequency[chr(k + ord('A'))] = 0

            # find the frequencies of each position of the key.
            # start at the letter position of the key to be found
            # and loop through the cipher text
            k = i
            while(k < textLen):
                letter = text[k]
                textFrequency[letter] += 1 / textLen * 100  # set frequency
                k += keyLen

            # Sum the frequency values of textFrequency * rolledArray
            # textFrequency -> frequency in cipher textFrequency
            # rolledArray   -> frequency of english alphabet
            sumVal = 0
            for k in range(0, 26):
                sumVal += textFrequency[chr(k + ord('A'))] * rolledArray[k]

            # max of sumVal's during a parse of the alphabet wins
            # maxPosition -> position of the letter found
            if(sumVal > maxVal):
                maxVal = sumVal
                maxPosition = j

        # add the letter to the key
        key += chr(maxPosition + ord('A'))

    return key
```

The key of the encrypted text given is **EMPEROR** and the decrypted text is:

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 UP ON 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!

Exercise 4

The numerical representation of the cipher text "AJZBPMDLHYDBTSMFDXTQJ" is 1, 10, 0, 2, 16, 13, 4, 12, 8, 25, 4, 2, 20, 19, 13, 6, 4, 24, 20, 17, 10.

For each letter in the cipher text, we will use E(5), K(11) and Y(25) to decrypt the message and find which combination produces a meaningful plain text.

The decryption method is the following: substract the key letter from the cipher text letter and perform modulo 26. For example if the cipher text character is 'A' = 1 and the key character is 'E' = 5, we have: $1-5=-4$ and then $-4 mod 26 = 22$, which corresponds to V.

In the following tables, all possible decryptions can be seen. Letters marked with red, mark the correct letters of the plain text.

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

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

The message is **PEACE BEGINS WITH A SMILE** and the key is **KEYYKKYEYKK EKYEEKKKEE**.

Exercise 5

The code for this exercise can be found in the file "ex5.py".

A dictionary attack is a method of breaking into a password-protected computer or server by systematically entering every word in a dictionary as a password. It can also be used in an attempt to find the key necessary to decrypt an encrypted message or document.

In our case, a password for a locked zip file has to be found. In order to do that, we need to test all the words from a given dictionary, to decide which one is the correct one.

Once the neccessary python libraries are imported, the target zipfile and the dictionary are specified. The zipfile object is initialized, and the words in the dictionary are counted.

The dictionary is opened, and each word in it is read and tried as a password to unlock the zip file. Reading the entire line of the word list will add the new line character in the end, which will be marked as wrong by the zip file. To solve this problem, strip() method is used to remove white spaces and new lines.

The method extractall() seen in the code segment below, raises an exception whenever the password is incorrect. Once the exception is raised, we pass on to the next password. Otherwise the correct password is printed, along with progress information: percentage of the dictionary parsed, number of correct password, total number of words.

The code used can be seen below:

```python
dictionary = "english.txt"
zipFile = "locked.zip"

zipFile = zipfile.ZipFile(zipFile)
# count the number of passphrases
words = len(list(open(dictionary, "rb")))

with open(dictionary, "rb") as dictionary:
    for word in tqdm(dictionary, total=words, unit="word"):
        try:
            # extractall => exception whenever pwd incorrect
            #            => if correct - print pwd
            # strip => avoid newlines

            zipFile.extractall(pwd=word.strip())
        except:
            continue
        else:
            print("\nPassword:", word.decode().strip())
            exit(0)
print("Password not found.")
```

The correct password is **secret**. The password was found after 267936 trials (words) out of 394740. The percentage of the parsed dictionary is 68%.

Exercise 6

The code for this exercise can be found in the file "ex6.py".

The code development was inspired by Konstantinos Draziotis' python project "lfsr_project.py" (3/3/2016), found in https://github.com/drazioti/python_scripts/tree/master/numtheory

A feedback shift register is made up of two parts: a shift register and a feedback function. The shift register is a sequence of bits. Each time a bit is needed, all of the bits in the shift register are shifted 1 bit to the right. The new left-most bit is computed as a function of the other bits in the register. The output of the shift register is 1 bit, often the least significant bit. The period of a shift register is the length of the output sequence before it starts repeating. The simplest kind of feedback shift register is a linear feedback shift register (LFSR). The feedback function is simply the XOR of certain bits in the register; the list of these bits is called a tap sequence.

In this exercise, there is a 10-bit shift register, and the feedback function used is $x^{10} + x^9 + x^6 + 1$. The goal is, given the encryption of a known plain text, to decrypt the text : i!))aiszwykqnfcyc!?secnncvch.

Given that ENC(ab) = .s, we can easily find the seed used to encrypt the above message. Plain text 'ab' and cipher text '.s' are XORed, producing the output of the LFSR function, which is, as already mentioned, a list of the least significant bits, of the shift register.

The initial input on the LFSR funtion is the seed, which is tapped in the $10^{th}, 9^{th}, 7^{th}$ and 6^{th} place according to the feedback function. The output of the 'ab' XOR '.s' bit representation according to the table give, is the seed used inverted. This happens because, each time the sum - XOR is calculated, it is placed at the start of the list, giving away each time the last digit.Once the seed is calculated, we can decrypt the given message, applying the LFSR algorithm.

The steps followed to decrypt the message, after the calculation of the seed are explained below.

Step 1

Convert decrypted message to its 5-bit representation according to the table.

```
def convertToBits(plainText, table):
    bitStream = 0b0
    n = 1
    for c in plainText.upper():
        n = n + 1
        bitStream = bitStream << 5
        bitStream = (bitStream | table[c])
    return bitStream
```

The table is initilized as:

```
table = {'A' :0b00000, 'B' :0b00001, 'C' :0b00010, 'D' :0b00011, 'E' :0b00100,
         'F' :0b00101, 'G' :0b00110, 'H' :0b00111, 'I' :0b01000, 'J' :0b01001,
         'K' :0b01010, 'L' :0b01011, 'M' :0b01100, 'N' :0b01101, 'O' :0b01110,
         'P' :0b01111, 'Q' :0b10000, 'R' :0b10001, 'S' :0b10010, 'T' :0b10011,
         'U' :0b10100, 'V' :0b10101, 'W' :0b10110, 'X' :0b10111, 'Y' :0b11000,
         'Z' :0b11001, '.' :0b11010, '!' :0b11011, '?' :0b11100, '(' :0b11101,
         ')' :0b11110, '-' :0b11111}
```

Step 2

Calculate the LFSR output. The seed previously found is used, along with the feedback function in the form $[0, 0, 0, 0, 0, 1, 1, 0, 1, 1]$, and the length of the cipher text (length of plain text * 5).

```python
def lfsr(seed, feedback, bits):
    newFeedback = []
    for i in range(len(feedback)):
        if 1 in feedback:
            newFeedback.append(feedback.index(1))
            feedback[feedback.index(1)] = 0
    seed = deque(seed)
    output = []
    for i in range(bits):
        xor = sumxor([seed[j] for j in newFeedback])
        output.append(seed.pop())
        seed.appendleft(xor)
    return output
```

Step 3

The output of the LFSR function is converted from list to bits, in order to be XORed with the cipher text.

```python
def listToBits(l):
    binary = 0b0
    for i in range(len(l)):
        n = l[i]
        binary = binary ^ n
        binary = binary << 1
    binary = binary >> 1
    return binary
```

Step 4

The XORed text (cipher text XOR LFSR Ouptut) is converted from binary to string, with the function binaryToString.

```python
def binaryToString(binary, length, table):
    mask = 0b11111
    charList = []

    for i in range(length):
        # isolate last 5 bits of the bit stream
        last = binary & mask
        binary = binary >> 5

        # find the element on the table that matches the isolated bits
        for element in table:
            if (table[element] == last):
                charList.append(element)

    # reverse the charList / characters are encoded from end to start
    result = charList[::-1]
    return result
```

Step 5

The result text is: **SIMPLECANBEHARDERTHANCOMPLEX**

Exercise 7

The encryption formula of a 16-bit message 'm' is the following:
c = m XOR (m « 6) XOR (m « 10)

The result of this encryption is a 16-bit text. In order to decode this text, we use some of the bits of the cipher text, which are the same as the plain text. These bits are the last 6 bits, as the message is shifted in the encryption, and then XORed, not all the bits overlap during the XOR operation.

Assuming the following blocks of bits with a total of 16 bits: A = 6 bits, B = 4 bits, C = 4 bits.

The cipher text is then:

C	B	A

with C representing the right-most bits, and A the left-most bits of the cipher text.

As mentioned before, the A - block of bits are already known. So based on these, we can find the plain text, based on operations between the blocks.

The plain text is:

A XOR B XOR C	A XOR B	A

with A representing the 6 right-most bits, A XOR B XOR C = 6 bits and A XOR B = 4 bits.

If the cipher text is represented with 'c', the components of the plain text are calculated as seen below. The numbers used (1023, 63) are masks of the binary form 111... used to isolate the 10 and 6 last bits of the cipher text accordingly.

C = c » 10
B = (c AND 1023) » 6
A = c AND 63

As a result, the decryption formula is:

d = ((c»10) « 10) XOR (((c AND 1023) » 6) « 6) XOR (c AND 63)

Exercise 8

To describe the decryption of the Fischer Spiffy Mixer mode on block ciphers we first XOR both sides of the encryption equation with m_{i-1}:

$$c_i \oplus m_{i-1} = m_{i-1} \oplus E_k(m_i \oplus c_{i-1}) \oplus m_{i-1} =$$

$$c_i \oplus m_{i-1} = E_k(m_i \oplus c_{i-1})$$

We apply the decryption function on both sides

$$D_k(c_i \oplus m_{i-1}) = D_k(E_k(m_i \oplus c_{i-1})) =$$

$$D_k(c_i \oplus m_{i-1}) = m_i \oplus c_{i-1}$$

As a result

$$m_i = c_{i-1} \oplus D_k(c_i \oplus m_{i-1})$$

Exercise 9

(i) To find the Greatest Common of two numbers we use the Euclidean Algorithm.

The Euclidean algorithm for finding gcd(A,B) is as follows:

- If A = 0 then gcd(A,B) = B, since gcd(0,B) = B
- If B = 0 then gcd(A,B) = A, since gcd(A,0) = A
- Write A in quotient remainder in the form (A = B · Q + R)
- Find gcd(B,R) using Euclidean Algorithm, since gcd(A,B) = gcd(B,R)

According to this algorithm, the gcd(126048,4564) = 4 because:

126048 = 27 · 4564 + 2820

4564 = 1 · 2820 + 1744

2820 = 1 · 1744 + 1076

1744 = 1 · 1076 + 688

1076 = 1 · 688 + 408

688 = 1 · 408 + 260

408 = 1 · 260 + 148

260 = 1 · 148 + 112

148 = 1 · 112 + 36

112 = 3 · 36 + 4

36 = 9 · 4

In order to find Bezout's identity, that is x and y such that $ax + by = gcd(a, b)$, we use the Extended Euclidean Algorithm, based on the previous calculations, solving for the remainder of each equation.

2820 = 126048 - 27 · 4564

1744 = 4564 - 1 · 2820

1076 = 2820 - 1 · 1744

688 = 1744 - 1 · 1076

408 = 1076 - 1 · 688

260 = 688 - 1 · 408

148 = 408 - 1 · 260

112 = 260 - 1 · 148

36 = 148 - 1 · 112

4 = 112 - 3 · 36

Let a = 126048, and b = 4564. As a result,

2820 = a - 27b

1744 = b - (a - 27b) = 28b - a

1076 = a - 27b - (28b - a) = 2a - 55b

688 = 28b - a - (2a - 55b) = 83b - 3a

408 = 2a - 55b - (83b - 3a) = 5a - 138b

260 = 83b - 3a - (5a - 138b) = 221b - 8a

148 = 5a - 138b - (221b - 8a) = 13a - 359b

112 = 221b - 8a - (13a - 359b) = -21a + 580b

36 = 13a - 359b - (-21a + 580b) = 34a - 939b

4 = -21a + 580b - 3(34a - 939b) = -123a + 3397b

As a result,

$$ax + by = gcd(a, b) \Rightarrow -123a + 3397y = 4$$

and so the Bezout's coefficients are **-123** and **3397**.

(ii) In order to calculate the the modular multiplicative inverse of the given integer 809 modulo 1001 we once again use the Extended Euclidean Algorithm. At first we have to find whether 809 and 1001 are co-prime numbers:

$1001 = 1 \cdot 809 + 192$

$809 = 4 \cdot 192 + 41$

$192 = 4 \cdot 41 + 28$

$41 = 1 \cdot 28 + 13$

$28 = 2 \cdot 13 + 2$

$13 = 6 \cdot 2 + 1$

$1 = 1$

Since gcd(1001, 809) = 1, the numbers 1001 and 809 are co-prime, and thus we can now proceed to calculate the modular multiplicative inverse of 809 modulo 1001.

We calculate the Bezout's idendity of the two integers:

$192 = 1001 - 1 \cdot 809$

$41 = 809 - 4 \cdot 192$

$28 = 192 - 4 \cdot 41$

$13 = 41 - 1 \cdot 28$

$2 = 28 - 2 \cdot 13$

$1 = 13 - 6 \cdot 2$

Let a = 1001, and b = 809. As a result,

192 = a - b

41 = b - 4(a - b) = 5b - 4a

28 = a - b - 4(5b - 4a) = 17a - 21b

13 = 5b - 4a -(17a - 21b) = -21b + 26b

2 = 17a - 21b - 2(-21b + 26b) = 59a - 73b

1 = -21b + 26b - 6(59a - 73b) = -375a + 464b

As a result

$$ax + by = gcd(a, b) \Rightarrow -375a + 464y = 1$$

We apply the (modulo 1001) operation on both sides of the equation

$$-375a + 464y1(mod1001)$$

Since a = 1001, (-375 · 1001) (mod 1001) ≡ 0.

Thus,

$$464 \cdot 8091(mod1001) \Rightarrow 809 \equiv 464^{-1}(mod1001)$$

and as a result the modular multiplicative inverse of 809 modulo 1001 is **464**.

(iii) Based on Fermat's little theorem, if p is a prime number, then for any integer a, the number $a^p - a$ is an integer multiple of p:

$$a^p \equiv a \mod p$$

If a is not divisible by p, Fermat's little theorem is equivalent to the statement that $a^{p-1} - 1$ is an integer multiple of p:

$$a^p \equiv a \mod p \Rightarrow$$

$$a^{p-1} \equiv 1 \mod p$$

In our case, $a = 2$ and $p = 101$, with p being a prime number.
From Fermat's little theorem we have

$$2^{101} \equiv 2 \mod 101 \Rightarrow$$

$$2^{100-1} \equiv 1 \mod 101 \Rightarrow$$

$$2^{100} \equiv 1 \mod 101 = 1$$

(iv) In the following code segment the implementation of the fast modular exponentiation algorithm can be seen. The algorithm as explained in the project description, was slightly modified in order to be more optimized.

```python
def fast(b,e,m):
    x = b
    g = e
    d = 1

    while g > 0:
        if g % 2 == 0:
            x = (x * x) % m
            g = g / 2
        else:
            d = (x * d) % m
            g = g - 1
    return d
```

The results of the requested equations are:

$$2^{1234567} \mod 12345 = 8648$$

$$130^{7654321} \mod 567 = 319$$

Exercise 10

The computation of the discrete logarithm is implemented with the use of Shank's algorithm and the baby-step giant-step algorithm that relies on multiple operations in sequence.

```python
# find x such that
# g^x = y mod p
def shank(g, y, p):

    # ceil func: the smallest integer not less than sqrt of p
    m = math.ceil(math.sqrt(p))

    table = []

    # Baby step
    # map of g^1,...,g^m (mod p) : 1,..m .
    table = {pow(g, i, p): i for i in range(m)}

    # Giant Step Precomputation c = g^(-m) mod p
    # Fermat's little theorem
    c = pow(g, m * (p - 1), p)

    # Giant step
    # Search equivalent value in the table
    for j in range(m):
        x = (y * pow(c, j, p)) % p

        if x in table:
            return j * m + table[x]
```

The result of the equation $2^x \equiv 9077 \mod 12161$ is $x = 59$.

Exercise 11

The cryptographic algorithm RSA is the following:

Two large primes p and q are chosen. Let $n = p \cdot q$. Choose e such that $gcd(e, \varphi(n)) = 1$ (where $\varphi(n) = (p-1)(q-1)$). Find d such that $d = 1 \mod \varphi(n)$. In other words, d is the modular inverse of e ($d = e^{-1} \mod \varphi(n)$).

The public key is (e,n) and the private key is (d,n).

To encrypt a message m, we compute $c = m^e \mod n$ and to decrypt a ciphertext c, we compute $m = c^d \mod n$.

To decrypt the secret message C = **[3203, 909, 3143, 5255, 5343, 3203, 909, 9958, 5278, 5343, 9958, 5278, 4674, 909, 9958, 792, 909, 4132, 3143, 9958, 3203, 5343, 792, 3143, 4443]** we first have to calculate the private key.

```
phi = phi(N)
d = modularInverse(e,phi)
privateKey = [N, d]
```

The calculation of the private key uses the functions that find the Euler's Totient φ of a number, the greatest common divisor of two numbers, and the modular inverse of a number $a \mod m$. The code implementation of the above mentioned operations can be seen below.

```
# Find gcd (greatest common divisor) of two numbers
def gcd(a, b):

    if (a == 0):
        return b
    return gcd(b % a, a)

# Find Euler's Totient Function (   )
def phi(n):
    result = 1
    for i in range(2, n):
        if (gcd(i, n) == 1):
            result += 1
    return result

# Find modular inverse of a mod m
def modularInverse(a, m):

    for x in range(1, m):
        if (((a % m) * (x % m)) % m == 1):
            return x
    return -1
```

The private key given the numbers $N = 11431$ and $e = 19$ is [11413, 1179].

Using this information, the fast algorithm analyzed in a previous exercise is used and the cipher text is decrypted successfully giving the following message:

welcowe to the real world

Exercise 12

In this exercise, code was developed to implement Wiener's attack against RSA. This attack uses the continued fraction method to recover the private key d.

Three main functions are used to implement this attack: one to convert a rational fraction into a list of partial quotients [a0, ..., an], one to convert a finite continued fraction [a0, ..., an] to a rational fraction and one to compute the list of convergents using the list of partial quotients. These functions can be seen in the code segment below.

```python
# Converts a rational x/y fraction into
# a list of partial quotients [a0, ..., an]
def rationalToContFrac(x,y):
    a = x // y
    q = []
    q.append(a)
    while a * y != x:
        x, y = y, x - a * y
        a = x // y
        q.append(a)
    return q

# Converts a finite continued fraction [a0, ..., an]
# to an x/y rational.
def contFracToRational (frac):
    if len(frac) == 0:
        return (0, 1)
    num = frac[-1]
    denom = 1
    for i in range(-2, -len(frac) - 1, -1):
        num, denom = frac[i]*num+denom, num
    return (num, denom)
```

```python

# computes the list of convergents
# using the list of partial quotients
def convergentsFromContFrac(frac):
    c = []
    for i in range(len(frac)):
        c.append(contFracToRational(frac[0:i]))
    return c
```

Given the values $N = 194749497518847283$ and $e = 50736902528669041$, the private key is **[194749497518847283, 20881]**.

Once the private key is calculated, the cipher text can be decrypted. The characters of the given ciphertext are encoded using base64. So in order to find the plaintext, we first need to convert them to UTF-8.

```python
cipher = base64.b64decode(cipher).decode('utf-8')
```

After the decryption, we convert the integers to characters using ASCII and the message is:

Just because you are a character doesn't mean that you have character

Exercise 13

(i) Fermat's Primality Test

Fermat's primality test is implemented in the following function.

```
while(found == False):
    p = generateLargePrime(400)
    if(isPrime(2 * p + 1) == True):
        found = True
```

To generate a 2048-bit prime number using the Fermat method, we first generate a random 2048-bit number and then we check whether it is prime.

```
def generateLargePrime(k):
     # k is the desired bit length
     r = 100 * (math.log(k,2) + 1) # max number of attempts

     while r > 0:
         n = random.randrange(pow(2, k - 1), pow(2,k))
         r -= 1
         if isPrime(n) == True:
            return n
     return r
```

Fermat's Primality test i The following function checks if the random generated number is prime using Fermat's Method, which is shown in the first code segment of this exercise.

```
# decrease the number of potential primes to be tested
def isPrime(n):
    # lowPrimes => all primes under 1000
    lowPrimes =   [3, 5, 7,  11,  13,  17,  19,  23,  29,  31,  37,  41,  43,
     47,  53,  59,  61,  67,  71,  73,  79,  83, 89, 97, 101, 103, 107, 109,
   113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193,
   197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277,
   281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373,
   379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461,
   463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569,
   571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653,
   659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757,
   761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859,
   863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971,
   977, 983, 991, 997]

    # numbers greater than 3
    if (n >= 3):
        # n & 1 = 0 +> n is composite
        if (n & 1 != 0):
            for p in lowPrimes:
                if (n == p):
                    return True
                if (n % p == 0):
                    return False
            return fermat(n)

    return False
```

The output 2048-bit prime number was created in **27.63568353652954** seconds.

164843921204119251714414732601505767098327831329718205713343404491325472766798335850079846745402953959105645521874243522971644197301119020799077282562065883455319202837948676023717148649220123904550446685411070

5475633364611107042816604461290502019196550736333232308333969725315799
8229850005963330008362943215377791303602517373556005442382489964065580
6501943668875804465193681652780221056418921736547219123898724966025693
3733949074715744173033183365516226440253561878416421167164286475401854
9885906440728655434074750601310373395279815413135307112415079715275389
030371746223780973982511498047603132141194563611308941593

(ii) Miller-Rabin Primality Test

The creation of 1024-bit prime number using the Miller-Rabin primality test, is done in a very similar way to Fermat's method. The difference is the algorithm that tests if a number is prime or composite, and it can be seen in the following code segment.

```python
# Miller Rabin Primality Test
def rabinMiller(n):
    s = n-1
    r = 0
    # s is composite
    while s & 1 == 0:
        s = s // 2
        r += 1

    k = 0
    while k < 128:
        a = random.randrange(2, n - 1)
        x = fast(a, s, n)

        # if x composite
        if x != 1:
            i = 0

            while x != (n - 1):
                if i == r - 1:
                    return False
                else:
                    i = i + 1
                    x = fast(x, 2, n)
        k += 2
    return True
```

The output 1024-bit prime number was created in **1.673835277557373** seconds.

10560156468501374854886177055155136747196269453392067469838304920170447956479625552064626986390510031893604658200276393123448879760515274413286357681578286324105776497857566820964569079503270573990299576155933533749401934487285645575676901600505008347353628907161392485877584083504669408763189911012385123091 1

(iii) Safe and Sophie Germain Primes

A prime number p is called a Sophie Germain prime if $2p + 1$ is also prime. The number $2p + 1$ associated with a Sophie Germain prime is called a safe prime.

The only difference in the code is the following addition, which checks if a prime is safe or not.

```
while(found == False):
    p = generateLargePrime(400)
    if(isPrime(2 * p + 1) == True):
        found = True
```

The implementation is not optimized, and as a result the time needed for the program to run in order to generate a 1500-bit safe prime is a lot. Instead, the number shown is a 300-bit safe prime: 197378929604113811878639070594639242261550821288350200785333981146348923261259040086801059 3

Exercise 14

The Tonelli–Shanks algorithm is used in modular arithmetic to solve for r the equation $r^2 = n \mod p$, where p is a prime. In other words, it calculates a square root of n modulo p.

To apply the algorithm, we need the Legendre symbol. The Legendre symbol $(a \mid p)$ denotes the value of $a^{(p-1)/2} \mod p$. Some useful properties of Legendre symbol are:

- $(a \mid p) \equiv 1$ if a is a square $\mod p$
- $(a \mid p) \equiv -1$ if a is not a square $\mod p$
- $(a \mid p) \equiv 0$ if $a \equiv 0 \mod p$

```
def legendreSymbol(a, p):
    return pow(a, (p - 1) // 2, p)
```

The Tonelli-Shanks algorithm is explained below.

We first check if integer a is a perfect square using the Legendre Symbol Properties and them we factor out powers of 2 from $p-1$, to find q and e such that $p - 1 = q^{2^e}$ with q being odd. Then, we select a non-square z such that $(z \mid p) \equiv$ -1 and set $c \equiv z^q$

```
# check that a is a square: (n | p) = 1
if (legendreSymbol(a,p) == 1):

# Partition p-1 to q * 2^e for an odd q
# (reduce all the powers of 2 from p - 1)
q = p - 1
e = 0
```

```python
while q % 2 == 0:
    q = q // 2
    e += 1

# find z so that z|p = -1
z = 2
while (p - 1 != legendreSymbol(z, p)):
    z += 1
```

We then set some values as explained in the comment section of the following code segment.

```python
# set c = z^q (mod p)           => successive powers of z to update a and t
# set r = a ^((q+1)/2) (mod p) => a guess of the square root
# set t = a^q (mod p)           => how much guess is off
# set m = e : e is the exponent - decreases with each update
c = pow(z, q, p)
r = pow(a, (q + 1) // 2, p)
t = pow(a, q, p)
m = e
v = 0
```

Lastly, we loop until $t = 1$, and return the value r which is the result of the initial calculation.

```python
# loop until t = 1
while (t - 1) % p != 0:
    v = (t * t) % p

    # if t != 1 : find the lowest i (0 < i < m) so that t^(2^i) = 1
    for i in range(1, m):
        if (v - 1) % p == 0:
            break
```

```

      v = (v * v) % p

    # set b = c^(2^(m-i-1)) (mod p)
    # set r = r * b (mod p)
    # set t = t * b^2 (mod p)
    # set c = b^2 (mod p)
    # set m = i.
    b = pow(c, pow(2, m - i - 1), p)
    r = (r * b) % p
    c = (b * b) % p
    t = (t * c) % p
    m = i
```

The solution of x of the quadratic equation $x^2 \equiv 17592194433025 \mod 3094850098213450687$ is :

$$x = 4194305$$

Exercise 15

(i) The Miller-Rabin Primality Test has already been explained in exercise 13. The code used in this exercise concerning the determination of a prime number is the same.

Using the function $f(x) = x^2 + x - 1354363$ and 1000 random numbers between 1 and 10^5, we apply the Miller-Rabin primality test in $f(x)$ and we keep count of how many numbers are actually prime. The average number of prime numbers is approximately 400 out of the 1000 numbers tested.

(ii) There is no known rational algebraic function which always gives primes.

As Goldbach proved for $f(x) \in [x]$ such that $f(n)$ is prime for all integers $n \geq 0$, the $f(x)$ has to be constant.

The proof of this theorem is the following:

Assume that $n_0 > 0$ and consider the prime $p = f(n_0)$.
Since $f(n_0 + pk) \equiv f(n_0) \mod p$ it follows that $p \mid f(n_0 + pk)$ for all $k \in$ and thus $f(n_0 + pk) = p$ since $f(n)$ is always prime.
But if $f(n) - p$ has infinitely many roots it must be the zero polynomial, which implies that $f(n) = p$ identically, so $f(x)$ is constant.

Therefore there is no integer polynomial that remains prime for all integer inputs.

Exercise 16

We know that $p = 5$, $q = 11$, $c = 14$ and $m < 20$.

In order to find m we have to solve the equation $m^2 \equiv c \mod pq$.
This is performed by solving $m^2 \equiv c \mod p$ and $m^2 \equiv c \mod q$.

$$m_p^2 \equiv 14 \mod 5 \Rightarrow m_p = 2,\ m_p = 3$$

$$m_q^2 \equiv 14 \mod 11 \Rightarrow m_q = 5,\ m_p = 6$$

So, we have to solve the following system of equations to find the value of m. This is done with the use of the Chinese Remainder Theorem.

$$\begin{cases} m \equiv 2 \mod 5 \\ m \equiv 3 \mod 5 \\ m \equiv 5 \mod 11 \\ m \equiv 6 \mod 11 \end{cases}$$

We set $m = 55$, $M_1 = 11$ and $M_2 = 5$ and from the Euclidean Algorithm we have that $\overline{\mathrm{M}_1} = 1$ and $\overline{\mathrm{M}_2} = 9$.
We solve the system selecting two equations each time, resulting in four total solutions of m.

- Solution of the first subsystem:

$$\begin{cases} m \equiv 3 \mod 5 \\ m \equiv 5 \mod 11 \end{cases}$$

$3 \cdot \overline{\mathrm{M}_1} \cdot M_1 + 5 \cdot \overline{\mathrm{M}_2} \cdot M_2 = 51$ resulting in $51 \mod 55 \equiv 51 > 20$.
This solution is rejected.

- Solution of the second subsystem:

$$\begin{cases} m \equiv 2 \mod 5 \\ m \equiv 5 \mod 11 \end{cases}$$

$2 \cdot \overline{\mathrm{M}_1} \cdot M_1 + 5 \cdot \overline{\mathrm{M}_2} \cdot M_2 = 247$ resulting in $247 \mod 55 \equiv 27 > 20$.
This solution is rejected.

- Solution of the third subsystem:

$$\begin{cases} m \equiv 3 \mod 5 \\ m \equiv 6 \mod 11 \end{cases}$$

$3 \cdot \overline{\mathrm{M}_1} \cdot M_1 + 6 \cdot \overline{\mathrm{M}_2} \cdot M_2 = 303$ resulting in $303 \mod 55 \equiv 28 > 20$.
This solution is rejected.

- Solution of the fourth subsystem:

$$\begin{cases} m \equiv 2 \mod 5 \\ m \equiv 6 \mod 11 \end{cases}$$

$2 \cdot \overline{\mathrm{M}_1} \cdot M_1 + 6 \cdot \overline{\mathrm{M}_2} \cdot M_2 = 292$ resulting in $292 \mod 55 \equiv 17 < 20$.
This solution is accepted.

As a result, the value of m is 17.

Exercise 17

A PGP was generated with the use of "pgp –gen-key".

Then the message "Fun Fact: Cicadas are insects that develop underground for 13 or 17 (both primes) before surfacing. The prime life cycles do not align with the life cycles of the cicadas predators, protecting the broods of cicadas from extinction" was created.

The following command was used after that: "gpg –output message.gpg –encrypt –sign –recipient drazioti@csd.auth.gr message"

Exercise 18

(i) The goal is to create RSA keys, meaning that p, q, e, d, N, DP, DQ and Qinv have to be calculated.

Firstly two large primes (p and q) are calculated with the use of the Miller-Rabin Primality Test.

```
# Step 1: Create two prime numbers, p and q.
#         Calculate n = p * q.
  p = generateLargePrime(keySize)
  q = generateLargePrime(keySize)
  n = p * q
```

Then, the number e is created, which is a relatively prime to $(p-1)(q-1)$.

```
# Step 2: Create a number e that is relatively prime to (p-1)*(q-1).
while True:
    e = random.randrange(2 ** (keySize - 1), 2 ** (keySize))
    if gcd(e, (p - 1) * (q - 1)) == 1:
        break
```

After that, d is calculated, which is the modular inverse of d. Alongside with d, DP, DQ, Qinv are calculated.

```
# Step 3: Calculate d, the mod inverse of e.
d = modularInverse(e, (p - 1) * (q - 1))

# dp = d mod (p - 1)
dp = d % (p - 1)

```

```python
# dq = d mod (q - 1)
dq = d % (q - 1)

# qinv = modular inverse of q mod p
qinv = modularInverse(q, p)
```

A sample output of the program for 100-bit prime numbers p and q is:

p: 696415967177466918570116648897
q: 937287722160230599800253569527
N: 652742135551781885968263358393482929454332290415595637361719
e: 865973251219381285755960206059
d: 484484505055664326932714795547121503749275826138825673992899

dp: 680793657771539166414957120579
dq: 439093951328051848110031322185
qinv: 457556890223704426681840487201

(ii) The RSA encryption algorithm can be seen in the next code segment.

```python
def encrypt(publicKey, text):
    n,e = publicKey
    x = []

    for i in text:
        c = fast(i, e, n)
        x.append(c)

    return x
```

(iii) For the decryption of RSA the Chinese Remainder Theorem is used as shown below.

```python
def decrypt(p, q, d, c):
    # dp = d mod (p - 1)
    dp = d % (p - 1)

    # dq = d mod (q - 1)
    dq = d % (q - 1)

    # qinv = modular inverse of q mod p
    qinv = modularInverse(q, p)

    m1 = pow(c, dp, p)
    m2 = pow(c, dq, q)

    h = (qinv * (m1 - m2)) % p
    m = m2 + h * q
    return m
```

The program having a message as input, it encrypts it using the RSA algorithm and then successfully decrypts it using the Chinese Remainder Theorem.

Exercise 19

The hint states that all the needed information is in the course-1.pdf . We open the file online, and we click the inspect tool. In the console field there is the text: PDF 943d1e2975e9ed7884cbd802e8db2d2b [1.5 LibreOffice 6.4 / aHR0cHM6Ly9jcnlwdG9 sb2d5LmNzZC5hdXRoLmdyOjgwODAvaG9tZS9wdWIvMTUvCg==] (PDF.js: 2.8.243).

We decode this aHR0cHM6Ly9jcnlwdG9 sb2d5LmNzZC5hdXRoLmdyOjgwODAvaG9t ZS9wdWIvMTUvCg== from Base64 format and the result is the link:

$$https://cryptology.csd.auth.gr:8080/home/pub/15/$$

There is a message saying that "#2-75-22-6!". Knowing that Walter White was a chemist, we find the representation of each chemical element with the numbers above. The message that appears is "Heretic".

In the password field of the locked file "secret.txt" we type "#heretic!" and the file opens up. If we decode using Base64 the text in this file, we get some clues about a famous chess player from 1960. We now know that the next clue is about chess.

We go to the previous link, and click on "Other published documents". One of the links contains the moves of a chess game, and asks to find the best move for the white. The best move in chess terminology is Rf7. We put the move in the MD5 hash converter, and the result is f1f5e44313a1b684f1f7e8eddec4fcb0.

Once we enter this string as password in the "mastersecret.txt" document, the text file opens and the hash - answer is:

be121740bf988b2225a313fa1f107ca1

Exercise 20

(i) Once the https://cryptology.csd.auth.gr:8080 certificate is downloaded and saved as "csdauth.cer", we run the command in the command line we can obtain information about the modulus and the exponent.
The modulus in hexadecimal form has the value of: **" BBE169B489554736F4D8FF7B EDF00D4B2DB1EB4C87ED11551CEB4A66EC5786BC0A2DDED8E907F2DA1E5F9 9A9099032493F7501F9F08B784BDEE12AAB7DD3ECF1A93B211FB2DA5A8F16 C0333459B9744A9BF6995284860A5B646D5E767DBF648DC8315F4A954C285 2E37C33D3660E850FFB9A0D25B6290DDE1557F210C2D8001FE9CC9FE4735D A6C3BB96E431EB6C4468A0FD08B71044004198D3AB19CB63C5CF9FDDC69AD B36918E77BA7389C6F00EEB1F5E8FB5AFE8C0677B595C2D799EAAADA7AC07 7D975EC030C079CD9FE17BE35D149E25C20737EFA60D84993F74D83CB9602 D1732AFB12B924E05713279F57E5234602AEC07EF053583B2E883B66D417"**

This number in decimal form has the value of: **237177182778886746550451797068010 3746967679420129996216301187069860386949046980233396981461619825321222 6808329767940731504240476405886757088358089010763735405958500169408834 6387932750235146303297102860661469011960202811789498549573026483787713 8665472161583045335782914092071676402732942542039265311427727213225024 5197258328541292880473144810643161897154818048293578009886727864016409 5323648493600690957488769959169826320449256952494728739652239305103095 8232480585596101186612010417297182032217461058287273979626169086519605 4907416042643981566390691418075271354059019902666618284865249610892809 417078830770485788595223**
Finally the exponent is **65537 (0x10001)**

(ii) The certificate was created on **Jun 2 21:47:22 2021 GMT** and its expiration date is on **Aug 31 21:47:22 2021 GMT**.

Exercise 21

In order to lower the entropy of the system, we use two terminals. In the one we run the command:
"cat /dev/random"
and in the other:
"cat /proc/sys/kernel/random/entropy_avail"
to see the number of the entropy.

Once the output number is low enough (in my case 2) we generate a key with the command **"gpg -–gen-key"**.

The key generated in hexadecimal form is 1BA4333EE6390D3FCAA071ADF4BD238 64B9DE6E8 and when converted to decimal it has the value of 157804537154910626897146613809815362162069989096.

The prime factorization of the number 157804537154910626897146613809815362162069989096 is "$2^3 \cdot 23 \cdot 7204835546738257 \cdot 119035798740894138725021268667$".

The prime factors of this number are significantly large numbers, meaning that in case to find them lots of computational power is needed.

Exercise 22

(i) The **P** versus **NP** (polynomial versus non - deterministic polynomial) is a problem is a problem in computer science that asks whether every problem whose solution can be quickly verified can also be solved quickly. P is the set of problems whose solution times are proportional to polynomial and NP is the set of problems whose solutions can be verified in polynomial time.

By proving that P equals NP, it would mean that cryptography cannot be provably secure. It would become possible to crack the key to any digital cipher regardless of its complexity, thus rendering all digital encryption methods worthless, meaning that it is easy to break any algorithm.

For example, the RSA algorithm , is based on the factoring problem, meaning the division of a composite integer into a number of primes, which is hard to do for large numbers. If there existed a proof for easy and quick factorization of large numbers, RSA algorithm would be breakable, alongside with most of the cryptographic algorithms today.

(ii) The **OAEP** (Optimal Asymmetric Encryption Padding) algorithm uses a pair of random oracles G and H to process the plaintext before it is asymmetrically encrypted, resulting in being semantically secured under chosen cipher text and plain text attack.

In OAEP two hash functions are used, G and H alongside with r which is a random nonce of bits.

$$\mathbf{G} : hash function outputting\ \mathbf{g}\ bits$$

$$\mathbf{H} : hash function outputting\ \mathbf{h}\ bits$$

$$\mathbf{r} : random\ nonce\ of\ \mathbf{g}\ bits$$

Then, the message m is padded with zeros until the length of the m' is g bits.

$$m' = m \parallel 0^{\,g-len(m)}$$

After that, X is calculating by concatenating two parts: the first part is the message XORed with the hash function of r, and the second part is r XORed with the hash function of the first part.

$$X = m' \oplus G(r) \parallel r \oplus H(m' \oplus G(r))$$

Lastly, X is encrypted using the RSA algorithm.

As explained, the security of this algorithm is good, because to recover the message m, one must recover the entire m' and the entire X. X is required to recover r from m', and r is required to recover m from X. Since any changed bit of a cryptographic hash completely changes the result, the entire X, and the entire m' must both be completely recovered in order to find m.

This is known as an *all-or-nothing* encryption mode, because the data can only be understood only if all of it is known.

Exercise 23 [3.1]

(**i**)Assume that $x^2 = 4n + 3$.

The number $4n + 3$ is odd because it is of the form $2m + 1$:
$4n + 3 = 2(2n) + 2 + 1 = 2(2n + 1) + 1 = 2m + 1$

So, $x^2 = 2m + 1 = 4n + 3$ and thus x is odd because its square is odd and it is of the form $x = 2a + 1$:

$x^2 = 4n + 3 \Rightarrow$
$(2a + 1)^2 = 4n + 3 \Rightarrow$
$4a^2 + 4a + 1 = 4n + 3 \Rightarrow$
$4(a^2 + a - n) = 2 \Rightarrow$
$2(a^2 + a - n) = 1$

which is impossible, ans as a result no number of the form $4n + 3$ can be a perfect square.

(ii) Every integer is of the forms:

$4k$

$4k+1$

$4k+2$

$4k+3$

Then, the square root of an integer is of the forms:

$(4k)^2 = 16k^2 = 4(4k^2) = 4k_1$

$(4k+1)^2 = 16k^2 + 8k + 1 = 4(4k^2 + 2k) + 1 = 4k_2 + 1$

$(4k+2)^2 = 16k^2 + 16k + 4 = 4(4k^2 + 4k + 1) = 4k_3$

$(4k+3)^2 = 16k^2 + 24k + 9 = 4(4k^2 + 6k + 2) + 1 = 4k_4 + 1$

So, a perfect square modulo 4 is equivalent to 0 or 1.

Each number of the sequence $11, 111, 1111, \dots$ can be written as $4n+3$. For example:

$111 = 100 + 11 = 100 + 8 + 3$

$1111 = 1100 + 11 = 1100 + 8 + 3$

The first number of the sum (e.g. 100, 1100 etc) ends with double zeros, and thus it is perfectly divisible by 4. So each number of the sequence takes the form of $4n+3$ which is not a perfect square as already proven.

Exercise 24 [3.5]

We have tow lists: $P = [p_1, p_2, \ldots, p_n]$ (p_i : primes) and $e = [e_1, e_2, \ldots, e_n]$ (e_i : positive integers) and we want to find all the divisors of the number $N = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_n^{e_n}$.

To find all the divisors we start from the last factor which is $p_n^{e_n}$ and we reduce its power until it is e^0. Once the last power is fully reduced, we do the same thing in the previous prime number $p^{e}_{p-1}{}_{n-1}$ until its power is also fully reduced. This process continues until all the powers of the primes are fully reduced.

So in the first run we'll take the numbers
$d_1 = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_n^{e_{n-1}}$
$d_2 = p_1^{e_1} \cdot p_2^{e_2} \cdot \ldots \cdot p_n^{e_{n-2}}$
up until
$d_k = p_1 \cdot p_2 \cdot \ldots \cdot p_n$

In the next run, we repeat the same process but this time we start from the factor which is $p_{n-1}^{e_{n-1}}$.

All the numbers d that come out of this process are the divisors of N.

The complexity of the algorithm is $n \cdot \frac{n(n-1)}{2} \cdot n \approx O(n^4)$.

Exercise 25 [3.12]

In order to plot the two sequences given, python code was developed.

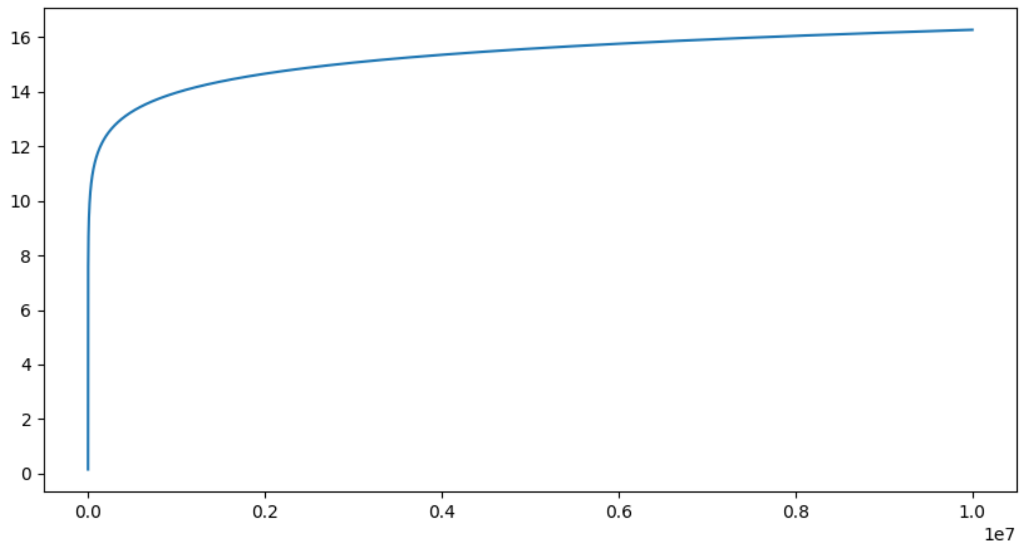
The number of the positive integer divisors on another integer, known as $\tau(n)$ is calculated through the following function.

```python
# count divisors of an integer
def t(n) :
    count = 0
    for i in range(1, (int)(math.sqrt(n)) + 1) :
        if (n % i == 0) :

            # if divisors are equal,
            # count only one
            if (n / i == i) :
                count = count + 1
            else : # otherwise count both
                count = count + 2

    return count
```

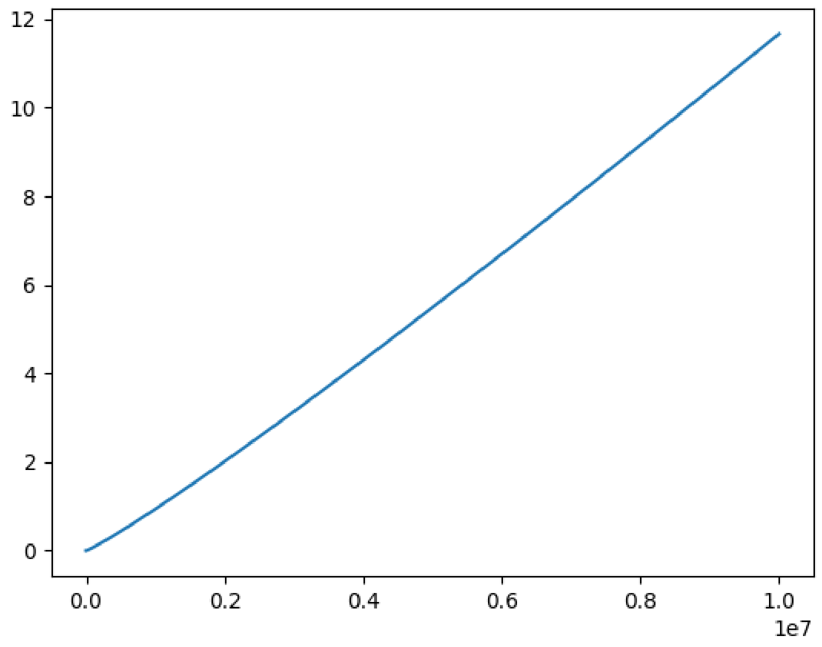
The code for the plotting of the two sequences can be seen below.

```python
# sequence a_n
a = np.log(n) + 2*g - 1

# sequence b_n
def b(n):
    sum = 0
    sumArray = []
    for r in range(int(n)):
        sum = sum + (1/n) * t(r)
        sumArray.append(sum)
    return sumArray
```

The graph of $a_n = \ln n + 2\gamma - 1$, where $\gamma \approx 0.577$ is:



The graph of $b_n = \sum_{r=1}^{n} \tau(r)$ is:



Exercise 26 [3.16]

For $n = 1$, $P_1 = 2 \geq 1 + 1 = 2$.

Assume that for some $k > 1$, $P_k > k$.

Then $P_{n+1} \geq P_n + 2 > n + 2 > n + 1$ so $P_{n+1} > n + 1$.

By induction, $P_n > n \Rightarrow P_n \geq n + 1$ for all $n \in$.

Exercise 27 [3.23]

Suppose that $\sqrt{p}$ is rational.

Let $a, b \neq 0$ be rational as well, and assume that $gcd(a, b) \neq 1$.

Then $\sqrt{p} = \frac{a}{b}$.

On squaring both sides we get:

$(\sqrt{p})^2 = \frac{a^2}{b^2} \Rightarrow$

$p = \frac{a^2}{b^2} \Rightarrow$

$b^2 = \frac{a^2}{p}$ (p divides a^2, so it divides a)

Let $a = kp$ for some integer k.

Then, $b^2 = \frac{(kp)^2}{p} = (kp)^2 p \Rightarrow k^2 = \frac{b^2}{p}$ (p divides b^2, so it divides b)

Thus, p is a common factor of a and b, which means that a and b are not coprime numbers/ This contradiction arises because we assumed that $\sqrt{p}$ is rational.

Therefore $\sqrt{p}$ is irrational.

Exercise 28 [3.28i]

We know that $gcd(a, b) = 1$.

Let $d_1 = gcd(c, b)$ and $d_2 = gcd(ac, b)$.

So we have from Bezout's identity

$cx_1 + by_1 = d_1$

$acx_2 + by_2 = d_2$

$ax + by = 1$

We multiply $ax + by = 1$ by d_1 using $cx_1 + by_1 = d_1$.

$d_1(ax + by) = d_1 \Rightarrow$

$\mathrm{ax}(\mathrm{cx}_1 + by_1) + byd_1 = d_1 \Rightarrow$

$\mathrm{ac}(\mathrm{xx}_1) + b(axy_1 + yd_1) = d_1$

$d_2 = gcd(ac, b)$ divides an integer linear combination of ac and b, so we know that d_2 divides perfectly d_1.

Similarly, we multiply $ax + by = 1$ by d_2 using $acx_2 + by_2 = d_2$.

$d_2(ax + by) = d_2 \Rightarrow$

$\mathrm{ax}(\mathrm{acx}_2 + by_2) + byd_2 = d_1 \Rightarrow$

$\mathrm{c}(\mathrm{a}^2 xx_2) + b(axy_2 + yd_2) = d_2$

Since we know that any integer linear combination of c and b is perfectly divided by $d_1 = gcd(c, b)$, we conclude that d_1 divides perfectly d_2.

As a result, we have that $d_2 \mid d_1$ and $d_1 \mid d_2$ with d_1, d_2 being non-negative integer numbers. Thus, $d_2 = d_1$ and $gcd(ac, b) = gcd(c, b)$

Exercise 29 [3.42]

(i) The prime factorization of 1134 is $1134 = 2 \cdot 3^4 \cdot 7$.

From the Euler's Totient theorem, we know that $\varphi(p) = p-1$ so $\varphi(2) = 1$ and $\varphi(7) = 6$.

For any prime p, $\varphi(p^n) = p^{n-1}(p-1)$, for any positive integer n.
$\varphi(3^4) = 3^{4-1}(3-1) = 3^3 \cdot 2 = 54$

Thus, $\varphi(1134) = \varphi(2) \cdot \varphi(3^4) \cdot \varphi(7) = 1 \cdot 54 \cdot 7 = 324$

(ii) The prime factorization of 2457 is $2457 = 3^3 \cdot 7 \cdot 13$.

From the Euler's Totient theorem, we know that $\varphi(p) = p - 1$ so $\varphi(7) = 6$ and $\varphi(13) = 12$.

For any prime p, $\varphi(p^n) = p^{n-1}(p-1)$, for any positive integer n.
$\varphi(3^3) = 3^{3-1}(3-1) = 3^2 \cdot 2 = 18$

Thus, $\varphi(2457) = \varphi(3^3) \cdot \varphi(7) \cdot \varphi(13) = 18 \cdot 12 \cdot 7 = 1296$

Exercise 30 [3.43]

The difference $n_1 - n_2$ is a multiple of $\varphi(m)$ because $n_1 \equiv n_2 \mod (\varphi(m))$.

So,
$n_1 - n_2 = k\varphi(m)$ for some $k \in \Rightarrow$
$n_1 = n_2 + k\varphi(m) \Rightarrow$
$a^{n_1} = a^{n_2 + k\varphi(n)} = a^{n_2} \cdot a^{k\varphi(m)} = a^{n_2} \cdot (a^{\varphi(m)})^k$

By applying Euler's Theorem which states that for every integer m relatively prime to an integer a, the following equation is holds:
$a^{\varphi(m)} \equiv 1$

Concluding, we have proved that $a^{n_1} \equiv a^{n_2} \mod m$.

Exercise 31 [3.68]

$a^x \equiv a^y \mod n \Rightarrow$

$a^x \cdot a^{-y} \equiv a^{-y} \cdot a^y \mod n \Rightarrow$

$a^{x-y} \equiv 1 \mod n$

In the last equation we set $k = x - y$ and we have $a^k \equiv 1 \mod n$.
We also have that

$k = x - y \Rightarrow$

$x - y = 0 \mod k \Rightarrow$

$x = y \mod k$

Since $a^{x-y} \equiv 1 \mod n$, we can say that $gcd(a, n) = 1$ and consequently $k = ord_n(a)$.

Thus, $x = y \mod k$ becomes $x \equiv y \mod ord_n(a)$
.

Exercise 32 [3.75]

(i) The $gcd(341, 2) = 1$, so 341 and 2 are relatively prime and 341 divides perfectly $2^{341-1} - 1$.

Also, $341 = 11 \cdot 31$, meaning that it is a composite number.

Thus, 341 is a Fermat pseudoprime to the base 2.

(ii) From Korselt's criterion a positive composite integer n is a Carmichael number if and only if n is square-free, and for all prime divisors p of n, it is true that $p-1 \mid n-1$.

The number 341 is square free and $341 = 11 \cdot 31$ as already stated. $11 - 1 = 10$ divides perfectly $341 - 1 = 340$.

However, $31 - 1 = 30$ does not divide perfectly $341 - 1 = 340$, and thus 341 is not a Charmichael number.

(iii) The $gcd(3914864773, 2) = 1$, so 3914864773 and 2 are relatively prime and 3914864773 divides perfectly $2^{3914864773-1} - 1$, as calculated by the fast algorithm which has already been explained in a previous exercise.

Also, $3914864773 = 29 \cdot 113 \cdot 1093^2$, meaning that it is a composite number.

Thus, 3914864773 is a Fermat pseudoprime to the base 2.

Exercise 33 [3.84]

(i) From Korselt's criterion a positive composite integer n is a Carmichael number if and only if n is square-free, and for all prime divisors p of n, it is true that $p-1 \mid n-1$.

In this case 8911 is square free because $\sqrt{8911} = 94.398\ldots$

Its prime factorization is $8911 = 7 \cdot 19 \cdot 67$.
The divisions $\frac{8911-1}{7-1} = 1485$, $\frac{8911-1}{19-1} = 495$ and $\frac{8911-1}{67-1} = 135$ are perfect, so we conclude that 8911 is a Carmichael number.

(ii) The $gcd(8911, 3) = 1$, so 8911 and 3 are relatively prime and 8911 divides perfectly $3^{8911-1}-1$, as calculated by the fast algorithm which has already been explained in a previous exercise.

Also, $8911 = 7 \cdot 19 \cdot 67$, as already explained, meaning that it is a composite number.

Thus, 8911 is a Fermat pseudoprime to the base 3.

Exercise 34 [3.89]

In order to prove that all primes between 1000 and 2000 are the sum of three primes, a python function was created to calculate all primes in an interval. This funstion is called two times, one to create all the primes between 1000 and 2000, and one to create an array with all the trimes to be tested for the sum.

```python
def calculatePrimes(lower, upper):
    primes = []
    for num in range(lower, upper + 1):
    # all prime numbers are greater than 1
        if num > 1:
            for i in range(2, num):
                if (num % i) == 0:
                    break
            else:
                primes.append(num)
    return primes
```

Then a method was created that checks if there are any numbers from the array of all the primes that if summed are equal to a prime chosen. This function returns 1 if the search was successfull and -1 otherwise.

```python
def findPrimes(num, primes):
    flag = False
    first = -1
    second = -1
    third = -1

    for i in range(len(primes)):

        # Take the first prime number
        first = primes[i]
        for j in range(len(primes)):

            # Take the second prime number
            second = primes[j]

            # Subtract the two prime numbers
            # from the N to obtain the third number
            third = num - first - second

            # If the third number is prime
            if (third in primes) :
                flag = True
                break

        if (flag):
            break

    #return 1 if the solution exists
    if (flag):
        return 1
    else :
        return -1
```

The number of all the primes in the interval 1000 to 2000 is 135. The outputting numbers of the findPrimes() function are stored, and the total sum of 1's outputted are equal to the total number of primes (135).

Thus, it is proved that all primes between 1000 and 2000 can be expressed as the sum of three prime numbers.

Exercise 35 [3.105]

We want to solve the following system using the Chinese Remainder Theorem.

$$\begin{cases} x \equiv 2 \mod 3 \\ x \equiv 3 \mod 5 \\ x \equiv 2 \mod 7 \end{cases}$$

We set $m_1 = 3$, $m_2 = 5$, $m_3 = 7$, $m = 3 \cdot 5 \cdot 7 = 105$, $M_1 = 5 \cdot 7 = 35$, $M_2 = 3 \cdot 7 = 21$, $M_3 = 5 \cdot 3 = 15$.

With the use of the Euclidean Algorithm we calculate $\overline{\mathrm{M}_1}$ using M_1.

$35 = 11 \cdot 3 + 2$

$3 = 2 + 1$

Thus,

$1 = 3 - 2 \Rightarrow$

$1 = 3 - (35 - 11 \cdot 3) \Rightarrow$

$1 = -35 + 12 \cdot 3 \Rightarrow$

$-1 \cdot 35 = 1 - 12 \cdot 3 \Rightarrow$

$-1 \cdot 35 = 1 \mod 3 \Rightarrow$

$-1 \cdot M_1 = 1 \Rightarrow$

$\overline{\mathrm{M}_1} = -1$

Similarly, we calculate that

$21 \equiv 1 \mod 5 \Rightarrow \overline{\mathrm{M}_2} = 1$

$15 \equiv 1 \mod 7 \Rightarrow \overline{\mathrm{M}_3} = 1$

As a result, the system has the following solution:

$$x = 2 \cdot \overline{\mathrm{M}_1} \cdot M_1 + 3 \cdot \overline{\mathrm{M}_2} \cdot M_2 + 2 \cdot \overline{\mathrm{M}_3} \cdot M_3 = 23$$

Thus, $23 \mod 105 \equiv 23$ and so $x = 23$.

Exercise 36 [12.7]

The program created to calculate the probability of a divisor d of a random n-bit positive integer number to be in a certain interval stated in the exercise description can be seen below.

The first function returns all the factors of a number, and the second one returns 1 if the binary length of a divisor is between certain margins.

```python
# find all divisors of integer number
def divisors(n):
    i = 1
    factors = []
    while i <= n :
        if (n % i==0) :
            factors.append(i)
        i = i + 1

    return factors

# calculate probability
def probability(L, n):
    for i in L:
        if (len(bin(i)) - 2 >= math.floor(n/2) and
           (len(bin(i)) - 2 <= math.floor(n/2) + 1 or
            len(bin(i)) - 2 <= math.floor(n/2) - 1)):

            return 1
```

The output is approximately 0.5 which is not correct according to the theory (it should be around 0.25). There might be an error in the calculations in the code.

Exercise 37 [12.9]

According to the theory the complexity of each factorization method is given by:

Coppersmith: $\text{O}(\log(N)^{O(1)})$

Quadratic sieve: $e^{(1+O(1))\sqrt{\ln n \ln \ln n}}$

General number field sieve (GNFS): $e^{\left(\sqrt[3]{\frac{a}{b}}+O(1)\right)\sqrt[3]{\ln n}\,\sqrt[3]{\ln \ln n}^{2}}$

The calculations for the numbers 100, 200, 1024, 2048, 4096 were made with the following program.

```python
def coppersmith(n):
    return math.log10(n)

def quadraticSieve(n):
    return math.exp(math.sqrt(math.log(n) *
                    math.log(math.log(n))))

def gnfs(n):
    return math.exp(math.pow(math.log(n), 1/3) *
                    math.pow(math.pow(math.log(math.log(n)), 2), 1/3))
```

The results can be seen in the following table.

	COPPERSMITH	**QUADRATIC SIEVE**	**GNFS**
100	2.0	14.181888622403854	9.082546300486353
200	2.3010299956639813	19.53611361554148	11.603914534697761
1024	3.010299956639812	38.99013126724143	19.332799168167053
2048	3.3113299523037933	51.19005396003301	23.498723225866637
4096	3.612359947967774	66.530104609772	28.27567472762879

References

Boneh, Dan, and Victor Shoup. "A Graduate Course in Applied Cryptography." (2017).

Letter frequencies of English language, Cryptographical Mathematics, Robert Edward Lewand : http://cs.wellesley.edu/~fturbak/codman/letterfreq.html

Michignan Technological University : https://www.mtu.edu/

Schneier, Bruce. "Applied cryptography protocols." Algorithms and Source Code in C (1995).

Smart, Nigel Paul. "Cryptography: an introduction", 3rd Edition. New York: McGraw-Hill (2003).

Stallings, William. Cryptography and network security, 4th Edition. Pearson Education India (2006).

Sweigart, Al. "Cracking Codes with Python: An Introduction to Building and Breaking Ciphers". No Starch Press (2018).