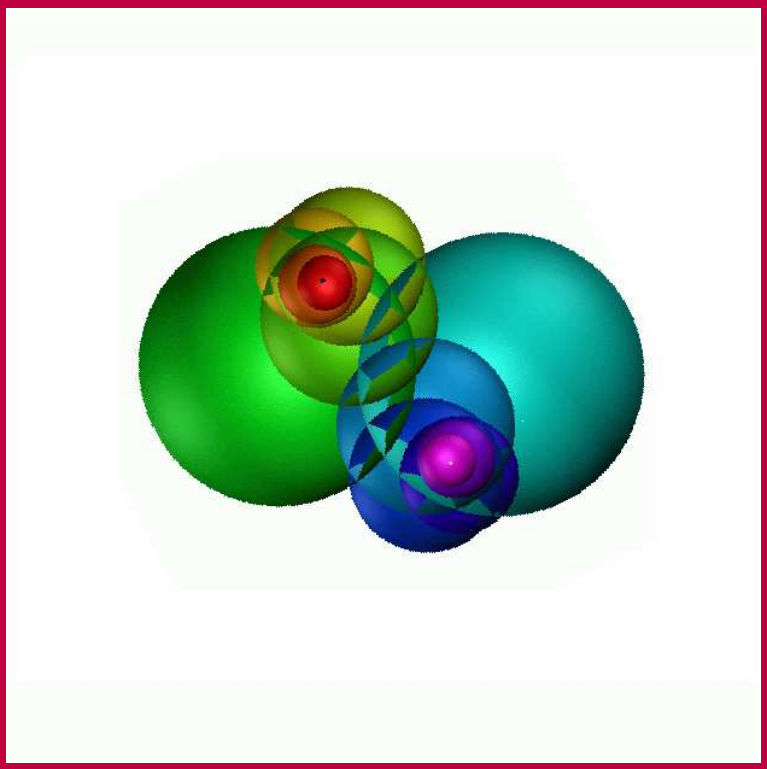


Open Workbook of Cryptology

A project-based introduction to crypto in Python

Jonathan A. Poritz

Department of Mathematics and Physics
Colorado State University, Pueblo
2200 Bonforte Blvd.
Pueblo, CO 81001, USA
E-mail: jonathan@poritz.net
Web: poritz.net/jonathan

11 Jun 2021 10:35MDT

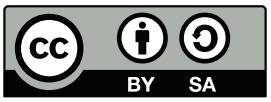
Release Notes

This is a first draft of a free (as in speech, not as in beer, [**Sta02**]) (although it is free as in beer as well) textbook for a one-semester, undergraduate cryptology course. It was used for CIS 491 at Colorado State University Pueblo in the spring semester of 2021.

It's not clear that any other instructor would feel comfortable using this version, without a lot of adaptation. In fact, I usually find that when I write an open textbook like this one, during the semester in which I am teaching the class, what results is a little rough and has a few real gaps. After another semester teaching with this book, the second version (which could well be twice as long!) will likely be much easier to use by others.

Nevertheless, in the spirit of openness, I'm sharing this first draft as well: if you want to use it for self-study or in a class you are teaching, feel free to do so – just makes sure your seatbelt is fastened, as the ride might be a little bumpy at times. [That said, if you do use this and find typos, thinkos, or other issues, I would be grateful if you would tell me about them!]

As I make improvements and additions to this work, you will always be able to find them on the page `https://poritz.net/jonathan/share/owoc/`, so please look there for the best version.



This version: 11 Jun 2021 10:35MDT.

Jonathan A. Poritz
Spring Semester, 2021
Pueblo, CO, USA

Contents

Preface

Many children want to keep secrets.

Lovers on opposite sides of the globe want to whisper sweet nothings to each other over an international telephone system whose wires and fiber optic cables run under neighborhoods, city streets, fields, and forests, or over mountain passes and on deep sea beds.

Continent-spanning empires need to store secrets and issue commands to distant government agents and agencies which cannot be modified in transit from the imperial center to its far-flung agents.

Consumers want to buy goods from merchants in other time zones, giving their credit card numbers to those merchants in messages over the open Internet – which is, without some system for hiding the contents of those messages, about as safe as writing them on the sides of long-haul trucks as they go by on the highway.

Everyone needs a little cryptology.

The problem with crypto (as we shall call cryptology in this book) is that it has a reputation of being very hard and mysterious, as well as very easy to get wrong. While there are aspects of crypto that are connected to quite modern and complex theories – such as number theory, an old and deep branch of mathematics; complexity theory, a new(er) and subtle branch of computer science; and even quantum computation, a quite new wrinkle on a 100 year-old version of physics which is famously counter-intuitive – that are not particularly friendly to the novice, much of the over-all framing of crypto is perfectly easy to comprehend and to use.

We contend, further, that this straightforward comprehension of the important basics of cryptology is most easily acquired by actually *working with cryptographic primitives*, by doing actual coding projects to implement, or use others' implementations of, basic cryptographic ideas.

That is the subject of this book.

This version uses **Python** and some standard cryptographic libraries in **Python** to explore these cryptological ideas. It should be accessible to students with a solid basic comfort level with **Python** – but could also be used as a way to solidify **Python** knowledge in more beginning users of that language, particularly if those beginners had a friendly instructor or peers with whom to collaborate.

CHAPTER 1

Preliminaries

Here are some Greek roots:

kryptos, $\kappa\rho\upsilon\pi\tau o\varsigma$: secret, hidden
logos, $\lambda\acute{o}\gamma o\varsigma$: word, study, speech
graph, $\gamma\rho\acute{\alpha}\varphi\omega$: write, written
analysis, $\alpha\nu\acute{\alpha}\lambda\upsilon\eta$: analysis

From these (and others), modern (technical) English gets the words

cryptosystem: a set of algorithms for protecting secrets, for hiding information so that only the intended persons have access to it
cryptography: work done to make cryptosystems
cryptanalysis: work done to circumvent the protections of cryptosystems – to "crack" a cryptosystem
cryptology: the combination of cryptography and cryptanalysis, often abbreviated simply to *crypto*.

Beware that *cryptography* is widely (but inappropriately!) used as a synecdoche for *cryptology*. We will use these words more carefully, except when the differences between the words is not so important, in which case we will use the word *crypto* as short for whichever of cryptology, cryptography, or cryptanalysis is most appropriate[1]

[1]Note this use of "crypto" conflicts a little with a silly modern usage of that word as short for *cryptocurrency*. We will return to that topic much later, when we talk about *blockchains*.

1.1. Some speculative history

Perhaps there was a form of deception that preceded language – certainly many a house pet has feigned innocence despite the clear evidence of involvement in stealing treats. And even apiologists may not know if some lazy bees make up a story about a long excursion to a new flower patch when their Queen demands an accounting.

But among *homo sapiens*, probably as soon as there was language, there was lying. Of course, when two humans are face to face, both parties have some control, such as: the listener can make an attempt to evaluate the trustworthiness of speaker, they can both form their own judgments of the other's identity and therefore choose what they wish to share with each other, and the words of the speaker pass directly from their lips to the listener's ears without the possibility of change of meaning in flight (absent considerations of ambient noise and so on).

A great deal changed with the invention of writing more than 5000 years ago. Words frozen in physical form, and the ideas they represent, can be taken and shared with a wide range of parties other than those with whom the original author wanted to communicate. In addition, if an author is not able to hand her work directly to the intended reader and instead the written words are out in the world on their own for a while, then both intended communicants can no longer be sure that the other is who the writing claims them to be nor that the writing remains the unchanged symbols that the other party originally set down.

Let us formalize some of these issues of *information security* (as it is called now), in the context of a message to be sent from someone named *Alice* to someone named *Bob*. The role of the possibly disruptive and overly intrusive environment is played in our little drama by *Eve*. (Traditionally one skips directly to a character whose name starts with *E* to symbolize both the *environment* and also someone who is potentially an *eavesdropper*[2].) Note that another way to think of what Eve is doing is listening to what is going by on some unfortunately (maybe unintentionally) public *communications channel* that Alice and Bob are using.

Here is some basic terminology:

DEFINITION 1.1.1. **Confidentiality** means that only the intended recipient can extract the content of the message – Alice wants only Bob to get her message and not Eve, no matter if she listens on the eavesdrop or intercepts the message as it travels in some form from Alice to Bob.

DEFINITION 1.1.2. Message **integrity** means that the recipient can be sure the message was not altered – Bob wants to know that what he gets is what Alice wrote, not what the mischievous Eve intended to change it into.

[2] *Eavesdropper* apparently comes from the Old English *yfesdrype*, meaning literally *one who stands on the eavesdrop* [ground where water drips from the eaves of the roof] *to listen to conversations inside a house.*

DEFINITION 1.1.3. Sender **authentication** means that the recipient can determine from the message the identity of the sender – Bob wants to be sure this message did in fact originate with Alice.

DEFINITION 1.1.4. Sender **non-repudiation** means that the sender should not be able to deny sending that message – Bob wants to be able to hold Alice to her promises.

Note that Alice and Bob may actually be the same person sending a message from a past self to their future self. For example, someone may want to keep records which must be *confidential* and whose *integrity* must be reliable, even if there is a break-in to the site keeping those records.

In fact, information storage can always be thought of, in this way, as an attempted communication from the past to the future via a channel that may unexpectedly become public if, for example, your laptop is stolen or someone breaks into your business and steals the hard drive from your server. Unless you can be absolutely sure that in all possible futures, no bad actor will ever possibly have physical access to your computer, it's always a good idea to encrypt all of your data on your personal devices!

Before we go on, a few more technical terms:

DEFINITION 1.1.5. The message that Alice wishes to send to Bob, in its actual original form is called the **plaintext**.

DEFINITION 1.1.6. An algorithm that Alice uses to transform (scramble, obfuscate) the plaintext into a form that will remain confidential even if observed by Eve while in flight is called a **cipher**. We also say that Alice **encrypts** the (plaintext) message.

DEFINITION 1.1.7. The encrypted form of a plaintext message is called the **ciphertext**.

DEFINITION 1.1.8. When Bob receives the ciphertext, he applies another algorithm to **decrypt** it and recover the plaintext.

Basic crypto terminology, graphically:

Alice	on public network/channel (where **Eve** is watching)	**Bob**
plaintext/cleartext message m **encrypts** m to c transmits c	$\rightarrowtail$ **ciphertext** c $\rightarrowtail$	receives c **decrypts** c to recover plaintext m

Hundreds of years of experience with cryptology have shown, again and again, that a cryptosystem which one person dreams up, thinking they have invented a system with wonderfully strong security, may easily fall to cryptanalysis done by someone else. So it has gradually come to be understood that the best way to make sure one is using a really solid cryptosystem is to share the algorithms widely and publicly, so that others in the community can try to break them. As is said in the FLOSS[3] community: *with enough eyes, all bugs are shallow*. Therefore, we should only trust a cryptosystem if its security remains strong even after everyone has tried their best to break it!

If the algorithm is public, how does it provide any security at all? The answer is that the algorithm may be public, but it is designed to take as input one more piece of information, beyond just the plaintext to encrypt or the ciphertext to decrypt:

DEFINITION 1.1.9. Additional information (some of which is) used in encryption and (some of) which is necessary for successful decryption is called a **key**.

Think of a key as something that Alice uses to lock her message in a box, the box being the message as it is transmitted over a public network and even possibly seen by Eve, and then used by Bob to unlock the box and find out what the actual message was that Alice wanted to send. Clearly, the keys must be kept carefully secret for the cryptosystem to do its job of protecting Alice and Bob's communications from the eavesdropping Eve!

This idea – that the security of a cryptosystem should be based on the secrecy of the key but not of the algorithm – has come to be called **Kerckhoff's Principle**, after a set of cryptosystem design ideas written down in 1883 by a French military cryptographer. The opposite of a system made with Kerckhoff's Principle in mind is one whose algorithms are simply kept secret, and in which it is very foolish to place any trust: such weak systems are said to reply upon the ill-advised **security through obscurity** paradigm.

Reading Response 1.1.1

Does all of this make sense? What was new or interesting, or what was old and uninteresting?

Reading Response 1.1.2

Can you think of an example not mentioned above where the terms like **confidentiality**, **integrity**, **authentication**, and/or **non-repudiation** could be used to describe something you've read about or experienced? Or else, can you think of a situation where these kinds of information security concepts apply to some version of **Alice**,

[3]"FLOSS"="Free/Libre/Open-Source Software, " a better term than just "open source"

Bob, **Eve**, and a **public channel**? – if so, say what that situation was and what corresponded to each of those characters.

1.2. The Caesar cipher and its variants

A cryptosystem which dates to ancient times was supposedly used by Julius Caesar.

DEFINITION 1.2.1. Alice takes her message, removes all spaces and punctuation, and puts it all in one case (maybe lower case). Then she moves each letter k places down the alphabet, wrapping around from **z** to **a** if necessary, where k is a fixed whole number known to both Alice and Bob but no one else, called the **key**. (Often the ciphertext is written entirely in upper case, so it is easier to tell plaintext from ciphertext.)

To decrypt, Bob simply moves each letter k places earlier in the alphabet, wrapping past **A** to **Z** if necessary: in other words, Bob *encrypts* the ciphertext with key $-k$ to get the plaintext.

This is called the **Caesar cryptosystem** or **Caesar cipher**.

Apparently Julius Caesar usually used the key value $k = 3$. His nephew Octavian, who later became the emperor Augustus, liked to use $k = -1$.

When using what is called the Latin alphabet (which is not what was used in ancient Rome, though!), with its 26 letters, there is one particularly nice key value: 13. The nice thing about that value is encryption and decryption are exactly the same transformation. In modern times, this transformation is called **ROT13**, and it has a small role in the modern history of the Internet. In particular, posts on early chat rooms and bulletin boards would sometimes want to have a bit of content that should not be automatically available to anyone who looks at the post, but would be there for the determined reader (such as, for example, a spoiler in a review of some popular new game, book, or film). These were often included in the post, but only after they had been run through ROT13.

A few commercial products used ROT13 for actual security, despite it actually being completely insecure, such as certain parts of some versions of the Windows operating system.

Reading Response 1.2.1

Have you heard of the Caesar cipher before? Is it clear to you how to use it?

1.2.1. Keys only matter "mod 26". Notice that if Alice uses the Caesar cipher and key $k_{\text{Alice}} = 1$, her encryption does exactly the same thing as Ann who was using Caesar with key $k_{\text{Ann}} = 27$, because of that part of the definition of the Caesar cryptosystem which says "wrapping around from **z** to **a**": Ann will always go all the way around the alphabet once, but then end up at the same place Alice will. The reason for this is that $27 = 26 + 1$, and encrypting with 27 goes around the alphabet once – from the 26 – and then goes one more step. The same thing would happen if Ann used the key $k_{\text{Ann}} = 53 = 2 * 26 + 1$, in

that Ann's encryption process would go around the alphabet – twice, in this case, from the $2 * 26$ – and then end up at the same place as Alice's.

In fact, if difference between Alice's and Ann's key is any number of 26s, then their encryptions will look exactly the same! Mathematicians have a bit of terminology and notation for this:

DEFINITION 1.2.2. Suppose a, b, and n are three whole numbers. Then we say that a **is equal to** b **modulo** n if $a - b$ is a multiple of n; that is, if there is some other whole number k such that $a - b = k * n$. The notation for that is

$$a \cong b \pmod{n}$$

EXAMPLE 1.2.3. As we just noticed, $1 \cong 27 \pmod{26}$.

EXAMPLE 1.2.4. ROT13 was so convenient because doing it twice has the effect of doing nothing, because

$$13 + 13 \cong 0 \pmod{26}$$

Or, thought of another way, since

$$-13 \cong 13 \pmod{26}$$

ROT3 encryption (Caesar with key 13) is exactly the same thing as ROT13 decryption (Caesar with key -13).

Reading Response 1.2.2

Does all of this make sense? What was new or interesting, or what was old and uninteresting?
If you feel comfortable with this new terminology and notation, try to answer the following question: If you have a whole number x and you know that $x \cong 0 \pmod{2}$, what would be a more common way of talking about that number? *I.e.,* you would say that "x is _____."

Computer languages usually have a way to work with numbers mod n. For example, **Python** has a built-in arithmetic operation which is useful here: `a%n` gives the smallest non-negative whole number which is equal to `a` mod `n`.

EXAMPLE 1.2.5. From what we noticed above, in **Python**

```
>>> 27%26
1
```

and

```
>>> -13%26
13
```

Even more, we can use this operator to implement the Caesar cipher!

First, notice that we can't just add a key to a letter, because the types don't work:

```
>>> 'a'+2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int
```

So we must first convert a character to an `int` and then back to a character:

```
>>> chr(ord('a')+3)
'c'
```

Note that `ord()` takes a single character and converts it to an `int`, while `chr()` takes an `int` and converts it to a single character. It's not so important what are the details of how that works, except to note that the encoding includes all of the lowercase letters, in alphabetical order, in one numerical block from the value `ord('a')` to the value `ord('z')`, and all of the uppercase letters, in alphabetical order, in another numerical block from `ord('A')` to `ord('Z')`.

Actually, that's not going to work with a letter too close to the end of the alphabet, *e.g.,*

```
>>> chr(ord('y')+3)
'|'
```

because it didn't "wrap around the alphabet."

So how about if we take a character in a variable `x` and do the following:

(1) make it into a number using `ord()`
(2) shift it so that it's in the range from `0` to `25` by subtracting `ord('a')`
(3) do the Caesar by adding the value `k`
(4) get it back in the range `0` to `25` by doing `%26`
(5) get it in the range `ord('A')` to `ord('Z')` by adding `ord('A')`
(6) make it back into a letter using `chr()`

E.g., if `x='y'` and `k=4` then we would get

```
>>> chr((ord(x)-ord('a')+k)%26+ord('A'))
'C'
```

Let's now implement the Caesar cipher in **Python**, in steps. First, we should clean up the plaintext string.

Code Task 1.2.1

Write a **Python** function which takes an input string and returns a string which is the same except all the non-letter characters have been removed and all the letters have been made to lower case.

It might help to know that you can loop through the characters of a string `s` by doing

```
for x in s:
```

and you can build up a string `o` for output by starting with

```
o=''
```

and then adding one or more characters in a string variable `t` to the end of `o` by doing

```
o=o+t
```

Another, more "Pythonic" approach is probably to build up a list by the construction

```
[ c for c in s ]
```

which just makes a list each of whose elements is a single character out of the string `s`. Note you can also do something to the characters as you put them in the list, and you can add a condition to the looping part, such as

```
[ ord(c) for c in s if c.isalpha()]
```

– try it!

Code Task 1.2.2

Caesar encryption: write a **Python** function `Caesar_encrypt(s, k)` which takes an input string `s` – the plaintext – and an int `k` and returns the ciphertext from that plaintext under the Caesar cipher with key `k`.
For example, if you do

```
Caesar_encrypt('All the animals at the zoo!', 3)
```

the output should be

```
'DOOWKHDQLPDOVDWWKHCRR'
```

.

Code Task 1.2.3

Caesar decryption: Write a **Python** function `Caesar_decrypt(s, k)` which takes an input string `s` – the ciphertext – and an int `k` and returns the plaintext from that ciphertext under the Caesar cipher with key `k`.
For example, if you do

```
Caesar_decrypt('DOOWKHDQLPDOVDWWKHCRR', 3)
```

the output should be

```
'alltheanimalsatthezoo'
```

.
You might want to do a sanity test on your input and return an error or a null string if the input is not a well-formed ciphertext, *i.e.,* a string consisting only of uppercase letters.

1.2.2. Modernizing the Caesar cipher. It seems unfortunate that the Caesar cipher, as we have used it, looses the punctuation in the messages and case of the letters of the message. One way to get around this would be to imagine all of the letters of both cases and all punctuation symbols to be part of a much bigger alphabet, and then to use the same

old Caesar with this much bigger alphabet, correspondingly using `%N`, where `N` is the new alphabet size, in place of the `%26` we were using before.

Technically, in **Python 3**, strings are made of characters, and characters are encoded using **Unicode**. Unicode is a way of turning numbers, represented by one, two, or four bytes, into many special symbols as well as letters from many writing systems used around the world.

The first 128 symbols in Unicode, which therefore fix into one byte (in fact with the highest-order bit being 0), are just the old ASCII encoding scheme, dating back to 1963, for the English alphabet plus some additional punctuation, symbols, and even non-printing symbols like "EOF." The name of the encoding scheme for the subset of Unicode which fits in 8 bits is "UTF-8," which includes all of the old ASCII code.

Full Unicode in two or four bytes includes many non-Latin alphabets (such as Greek, Cyrillic, hangul, *etc.*) and even non-alphabetic writing systems like Chinese and Japanese (actually, part of Japanese is non-alphabetic, called "kanji," but Japanese also has two different writing systems based on syllable sounds, called "hiragana" and "katakana,", each with 49 symbols), with thousands of ideograms.

Reading Response 1.2.3

Have you heard of Unicode?
If not, you might also read, or at least skim, the Wikipedia page about Unicode and say what you think of it.
Have you ever worked with Unicode in **Python**?
If so, what did you think of it? If not, you might also read, or at least skim either the **Python** documentation Unicode HOWTO or the more detailed Unicode & Character Encodings in Python: A Painless Guide and say what you think about it.

Bonus Task 1.2.1

Implement in **Python** a Caesar-style encryption and decryption which works for a different alphabet than English. Maybe do Spanish (one extra letter: ñ) or German (four extra letters: ä, ö, ü, ß)....
Or, maybe implement Caesar with the English alphabet, but figure out some way to keep the punctuation as well. It seems like it might be giving away too much information to leave the spaces and punctuation the same in the ciphertext as it was in the plaintext, but can you do something interesting with it?
Or, maybe implement a Caesar-style cryptosystem which handles all of ASCII, or UTF-8, or all of Unicode!

1.3. Cryptanalysis of the Caesar cipher

Let's think about cryptanalysis of the Caesar cipher; that is, let's figure out how to get a plaintext back from a Caesar ciphertext when we don't know the key.

The first thing we should notice is that, as we saw in §1.2.1, the key only matters "up to mod 26." But every possible whole number k for a key is equal mod 26 to a whole number in the range from 0 to 25: just try running

```
for i in range(100):
  print(i%26)
```

to get a feel for what's happening.

So, effectively, there are only 26 possible keys for a Caesar cipher! In fact, it would be pretty silly to use the key $k = 0$, since then the ciphertext would be exactly the same as the plaintext (well, without the non-letter characters, and all in upper case). So there are really only 25 reasonable Caesar cipher keys. It seems like this kind of thing is important enough to have its own name:

DEFINITION 1.3.1. The set of possible keys for some cryptosystem is called the **keyspace** of the cryptosystem.

What we have just seen is that the keyspace of the Caesar cipher is just the whole numbers from 1 to 25. In **Python**, it is the elements of the list `range(1,25)`.

An enemy of Caesar's who intercepted a ciphertext of his and who wanted to know what it said could assemble a team of 25 literate people (maybe those were hard to find at the time?) who could count (also hard?) and use them to make a *Caesar-cracking computer* as follows:

- each person would be assigned one of the numbers from 1 to 25
- each person would write out the Caesar-decryption of the intercepted ciphertext, using their assigned number as key
- if one of them got something which looked like it was written in good Latin, they would shout "Success!" and read out their decryption

This would be a pretty fast and economical way to crack the Caesar cryptosystem only **because the keyspace is so small**. This kind of attack has a name:

DEFINITION 1.3.2. An attempt to cryptanalyze a cryptosystem by trying all of the possible values of some parameter of the system (such as all possible keys or all possible messages) is called a **brute-force attack**.

Reading Response 1.3.1

Does all of this make sense? What was new or interesting, or what was old and uninteresting?
Can you think of an example of a **brute-force attack** used in some information security context that you know about from other classes or other readings or just from your life?

Code Task 1.3.1

Write a **Python** function `CaesarBF(s)` implementing a brute-force attack on the Caesar cryptosystem. Given an input ciphertext, it should try all possible keys for decryption and print out the attempted key and the corresponding decryption.
E.g., the output of `CaesarBF('DOOWKHDQLPDOVDWWKHCRR')` should be

```
 1:  cnnvjgcpkocnucvvjgbqq
 2:  bmmuifbojnbmtbuuifapp
 3:  alltheanimalsatthezoo
 4:  zkksgdzmhlzkrzssgdynn
 5:  yjjrfcylgkyjqyrrfcxmm
 6:  xiiqebxkfjxipxqqebwll
 7:  whhpdawjeiwhowppdavkk
 8:  vggoczvidhvgnvooczujj
 9:  uffnbyuhcgufmunnbytii
10:  teemaxtgbfteltmmaxshh
11:  sddlzwsfaesdksllzwrgg
12:  rcckyvrezdrcjrkkyvqff
13:  qbbjxuqdycqbiqjjxupee
14:  paaiwtpcxbpahpiiwtodd
15:  ozzhvsobwaozgohhvsncc
16:  nyygurnavznyfnggurmbb
17:  mxxftqmzuymxemfftqlaa
18:  lwwesplytxlwdleespkzz
19:  kvvdrokxswkvckddrojyy
20:  juucqnjwrvjubjccqnixx
21:  ittbpmivquitaibbpmhww
22:  hssaolhupthszhaaolgvv
23:  grrznkgtosgrygzznkfuu
24:  fqqymjfsnrfqxfyymjett
25:  eppxliermqepwexxlidss
```

That's all well and good, but we have a huge problem: human intervention is required here, to pick out which possible decryption, in the brute-force attack, is the correct one!

For example, suppose bad actors invaded your organization's network and encrypted all of your files with the Caesar cipher, but choosing a different key for each one. Your organization probably has hundreds of thousands of files, and while any individual one could be easily decrypted, it would take an insane number of person-hours to decrypt your whole network ... you might be willing to pay a fairly large ransom to get all of your files back without all that human life invested in brute-force decryption.

1.3.1. Frequency analysis. Let's think about how to automate the choice of the correct decryption in a brute-force attack on the Caesar cipher.

The key idea here is just the observational fact that *the relative frequencies of different letters in English are relatively stable across almost all pieces of English text*. In particular, in English, the most common letter is usually "e," the second most common is usually "t," *etc.*

This suggests an approach to detecting automatically which decryption in the brute-force attack is the correct one: declare victory when the brute-force approach has found a possible decryption where the most common letter is "e."

Code Task 1.3.2

Write a **Python** function `freq_table(s)` which makes the *frequency table* for the letters of a string `s`. That is, first clean up `s` by removing all non-letters and making all remaining letters be in the lower case. Then `freq_table()` should returns a list of 26 `int`s where the value at location `i` in this list tells how many times the letter `chr(i+ord('a'))` occurred in the cleaned-up input string.
For example, executing
`freq_table('alltheanimalsinthezoo')`
should return the list
`[3,0,0,0,2,0,0,2,2,0,0,3,1,2,2,0,0,0,1,2,0,0,0,0,0,1]`.

Code Task 1.3.3

Write a **Python** function `CaesarBF_findE(s)` uses your previous `CaesarBF(s)` but only prints out the possible decryption if the letter "e" is the most frequent in the proposed cleartext. That is, each time your function computes a possible cleartext, you should get the frequency table `freqs` from that string, using your `freq_table()` function. Then do the test

`freqs[4]==max(freqs)` to see if the letter at index 4, which is "e," has the maximum frequency. Only if the test passes will you print out that possible cleartext.

Clearly, this strategy doesn't seem to work too well, because it would find that the string `'teemaxtgbfteltmmaxshh'` from key `10` and `'eppxliermqepwexxlidss'` from key `25` were both possible decryptions of `'DOOWKHDQLPDOVDWWKHCRR'`, neither of which is correct!

The problem is that the single letter which is most frequent is too coarse a piece of information about a string to be useful to determine whether the string is in good English. However, the entire frequency table does a much better job. Let's implement that in **Python** now, in several steps.

Code Task 1.3.4

Modify your **Python** function `freq_table(s)` to make one called `rel_freq_table(s)` which computes the *relative frequency table* for an input string `s`: a table which computes the fraction of the letters in `s` which are each particular letter. If the list computed by `freq_table()` is `f`, then the relative version will have `float`s rather than `int`s, with values that are the values of `f` divided by `sum(f)` (or, what should be the same thing, divided by `len` of the input string).
For example, executing `rel_freq_table('zbazb')` should return the list `[0.2,0.4,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0, 0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.4]`.

Code Task 1.3.5

Run your function `rel_freq_table()` on some very long pieces of English text, to get the model relative frequency table of the English language.
You might compare the values you get from what Wikipedia says should be the right table of relative frequencies in the article Letter Frequency. Did you get a similar answer? If not, do you have a guess of why?

Code Task 1.3.6

You will need, in a moment, a tool which computes the "distance" between two relative frequency tables. Those tables are lists of numbers of length 26. Well, if you have a list of two numbers, you can think of those numbers as the coordinates

in two-dimensional space. If you have a list of three numbers, you can think of them as coordinates in three-dimensional space. So a list of 26 numbers could be thought of as the coordinates in 26-dimensional space. Weird, yes, but go with it....
In two- or three-dimensions, the distance between two points is given by the square root of the sum of the squares of the differences between the coordinates. That is, in two dimensions, the distance between two points with coordinates (x_1, y_1) and (x_2, y_2) is, by the Pythagorean Theorem, given by the formula $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$. The same thing also works in three ... or in 26 dimensions: that's just the Pythagorean Theorem 25 times!
So write a **Python** function which takes as input two lists of numbers of the same length, say they are `a` and `b`. It should then take the first element of `a` minus the first element of `b` and square that difference. It should add to that square the square of the difference of the second elements, and so on until all the elements are used up. Finally, it should return the square root of that total, which will be the value of `freq_table_dist(a,b)`.

Code Task 1.3.7

Now implement a **Python** function `CaesarBF_rel_freq(s)` tries all possible Caesar decryptions of the input ciphertext string `(s)`, and picks the one whose relative frequency table is least different from the standard relative frequency table of English which you just found.
Here are the steps: do the `CaesarBF(s)` approach from before, and compute the relative frequency table for each possible decryption. Now compute the "frequency table distance," in the sense of the previous coding task, between the relative frequency table for this possible decryption and the standard relative frequency table for English.
When you're done, find the smallest distance of all those possible decryptions and announce that one as the best guess of the real decryption of the input ciphertext.

Bonus Task 1.3.1

Now let's stress-test the brute force Caesar cipher cracker which uses frequency analysis.

First, try taking a big piece of text in another language, encrypt it with the Caesar cipher and some random key, and then use your function

`CaesarBF_rel_freq(s)` to try to decrypt it automatically. For example, you might take some of the Spanish text of the famous novel Don Quixote or simply cut-and-paste from the French newspaper *Le Monde* or the German newspaper *Deutsche Tageszeitung*. **Note:** for other languages, you may have an issue with the alphabet! Letters with accents may mess up your constructions in **Python** which assume the alphabet will be sequentially ordered from 0 to 25 when you take each letter `c` and do `ord(c)-ord('a')`. Figure out how to handle this situation (probably by turning each special letter like into a letter in the English alphabet (like just make "ñ" into a plain "n").

Or you could try decrypting a text which was originally in Latin, like the original Latin text of Isaac Newton's famous *Philosophiae Naturalis Principia Mathematica*.

Another thing to do would be to try some very unusual text, but in English. For example, a man named Ernest Wright published a book called *Gadsby* in 1939 which did not have the letter "e" anywhere in it. Since "e" is the most common letter in normal English, the relative letter frequencies in this novel must be quite different from those in normal English. Try encrypting and then automatically decrypting some of the text of *Gadsby* (but make sure you go down far enough in that file to get to the text of the novel: the introductory material does have the letter "e").

1.4. Defending Caesar against frequency analysis: Vigenère and the one-time pad

Sometimes a very short message is very important: "yes" or "no" is a big difference in many situations! But frequency analysis attacks on the Caesar cipher depend upon having a relative frequency table to compare to the relative frequency table of English ... and without a long enough enough ciphertext string, there won't be much of a frequency table at all.

There must be some sort of threshold k such that when a ciphertext string has length k or less, the relative frequency table doesn't give enough information to do successful cryptanalysis. There should also be another threshold K such that when a ciphertext string has length of K or more, the cryptanalysis of the Caesar cipher using relative frequency tables usually works.

For messages of length between k and K, presumably, sometimes the approach to cryptanalysis based on relative frequency tables is successful, and sometimes not.

Bonus Task 1.4.1

Let's try to figure out what those thresholds k and K are.

Do a big loop as some number `l` goes from 1 to 150 (or so). For each value of `l`, you are going to figure out how often your relative frequency table-based Caesar cracker `CaesarBF_rel_freq` is successful on strings of length `l`.

Make a very long string `big_string` with some text. You probably want to take all of the non-letters out of `big_string` and make all of the letters lower case.

Now make a loop to try decrypting maybe 100 randomly chosen strings from `big_string`. Each time through the loop, pick a random substring from `big_string` of length `l` – the way to do this is to do

```
from random import *
```

once near the beginning of your program, then in this loop set

```
rand_start = randint(0,len(big_string)-l)
```

to be the random starting point for the substring, and

```
rand_cleartext = big_string[rand_start:rand_start+l]
```

to be the corresponding substring of length `l`.

Now pick a random Caesar key `k=randint(1,26)`, encrypt `rand_cleartext` to `rand_ciphertext` with that key, and try your relative frequency table-based Caesar cracker on `rand_ciphertext`. If its proposed decryption equals `rand_cleartext`, count that as a success. If not, count it as a failure.

See how many times out of the 100 (*i.e.*, the percentage) in this inner loop you got success.

Finishing the outer loop, you should now have a list of success percentages for each possible string length from 1 to 150, maybe in a list `successes`.

Plot that list against the numbers from 1 to 150. This can be done by first getting the plotting package with `import matplotlib.pyplot as plt`, then making the plot with `plt.plot(x,successes)` where `x=[i+1 for i in range(149)]`. You will need also to run `plt.show()` to display the plot.

The way to prevent relative frequency table-based attacks on the Caesar cipher is to only use it to encrypt short messages (less than that threshold k we just mentioned). This is a problem if it is important to encrypt a longer message.

The approach to use with longer messages is simply to change the Caesar cipher key frequently, so that not all that much cleartext is encrypted with each particular key value. For example, to make the string half as long, so that relative frequency table methods are only half as good, we could use two keys a and b, and encrypt the first half of the message with Caesar under the key a and the second half with Caesar under key b.

Historically, this approach was implemented in a slightly different way: rather than using a for the first half of the message and b for the second half, people used a for the letters in the first, third, fifth, *etc.*, positions in the message, and b for the letter in the second, fourth, sixth, *etc.* positions.

To cut the message pieces down even more, one can use even more keys than just two. There is a name for this cryptosystem:

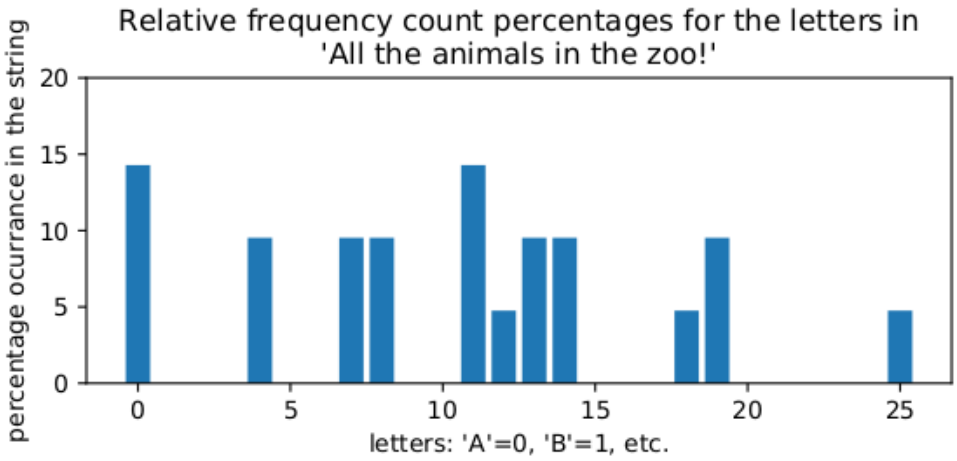
DEFINITION 1.4.1. Fix ℓ numbers $k_1, k_2, \ldots, k_\ell$. The **Vigenère cryptosystem** does encryption as follows: given a cleartext c, encrypt the first letter of c using the Caesar cipher with key k_1, the second letter of c using Caesar with key k_2, *etc.* When all of the ℓ key values have bee n used, start again with k_1 for the next letter, then k_2, *etc.* Vigenère decryption works in the same way, except Caesar decryption is used at each step.

For example, the Vigenère encryption of "All the animals in the zoo" with three keys $3, 20, 4$ is "DFPWBIDHMPUPVCRWBICIS," since

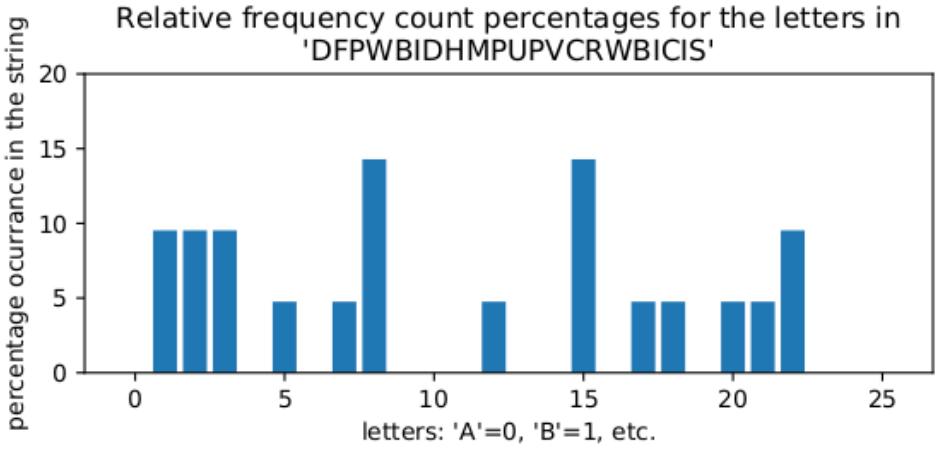
- 3 characters after "a" in the alphabet is "D,"
- 20 characters after "l" in the alphabet, wrapping around to the beginning of the alphabet when falling off the end, is "F,"
- 4 characters after "l" is "P,"
- 3 characters after "t" is "W,"
- 20 characters after "h" in the alphabet, wrapping around to the beginning of the alphabet when falling off the end, is "B,"
- *etc.*

Notice that we have achieved the goal of messing up frequency analysis approaches to breaking our cryptosystem: for example, the repeated letter "l" in the cleartext word

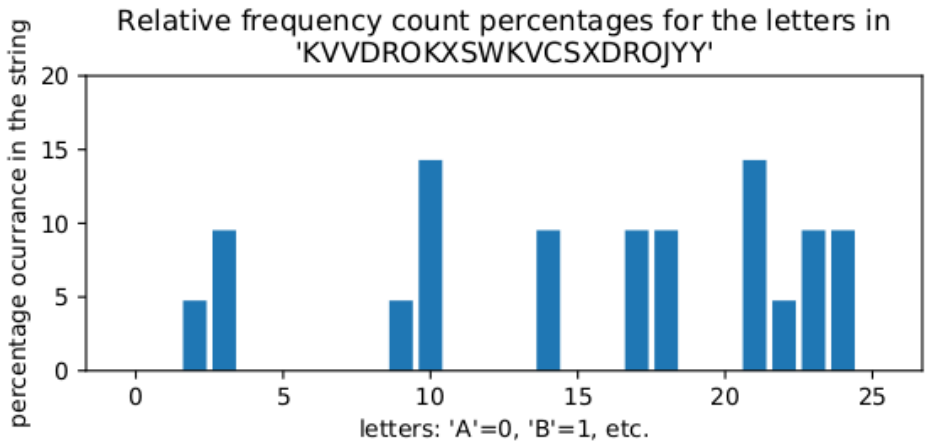
"all" has been encrypted into the two different letters "F" and "P" – but the "l" later in the cleartext, in the word "animal," has also been encrypted to "P" ... the frequency tables have been totally scrambled! In fact, we can see that by looking at the relative frequency charts of the cleartext



and for the ciphertext from Vigenère encryption with keys $3, 20, 4$



which looks completely different! Contrast this with the relative frequency chart for a Caesar encryption (with key 10) of the same cleartext



which looks like the relative frequency chart of the original cleartext, shifted horizontally.

Reading Response 1.4.1

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you feel like you understand the Vigenère cryptosystem?

Reading Response 1.4.2

To show you understand Vigenère, can you say what the keyspace is for a the Vigenère cryptosystem when it uses ℓ key values? And how big is that keyspace? (How big it is will give us some information about how hard it is to do a brute-force attack on this cryptosystem.)

Code Task 1.4.1

Write a **Python** function `Vigenere_encrypt(s, k)` which takes a cleartext `s` and list `k` of key values and does Vigenère encryption.
Likewise implement a **Python** function `Vigenere_decrypt(s, k)` which does the decryption for ciphertext `s` and list `k` of key values.

Using the Vigenère cipher with ℓ keys basically divides the cleartext into ℓ different pieces and uses the Caesar cipher on each of these different pieces, each time with a different key. As long as these pieces are sufficiently long, we can use our automated Caesar cipher cracker based on relative frequency counts on each piece separately, figure out what the separate Caesar keys are, and then do Vigenère decryption with those ℓ keys we've figured out.

Notice that the ℓ pieces of the cleartext into which the Vigenère cryptosystem are all the same size, that being `len(s)`$/\ell$ if the cleartext is `s`. Since an attack on the Vigenère cryptosystem is based on relative frequency-based attacks on the Caesar ciphers of those pieces, and such relative frequency-based attacks only work if there is enough data to make a good relative frequency table, if `len(s)`$/\ell$ is a small number, then the attack will fail.

In particular, if $\ell =$ `len(s)` then those ℓ pieces will have only one character in them and relative frequency-based attacks will fail miserably. In other words, if you use a Vigenère cryptosystem with ℓ keys on a cleartext of length ℓ, then it is perfectly secure! This is so important that this cryptosystem is worth giving its own name:

DEFINITION 1.4.2. A cryptosystem which is Vigenère where there are as many keys as letters in the cleartext is called a **one-time pad cryptosystem**, and the **pad** in this cryptosystem is the name used for the collection of all of those Vigenère keys.

It is a very important thing when using a one-time pad never to reuse the pad – that's why it's called a *one-time* pad – because if Eve guesses that the pad has been reused, she can crack all of the messages which were sent with that pad. We will not discuss here how that attack on encryption with a reused one-time pad works, but suffice it to say that the attack is based, again, on frequency analysis.

One final note about one-time pads: typically, when using a one-time pad, the keys are picked purely randomly. The advantage of doing that is that there is no way Eve can predict what the keys in the pad might be, even if she knows Alice and Bob very well. With a randomly chosen one-time pad, this cryptosystem is called *information theoretically secure*, meaning that is secure no matter how much computation power Eve can throw at the problem.

Reading Response 1.4.3

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you feel like you understand how one-time pads work?

Code Task 1.4.2

Write a **Python** function `OTP_keygen(n)` which generates a one-time pad of size `n`. As we said above, it's best if these pads are as random as possible, so you might want to do

```
from random import *
```

and then use the command `randint(a,b)` which returns a random integer in the range `a` to `b` (including both endpoints, which is unusual in **Python**!).
As an example of using your new `OTP_keygen(n)`, send your instructor an email with a one-time pad of at least 100 elements.

Code Task 1.4.3

Write a **Python** function `OTP_encrypt(s,k)` which takes a cleartext `s` and one-time pad `k` and does the encryption.
Also write a **Python** function `OTP_decrypt(s,k)` which takes a ciphertext `s` and one-time pad `k` and does the decryption.
As an example of using your new `OTP_encrypt(s,k)`, send your instructor an email with your first **Reading Response** from the course, but encrypted using that one-time pad you sent for Coding Task 1.4.2. (If that **Reading Response** was longer than 100 characters, you will have to truncate it to the first 100 characters so that you have enough one-time pad to do the encryption.)

Bonus Task 1.4.2

Since the quality of the randomness in a one-time pad can actually matter if you plan on encrypting very sensitive data, you might consider using a more sophisticated random number generator than the basic `randint(a,b)` from the `random` package ... which is fine for many uses, but maybe not good enough if you are trying to keep a large document secret from a powerful adversary.
Python has a function `urandom` in the package `os` which generates much stronger randomness. Unfortunately, it returns just a string of random bytes, which you need to convert to `int`s by using in the **Python** function `unpack` in the package `struct`. Here is a web page which shows how to use these two functions.
Write a **Python** function `OTP_keygenCS(n)` which generates a one-time pad of size `n` with *cryptographically secure randomness*.

Code Task 1.4.4

Write a **Python** function `OTP_key_finder(s, t)` which takes a string `s` consisting only of lower case letters and string `t` consisting only of upper case letters, where `len(s)` is the same as `len(t)` and computes what must have been the one-time pad which was used to get `t` as the ciphertext from `s` using the one-time pad cryptosystem.
The fact that you can do this means that when Eve sees a ciphertext from a one-time pad cryptosystem, she cannot deduce anything about the original cleartext other than its length. This is why this cryptosystem is information theoretically secure.
Hint: If you're looking at a letter `x` of cleartext and a letter `y` of ciphertext and wondering which number `k` in a one-time pad key could have encrypted `x` to `y`, you're really asking the question "what value of `k` makes

`chr((ord(x)-ord('a')+k)%26+ord('A'))==y`

be true?" Solving this for `k` would give one answer as

`k=ord(y)-ord('A')-ord(x)+ord('a')`.

Bonus Task 1.4.3

Suppose we somehow know that the following string is the Vigenère encryption of some English cleartext with a Vigenère cryptosystem having $\ell = 2$ keys:

```
WUZHEGJUFPRQJFFXEWIBDHEOVQUPVBFXIHRUJLYDMHTRDHKRSXIBTDVV
RUERKWFSIDZVVKZPKKVHMLCWYDKPVQURCLMHJDWWVUKKVPKKVJFRULJR
WWZQKHIUWGNLKKKKVLIEFQVVJRCHKLKEVZZWYFRHJDIWYHERSOVEIXKX
JKRWYWFOUBFXTDVVRUNDJDDEZWZRLVZIZWNHIHJRZWNDJDXUZHMRLVWD
LOKDEGXUZHMRLVCBYDKKTDVVRURQJZVUVGZWYHIHLQUHIOVDMHFISULW
LVRQUWYHIHJWWRIEIXKXJLJDEKFQFURECHDDEVFDIHKKVBROCDCOYRER
IDSOVPVQTRDHZWFVGHRNZQTDVVRUJILQVUROYHNDJPPIILVQUIRLKKWX
CDEGAXJWKRDHSXKEIXKXJVRBJKVZRVRPSLKLFXJDEGSULWLVZVRQYRER
IDSOVPRQYHYDKKSUFXXKKPRQPFRSKLMHJKFPVWFUFPVZYRJHIDEVFPJG
ZGKKVJVQVUROTRWIVUJIZOCGZGKKZVZQTDVVRUJHVPRPSLKLFXJZYHEW
YDKWYHGRFUYDMHTUZHUFRHJDIKRWYZVSKDDEZWZREVYRLOUEVPRGVRWV
KHIQVUJWLIWBVWSULWLVJDPVYHNDJDDEZWZRLVRQUEIXKXJLJDEKFQFU
RECHDDEBFXROCGZGJHVWYDKREWYHCXGHIFROZWYUZFVSIHJHEWVGYLDD
BLEJCBTUFZEZYLTKYHULUWYUZFVUVILVVZRVKKZVRPSLKLFQPHKEIXKX
JVRBJKVZRVRPSLKLFXJDEGJXIHYHZVRQYRERIDSOVPRQZVGHRNERKWFG
ZVGUFYVZYDKEIXKXJVGRBHSXKKVUVLRPKRJSVDBZYDKLURBQFZPRLDCO
ULUOFYVKZPFQTHERKZZWYRLWTDLVVZYDKFRXJHNLKKYRCGJBFXKKVQKR
DRLUEIFUYLDRAXUJDHEWKKFXRUKICHUWFEIXKLJKSHRVKVRQUPVQYDMH
CRJWKKVLIUVDJREEVDIZZWYPVPPKVDIWZVZQKKVFFIWLEWYHIHNLKKTD
VVRURQULDXJWGDLVVWZOCLKFFPVERFBWFPV
```

Can you figure out what the two Vigenère keys must have been, and what the original cleartext was?

1.5. Preliminary conclusion of preliminaries

We have seen that a very old cryptosystem, the Caesar cipher – which even has lived into the modern age in the form of ROT13 – helps illuminate some basic ideas of the design of cryptosystems and some first steps in attacking them: basics of cryptography and cryptanalysis.

The attack on the Caesar cipher which we discussed was based on *frequency analysis*. There's actually some interesting history here: As was mentioned above, the Caesar cipher was used by Julius and his nephew in the first century BCE. The approach of frequency analysis was first described nearly a thousand years later, in a book *Manuscript on Deciphering Cryptographic Messages* by the Muslim philosopher, mathematician, physician, and general polymath Al-Kindi. Like the mathematician and astronomer Al-Khwarizmi (whose name came down to modern times in the word *algorithm* and whose book *The Compendious Book on Calculation by Completion and Balancing* gave us the word *algebra*, from it's title in Arabic), Al-Kindi worked at one of the greatest centers of scholarship in the world in the 8th century CE, the House of Wisdom in Bagdad under the Abbasid Caliphate. Al-Kindi and Al-Khwarizmi seem to have both been involved in bringing an Indian system for writing numbers using positional notation (so: 101 means one 100 plus no tens plus one 1, whereas the Romans would have written that number CI) to the Arab world. This notation later spread to Christian Europe in the early 13^{th} century through a book called *Liber Abaci* (title in Latin; which would be *The Book of Calculation* in English) written by a man Leonardo of Pisa ... whom we know today as *Fibonacci*. Some have claimed that Fibonacci's notation helped bring about the Renaissance in Europe, since without good numerical notation, it would have been very hard to do double-entry bookkeeping, which was used by great trading houses like the Medici, who then had the funds to support the tremendous flowering of the arts which was the Renaissance....

The Vigenère cipher was invented in the mid-15^{th} century (**not** by Vigenère!), used for hundreds of years by European governments and even by the Confederacy during the American Civil War, and was described as recently as the early 20^{th} century as unbreakable (by none other than Charles Dodgson, the mathematician known to us today mostly as Lewis Caroll, the author of *Alice in Wonderland*!).

Reading Response 1.5.1

Had you heard of any of this history before? Did you know of Al-Kwarizmi and his connection to the words *algorithm* and *algebra*? Had you heard of the House of Wisdom in Bagdad? Or what about the theory that the Renaissance was made possible by the invention of double-entry bookkeeping (and therefore, indirectly, by the importation into Christian Europe of Hindu-Arabic numbers which made calculations so much easier than other systems like the Roman numerals)?

An enormously important thing we have learned in this chapter, however, is that:

There is a perfectly secure cryptosystem, the one-time pad.

The next time you are watching a movie or TV show which has some line like "There's no such thing as an unbreakable code," you should therefore laugh derisively.

Code Task 1.5.1

Before leaving this topic, we should implement a full, one-time pad cryptosystem which will work for any kind of data in any language.
The key to doing this is to think of any data, in any language (or not in a language at all), as being simply a string of bits. Bits, of course, are like letters from an alphabet which has only two symbols **0** and **1**. We know from §1.2.1 that Caesar – and, therefore Vigenère and one-time pads – only care about keys up to mod of the size of the alphabet, which is mod 2 for bits. So shifting the "letters" **0** and **1** by a key of 0 would do

$$\mathbf{0} \quad \mathbf{1} \overset{+0}{\mapsto} \mathbf{0} \quad \mathbf{1}$$

and shifting by a key of 1 would do

$$\mathbf{0} \quad \mathbf{1} \overset{+1}{\mapsto} \mathbf{1} \quad \mathbf{0}$$

Therefore, if the cleartext bit is a and the key bit is b, then the encrypted bit is $a \wedge b$ (this is the "exclusive or", XOR, operation).
Write a **Python** function `OTP_gen_keyfile(f, n)` which takes a string `f` and uses it as a filename into which to deposit `n` random bytes.
Write a **Python** function `OTP_fileencrypt(f,k)` which a string `f` and uses it as a name of a file of cleartext data to be encrypted. The encryption should use the one-time pad kept in the file with name `k` and the output should be put into the file with name `f+'.enc'`.
Some hints:

- You can open a file for reading as bytes with the command `file_handle = open(filename,'rb')`.
- You can get the entire contents of such a file into a byte string with the command `data=file_handle.read()`.
- If `a` and `b` are `int`s, then `a^b` is the bitwise XOR of `a` and `b`.
- If `bs` is a byte string, then `bs[i]`, (where `i` is in `range(len(bs))`) is an `int`.
- If you open a file with the flag `'wb'`, then you can write a byte string `bs` to that file with `file_handle.write(bs)`.

However, the enormous problem with one-time pads is that:

To remain secure, a one-time pad must have as many key digits as there are letters in the plaintext to be encrypted, and reuse of pads completely destroys the security of this cryptosystem.

What this means is that the distribution of very large keys is supremely important in the use of this perfectly secure cryptosystem.

There's a nice story of how important the management of one-time pads can be from a wonderful book called *Between Silk and Cyanide: A Codebreaker's War* [**Mar99**] on cryptology during (part of) World War II by the author Leo Marks. Marks ran spies into Nazi-occupied Holland during the war, and when he took over, the spies were given cyanide pills to take so that they could commit suicide if ever captured. That way they would not give away the cryptosystems they used to radio back to England all the secrets they uncovered. Instead, under Marks's direction, they were given large one-time pads, written on sheets of silk thin enough to be worn underneath their clothes when they were first sent to Holland. That way, if captured later, they could tell everything they new about the codes they used and without having the specific one-time pad used by other spies or the one they intended to use later, the Nazis could get no information from intercepted ciphertexts. Which all meant that captured spies had no particular reason to commit suicide, and instead could cooperate and perhaps live out the rest of the war in a POW camp, if captured.

This story of silk undergarments with cryptosystem keys written on them is the first time we see what will become a recurring theme in this subject: key distribution, management, and security can often be the most important practical consideration in applied cryptography.

CHAPTER 2

Block Ciphers

Most of the encryption today on the Internet is probably done with one of the cryptosystems called *block ciphers*, which do not provide as perfect cryptographic security as a one-time pad, but which have a much, much more modest level of difficulty in key management. Block ciphers are also very fast and have some nice security features that cryptosystems structured like the ones we've seen so far do no.

Examples of block ciphers you may have encountered might include:

- *Advanced Encryption Standard [AES]*, authorized by the National Institute of Standards and Technology [NIST] for use by the US federal government in unclassified situations;
- the *Data Encryption Standard [DES]*, which was the NIST-authorized block cipher from 1977 to 2001; and
- *3DES* or *triple DES*, a block cipher pretty much consisting of doing DES three times, which was used for a while after it was known that DES was nearly broken but before AES was standardized.

In this chapter, we will very briefly touch upon some of the most basic design principles for block ciphers, do a little **Python** implementation to explore these principles, and use some code libraries to do real encryption with state-of-the art block ciphers.

2.1. Why encrypt blocks of data: Shannon's *confusion* and *diffusion*

Our approach to the cryptanalysis of the Caesar and Vigenère ciphers was based on the frequencies that individual letters appear in the English language and also in a ciphertext we are trying to crack. For some cryptosystems, similar approaches might be used for longer strings of characters:

DEFINITION 2.1.1. A single letter (from some alphabet) in a piece of text such as a particular plaintext or ciphertext, or in a large collection of all English texts, is called a **monograph**.

Similarly, a pair of adjacent letters in the same kind of text source is called a **digraph**, and a sequence of three letters is called a **trigraph**.

So one way to describe our attacks on Caesar and Vigenère is to say that they compared the statistics of monographs in (various substrings of) a ciphertext with the same statistics for the English language, which turns out to be a fairly stable distribution of letter (monograph) frequencies.

Similarly, the frequencies of digraphs in English are fairly stable – *e.g., th* is quite common, as is *qu*, while both *tx* and *ql* are very common. Thus the statistics of digraphs (and, eventually, trigraphs) can be useful in attacking some cryptosystems.

Bonus Task 2.1.1

Ignoring non-alphabetic characters and the difference between upper and lower case, there are $676 = 26^2$ possible digraphs in English (26 choices for the first letter of the digraph and 26 choices for the second letter).

Write a **Python** function `digraph_freq(f, n)` which makes a relative frequency table of the digraphs from the text in the file with name `f`, printing out the most frequent `n` digraphs and their relative frequencies. You will want to read in the whole file but, as we have done before, discard all the non-alphabetic characters and make the remaining letters all lower case, before counting the digraphs.

Note also that the digraphs can overlap: that is, the string

humperdink

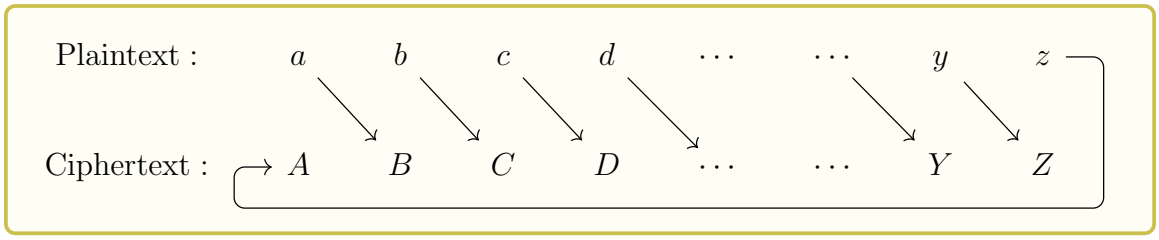
has these digraphs:

hu, um, mp, pe, er, rd, di, in, and *nk*

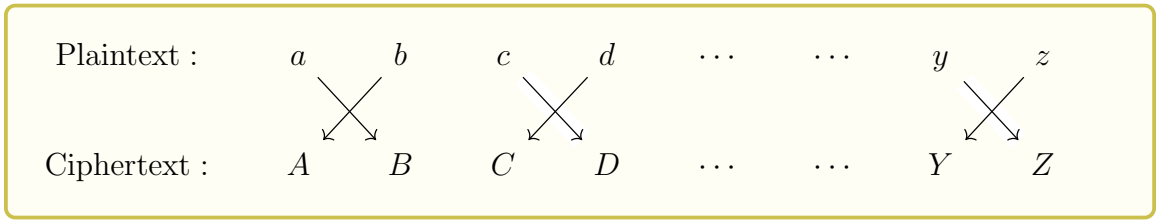
and not just the non-overlapping ones *hu, mp, er, di*, and *nk*.

Printing out the entire relative frequency table is easier than just the `n` most frequent digraphs. To do that, you will want to sort the relative frequency values, and then loop through them in the order of decreasing frequency. For each value of a frequency, you will then want to find all digraphs with that frequency and print it out, increasing a counter as you do. When the counter gets to `n`, you will stop.

The reason our attack on Caesar using monographs was successful is that Caesar doesn't hid the statistics of its cleartext letters very well: they're all in the ciphertext, just shifted over in the alphabet. In fact, so long as the way we scramble the letters in a cleartext to get the letters of the corresponding ciphertext is the same for every letter – maybe not simply shifting, like



but even other schemes like



which switches the even-numbered letters in the alphabet with the odd-numbered ones – the statistics of monographs in our ciphertext will be the same as in the cleartext, only in some different order. There's a name for cryptosystems that do this:

DEFINITION 2.1.2. A cryptosystem where a single scrambling transformation of letters in the cleartext to the letters in the ciphertext is called a **monoalphabetic cryptosystem**.

A cryptosystem where, instead, the scrambling transformation of letters in the cleartext to the letters in the ciphertext changes from letter to letter of the cleartext – so that, for example, sometimes a in the cleartext is encrypted to K in the ciphertext, but other times in the same cleartext, a might be encrypted to B or M or some other letter – depending upon perhaps the other letters in the message or the location of the cleartext letter in the entire cleartext, is called a **polyalphabetic cryptosystem**.

Reading Response 2.1.1

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you feel like you understand which cryptosystems are monoalphabetic and which are polyalphabetic – if so, which of the ones we've discussed so far are of which type?

So (monograph) frequency analysis will make good progress in cryptanalysis of monoalphabetic cryptosystems, but the relative frequency tables for ciphertexts from polyalphabetic cryptosystems are smeared out by adding together lots of differently scrambled versions of the frequency table and so cryptanalysis this way will be much harder.

In the end, what we are seeing is that when the ciphertexts produced by some cryptosystem have statistics that somehow reflect the statistics of the cleartext, then there is a possible avenue of attack. The best thing, then, would be if the cleartexts and ciphertexts seemed to be statistically unrelated to each other

The great mathematician, computer scientist, and cryptologist Claude Shannon[1] in his highly influential (and, initially classified by the US Government) paper *A Mathematical Theory of Cryptography*[**Sha45**]

DEFINITION 2.1.3. We say that a cryptosystem has the property of **diffusion** if, for a fixed choice of key, every time any single bit in a cleartext is flipped, half of the bits of the ciphertext should change (in general, statistically), and vice-versa. (Here, "statistically" means that this doesn't have to be precisely true every time, but when we average over many cleartexts and keys, on average half of the ciphertext bits will change every time we change any single cleartext bit.)

The point of diffusion is to minimize the information about the cleartext which is *leaked* by he ciphertext. The very fact that there is some communication between Alice and Bob is, of course, leaked. Looking at the flows of messages between individuals in some group – who speaks to whom and when or, in other words, the **metadata** of the communication – is called **traffic analysis** and can be very important. Seeing which individual is the central point through which all messages flow, and related issues from traffic analysis, can give information about who is the leader of the group, *etc.*. As former head of the US National Security Agency General Michael Hayden said[**Col14**], "We kill people based on metadata," referring to US drone strikes on individuals though to be terrorist leaders based on traffic analysis.

If if we are not trying to understand the organization of some group of communicating individuals on the basis of their pattern of communications, and are just looking at Alice and Bob's exchange of encrypted messages, some things are hard to conceal. For example, it is nearly impossible to send a very large piece of information (cleartext) through an encrypted channel unless the corresponding ciphertext is quite large.

Although, to be fair, the opposite, that a large ciphertext means the cleartext had a lot of information in it, is not necessarily true. For example, Alice could pad a short message with a very large amount of random extra data and send to Bob the encrypted version of

[1] Shannon was an enormously influential figure in the history of computer science – for example, he single-handedly invented the field now known as *Information Theory*. He was also a very colorful character: see his Wikipedia page.

that long cleartext. That would be a large ciphertext which did not correspond to a large amount of (useful) information in the cleartext.

Reading Response 2.1.2

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you feel like you understand how an observer could use traffic analysis to guess who was the leader of a group of communicating individuals? Can you think of another situations where something like traffic analysis comes up in regular life – maybe when friends or romantic partners suddenly are communicating with different people more or less frequently...?

Since it is very hard to prevent the leakage of at least the fact that communication has occurred, as well as, probably, some idea of how much data has be transmitted, a good cryptosystem will try to prevent as much other leakage as possible: Shannon's diffusion is one way to do that.

Code Task 2.1.1

On our way to seeing if the cryptosystems we've met so far have diffusion:
Write a **Python** function `str_diff_display(s,t)` which takes two string `s` and `t` and shows how much they differ.
Your function should first check if the strings have the same length, printing an error and returning 0 if not.
Next, your function should loop through the strings and print a "*" if they differ at each location and a "_" if they are the same. (You might want to print these symbols in rows of some determined length, like 30 characters. You can do this by printing with commands like `print('*',end='')` and `print('_',end='')` to stay on the same line, and then using `print('')` to get to the next line.)
Your program should then return the number of places where the two strings differ (*i.e.*, the number of "*"s printed).
For example, running

```
str_diff_display("abc"*30,("abc"*15)+"Abc"+("abc"*13)+"aBc")
```

should produce the output

```
______________*______________
______________________*_
2
```

Code Task 2.1.2

Actually, displaying where two strings have different characters, as `str_diff_display` does, is a pretty crude approximation of showing where those strings have different *bits*. So let's improve the string difference display and calculation code to make a bit difference display and calculation program.

In principle, the easiest way to do this is to convert a string of characters into a string of characters displaying all of the bits in the first string, and then use the `str_diff_display` you built before.

To get a character into its binary expression, you can convert it first to an `int` with `ord`, and then convert that `int` into a string of binary digits with `bin`. The problem with `bin` is that it always puts a `0b` at the front of its binary strings, and it also drops off leading 0s. (That makes sense: when writing the number 17 in base 10, we don't write it as "017," to indicate there are no 100s, 1 ten, and 7 ones ... we just write "17;" *i.e.,* we drop leading 0s. The same happens in base two.)

Of course, you can skip the first two characters (like that "`0b`") of a string `s` with `s[2:]`, and if you have a string `b` of 0s and 1s that you want to pad on the left with the right number of 0s to make it eight symbols long (to show all the bits in that byte), you can do `('0'*(8-len(b)))+b`.

Use the above ingredients to make a **Python** function `bits_diff_display(s,t)` which takes two string `s` and `t` and shows how much their bits differ. Since there will be eight times as may bits to show as there were just for `str_diff_display(s,t)`, you might want to make your lines a bit longer, and maybe show if the bits are the same with a "." and if they differ with an "x."

For example, running `bits_diff_display("abcdef", "CbcdeF")` should produce the output

```
..+...+.....................
...........+.....
3
```

(Even though running `str_diff_display("abcdef", "CbcdeF")` would produce the output

```
*___*
2
```

The difference is because while the strings `"abcdef"` and `"CbcdeF"` differ only in two characters, the first and the last, the bits for `"a"` are `01100001` though the bits for `"C"` are `01000011` – so there are **two** different bits in this **one** different character – and the bits of `"f"` are `01100110` though the bits for `"F"` are `01000110` – so there is only one different bit in this one different character!)

Code Task 2.1.3

Which of the cryptosystems that we have studied so far – Caesar, Vigenère, and one-time pads – satisfies Shannon's idea of diffusion?
Take a fairly long – maybe 100 characters or so – cleartext from some source, and make two versions of it which differ by only one bit.
[Note: think a bit about how to make that one-bit change. Changing one letter may not do it! For example, the letter **a** is represented in Unicode by the number 97, which is has binary expression **01100001**. On the other hand, **b** in Unicode is 98, which is **01100010** in binary – so changing **a** to **b** changes *two* bits!]
Once you've got those two strings, make appropriate keys and encrypt both strings, then run your `bits_diff_display` on the encrypted versions to see where and how much they differ. Try this for each of the cryptosystems we've done in this class.
You should now be able to answer the question if those cryptosystems satisfy Shannon's diffusion.

Another thing Shannon worried about was if the ciphertexts leaked some information about the *key* that was used in their encryption. In particular, he made a definition similar to diffusion, but now looking at keys, as follows:

DEFINITION 2.1.4. We say that a cryptosystem has the property of **confusion** if, working with a fixed cleartext, every time any single bit in a key is flipped, half of the bits of the ciphertext should change (in general, statistically), and vice-versa. (Here, 'statistically" means that this doesn't have to be precisely true every time, but when we average over many ciphertexts and keys, on average half of the ciphertext bits will change every time we change any single bit of the key.)

Code Task 2.1.4

Which of the cryptosystems that we have studied so far – Caesar, Vigenère, and one-time pads – satisfies Shannon's idea of confusion?
Repeat the previous Code Task 2.1.3 with now a fixed cleartext but trying encryptions with only one bit changed in the key, and see how much of the ciphertext changes.

The conclusion of this discussion of information leakage is that we should seek cryptosystems that satisfy confusion and diffusion: they must smoosh up and mix up the bits of the cleartext and the key, so that, in general, (nearly) every bit of the ciphertext depends upon (nearly) every bit of both the cleartext and the key. This means that we must process the whole cleartext at once, we cannot handle the bits (characters) of the cleartext one at a time, as the cryptosystems we've seen so far have done.

This is not easy to do with a fixed cryptographic algorithm for a general cleartext, so a common approach has been work with chunks of the cleartext at a time, called blocks, where the blocks all have the same. At least on the level of blocks, we can hope that our new cryptographic algorithm will satisfy confusion and diffusion and therefore reduce information leakage.

This leads us to:

DEFINITION 2.1.5. We say that a cryptosystem is a **block cipher** if it only is defined when it's input cleartexts and ciphertexts are of some fixed size, called the **block size** and if it is deterministic (*i.e.,* it has not built-in randomness) so it's result depends only upon the input clear- or ciphertext and the key.

Reading Response 2.1.3

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Does it make intuitive sense to you that confusion and diffusion would be desirable characteristics for a cryptosystem, and how block ciphers might relate to that?

We mentioned at the very beginning of this chapter several well-known block ciphers. Such cryptosystems – or at least the ones that have been widely used – tend to be based on very primitive but very fast mixing operations on the bits in the messages and the key. One particular strategy is to take pieces of the key and do various operations on it and the message block, to make something called the *round function*. Then the full encryption algorithm simply applies the round function some number of times, each time with a new piece of key data, again and again to the slowly transforming message block.

2.2. Encrypting a block at a time with AES

Let's get some practice with using a block cipher by looking at the case of the *Advanced Encryption Standard [AES]*. The key for AES must be of length 16, 24, or 32 bytes [to say it another way: 128, 192, or 256 bits] while its block size is 16 bytes [or 128 bits].

Standard **Python** distributions should come with the `Crypto` package, which includes `AES` among the set of implemented ciphers in `Crypto.Cipher`. Therefore before you can use AES, you must do

```
from Crypto.Cipher import AES
```

You can then create an object to work with this cipher by

```
cipher=AES.new(key, AES.MODE_ECB)
```

The `key` must be one of AES's recognized key sizes, or else this attempted object creation will throw an error (and we'll discuss what that `AES.MODE_ECB` is doing later...).

An AES object created this way can be used for encryption or decryption (or both) of a block of 16 bytes of cleartext or ciphertext, always with the key you used to create the object. For instance, if your key was `"asdfghijklmnotuv"` then here is an example of encryption

```
>>> cipher.encrypt("16 bytes of fun!")
b'~\x94\xaa\xf6\xc20\x85\xce\x83\x91\xc9\}\xdf\x96\xb8\xcf'
```

and here is an example of decryption (assuming the key is still `"asdfghijklmnotuv"`)

```
>>> cipher.decrypt(
... b'~\x94\xaa\xf6\xc20\x85\xce\x83\x91\xc9}\xdf\x96\xb8\xcf')
b'16 bytes of fun!'
```

Code Task 2.2.1

If AES is going to be a useful standard cipher for wide use on the Internet, it should run very fast. Let's test how fast this by timing how long it takes to encrypt one block. ...Actually, to be safe – *e.g.,* in case some other process takes some time on your computer while you think it is only doing an encryption, it is best to do something like 1000 encryptions, measuring the total time, and then dividing that by 1000.

A quite direct way to do measure how long something takes is first to run

```
import time
```

You can then save into a variable the number of seconds since the so-called "Unix Epoch," (which was 00:00:00 UTC on 1 January 1970) by

```
start_time=time.time()
```

If you then do some processing, you can calculate how long it took with

```
elapsed_time=time.time()-start_time
```
.

When using this approach to measuring computational speed, you should be careful only to do the repeated runs of the commands you want to time between the two invocations of `time.time()` – run nothing superfluous! For example, make one AES cipher object before you start the timing, so that the computational cost of building that object doesn't get counted when you are trying to figure out the cost of encryption (or decryption).
With all that in mind, make some 16 byte AES key and 16 byte cleartext, time how long it takes to do 1000 encryptions and 1000 decryptions, and report on how long it takes, on average, for each encryption or decryption. It would be nice to change the message and/or the key each of the 1000 times – but call some random number generator in the loop, then you'll be timing *that* as much as encryption/decryption!

Code Task 2.2.2

Does AES satisfy Shannon's idea of diffusion?
To see, take some cleartext of length 16 bytes and make two versions of it which differ by only one bit.
Then make a 16 byte key, create an instance of AES, and make ciphertexts for the two different cleartexts (but the same key). Use your program `bits_diff_display` to see where and how much the two ciphertexts differ.
You might also try the above for several different keys, just to make sure that it's not a fluke which is special for the choice of key you happened to make.

Code Task 2.2.3

Does AES satisfy Shannon's idea of confusion?
To see, take some cleartext of length 16 bytes. Make two different keys of length 16 bytes which differ by only one bit. First with one key, make an AES cipher object and encrypt the fixed cleartext, then do the same using the other key. Use your program `bits_diff_display` to see where and how much the two resulting ciphertexts differ.
You might also try the above for several different ciphertexts, just to make sure that it's not a fluke which is special for the choice of key you happened to make.

Reading Response 2.2.1

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Was it easy or hard to use **Python**'s AES? What do you find most annoying and/or convenient about this implementation of AES?

2.3. Encrypting more or less than a block with a block cipher

If Alice and Bob know they will always be exchanging messages which are exactly the number of bytes in the block size of the block cipher they are using, they're in very good shape. What is much more likely, though, is that they will send smaller or (much) larger messages, and we ought to help them do that securely.

2.3.1. Very small messages need some *padding*. When a message is smaller than the block size, it seems pretty easy simply to add some extra stuff to fill up an entire block: this extra data is called **padding**. The problem is, if the message can could be arbitrary data, then how do we tell what is message and what is padding?

A common way to do this, when we know that there will have to be some padding (maybe metadata that goes along with the message, like in the subject line of an email or in the packet metadata for an encrypted packet sent on a packet-switched network), is to assume that there is always at least one byte of padding: the last one. That byte can carry information, though, so it can simply tell the number of bytes of padding or, conversely, the number of actual non-padding bytes in the block.

Code Task 2.3.1

Write a **Python** function `pad_string(s, n)` which takes as input a string `s` and block size `n` and returns a string of length `n`, using padding as described above. If `len(s)>n-1`, you should print an error and return the empty string, since there will be no room for padding. Otherwise, return a string whose first characters are just a copy of `s` and whose remaining bytes are just some copies of the number `len(s)`.
Also write a function `unpad_string(s)` which takes as input a string `s`, assumed to come from the `pad_string()` function, and returns the original string without padding.
For `unpad_string`, you can assume the block size used in the `pad_string` was simply `len(s)` and the last byte of `s` is a number telling how much of `s` is the original string. Therefore, if `ord(s[-1])>len(s)-1` then there's some sort of problem and you should print out an error message and return the null string. Otherwise, return the first `ord(s[-1])` character substring of `s`.
E.g., `pad_string('test',16)` should return

`'test\x04\x04\x04\x04\x04\x04\x04\x04\x04\x04\x04\x04'`

and `unpad_string('fishhead\x08\x08\x08\x08\x08\x08\x08\x08')` should return

`'fishhead'`.

Code Task 2.3.2

Write a **Python** function `smallAESencrypt(k, s)` which takes as input an AES key `k` and cleartext string `s`, where `len(s)<16` and returns the AES encryption of the padded version of `s`.
If `len(s)>15` or if `k` is not a valid AES key, you should print an error and return the empty string.
Also write a function `smallAESdecrypt(k, t)` which takes as input an AES key `k` and AES ciphertext `t`, assumed to come from `smallAESencrypt(k, s)` function, and returns the original cleartext string `s` (without padding).

Reading Response 2.3.1

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Can you think of a different (or better) padding scheme?

2.3.2. Larger messages require *block chaining*. What if a cleartext is longer than the block size of the block cipher we are using? The obvious thing would be simply to break the cleartext into some number of full blocks and then one last block which is smaller, and to use regular block encryption on the full blocks and padding-plus-encryption on the last, smaller block.

Code Task 2.3.3

Implement that scheme for the AES block cipher. That is:
Write a **Python** function `naiveAESencrypt(k, s)` which takes as input an AES key `k` and cleartext string `s`. It should then build up a return value as follows:

(1) for each full block of cleartext (there may be none such, if the `len(s)<16`), append to the return value the regular AES encryption of that block with key `k`
(2) append to the return value the output of `smallAESencrypt(k, small)`, where `small` is the last piece of the cleartext, whose length is less than 16 bytes.

Also write a **Python** function `naiveAESdecrypt(k, t)` which takes as input an AES key `k` and ciphertext string `t`. It should then build up a return value as follows:

(1) if `t` has more than 16 bytes, for each block of 16 bytes in `t` except the last, append the plain AES decryption of that block with key `k` to the return value
(2) when done with that, append `smallAESdecrypt(k, last)` to the return value, where the variable `last` contains the last 16 bytes of `t`

But now we've got to wonder if this naive approach to AES on large messages still satisfies Shannon's diffusion condition.

Code Task 2.3.4

Re-do Code Task 2.2.2 but now using two cleartexts of length 80 bytes which differ by one bit near the middle, and using the `naiveAESencrypt` function.

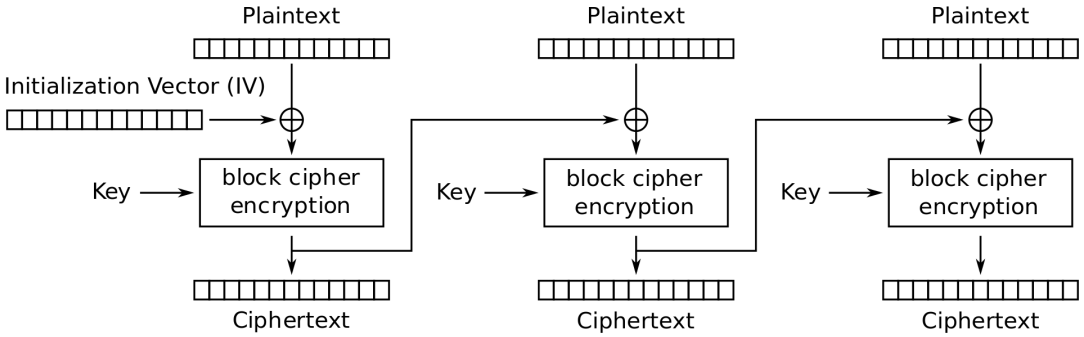
How are we going to get a block cipher that satisfies Shannon's diffusion condition? A good block cipher will diffuse nicely ... within a block. But the block size is fixed, and we have to somehow spread the diffusion – which, remember, basically means that changing on bit in the cleartext changes around half of the bits of the ciphertext – around in each block. One way might be just to design a new block cipher for each new cleartext whose block size is larger than the size of the cleartext and which does nice diffusion across that block. But it's so hard to verify that ciphers are really secure, this is completely impractical.

Instead, the most common approach is to use a block cipher of some fixed size (like AES) and to make the encryption of each block depend also on what happened with previous blocks: this is called **block chaining**.

There are two things we need to think about for block chaining.

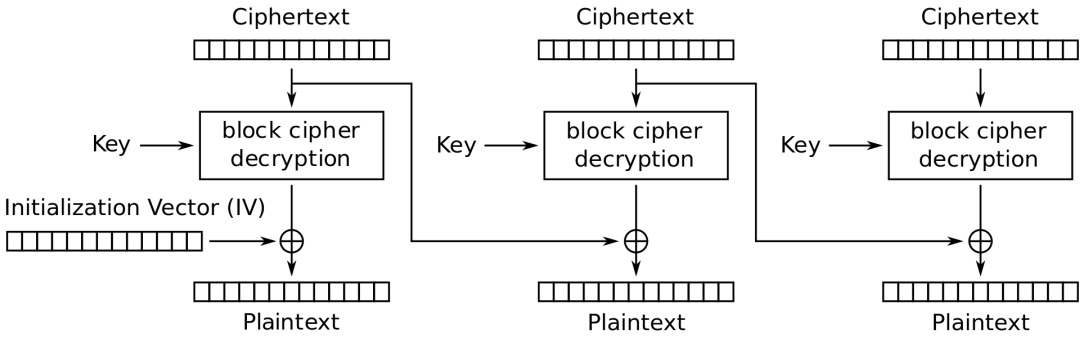
(1) **How exactly to chain the blocks:** There are various ways to chain the encryption of blocks and achieve diffusion. Here's one: Suppose we change one bit in a block of cleartext. If we're using a well-designed block cipher, about half of the bits of the ciphertext will change as well. We can use these different bits to change the bits of the next block of plaintext, by bitwise XORing (*i.e.*, bitwise addition mod 2) the ciphertext block with that next block of cleartext. That will propagate the change to half of the bits of the corresponding ciphertext, and repeating for each successive block, we will have spread the changed bits all down the sequences of blocks of the ciphertext coming from the long cleartext.
(2) **How to start the chaining:** The very first block will not have a predecessor block off of which to chain. It turns out that this can cause security issues, so the way to handle this is to make a block's worth of random data, called the **initialization vector (IV)**, and to start chaining as if that IV were the ciphertext from the previous block. Alice should then send the random IV that she chose along with the ciphertext so that Bob can decrypt the long message.

DEFINITION 2.3.1. The block chaining approach just described is called **Cipher Block Chaining** or **CBC mode** block chaining. Symbolically, it works like this:



Cipher Block Chaining (CBC) mode encryption

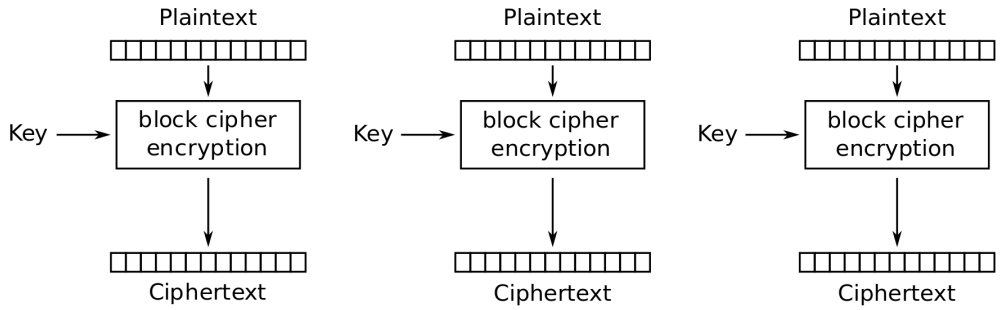
for encryption and like this:



Cipher Block Chaining (CBC) mode decryption

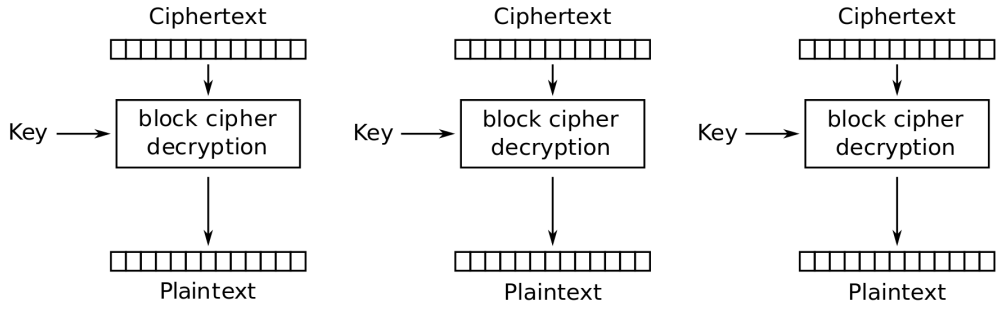
for decryption. (In diagrams like this, the symbol " $\oplus$ " stands for bitwise XOR (*i.e.*.)

Note that the way we discussed before of doing encryption of long messages with a block cipher, which simply treated each block entirely separately and failed Shannon's diffusion condition is, in the context of block chaining schemes, called **Electronic Codebook mode** or **ECB mode** chaining and has the simple diagram:



Electronic Codebook (ECB) mode encryption

for encryption and



Electronic Codebook (ECB) mode decryption

for decryption.

The way a block chaining mode is chosen in **Python** is in the creation of a `Crypto.Cipher` object. Where, in the past, we have used `AES.MODE_ECB` for Electronic Code Book chaining, we can use instead `AES.MODE_CBC` for cipher block chaining as described above. Since that mode requires an initialization vector, Alice must specify the 16-byte IV they want to use, as follows:

```
from Crypto.Cipher import AES
key=b'randobytes 4 key'  # not a good choice of key
iv=b'IV of randobytes'   # terrible choice of IV
cipher=AES.new(key,AES.MODE_CBC,iv)
ciphertext=cipher.encrypt(b'some cleartext w/ len two blocks')
message=iv+ciphertext    # put together IV and ciphertext
                         # to be transmitted to Bob
```

The original message can then be recovered with

```
from Crypto.Cipher import AES
key=b'randobytes 4 key' # Bob must know the same key
iv=message[:16]         # Bob grabs the IV out of the message
ciphertext=message[16:] # the rest of the message is ciphertext
cipher=AES.new(key,AES.MODE_CBC,iv)
cleartext=cipher.decrypt(ciphertext)
```

Note that the `Crypto.Cipher` object is **stateful** in CBC mode – meaning that it remembers what it has seen before. When it is created, it uses the given IV to start the chaining, but thereafter, every call to `encrypt` chains off of the previous block. That is, the way of encrypting the two-block cleartext `b'some cleartext w/ len two blocks'` described above yields exactly the same result as doing instead

```
from Crypto.Cipher import AES
key=b'randobytes 4 key'  # not a good choice of key
iv=b'IV of randobytes'   # terrible choice of IV
cipher=AES.new(key,AES.MODE_CBC,iv)
ciphertext_part1=cipher.encrypt(b'some cleartext w')
ciphertext_part2=cipher.encrypt(b'/ len two blocks')
ciphertext_full=ciphertext1+ciphertext2
message=iv+ciphertext    # put together IV and ciphertext
                         # to be transmitted to Bob
```

Reading Response 2.3.2

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Can you think of a different (or better) approach to block chaining? Do you see why chaining is good, and why an initialization vector is a good idea?

Finally, we can see to what extent block chaining makes a version of AES which satisfies Shannon's diffusion condition ... to some extent.

Code Task 2.3.5

Re-do Code Task 2.3.4 but now using AES in CBC mode.

2.4. Some concluding observations for block ciphers

Modern block ciphers like AES can run quite fast (that should have been the conclusion of Coding Task 2.2.1). Using some, but not all, block chaining modes, they can satisfy Shannon's diffusion condition, at least a sort of *forward diffusion* in that when two messages differ by one bit, then about half the bits in the corresponding blocks in the two ciphertexts are different, and also for all blocks after (but not before!) that one.

This makes block ciphers quite good to use in practice... assuming they're secure. Fortunately, at the moment, there is no known, general-purpose attack against the AES cryptosystem better than brute force.

Let's think through some issues around a brute-force attack on AES. The key size we've been using is 16 bytes, or 128 bits. If a random key is chosen, then 2^{127} keys will have to be tried, on average, in a brute-force attack before the correct one is guessed. 2^127 is about 1.7×10^{38}, which means that if you could check a billion possible keys a second, then it would take roughly 100 billion times the life of universe (which is around 4.3×10^{17} seconds) to do one brute-force attack on AES.

Code Task 2.4.1

We should at least try a brute-force attack against AES ... at least in a very special case (we don't want to wait billions of times the life of the universe!).

Suppose Eve thinks Alice has sent her password to Bob in a message which was encrypted with the AES cipher using Electronic Code Book mode chaining. Eve also knows that Alice and Bob's favorite number is 17, so she thinks that they will probably use a key which is mostly 17s. In fact, Eve guesses that the key will be 15 bytes all with the value 17 followed by one new, random byte.

Eve can try all possible keys of this form with code like this:

```
key_ints=[17]*16  # last 17 is probably not right,
                  # it's just here to make a list of
                  # the correct length
for i in range(256):
  key_ints[15]=i
  possible_key=bytes(key_ints)
  # make an AES object with that possible_key
  # and use it to try decrypting the ciphertext
```

Eve also guesses that the word "password" will appear in the cleartext. **Python** has a nice command to check if a substring is present in a string, the command `in`, which works both with normal strings and substrings and also byte strings and

potential byte substrings. Eve can use this as follows:

```
  # inside a loop making attempted decryptions which
  # are in a variable possible_cleartext
  if b'password' in possible_cleartext:
    print('Found the cleartext!  It should be: ',end='')
    print(possible_cleartext)
```

If Eve were to try the above approach to decrypt the stolen ciphertext
`b"h9\xf2\x12,\x5e.\xab\xf4M\x1e\xf5\xc9\xcea\r"`
what password would she find, and what would be the key that unlocked it? [Show all of Eve's code as well as answering those questions!]

Code Task 2.4.2

Repeat the previous Code Task 2.4.1, but suppose now that Eve thinks that only the first 14 bytes of the key are 17s, the last two might be anything.
In this situation, try to use the same general approach (but probably with two nested loops, to try all 256 possible bytes in both of the last two bytes of the key) to decrypt the stolen ciphertext
`b"\xe0\xb3W\x94/X\xe7.\x93\xda\xe9\xed\xde\xedT\xfe"`

Bonus Task 2.4.1

Repeat the prior Code Task 2.4.1 one more time, but suppose now that Eve thinks that 15 of the bytes of the key are 17s, while the last byte – call it the "wildcard key byte" – might be anything ... and that wildcard byte might be at any location of the key.
In this situation, try to use the same general approach (but probably with two nested loops, one to loop through the 16 possible locations for the wildcard key byte, the other then to try all 256 possible values for the wildcard byte) to decrypt the stolen ciphertext
`b"0\xbd\x9e*\x9b\xd4\x03\xa58<\xfa\x0e\x86J\xfd@"`

In the Code Tasks 2.4.1 and 2.4.2 and the Bonus Task 2.4.1 above, we imagined that Eve would somehow know that the cleartext behind the ciphertext she had stolen contained the substring `b'password'`. This kind of thing has a name:

DEFINITION 2.4.1. If a cryptanalyst is trying to discover the cleartext which generated a particular ciphertext and knows (or at least hopes) some substring of the unknown full cleartext, then that substring is called a **crib**.

Cribs are actually not all that uncommon. Cleartexts often have a standard header which includes the date and time, which the cryptanalyst might know for other reasons, so the correctly formatted time-and-date string can be used as a crib. Famously, the Nazi military communications during WWII always included a well-known phrase signaling fealty to the leader of the Third Reich[2], which could be used as a crib.

Reading Response 2.4.1

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Can you think of a "crib" in messages you often send or receive, maybe to or from a family member, or someone (or some organization) with which you regularly communicate on the 'net for work, school, or entertainment?

Bonus Task 2.4.2

Here's an entirely non-coding Bonus Task:
Cribs were very important in cracking the Enigma cryptosystem which was used by the Nazis during WWII – which shortened the war by years, it has been estimated (see "Alan Turing: The codebreaker who saved 'millions of lives'" from The BBC). Write a one-page (or so) "mini-paper" on the Enigma, as a cryptosystem, its role in history, and something about how it was cracked. This should include

- A short description of how the Enigma cryptosystem worked – use as many of the technical terms from this course that you can (*e.g.*,, what was the *keyspace*, was it a *mono-* or *polyalphabetic cryptosystem* or neither, *etc.*); this should be about half of your "mini-paper."
- A very short description of the Enigma's historical role in the war; about one sixth of your "mini-paper."
- A short, very high-level description of the effort made by the Allies to crack the Enigma cryptosystem – mention *cribs*; about one third of your "mini-paper."

Sources you might use are:

- The BBC story mentioned above.
- The Wikipedia page "Enigma machine"
- The Wikipedia page "Cryptanalysis of the Enigma"
- The Crypto Museum's page "History of the Enigma"

We have seen that block ciphers can be fast, easy to use, and highly secure.

[2]Which we will not quote exactly here, because it would be unpleasant to have that odious phrase be a substring of this book!

But we did see that the security is based in part on the size of, and lack of structure in, the keys: it was that keyspace of size 2^128, along with a lack of some foolish structure like "15 of the 16 bytes are just representations of the number 17," which made AES so secure.

Which means that it is really important for Alice and Bob to be able to share – very accurately, since even a single incorrect bit with scramble their messages catastrophically! – a large, random-seeming key. In the modern, networked, world, where we often want to share valuable information very securely with people whom we may have never met in person, this is a serious defect. You may want to get your credit card number to an online store from your laptop, first through the open WiFi at a coffee shop, then across the (insecure by design) Internet! Cryptography is clearly a vital tool to do a thing like that, but the cryptosystems we have seen so far all require a shared secret, the key, which is hard to achieve in common modern situations like this.

In the next chapter we start to talk a solution to this problem of key sharing: cryptosystems where [part of] the keys can be public,

CHAPTER 3

Asymmetric [Public-Key] Cryptosystems

We have seen that there are secure, fast, (relatively) easy-to-use cryptosystems, but which do require a special activity before the regular communication can start: secure, secret key sharing. There are various approaches to getting around the practical difficulty this may present in the real world (such as, noted above, getting your credit card number securely to an online vendor with whom you have not had the opportunity to perform the secret key-sharing ritual), including:

- There are protocols which allow two parties to create a shared secret by exchanging messages on a public network: Alice and Bob must merely exchange two messages and they will end up with a common secret value that Eve, even if she saw both of the messages, will not be able to compute. That shared value can then be used, directly or after further processing, as the key for a block cipher for more secure communication. The most famous of these protocols is Diffie-Hellman key exchange, about which more information can be found, *e.g.*, here:
  - The Wikipedia page "Diffie-Hellman key exchange"
  - a nice survey article [**Hel02**]
  - the original paper about their discovery by Diffie and Hellman [**DH76**]

  The details of Diffie-Hellman are based on some modest mathematics: it is quite accessible to an advanced undergraduate mathematics major. It's security rests on a hypothesis which is widely believed – but not proven! – by many mathematicians and computer scientists.
- Instead of basing a secure way for parties to build a shared secret by communications over a public channel on theoretical hypotheses, there is way to do the same thing using some specialized hardware and a deep understanding of quantum mechanics. Called **quantum key distribution, [QKD]** and invented in the 1980s, this approach has been demonstrated first in the laboratory over ever longer distances since then, but is not yet in wide use in practice. For more information:
  - The Wikipedia page "Quantum key distribution"
  - a general introductory site: Quantum computing 101
  - the original 1984 research paper on this subject, by Bennett and Brassard [**BB20**]
- Another approach, with wide and interesting application is the subject of most of this chapter: asymmetric cryptosystems.

3.1. Symmetric, asymmetric, and salty cryptosystems: basics

Looking more closely, the problem we noticed at the end of the last chapter stems from the fact that Alice and Bob both have to have the same key to encrypt and decrypt, which they must therefore keep away from Eve. After all, if Eve had the key, she could encrypt and send messages to Bob pretending that they came from Alice, and Eve could also intercept and decrypt the real messages that Alice was sending to Bob. That's the problem with using the same key. Cryptosystems that do this have a name:

DEFINITION 3.1.1. A **symmetric cryptosystem** is one in which the same key is used both to encrypt cleartexts to ciphertexts and also to decrypt ciphertexts to cleartexts.

Sometimes symmetric cryptosystems are also called **private-key cryptosystems**.

Graphically, symmetric cryptosystems work like this:

Symmetric [private-key] cryptosystem, graphically:

Alice $\longleftrightarrow$	private agreement of shared key k $\longleftrightarrow$	Bob
	on public network (where Eve is watching)	
creates cleartext message m_A; using k, encrypts m_A to c_A and transmits c_A	$\rightarrowtail$ ciphertext c_A $\rightarrowtail$	receives c_A; using k, decrypts c_A and recovers cleartext m_A
receives c_B; using k, decrypts c_B and recovers cleartext m_B	$\leftarrowtail$ ciphertext c_B $\leftarrowtail$	creates cleartext message m_B; using k, encrypts m_B to c_B and transmits c_B
etc....		

The problem we're dealing with now is the initial private agreement of the secret key k.

A way to get around this problem would be if Alice and Bob used different keys for encryption and decryption. There's a name for this type of cryptosystem:

DEFINITION 3.1.2. An **asymmetric cryptosystem** is one in which one key k_e – called the **encryption** or **public key** – is used to encrypt cleartexts to ciphertexts, while a different key k_d – called the **decryption** or **private key** – is needed to decrypt ciphertexts to cleartexts.

Sometimes asymmetric cryptosystems are also called **public-key cryptosystems**, because usually when one is using them, one posts one's encryption key k_e in a public place to everyone to see and use.

Graphically, asymmetric cryptosystems work like this:

Asymmetric [public-key] cryptosystem, graphically:

Alice	on public network (where Eve is watching)	Bob
generate key pair (k_e^A, k_d^A) download k_e^B publish k_e^A	$\leftarrow\!\!\!\!\!\!\leftarrow$ public encryption key k_e^B $\leftarrow\!\!\!\!\!\!\leftarrow$ $\rightarrow\!\!\!\!\!\!\rightarrow$ public encryption key k_e^A $\rightarrow\!\!\!\!\!\!\rightarrow$	generate key pair (k_e^B, k_d^B) publish k_e^B download k_e^A
create cleartext message m_A; using k_e^B, encrypt m_A to c_A and transmit c_A	$\rightarrow\!\!\!\!\!\!\rightarrow$ ciphertext c_A $\rightarrow\!\!\!\!\!\!\rightarrow$	receive c_A; using k_d^B, decrypt c_A and recover cleartext m_A
receive c_B; using k_d^A, decrypt c_B and recover cleartext m_B	$\leftarrow\!\!\!\!\!\!\leftarrow$ ciphertext c_B $\leftarrow\!\!\!\!\!\!\leftarrow$	create cleartext message m_B; using k_e^A, encrypt m_B to c_B and transmit c_B
etc....		

Reading Response 3.1.1

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you understand how Alice and Bob would use an asymmetric cryptosystem?

The most basic part of designing *secure* asymmetric cryptosystems is that **it must be infeasible for Eve to compute the decryption key k_d even when she knows the [public] encryption key k_e**. After all, the encryption key is public, so if it were feasible to do this calculation, Eve would simply do that and be able to decrypt all ciphertexts herself.

Another consideration for the security of these systems is that the space of possible decryption keys must be very large. Otherwise, Eve could simply do the brute-force attack of trying all of the possible decryption keys.

Actually, for asymmetric cryptosystems there is another possible brute-force attack which goes like this: If Eve had a pretty good idea of a fairly small set of possible messages that Alice might be encrypting to send to Bob, Eve could simply use the public encryption

key herself and do all of those encryptions. If one her guesses for the original message was correct, then its encryption will be the same as the ciphertext which she saw on the public network, going from Alice to Bob. Therefore, the space of possible messages must be quite large to prevent this kind of brute-force attack.

There is actually a nice way to prevent this message space-based brute-force attack, by making a cleartext message artificially larger before encrypting it. One usually does this by adding some

DEFINITION 3.1.3. **Cryptographic salt** is random data added to a message Alice wants to sent to Bob before she encrypts it, which is automatically removed at Bob's end, after he decrypts the ciphertext.

Once Alice salts her message, if Eve wants to do the message space-based brute-force attack, she must try encrypting all possible messages Alice might have sent, *plus* all possible random choices of salt. With even a relatively modest number of salt bits, this will make this brute-force attack entirely too time-consuming.

Reading Response 3.1.2

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you understand this new, "message space-based brute-force attack" and how cryptographic salt prevents it?

One last remark about keys and *generating them*, for asymmetric cryptosystems: Keys for asymmetric cryptosystems are more delicate than for symmetric cryptosystems. Typically, one picks a private/decryption key of a certain size – large, to prevent brute-force attacks, and often using strong randomness, also to prevent brute-force attacks which would be possible if the adversary knew something about the structure of the keys you are likely to choose.

Then one does a computation using that private/decryption key to create the corresponding public/encryption key. This computation must be very hard to *undo*, since otherwise Eve could find the private key from knowing its public key!

Therefore the whole process of key creation can be quite complicated for asymmetric cryptosystems. Implementations must have a key generation step which can actually be quite time-intensive, but which is run only once by the person who wants to be able to receive encrypted messages from strangers on the 'net. This is of course in contrast with the situation of symmetric cryptosystems, where Alice and Bob simply agree upon some key with the correct number of bytes, however they want, and start encrypting and decrypting right away: there is no special "*key generation*" step for symmetric cryptosystems.

3.2. Using the RSA asymmetric cryptosystem in Python

One of the oldest and still most widely used[1] asymmetric cryptosystems is know as **RSA**, so named in honor of its three inventors: Ron **R**ivest, Adi **S**hamir, and Leonard **A**dleman. We're going to use a **Python** implementation of RSA to explore the basic features of using asymmetric cryptosystems such as key generation, encryption, and decryption, first in an entirely straightforward way that is not entirely secure. Then we will revisit these basic functions in a slightly more complex configuration that is actually secure.

3.2.1. Straightforward – not completely secure! – RSA in Python. The **Python** module `Crypto` has a submodule `Crypto.PublicKey` which implements a class `RSA`. You can use this class by the usual
`from Crypto.PublicKey import RSA`

The first thing to do with an `RSA` object is to generate a new one, using
`key=RSA.generate(n)`
where `n` is the number of bits in (part of) the key. The values allowed for this parameter must be at least 1024, and must be a multiple of 256.

As is often the case, the larger the key, the more secure is the cryptosystem but the slower are the calculations to generate the key and also to do encryption and decryption. Common values one sees in use today are 1024 and 2048, although also 4096 is not unusual.

Code Task 3.2.1

Write a **Python** function `timeRSAkeygen(n)` which takes as input a key size `n` and generates 100 RSA keys of that size. Time how long each key generation takes, using the approach to measuring execution time in Code Task 2.2.1, and append that new execution time to a list. `timeRSAkeygen(n)` should then return the average of that list of 100 times.
Then run that program `timeRSAkeygen(n)` for the key sizes which start at 1024 and go up to 4096 in steps of 256 – that is, for values of `n` ranging over the list `[x for x in range(1024,4097,256)]` – and print out for each key size the average key generation time produced by `timeRSAkeygen(n)` for that key size.

[1] In a sense, this is unfortunate: to get the same security with RSA as with elliptic curve cryptosystems, much larger keys must be used and so the computations are significantly slower. The advantage of RSA is probably that it is based on mathematics that is at the level of advanced undergraduates, while elliptic curve cryptosystems are based on mathematics at the graduate level.

Bonus Task 3.2.1

Make a plot of the timing information you just gathered. That is, for each key length, you computed the average amount of time it took per generation of a key of that size over 100 times. Now make a line plot where the x values are those key sizes and the corresponding y values are the average key generation times for that size key. Make sure your graph has a title and x- and y-axis labels.
If you are not familiar with making simple line plots in **Python**, just do a search for "`python line plot`" and there will be literally hundreds of pages with quick-start guides to making line plots with `matplotlib.pyplot`, many including code that you can copy and easily adapt.

Let's look at one of those keys generated with `RSA.generate`, using a method `exportKey`[2] of the `RSA` class:

```
>>> from Crypto.PublicKey import RSA
>>> key = RSA.generate(1024)
>>> key.exportKey()
b'-----BEGIN RSA PRIVATE KEY-----\nMIICXQIBAAKBgQDRpxdIHxJTp
uM3R/mHKurOrAXnJ5a/nlucxvst3lpHVcs9Yv+Q\nTTtOrDVvgs7BBq2pdjO
DOYxEmAX2GGDuJIFMKZ8Se2ZUGWeJZ0sJolUoANQ0ihIj\nD7zvFz40kuWLU
7imSMOlQpWk35lyN2BR4I6DvAo6f4JKzEJ1l2wuig+75QIDAQAB\nAoGAQsK
HamLijhq1fdQAhGdJMBidJJd5rHj7yTefomKMsuyB9IFCyiuduBakSWcI\n+
XRr9mt6Sc4YeXtDYrMuooajWQ6Ltg8sGqjpDtrYzPILiVgN8fXz1R2TKAePr
ZUO\nKazRnkMN6QcgX1xilfXFtr3Q2OB67dpnVwK3mtAI3E6tZ4ECQQDZWho
gDB3+xkkO\nES0mH0DtyeEbLSknAF84/xSylD58vVYvwWGma4dHEmZcjKEob
c8dQQrndBm8jVQS\niDblKjhhAkEA9u6EZfk+JqeuoUleirRI5X1Kwvy8jo7
7kEf3noVBE3Eve9eJgcX2\nO+ZrAWke3tKqqu6+Tdy9AKkgl1s5EdfiBQJBA
K3My7k2l0Gj4sT53SVvtmaumG83\nxIFoXbxg1Hcb7X+nkuRq+R+vOiQNxYZ
Z+YAvln8pBIQhpXbNeB29iE/lW+ECQFWB\nxqshIdp02k3TgD97qnp9ZnQa3
Jho/se5hA+KiTxYR18VBfLAQEIByjAU3LHANYU3\nYwLHW1NtPXHsDtkU7pk
CQQC4N+Q/IHEY/Sc+fvha5x8zdWmGwODheGJ6Q2i6GN+I\n9LvYjKW5hZ1Gi
rUBjonzNi1vrQsaaVZatRsLa1iwTRIa\n-----END RSA PRIVATE KEY-----'
```

Notice a couple of things about this:

- Apparently this is a **private key**: that is, it is the sort of thing, in an asymmetric cryptosystem, that must be kept secret.

[2] This method may be called `export_key` if you are using a different version of the `Crypto` module.

- Also, when one exports a key, it is a byte string. This means that if you want to write this to a file, you need to open the file with a command like
  `file_handle = open("filename", "wb")`
  The `"wb"` there means that the file is being opened for **w**riting in **b**yte mode.

If you can export a key, it stands to reason that you can also import one. In fact, this is quite easy: suppose that private key byte string were stored in a variable `exported_key`. Then you could make an identical RSA object by

```
>>> from Crypto.PublicKey import RSA
>>> new_key = RSA.importKey(exported_key)
```

and that `new_key` will have all of the properties and uses as the original `key`, above.

Code Task 3.2.2

Write a **Python** function `gen_save_RSA_private(n, name)` which takes as input a key size `n` and a string `name`. The function should generate an RSA key of size `n` and store it in a file with the given `name`. It should also return that instance. Then write a **Python** function `recall_RSA_private(name)` which takes as input a string `name`. It should open the file with that `name`, create an instance of the RSA class with the key value stored in that file, and it should return that instance. The ideas is that you should be able to use these functions in situations like:

```
>>> from Crypto.PublicKey import RSA
>>> key = gen_save_RSA_private(1024, "AliceRSA")
>>> # do lots of encryption and decryption with this key
>>> # ... then take a break and even quit Python and
>>> # reboot the computer
>>> # ... but eventually restart the computer and Python
>>> # and do this:
>>> from Crypto.PublicKey import RSA
>>> key = recall_RSA_private("AliceRSA")
>>> # and then lots more encryption and decryption with
>>> # the same key [identity] as before
```

As we mentioned near the end of §3.2, in an asymmetric cryptosystem, first one computes the private key and then, from the private key, one computes the corresponding public key. In fact, as was also mentioned, it is important for the security of the asymmetric cryptosystem that while computation

$$\boxed{\text{private key}} \quad \rightsquigarrow \quad \boxed{\text{public key}}$$

can be done in a reasonable amount of time – it doesn't have to be fast, though, since it will be done only once, when setting up the cryptosystem – the computation backwards

$$\boxed{\text{private key}} \quad \rightsquigarrow \quad \boxed{\text{public key}}$$

must be entirely impractical no matter how great are the computational resources available to the enemies that it is imagined will try to break this secure communication.

In the case of RSA, the easy direction of from private to public key is just *multiplication* and the hard, backwards direction is basically just *division*. As most of us remember from grade school, it is much easier and more direct to multiply two numbers than it is to divide two. For RSA, the two numbers one multiplies are **prime numbers**, meaning that they are numbers which have no positive, whole number factors other than one and themselves. Multiplication of primes is still just as easy (and fast) as for any other types of numbers, even if they are hundreds of digits long. However, if you are given a 600 digit number and told that it is the product of two 300 digit primes, then it would take an immense amount of time – a sizeable fraction of the age of the universe: *really an immense amount of time* – using the best algorithms and hardware we have today to find those primes.

It is, however, not a mathematical theorem (at this time) that there cannot be some great factorization algorithm out there, as yet undiscovered, which would make the step

$$\boxed{\text{private key}} \quad \rightsquigarrow \quad \boxed{\text{public key}}$$

relatively fast for RSA, the most widely used asymmetric cryptosystem on the planet right now. That may or may not make you uncomfortable....[3]

Reading Response 3.2.1

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you understand how RSA's security could rely upon how it is easy to multiply but hard to divide [factor]?

Regardless of these general considerations around the security of RSA, since it is a working asymmetric cryptosystem, there must be a way to get just that public key from the private key in **Python**. Here's how (continuing from an example given above):

[3] And it may or may not make you even more uncomfortable to learn that **quantum computers** – computers which can do a kind of general-purpose computation in a different model of what that means, based on quantum mechanics – *do* have a fast factorization algorithm. RSA may well be broken the minute the first real, full-sized quantum computer is turned on!

```
>>> from Crypto.PublicKey import RSA
>>> key = RSA.generate(1024)
>>> key.exportKey()
b'-----BEGIN RSA PRIVATE KEY-----\nMIICXQIBAAKBgQDRpxdIHxJTpuM3
R/mHKurOrAXnJ5a/nlucxvst3lpHVcs9Yv+Q\nTTtOrDVvgs7BBq2pdjODOYxEm
AX2GGDuJIFMKZ8Se2ZUGWeJZ0sJolUoANQ0ihIj\nD7zvFz40kuWLU7imSMOlQp
Wk35lyN2BR4I6DvAo6f4JKzEJ1l2wuig+75QIDAQAB\nAoGAQsKHamLijhq1fdQ
AhGdJMBidJJd5rHj7yTefomKMsuyB9IFCyiuduBakSWcI\n+XRr9mt6Sc4YeXtD
YrMuooajWQ6Ltg8sGqjpDtrYzPILiVgN8fXz1R2TKAePrZUO\nKazRnkMN6QcgX
1xilfXFtr3Q2OB67dpnVwK3mtAI3E6tZ4ECQQDZWhogDB3+xkkO\nES0mH0Dtye
EbLSknAF84/xSylD58vVYvwWGma4dHEmZcjKEobc8dQQrndBm8jVQS\niDblKjh
hAkEA9u6EZfk+JqeuoUleirRI5X1Kwvy8jo77kEf3noVBE3Eve9eJgcX2\nO+Zr
AWke3tKqqu6+Tdy9AKkgl1s5EdfiBQJBAK3My7k2l0Gj4sT53SVvtmaumG83\nx
IFoXbxg1Hcb7X+nkuRq+R+vOiQNxYZZ+YAvln8pBIQhpXbNeB29iE/lW+ECQFWB
\nxqshIdp02k3TgD97qnp9ZnQa3Jho/se5hA+KiTxYR18VBfLAQEIByjAU3LHAN
YU3\nYwLHW1NtPXHsDtkU7pkCQQC4N+Q/IHEY/Sc+fvha5x8zdWmGwODheGJ6Q2
i6GN+I\n9LvYjKW5hZ1GirUBjonzNi1vrQsaaVZatRsLa1iwTRIa\n-----END
RSA PRIVATE KEY-----'
>>> public_key = key.publicKey()
>>> public_key.exportKey()
b'-----BEGIN PUBLIC KEY-----\nMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBi
QKBgQDRpxdIHxJTpuM3R/mHKurOrAXn\nJ5a/nlucxvst3lpHVcs9Yv+QTTtOrD
Vvgs7BBq2pdjODOYxEmAX2GGDuJIFMKZ8S\ne2ZUGWeJZ0sJolUoANQ0ihIjD7z
vFz40kuWLU7imSMOlQpWk35lyN2BR4I6DvAo6\nf4JKzEJ1l2wuig+75QIDAQAB
\n-----END PUBLIC KEY-----'
```

If you use this public key byte string in an `RSA.importKey` then it will work, but it will build an instance of the RSA class which only has the public key and so can do encryption but not decryption. This is the kind of RSA key object which most users will be building to send encrypted messages to the owner of the private key, while the full, original RSA key object with both private and public keys should only be known to and used by the person receiving and decrypting those messages.

Code Task 3.2.3

Do a version of Code Task 3.2.2 which also saves the public key in a separate file. That is, make a **Python** function `gen_save_RSA_pub_priv(n, name)` which takes as input a key size `n` and a string `name`. The function should generate an RSA key of size `n` and store it in a file with name `name+".private"`. It should also store just the corresponding public key in a file with name

`name+".public"`. Finally, it should return the full instance (not just the public version).
Then write a **Python** function `recall_RSA_public(name)` which takes as input a string `name`. It should open the file with name `name+".public"`, create an instance of the RSA class with the key value stored in that file, and it should return that instance.
The idea is that you should be able to use these functions in situations like:

```
>>> # Bob does:
>>> from Crypto.PublicKey import RSA
>>> key = gen_save_RSA_pub_priv(1024, "BobRSA")
>>> # then puts the file "BobRSA.public" on an open
>>> # website, where Alice could download it
>>> # ... then, with this file on her machine,
>>> # Alice can do:
>>> from Crypto.PublicKey import RSA
>>> public_key = recall_RSA_public("BobRSA.public")
>>> # and Alice can encrypt using this new
>>> # public key and send messages which only Bob
>>> # will be able to decrypt using his original
>>> # key which has both public and private keys
```

We still haven't used these RSA key objects to encrypt and decrypt messages. Remember, it is an object like the `public_key` one in the example above which is all that is needed to do encryption (we use the same key as above, rather than generating a new one, just for consistency), which we could create with

```
>>> from Crypto.PublicKey import RSA
>>> public_key = RSA.importKey(b'-----BEGIN PUBLIC KEY-----\nMIGf
MA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDRpxdIHxJTpuM3R/mHKurOrAXn\nJ5a
/nlucxvst3lpHVcs9Yv+QTTtOrDVvgs7BBq2pdjODOYxEmAX2GGDuJIFMKZ8S\ne2
ZUGWeJZ0sJolUoANQ0ihIjD7zvFz40kuWLU7imSMOlQpWk35lyN2BR4I6DvAo6\nf
4JKzEJ1l2wuig+75QIDAQAB\n-----END PUBLIC KEY-----')
```

We can then use the `encrypt` method of this `public_key` object to do encryption. But note first that `encrypt` has a second argument, just for backwards compatibility with earlier versions of this module, which must be some long integer that is actually completely ignored. Notice also that the return value of `encrypt` is a tuple, the first element

of which is the byte string of the ciphertext, so:

```
>>> ciphertext = public_key.encrypt(b'This is a test',1)[0]
>>> print(ciphertext)
b'\xbfa\xb0\x8b\x08*\xdf\x96gK0B/\xa2\x8e\x1e\x86\xf2]8\xd9\xc
b\xc2w\xad\x82}\xb5\xf4\x97\xe1g\x91\xc8\x00\x14fr\x15\xc1\x8b
\xbd\x8di\x1f\xbc\tt\x8f\xfa\x1c\xad\xb8V\xca\xc7\xed\xb3X"\xb
7ra\xf8h\x01\x94c\xc9\xaf\xf6\xd5\x0f/\x83[\xea\xfd\x85\xdep9\
xf2\xee\xfc\xec\xd7\xfc\xfd\xe0\xf1\xcc\x12\x8fZ\xad\xf0|n@\xe
8vD\xb48}a\xd8^\xdb#\xbb\x01\x8e\x8c\x1a;[\x8e\xfd@\xc1\xb8Ohw
\xc4\x15'
```

As we mentioned above, this `public_key` object cannot do decryption. In fact, if you try it, you will get some error like

```
>>> public_key.decrypt(ciphertext)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3/dist-packages/Crypto/PublicKey/RSA.py
", line 174, in decrypt
    return pubkey.pubkey.decrypt(self, ciphertext)
  File "/usr/lib/python3/dist-packages/Crypto/PublicKey/pubkey
.py", line 93, in decrypt
    plaintext=self._decrypt(ciphertext)
  File "/usr/lib/python3/dist-packages/Crypto/PublicKey/RSA.py
", line 239, in _decrypt
    mp = self.key._decrypt(cp)
TypeError: Private key not available in this object
```

On the other hand, if we still had the `key` object around, which contained the private key – or if we saved off the exported version of that key into a file and then rebuilt `key` with `RSA.importKey` on the byte string in that file – then we could do decryption as simply as

```
>>> key.decrypt(ciphertext)
b'This is a test'
```

Code Task 3.2.4

Do a version of Code Task 2.2.1 for the RSA.

That is, make a new RSA `key` object of key size 1024. Pick a random cleartext of length around 100 bytes. Then encrypt it 100 times with `key.encrypt` and report on the average time it takes to do the encryption (using the timing method described in Code Task 2.2.1).
Likewise take the ciphertext coming from any of those encryptions with `key` and do 100 decryptions of that ciphertext, reporting on the average time it takes to do the decryption.

Code Task 3.2.5

Do a version of Code Task 2.2.2 for RSA.
That is, make a new RSA `key` object of key size 1024. Pick a cleartext you like of length around 80 or 100 bytes. Make two versions of the cleartext which differ by only one bit.
Encrypt both versions of the cleartext with `key.encrypt` and use your program `bits_diff_display` from Code Task 2.1.2 to see where and how much the two ciphertexts differ.
You might also try the above for several different keys, just to make sure that it's not a fluke which is special for the choice of key you happened to make.

Reading Response 3.2.2

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you feel comfortable using **Python**'s implementation of the RSA cryptosystem?

3.2.2. More secure RSA in Python using OAEP and PKCS #1. Unfortunately, the direct use of RSA, as we just did, is not considered secure. This is for several reasons, some to do with the special structure of RSA, some simply to do with the fact that the bare RSA we used above has no salt and so always encrypts the same cleartext to the same ciphertext, when using the same key – for (some) details, see the section Attacks against plain RSA in the Wikipedia article RSA (cryptosystem).

The cryptographic community has developed a theoretical structure which protects against these attacks on plain RSA called **Optimal Asymmetric Encryption Padding [OAEP]**. OAEP was incorporated into a standard called **PKCS #1, Public Key Cryptography Standard #1**, issued by RSA Laboratories, an American IT security firm founded

by the very same Rivest, Shamir, and Adleman who created the RSA cryptosystem. It is PKCS #1 that is considered safe to use in practice[4].

PKCS #1 is implemented in **Python** with a class `PKCS1_OAEP` in the `Crypto.Cipher` module. Since it is essentially just a padding for an asymmetric cryptosystem – the "P" in "OAEP" – we make a PKCS #1 cipher object by feeding it an RSA key object:

```
>>> # Bob does:
>>> from Crypto.PublicKey import RSA
>>> key = gen_save_RSA_pub_priv(1024, "BobRSA")
>>> # then puts the file "BobRSA.public" on an open website
>>> # ... then, Alice gets this file on her machine, and does:
>>> from Crypto.PublicKey import RSA
>>> public_key = recall_RSA_public("BobRSA.public")
>>> from Crypto.Cipher import PKCS1_OAEP
>>> cipher_for_encryption = PKCS1_OAEP.new(public_key)
>>> # and Alice can encrypt messages, e.g.,
>>> cleartext = b'This is a test. This is only a test.'
>>> ciphertext = cipher_for_encryption.encrypt(cleartext)
>>> print(ciphertext)
b'\x17\xf4\xaa\xc8\x84\xa9\xe5\x7f8\xa7\x86‘\xea\x13\x1c\x1d>d\
xd5\xaeI\xed\xf8~$Y\xa3\x7f\xb7h\x14W\x1a\x99\\\xdd]P\x08\x0f*\
xd4z\xc2u-\xd6\xbb\xf3\xe2;\xf6\xdaZ\xc9!0-\xce\xa1S\xb4\xcc%,\
x08\x0c#\x00v\x87Kn41\x84\xc8\x18\xdf\xc9\xd8\x83?\xb5\xce\xbf\
x9b<\xadr.6xa\xa8\xad\xca\x0c\x1b\xf4\xc3\x8c\xe0\x05\\U\xc3\x0
b\xdf\x02\xfdk*l\xa3*\xa5>\x7f\xcd\xfa\xb3\xdb\x91\xfd\xd5\x15\
x8b'
>>> # Bob, on his machine, where he the key object which also
>>> # contains the private key, can do decryption by
>>> from Crypto.Cipher import PKCS1_OAEP
>>> cipher_for_decryption = PKCS1_OAEP.new(key)
>>> cleared_text = cipher_for_decryption.decrypt(ciphertext)
>>> print(cleared_text)
b'This is a test. This is only a test.'
```

Notice a couple of things about this:

- If you, dear reader, type exactly the same commands into your computer, you will still get back the original cleartext when you do the decryption, but the intermediate step of the ciphertext you will get will be different from the one shown above.

[4]at least with large enough key sizes, and until someone turns on a fully functional quantum computer

The reason for this is that OAEP includes some cryptographic salt, so that every time you do encryption, even of the same cleartext, you get a different ciphertext. (This is **a good thing**, it increases security.)

- You cannot encrypt cleartexts which are very large with the approach described above. So in a practical situation, one must either break up a long message into small chunks and encrypt each of them, or else use some other approach which avoids the size limit. The second of these solutions will be described in the very next section, below.

As you can probably guess, it makes sense now to see if this new, more secure version of RSA is slow or fast, and if it satisfies Shannon's diffusion condition.

Code Task 3.2.6

Do a version of Code Task 3.2.4 for the RSA in the PKCS #1/OAEP version.
That is, make a new RSA `key` object of key size 1024. Use it to make a `PKCS1_OAEP` cipher object. Pick a random cleartext of length around 100 bytes. Then encrypt it 100 times with `cipher.encrypt` and report on the average time it takes to do the encryption (using the timing method described in Code Task 2.2.1). Likewise take the ciphertext coming from any of those encryptions and do 100 decryptions of that ciphertext, reporting on the average time it takes to do the decryption.

Code Task 3.2.7

Do a version of Code Task 2.2.2 for the RSA in the PKCS #1/OAEP version.
That is, make a new RSA `key` object of key size 1024. Use it to make a `PKCS1_OAEP` cipher object. Pick a cleartext you like of length around 80 or 100 bytes. Make two versions of the cleartext which differ by only one bit.
Encrypt both versions of the cleartext with `cipher.encrypt` and use your program `bits_diff_display` from Code Task 2.1.2 to see where and how much the two ciphertexts differ.
You might also try the above for several different keys, just to make sure that it's not a fluke which is special for the choice of key you happened to make.
You might also try just comparing two encryptions made in exactly the same way, of exactly the same cleartext, with your program `bits_diff_display` to see that the cryptographic salt which OAEP uses makes for quite different ciphertexts every time you do encryption, even starting from the same cleartext.

3.2.3. How to use RSA in a way that is both fast and secure. It should have been clear from Code Tasks above that RSA, while it has the nice features of an asymmetric cryptosystem with regards to key distribution and management, is very slow. Additionally, it has the problem that one cannot encrypt large messages – although, presumably, one could solve this problem by one of the block chaining methods discussed above. But there is another approach which takes advantage of the good features of asymmetric cryptosystems while avoiding the weaknesses.

For this, one uses the whole approach described in this chapter of asymmetric encryption, including generating a public and private key, posting the public key on some public website for anyone to use, *etc.* But rather than doing the entire communication in RSA (or some other asymmetric cryptosystem, one uses that slow but secure asymmetric cryptosystem to transmit a randomly chosen "session key". Then the rest of the communication is done by some fast, well-chained, block cipher (such as AES).

A "session" in this context might consist of some agreed-upon number of messages, or all of the messages necessary to complete some protocol (like making a credit card payment on the 'net), or all of the message which are send before some time limit expires. Whatever triggers the end of a session, when that happens, a new random session key must be generated for further secure communications between the same parties.

This approach is extremely common on the Internet today. In fact, most of the time customers watch a movie over the Internet, this is the way the video stream is sent to them by the provider. Providers want to send the video data in encrypted form, so that their valuable movies cannot be stolen by someone who is merely observing the packets as they go by on the Internet. Since customers probably have never met the providers in person to agree upon a key for a symmetric cryptosystem, it would seem that the providers would have to use an asymmetric cryptosystem for this encrypted video stream – but asymmetric cryptosystems are too slow to run video through without the watching human noticing! On the other hand, block ciphers like AES *are* fast enough. Since the session key negotiation at the beginning of the stream happens in a fraction of a second, the approach in this section allows for security, without difficulties of key distribution, but nevertheless using a fast block cipher (for the entire video stream, after the initial session key was sent using an asymmetric cryptosystem).

Below is a graphical depiction of this approach, assuming that Aragorn wants to initiate fast and secure communication with Bilbo, even though they have not met in person to share any key. Note that once the initial session key negotiation is finished, there can be any number of messages coming from either side until the session ends – the example in this diagram where the messages strictly alternate is only one possibility. In fact, often Aragorn sends her first two messages to Bilbo and ②, below) as one, larger, concatenated message $c_{k_{AB}} + c_1$ (using the **Python** notation "+" for concatenation of byte strings), since she does not need to wait for any response from Bilbo after ① before doing ②.

Fast and secure, session key-based communication, graphically:

Aragorn	on public network (where Gollum is watching)	Bilbo
	①	generate key pair (k_e^B, k_d^B) for asymmetric cryptosystem $\mathcal{A}$;
download k_e^B	$\longleftarrow$ public encryption key k_e^B $\longleftarrow$	publish k_e^B
choose random session key k_{AB} for symmetric cryptosystem $\mathcal{S}$; encrypt k_{AB} using $\mathcal{A}$ with public key k_e^B, getting c_{AB}; transmit c_{AB}	② $\longmapsto$ $\mathcal{A}$-encrypted session key c_{AB} $\longmapsto$	receive c_{AB}; decrypt c_{AB} using $\mathcal{A}$ with private key k_d^B, getting k_{AB}
create cleartext message m_1; encrypt m_1 using $\mathcal{S}$ with session key k_{AB}, getting c_1; transmit c_1	③ $\longmapsto$ $\mathcal{S}$-encrypted ciphertext c_1 $\longmapsto$	receive c_1; decrypt c_1 using $\mathcal{S}$ with session key k_{AB}, getting m_1
	④	create cleartext message m_2; encrypt m_2 using $\mathcal{S}$ with session key k_{AB}, getting c_2;
receive c_2; decrypt c_2 using $\mathcal{S}$ with session key k_{AB}, getting m_2	$\longleftarrow$ $\mathcal{S}$-encrypted ciphertext c_2 $\longleftarrow$	transmit c_2
create cleartext message m_3; encrypt m_3 using $\mathcal{S}$ with session key k_{AB}, getting c_3; transmit c_3	⑤ $\longmapsto$ $\mathcal{S}$-encrypted ciphertext c_3 $\longmapsto$	receive c_3; decrypt c_3 using $\mathcal{S}$ with session key k_{AB}, getting m_3
etc....		

Reading Response 3.2.3

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you understand how Alice and Bob would use this session-based approach?

It is important that the session keys used in the above scheme come from a good source of randomness. Fortunately, the **Python** module `Crypto` has a submodule `Crypto.Random` that implements a method `get_random_bytes(N)` which generates a good random byte string of length `N`. This method should be used when creating session keys.

Bonus Task 3.2.2

Implement the above, session-based approach to fast and secure communications. Since we're not interested (at the moment) in the technicalities of publishing material (such as public keys) on the web or of sending messages over the Internet, you will have to play the roles of both Aragorn and Bilbo. Instead of posting or transmitting files and data, you will just put these things into files and the next program can read in that file, which, we will imagine, came off the Internet.

Hoping to keep matters clear, we'll insist that every **Python** function which Aragorn should run will have a name that starts with `A`, while those to be run by Bilbo will have names starting with `B`. To enforce this idea of separation, the methods beginning with `A` should only use information that they create themselves or get out of files on the disk (which is playing the role of the public Internet in this scenario) with names ending in `.public`, and not internal information from the methods beginning with `B`; and vice-versa for the information used by the `B` methods.

Here are the steps you should write code for, labeled to correspond to the numbers steps in the diagram above, and with the initial of the actor who is to be imagined to be doing the work in that protocol diagram (*i.e.*, basically the initial tells which column of the diagram is being indicated):

B ①: Write a **Python** method `B_asym_keygensend(n)` which takes as input a key size `n`. The function should

(1) generate an RSA key of size `n`

(2) store the full version (public and private) in a file `B_asym_key.private`, and

(3) store just the public key part in a file `B_asym_key.public`.

Remember, saving data in a file whose name ends in `.public` is "sending" the file over the public network, in the metaphor of this problem. [Note, you can accomplish this in very few lines by just calling the function

`gen_save_RSA_pub_priv(n, name)` from Code Task 3.2.3 with the correct arguments!]

A ②: Write a **Python** method `A_session_keygensend()` which generates and "sends" (puts in a file on disk with a name ending in `.public`) a session key for the AES cryptosystem. This new method should

(1) use `get_random_bytes` to generate a high-quality session key k_{AB} of the right size for AES,
(2) encrypt k_{AB} with `PKCS1_OAEP` (you can reuse some lines of code from the Code Tasks 3.2.6 or 3.2.7 for this), and
(3) store the resulting ciphertext in a file `encrypted_session_key.public`
(4) store the unencrypted session key k_{AB} in a file `A_session_key` for later use by the `A` methods.

Remember, as discussed immediately before Code Task 3.2.2, above, that when you want to write encrypted data to a file, you should open the file the with the command `file_handle = open("filename", "wb")` and then write the date, as usual, with `file_handle.write(data_bytestring)`).

B ②: Write a **Python** method `B_session_keyreceive()` which reads and decrypts the session key which Aragorn just created and "sent." For this, it should

(1) open the file `encrypted_session_key.public` for reading bytes, and get its contents,
(2) read the key in the file `B_asym_key.private` (use your method `recall_RSA_private` from Code Task 3.2.2 to do this!),
(3) use that key object to create a `PKCS1_OAEP` cipher object,
(4) decrypt the session key previously read in, and
(5) save for later use (the bytes of) the decrypted session key in a file `B_session_key` .

A ③: Write a **Python** method `A_send_encrypted_message(m, n)` which takes as input a byte string `m` (the message), and an int `n` (the message number). This method should:

(1) open the file `A_session_key` for reading bytes, and get its contents,
(2) using that key, create a `Crypto.Cipher` object for AES in CBC mode (as we did in §2.2),
(3) encrypt the message `m` using that cipher object,
(4) open a file whose name is `''c''+str(n)` for writing bytes, and

(5) write (the bytes of) the encrypted message to that file.

B ③: Write a **Python** method `B_receive_encrypted_message(n)` which takes as input an int `n` (the message number) and returns the decrypted message. This method should:

(1) open the file `B_session_key` for reading bytes, and get its contents,
(2) using that key, create a `Crypto.Cipher` object for AES in CBC mode (as we did in §2.2),
(3) open a file whose name is `''c''+str(n)` for writing bytes,
(4) read (the bytes of) the encrypted message from that file,
(5) decrypt the message `m` using the previously built cipher object, and
(6) return the cleartext.

B ④: Write a **Python** method `B_send_encrypted_message(m, n)` which takes as input a byte string `m` (the message), and an int `n` (the message number). This method should do exactly the same thing as the `A_send_encrypted_message(m, n)` described above, only switching `A` ↔ `B` everywhere.

A ④: Write a **Python** method `A_receive_encrypted_message(n)` which takes as input an int `n` (the message number) and returns the decrypted message. This method should do exactly the same thing as the `B_receive_encrypted_message(n)` described above, only switching `A` ↔ `B` everywhere.

[Yes, this seems like a long Task, but it extensively reuses or builds on other code you've already written, and has very detailed instructions!]

3.3. Digital Signatures

The discovery of asymmetric cryptosystems – or, as they are more often called in non-technical contexts like this paragraph, public-key cryptosystems – has changed the face of the Internet by allowing secure communications between entities who have never met in real space to share a secret key. Without this innovation, commerce over the Internet would be infinitely more difficult, for example. But beyond this straightforward application of public-key crypto to securing communication against eavesdroppers, other powerful applications have also been discovered. A very nice one, which has many uses in the networked world, is the idea of

DEFINITION 3.3.1. A **digital signature** is a chunk $\mathbb{S}$ of data which accompanies a message m. There is also an algorithm called **verifying the signature** which can be run with inputs $\mathbb{S}$, m, and some other public data which, if it returns ACCEPT, proves that the signature could only have been produced by someone who possesses a certain piece of secret data.

If we make some assumptions about behavior in the real world – which might or might not be justified! – then these digital signatures would have many extremely valuable applications. The assumptions we would need include:

- There is some reliable association between the public information used to verify a signature and the real-world person who claims they created the signature. For example, perhaps in order to have the public information listed on a well-known, trusted site next to a human name, that human must go to some government office, show an ID, and register their public data.
- The humans can be trusted to keep the secret data needed to generate those signatures reliably secure. For example, the don't keep the data on a machine which can ever be hacked, and they don't forget to erase it before they throw out an old hard drive, *etc.*

Now for some details.

3.3.1. Naive digital signatures.

It has been important since the beginning of our study of cryptology that encryption and decryption are *opposites:* what we would call "inverse functions" in mathematics[5], meaning that doing first one and then the other simply gets you back to where you started. Since inverse functions work in either order, that means that not only is the decryption of the encryption of some message just the original message, but so also is the encryption of the decryption of a message.

[5] *E.g.*, squaring and taking the square root are inverses (at least if you start with positive numbers), so that squaring a square root or square rooting a square gives back the original number.

That may seem like a mere computational curiosity – why would anyone decrypt a message which had not first been encrypted? – but it actually has a very nice feature: only the person who has the private key for some asymmetric cryptosystem can do the decryption, while anyone who has downloaded the corresponding public key can do the encryption. This situation where only the one person who has some secret information (the private key) can do something while everyone else can do some other thing using a piece of public information (the public key) sounds very much like what it supposed to happen in a digital signature scheme.

To be specific, suppose Bilbo wants to be able to digitally sign some message m, and Aragorn wants to be able to verify the signature. First, Bilbo should generate a public/private key pair and put his public key on his website. Then he should use the decryption algorithm, with his private key, on the message m, yielding a chunk of data we'll call s. Finally, Bilbo should send both the message m and the signature s to Aragorn.

Aragorn will have downloaded Bilbo's public key. What he can do is encrypt Bilbo's (purported) signature s, using that public key, to get a chunk of data m'. Then Aragorn will **accept** the signature as valid if his m' is the same as Bilbo's message m. Since only the person who has Bilbo's private key should be able to make a chunk of data s whose encryption is m, this method does implement the digital signature scheme as promised.

Graphically:

Naive digital signatures:

Aragorn	on public network	Bilbo
		generate key pair (k_e^B, k_d^B) for asymmetric cryptosystem $\mathcal{A}$;
download k_e^B	$\longleftarrow$ public encryption key k_e^B $\longleftarrow$	publish k_e^B
		with message m: decrypt m using $\mathcal{A}$ with private key k_d^B, getting s;
receive (m, s)	$\longleftarrow$ signed message (m, s) $\longleftarrow$	transmit (m, s)
encrypt s using $\mathcal{A}$ with public key k_e^B, getting m'; if $m' = m$ ACCEPT else REJECT		

Reading Response 3.3.1

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you feel comfortable with the way these naive digital signatures are generated and verified?

Code Task 3.3.1

Write a **Python** function `sign(key, m)` which takes as input a full (public and private) `key` for RSA, in the form of a byte string that might have been produced by `exportKey()` and a (byte string) message `m` and produces the signature `s` on `m` as described above. To do this, use `RSA.importKey` on the `key` variable to build an RSA object. Use the `decrypt` method of this RSA object on the variable `m` to produce the signature `s`, which your method should return.

Next, write a **Python** function `verify(pk, m, s)` which takes as input a public key `pk` for RSA, in the form of a byte string that might have been produced by `exportKey()`, a (byte string) message `m`, and a (byte string) signature `s` on that message, and returns a Boolean which indicates if the signature is valid or not, following the scheme above. To do this, use `RSA.importKey` on the `pk` variable to build an RSA object `public_key` and use the `encrypt` method of `public_key` to create the byte string `mprime`. Don't forget that `encrypt` takes a second, dummy variable which may as well have the value `1` and returns a tuple, the 0th element of which is the encryption. In other words, you need to use `mprime = public_key(s, 1)[0]`. Then simply return the (Boolean) value of `mprime == m`.

The idea is that you should be able to use these functions in situations like

```
>>> # Bilbo does:
>>> from Crypto.PublicKey import RSA
>>> key = RSA.generate(1024)
>>> full_key_exported = key.exportKey()
>>> public_key_exported = key.publicKey().exportKey()
>>> # Bilbo puts the public_key_exported on his website
>>> m = b'some message Bilbo wants to sign'
>>> s = sign(full_key_exported, m)
>>> # Bilblo sends (m,s) to Aragorn, maybe by email
>>> # Aragorn downloads the public_key_exported and
>>> # gets an email containing (m,s) and does:
>>> verify(public_key_exported, m, s)
>>> # if it is True, he can trust the message
```

3.3.2. Better digital signatures using hash functions. One big problem with the naive digital signatures described above is that they are (potentially) ... big. That is, the signature s on a message m is (essentially) the same size as the message itself – after all, it's a decryption, pretending the message is a ciphertext. So signing a large document, or maybe a large database or high-resolution image would basically require an unacceptable twice the space to store or transmit!

What we need to make much smaller signatures is some sort of algorithm which essentially *summarizes* the contents of the message m you want to sign. Then a very similar signature scheme to the one described above can do its verification algorithm by comparing the encrypted signature to the summary of the message, rather than to the entire message.

There's a name for this magical summarizing algorithm:

DEFINITION 3.3.2. A **cryptographic hash function** is an algorithm H which takes as inputs bit strings x of arbitrary length and produces outputs $H(x)$ of a fixed size, satisfying the properties:

(1) "**ease of computation**" computation of $H(x)$ is very fast;
(2) "**pre-image resistance**" starting with any bit string y of the size of the outputs of H, it is very difficult to find a bit string x with the property that $H(x) = y$;
(3) "**second pre-image resistance**" starting with any specific bit string x_1, it is very difficult to find another x_2 such that $H(x_2) = H(x_1)$;
(4) "**collision resistance**" it is very difficult to find any two bit strings x_1 and x_2 such that $H(x_2) = H(x_1)$.

In some sense, the key phrase in the above definition is "very difficult," in the sense that there are certainly very many possible inputs to a hash function which give the same output – in fact, there are an infinite number of such colliding inputs! Think about it: if the output size of a hash function is, say, 256 bits, then what happens if you look at input bit strings of length 257? There are twice as many input of that size as there are outputs, so the most efficient possible way to spread out the way the outputs come from inputs would have every output coming from two inputs! And the same argument says there are at least four inputs of length 258 bits giving the same output, *etc.* But to be a cryptographic hash function, it must be *very hard* to find such collisions.

In practice, what this means is that the algorithms used for hash functions do a lot of random fiddling around with the bits of their inputs, so there is not enough structure in the algorithm which might give an attacker a systematic way to try to find collisions or pre-images or second pre-images. Without structure, the only option left would seem to be a brute-force attack, but since there are roughly as many bit strings of length 256 as there are particles in the universe (leaving out photons and neutrinos, though), brute-force is not a viable attack.

Cryptographic hash functions are useful because if you have a message or document – think of it as a bit string m – the $H(m)$ acts a kind of fingerprint for m. For example: suppose Bilbo tells Aragorn what $H(m)$ is without revealing m on Monday, and only on Friday reveals m. Aragorn can be confident that it was the same m Bilbo was thinking of on Monday as he revealed on Friday, because it would have been "very difficult" to find another m' for which $H(m') = H(m)$.

Reading Response 3.3.2

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you feel like you understand what is useful about a hash function, and why there are theoretically infinitely many failures of the conditions pre-image resistance, second pre-image resistance, and collision resistance but nevertheless the fact that it is "very difficult" to find those failures makes hash functions practical?

There are a number of hash functions in use today, and there are some which were used for a while but which now are no longer considered secure. As mentioned above, the security of a hash function in part rests upon its lack of structure ... but any algorithm which is clear enough to be written down and accepted as a standard must have some structure, and eventually computer scientists have found the way to attack some of the older hash functions using only that little bit of structure. Here are some hash functions which were or are widely used:

- For around a decade starting in the early 1990s, the most widely used cryptographic hash function was called **md5**. This algorithm was developed by Ron Rivest and published in 1992. The output size of md5 is 128 bits.

  While md5 was thought to be flawed since the middle 1990s, a real attack was not published until 2004, when it was shown not to be collision resistant [**WY05**]. However, md5 is still used extensively today to verify that a large data transfer has not suffered a transmission error – *i.e.,* it is still a useful tool to test for non-malicious data corruption.
- The most widely used cryptographic hash function from the late 1990s until recently, and one which is built into many widely accepted and standardized cryptographic protocols, is **SHA-1**, with an output size of 160 bits.

  SHA-1 was developed by US National Security Agency in a semi-public process, and was adopted by the National Institute of Standards and Technology [NIST] as part of several US Federal Information Processing Standards.

  In 2004, some work was published which indicated that SHA-1 might be vulnerable to certain kinds of attack. (See [**PS04**].) For this reason, NIST required in 2010 many US federal data protection applications to move to another hash function.

- At the time of this writing, most security-conscious users and organizations recommend **SHA-2**, usually in its **SHA-256** variant, which has an output size of 256 bits. Given recent revelations of the NSA's involvement – and weakening of – cryptographic protocols, it might be a cause of concern that NSA participated in the development of SHA-2.

SHA-2 is implemented in **Python** with a class `SHA256` in the `Crypto.Hash` module. You can create this object with `SHA256.new(s)`, where `s` is the entire byte string to be hashed. Another way use this class, particularly useful if there is a lot of data to be hashed, is to pass `SHA356.new` the first piece of the data, and the to call the class's `update` method with the remaining pieces of data.

In either approach, when you are done feeding the `SHA256` instance your data to be hashed, you can call the `digest()` method to get the hash value as a byte string, or the `hexdigest()` method to get that same hash value as a character string written out in hexadecimal. For example:

```
>>> from Crypto.Hash import SHA256
>>> hash_object = SHA256.new(b'first bit of data')
>>> hash_object.update(b'some more data')
>>> hash_object.update(b'last bit of the data')
>>> hash_object.digest()
b':\x93"P`\xd2Y-\x8c/\xab\xfa=\x8e\xa7\xb7N\xdcF\xae[\xf9\xa4\xd
6.\xa1\xfeR\xe4\xcb\x89\xb8'
>>> hash_object.hexdigest()
3a93225060d2592d8c2fabfa3d8ea7b74edc46ae5bf9a4d62ea1fe52e4cb89b8
```

Note that you can get the `digest` from a `SHA256` instance and but still add more data by calling the `update` method and get a later `digest` which will be the hash function output for all of the input you've given that instance since it was created with `new`. What this means is that you must create a new instance with `new` every time you want to start fresh with computing the hash function of a new bit string.

Code Task 3.3.2

Since SHA256 is believed to have the properties that make it a good cryptographic hash function, it must be that small changes in the input make for drastic changes in the output, 256-bit hash value. In fact, the best thing would be for differences of just one bit in the input – by changing a bit, or adding one more bit to the sequence (somewhere: adding it at the beginning, end, or somewhere in the middle) to make approximately half of the bits in the hash value change. Let's test this!

First, pick some base input byte string that you like of about 50 bytes. Compute its SHA256 hash value. Now try changing one bit of this base input, computing the hash value of the new input string, and comparing the two hash values with your `bits_diff_display` from Code Task 2.1.2 to see where and how much the two hashes differ.
Repeat this several times (at least 10 times), changing bits at different locations in the input string, and see each time how much it changes the bits of the hash string. Also compute the average number of bits which were changed in the hash value in your experiments.

As mentioned above, the fact that the cryptographic hash of a message is a reliable fingerprint for the message allows us to make a much more efficient signature scheme. In this scheme, rather than sending the message and its decryption, we send the message and the decryption of the hash of the message. Therefore the signature is a data chunk which is as big as the output size of the hash function we're using, no matter how big is the message to be signed.

Graphically:

Better [smaller] digital signatures with hashing:

Aragorn	on public network	Bilbo
		generate key pair (k_e^B, k_d^B) for asymmetric cryptosystem $\mathcal{A}$;
download k_e^B	$\longleftarrow$ public encryption key k_e^B $\longleftarrow$	publish k_e^B
		with message m: hash m to get d; decrypt d using $\mathcal{A}$ with private key k_d^B, getting s;
receive (m, s)	$\longleftarrow$ signed message (m, s) $\longleftarrow$	transmit (m, s)
encrypt s using $\mathcal{A}$ with public key k_e^B, getting d'; hash m to get d; if $d' = d$ ACCEPT else REJECT		

Reading Response 3.3.3

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you feel comfortable with how this new approach makes a secure digital signature scheme, because of its structure and the properties of a cryptographic hash function, which is nevertheless much smaller than the naive digital signatures?

Code Task 3.3.3

Implement this new approach to making digital signatures that uses a hash function for smaller signatures.

That is, make new **Python** functions `signH(key, m)` and `verifyH(pk, m, s)` which have exactly the same inputs and outputs as the `sign(key, m)` and `verify(pk, m, s)` from Code Task 3.3.1 and can be used in the same ways as those original functions, but which use `SHA256` to make signatures which are always quite small.

3.4. Key management and the need for a robust PKI

We started this chapter by looking for a new approach which might solve a problem we had with symmetric cryptosystems: while they are fast and secure, there are many, many situations in which parties who have never met each other in real life, in a place and situation where they can share a high quality (meaning: large and hard for others to guess) secret key, nevertheless want to communicate securely over the Internet. Asymmetric cryptosystemss solved this problem by breaking a key into two pieces, one of which must be kept secret (the private key) and one of which must be made public (the public key.

A significant issue with remains key management, though: the problem with Aragorn and Bilbo never meeting in person to set up a shared secret key is that, basically, they each have to trust each other's websites where their public keys must be posted. If the website is hacked, or even if the communications channel between one of the participants and the other's website is compromised, then they may end up using not the public key of their intended message recipient, but rather a public key the hacker has substituted: this is called a *person-in-the-middle attack*[6], and it allows the hackers to read and even modify all communications between the parties without them even knowing they have no security!

Graphically, the person-in-the-middle attack works like this:

Person-in-the-middle attack on asymmetric cryptosystems:

Alice	Eve	Bob
	intercept $k_e^B \leftarrowtail$	generate keys: public k_e^B, private k_d^B; publish k_e^B
download k_e^E, thinking it is k_e^B	generate keys: public k_e^E, private k_d^E; $\leftarrowtail k_e^E$, spoof origin	
create cleartext message m_A; using k_e^E, encrypt m_A to c_A and transmit c_A	$\rightarrowtail c_A$, intercept	
	using k_d^E, decrypt c_A and recover cleartext m_A; read m_A, change to m_E if desired using k_e^B, encrypt m_E to c_E spoof origin $c_E \rightarrowtail$	receive c_E, thinking it is c_A
		using k_d^B, decrypt c_E and recover cleartext m_E thinking it is m_A from Alice

[6]Although usually a more gendered term than "person" is used.

There is also a person-in-the-middle attack on digital signatures, based on similar ideas of substituting Eve's public key for Bob's, which we leave to the reader to think through for themself.

Reading Response 3.4.1

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you feel comfortable with how the person-in-the-middle attack works, for communications? What about for digital signatures?

Bonus Task 3.4.1

Here's another entirely non-coding Bonus Task:
Write out an explanation of the person-in-the-middle attack for digital signatures. Include a diagram, like the diagram above called "Better [smaller] digital signatures with hashing" crossed with the diagram above called "Person-in-the-middle attack on asymmetric cryptosystems."

Apparently, all of the wonders of asymmetric cryptoasymmetric cryptosystem will collapse unless we can already be sure that the networks we use and the websites we visit are never hacked. That is, unless there is some way we can have some sort of **root of trust**, a public key connected to some real-world entity that we already implicitly trust, and whose security practices we also trust, so that we can be sure that the private key is kept private.

Modern operating systems and browsers come with built-in roots of trust, which the users don't even know about, but which can be used to verify digital signatures signed by that entity. Perhaps that entity will have done the work of meeting Jeff Bezos in real life, and then will issue a signature on Jeff's public key. Then if you want to use your credit card on `amazon.com`, you can download Amazon's public key along with the root of trust's signature on that key, which will enable you to trust that that the key which seems to be Amazon's is, in fact, Amazon's.

These digital signatures on someone's public key are usually called **certificates** and the root of trust, in this context, is called a **certificate authority**.

The broad infrastructure of certificate authorities and public keys which can be trusted, perhaps because of certificates, to be associated with certain real-world persons or organizations is called a **public-key infrastructure [PKI]**. If the Internet had a robust, reliable public-key infrastructure, then we will be able to have widespread secure communication as well as legally binding (digitally signed) documents on the web. This, then, is the last manifestation of a persistent issue throughout this book: key management is very hard, but very important, as this necessity of a robust PKI makes clear yet again.

Reading Response 3.4.2

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you feel comfortable with the ideas of a root of trust (or certificate authority), certificates, and, in general, a PKI?

3.5. Conclusions; consider the blockchain

At first the idea of an asymmetric cryptosystem can seem a bit like magic. In fact, it must be admitted that the security of all of such cryptosystems in wide use today is based on assumptions that seem very reasonable but are not known with mathematical certainty: for example, the security of RSA is based on the assumption that it is impossible to factor large numbers in a reasonable amount of time (for a certain precise meaning of the word "reasonable").

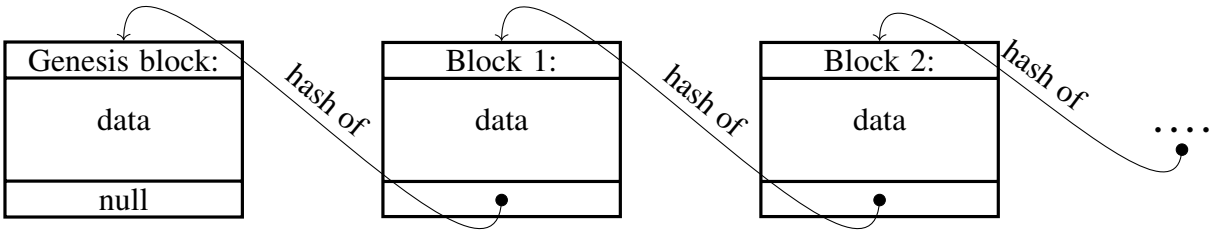
(Interestingly, the best known attack on this particular assumption, using a quantum computer, seemed for a long time like science fiction, but now seems like it definitely will start to be deployed at least by entities such as nation-states which have sufficient resources. Leave it to science fiction to be the best weapon against magic!)

How well all of the assumptions underlying asymmetric crypto stand the test of time, and whether practical, quantum-resistant asymmetric cryptosystems can be designed and implemented is a question of enormous importance in the next decade(s) of history of the Internet. While that cliff-hanger is being resolved by hard-working technologists behind the scenes, there are other exotic applications of asymmetric crypto which have been developed and which offer amazing potential benefits ... some which are, indeed, almost certainly too good to be true.

One example of this is something that has come to be widely discussed recently, called a **blockchain**, which you have probably heard of as the architecture underlying the cryptocurrency **Bitcoin**. If you worked in education, you would also have heard of blockchains as, supposedly, the best place for high school diplomas, college transcripts, and professional certificates to be stored.

A blockchain starts with a data structure called a **hash chain**, which is basically just a sequence of data blocks that can keep growing (potentially forever), with one additional field in each block: the hash (under some cryptographic hash function) of the previous block. Note that since that previous block contained the hash of the block which preceded it, these hashes implicitly link all the blocks back to the very first block of the chain, which is often called **genesis block**.

Graphically:



The security properties of cryptographic hash functions given in Definition 3.3.2 give hash chains a remarkable property: they are **immutable**, meaning they cannot be changed. Specifically, what that means is that if a whole bunch of people (call them "the hobbits") are sharing copies of a hash chain, then someone else (call this person "Gollum") cannot come to them and pretend that they have a valid copy of the chain which is the same at the beginning and the end, but has some extra blocks added in the middle. The reason this is impossible is that the first block – say it's block number 141592 near the end where Gollum agrees with the hobbits would have the hash of two different "previous" blocks: the hobbits would just have the same block they all agree is number 141591, while Gollum would have that extra block he was trying to sneak into the chain ... and those two different pieces of data cannot have the same hash (or, at least, it is very hard to figure out what data should be in the block to make it have the same hash value).

Reading Response 3.5.1

Does all of this make sense? What was new or interesting, or what was old and uninteresting? Do you feel comfortable with the ideas of a hash chain, and with why it is immutable?

Several additional features make a hash chain into a blockchain. These include the fact that parts of the data in the blocks must have valid digital signatures – a topic we have discussed in this chapter! – and that the new blocks are the result of a **consensus protocol**. Such protocols enable everyone who gets a copy of the blockchain to have confidence that they have the same chain of data blocks as everyone else who is using that blockchain.

Consensus protocols are a topic in even more advanced cryptology classes: don't stop here, there is much more that is fun, useful, and powerful to learn about in this field!

Bibliography

[BB20] Charles H Bennett and Gilles Brassard, *Quantum cryptography: Public key distribution and coin tossing*, arXiv preprint arXiv:2003.06557 (2020).

[Col14] David Cole, *We Kill People Based On Metadata*, The New York Review of Books **10** (2014), 2014.

[DH76] Whitfield Diffie and Martin Hellman, *New directions in cryptography*, IEEE transactions on Information Theory **22** (1976), no. 6, 644–654.

[Hel02] M. E. Hellman, *An overview of public key cryptography*, IEEE Communications Magazine **40** (2002), no. 5, 42–49.

[Mar99] Leo Marks, *Between Silk and Cyanide: A Codebreakers War*, HarperCollins, 1999.

[PS04] Jonathan A Poritz and Morton Swimmer, *Hash woes*, Virus Bulletin (2004), 14–16.

[Sha45] Claude Shannon, *A Mathematical Theory of Cryptography*, Bell System Technical Memo MM 45-110-02, `https://www.iacr.org/museum/shannon/shannon45.pdf`, Accessed: 22 February 2021.

[Sta02] Richard Stallman, *Free Software, Free Society: Selected Essays of Richard M. Stallman*, Lulu. com, 2002.

[WY05] Xiaoyun Wang and Hongbo Yu, *How to break md5 and other hash functions*, Advances in Cryptology–EUROCRYPT 2005, Springer, 2005, pp. 19–35.

Index