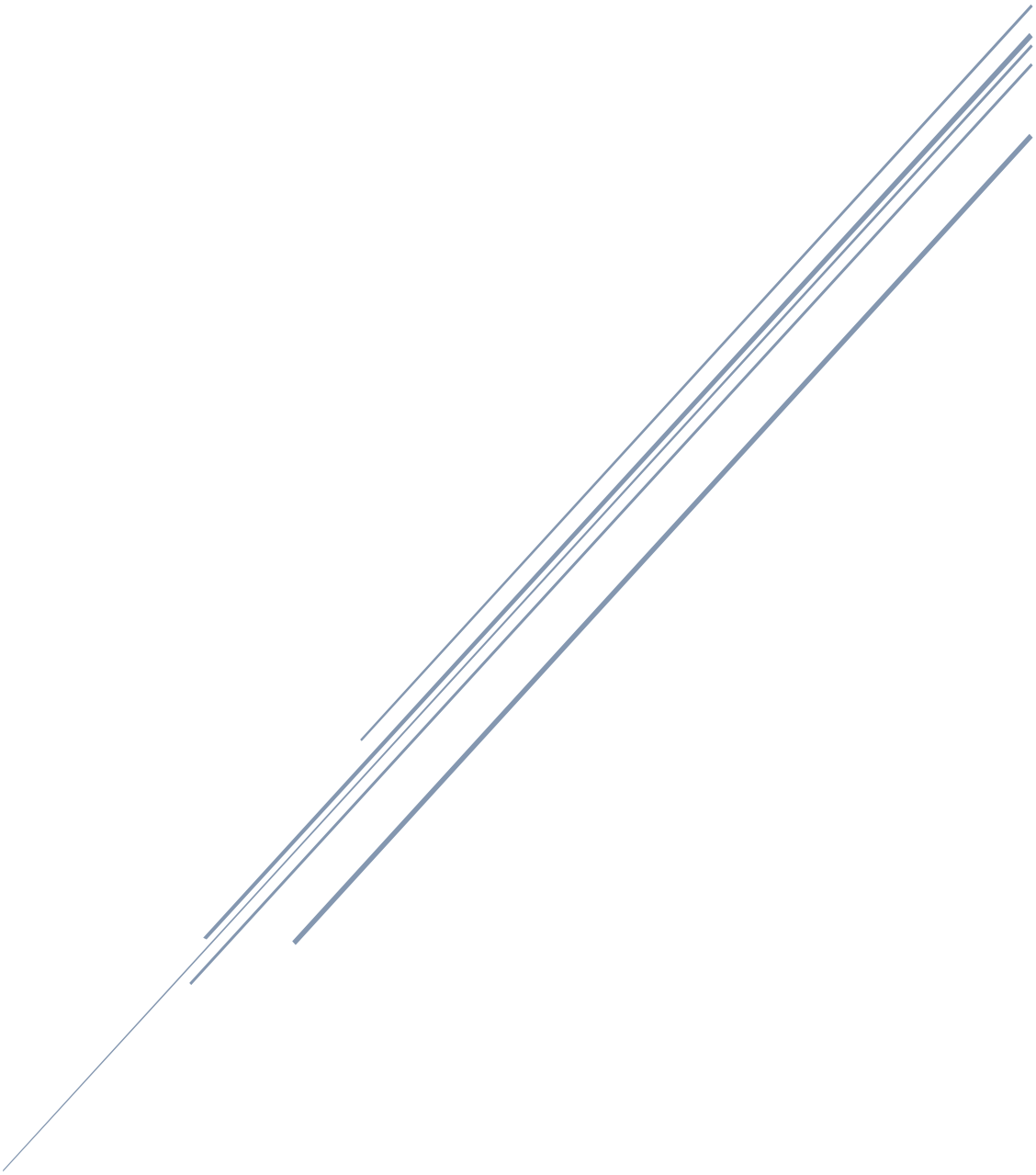
LA CRIPTOGRAFIA

Alexandre Ros i Roger



INS SERRALLARGA
Tutors: Nadina Palafoix i Albert Foradada

Agraïments

En primer lloc, vull agrair als meus tutors del Treball de Recerca, Albert Foradada i Nadina Palafoix, que m'han estat guiant, orientant i ajudant constantment el màxim possible. També vull agrair a tothom qui m'ha donat suport i s'ha interessat pel meu treball, incloent-hi alumnes, amics, professors i familiars.

Per últim, també dono les gràcies a totes aquelles persones d'arreu del món que s'han interessat per la criptografia i han aportat el seu gra de sorra en aquest estudi.

Síntesi

La Criptografia

En el dia d'avui, quan la societat avança cada cop més cap a un món digitalitzat i depèn més de les noves tecnologies, la criptografia i el seu futur són un element essencial per la nostra seguretat i privadesa. El meu Treball de Recerca se centra en aquest art; sobretot en investigar els mètodes criptogràfics que les civilitzacions antigues empraven. Viatjarem en el passat, present i futur, ens situarem en els contexts històrics i definirem matemàticament aquests mètodes. Els compararem i, per a acabar d'entendre els algorismes, els codificarem en *Python*. El meu objectiu és, doncs, donar una noció bàsica de la criptografia al lector.

La Criptografía

A día de hoy, cuando la sociedad avanza cada vez más hacia un mundo digitalizado y depende más de las nuevas tecnologías, la criptografía y su futuro son un elemento esencial para nuestra seguridad y privacidad. Mi *Treball de Recerca* se centra en este arte; sobretodo en investigar los métodos criptográficos que las civilizaciones antiguas usaban. Viajaremos en el pasado, presente y futuro, nos situaremos en los contextos históricos y definiremos matemáticamente estos métodos. Los compararemos y, para acabar de entender los algoritmos, los codificaremos con *Python.* Mi objetivo es, pues, dar una noción básica de la criptografía al lector.

Cryptography

Nowadays, when society goes towards a more digitized world and relies more on new technologies, cryptography and its future are an essential component for our security and privacy. My *Treball de Recerca* is based upon this art; especially taking a look at the cryptographic methods used by ancient civilizations. We'll travel back to the past, present and future, we'll discuss the historical backgrounds and we'll define these methods mathematically. We'll compare them and, in order to fully understand the algorithms, we'll code them in *Python.* My goal is, thus, to give the reader some basic cryptographic ideas.

Índex

Introducció

Des que vaig descobrir què és la criptografia tres anys enrere, no he deixat d'aprendre i apassionar-me per aquesta. Personalment, el fet de tenir un missatge illegible davant teu que pot ser desencriptat i cobrar sentit és quelcom impressionant i admirable. Quan se'm va donar l'oportunitat d'escollir un tema pel Treball de Recerca, no vaig dubtar més: la meva recerca aniria orientada a l'estudi de la criptografia. Per casualitat, el tema del treball fou proposat posteriorment pel Departament de Matemàtiques. Aquest fet em motivà encara més a escollir aquest nou tema.

La següent pregunta que sorgí fou quin seria el meu objectiu en aquest treball al qual dedicaria els meus pròxims cinc mesos. Inicialment, se'm va ocórrer que podria crear el meu propi xifratge o, fins i tot, crear la meva pròpia criptomoneda. Després d'unes quantes setmanes reflexionant sobre el meu objectiu, vaig arribar a la conclusió ideal: podria fer una recerca cronològica dels xifratges més destacats i, llavors, codificar aquests xifratges amb el programari Python.

El meu objectiu principal és, doncs, introduir el lector en el món de la criptografia explicant-la de la manera més entenedora i precisa per a qualsevol lector disposat a ampliar els seus coneixements. En el dors d'aquest document original, vostès hi trobaran una memòria USB que conté aquest mateix treball en format PDF i tota la codificació amb Python dels xifratges. A més a més, el document de text README.txt us explicarà com procedir amb la instal·lació del compilador Python.

Si no disposeu del document original, he creat un reservori a GitHub[1] que conté el mateix que la memòria USB. Podeu descarregar tots els fitxers del reservori en format ZIP si cliqueu "Clone or download" i, seguidament, "Download ZIP".

[1] Vegeu el reservori a https://github.com/alexland7219/La-Criptografia

1 Introducció a la criptografia

1.1 Definició i usos

La criptografia (del grec: κρυπτός "secret"; i γράφειν "escriure") és l'estudi i la pràctica de les tècniques per la comunicació segura que permeten que només siguin l'emissor i el receptor els únics que puguin obtenir i llegir el missatge enviat.

Hi ha indicis que suggereixen que els primers textos en els quals s'emprà la criptografia daten de l'antic Egipte. En la tomba de Khnumhotep II es trobaren jeroglífics escrits amb símbols inusuals amb l'objectiu d'amagar parcialment el significat de les inscripcions. (Cypher Research Laboratories Pty. Ltd. 2013)

Les civilitzacions cada vegada han anat perfeccionant els mètodes criptogràfics amb el temps. Els avenços tecnològics i de coneixements matemàtics han permès l'evolució d'aquests algorismes i els han anat millorant. De la mateixa manera que s'han anat millorant, però, els recursos per a desxifrar els algorismes també han evolucionat; sobretot avui dia quan la computació permet dur a terme tasques en centèsimes de segon.

La criptografia s'empra constantment en la nostra vida privada i pública. En l'àmbit diplomàtic i militar, la confidencialitat de les telecomunicacions és probablement el més important de tot abans de transmetre un missatge. La capacitat de transmetre i rebre informació sense que l'enemic pugui esbrinar les intencions d'aquest missatge poden decidir el destí d'un conflicte.

La invenció del *WWW* (*World Wide Web*; en anglès: "Xarxa d'abast mundial") el 1989 i la popularització de la Internet han obert una nova era humana: L'Era de la Informació. Aquesta nova era comporta, però, una necessitat primordial per la seguretat de la informació transmesa. La missatgeria instantània, els comptes bancaris i tota la nostra informació personal estaria a l'abast de tothom sense la criptografia.

1.2 Fonaments bàsics de la criptografia

L'objectiu de la criptografia és, com s'ha mencionat, transmetre un missatge a un receptor autoritzat sense que ningú més pugui rebre'l. A partir d'ara, utilitzarem la següent analogia:

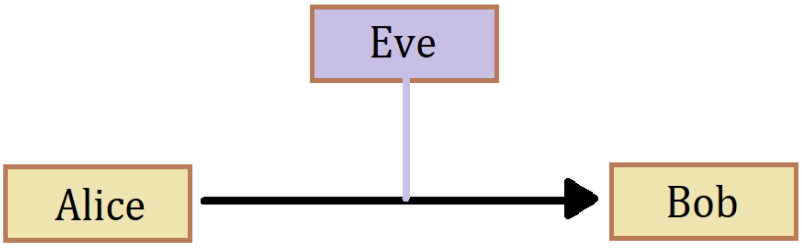


Figura I – Emissor, Receptor i Interceptor – Elaboració pròpia

On *Alice* és l'emissor, *Bob* és el receptor i *Eve* és un interceptor qualsevol. L'*Alice* vol enviar un missatge a en *Bob* sense que l'*Eve* pugui entendre'l. Aquesta analogia s'empra molt sovint en àmbits matemàtics com la criptografia, la teoria dels jocs i ciències com la física.

Per aconseguir-ho, l'*Alice* primer haurà de canviar la naturalesa del contingut a enviar. L'ha de canviar per tal que l'*Eve* no pugui entendre el missatge, però en *Bob* sí. El procés que consisteix a canviar el codi del missatge o *plaintext* s'anomena **encriptació**.

Aquest nou missatge, normalment anomenat text xifrat, missatge encriptat o *ciphertext*, ara és illegible per tothom. Quan l'*Eve* rebi el *ciphertext*, no podrà entendre'l.

Però llavors, com s'ho farà en *Bob* per obtenir el *plaintext* si el que rep és el text encriptat?

El procés que haurà de dur a terme en *Bob* s'anomena **desxifratge** o desencriptació**.** El desxifratge convertirà el *ciphertext* de nou en el missatge original. Aquesta operació és la inversa de l'encriptació.

L'encriptació i la desencriptació són, llavors, dos algorismes o funcions matemàtiques inverses que tenen la funció de convertir el missatge en un altre illegible i al contrari respectivament. Vegem-ho gràficament:

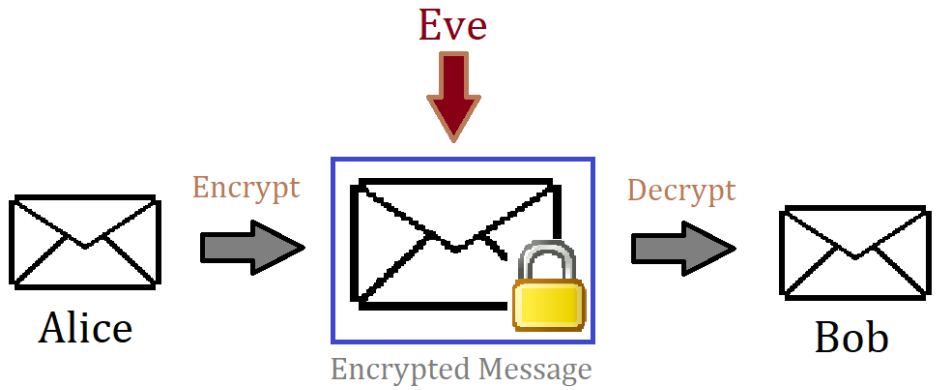


Figura II – Diagrama de l'encriptació i desencriptació – Elaboració pròpia

La desencriptació requereix dues variables: el missatge encriptat i una clau o *keyword* per a verificar que sigui en *Bob* qui intenti desxifrar el contingut i no l'*Eve*. Així doncs, només qui tingui la clau podrà llegir el missatge.

L'encriptació també demana una clau. La clau d'encriptació i desencriptació ha de ser la mateixa, d'aquesta manera l'*Alice* encriptarà el missatge amb la clau i en *Bob* el desxifrarà amb aquesta. Els algorismes criptogràfics que forcen l'emissor i el receptor tenir la mateixa clau s'anomenen xifratges amb clau simètrica.

Més endavant veurem algorismes de clau asimètrica o pública; on l'*Alice* i en *Bob* no han escollit cap clau amb antelació. En aquests casos, el protocol emprat per comunicar-se és considerablement de més complexitat. Tot i això, normalment són més segurs.

1.3 La teoria dels nombres

La teoria dels nombres és una branca de les matemàtiques pures que estudia el conjunt dels nombres enters positius

$$1, 2, 3, 4, 5 \ldots$$

Fins a mitjans del segle XX, aquesta branca es considerava la més pura de les matemàtiques; sense aplicacions directes al món quotidià. Els ordinadors i la computació demostren avui dia que la teoria dels nombres pot donar resposta a problemes del món real. Avenços en la informàtica han permès el progrés i l'estudi d'aquest camp.

(Dunham 2013)

Un dels estudiosos més antics fou Euclides d'Alexandria (segle III aC), un matemàtic grec conegut per ser "el pare de la geometria". Euclides escrigué una de les obres més conegudes i famoses de la història de les matemàtiques: Els *Elements*. Euclides va començar definint conceptes bàsics com un punt o una recta i, a partir d'aquests pilars o axiomes, va introduir conceptes més complexos.

A més a més de geometria, els Llibres VII-IX contenen elements de la teoria dels nombres. Euclides definí conceptes com un nombre parell, imparell o primer, i desenvolupà un algorisme per a trobar de manera senzilla el màxim comú divisor de dos nombres: el conegut *Algorisme d'Euclides.*

Figura III – Els Elements - Euclides (autor), Erhard Ratdolt (impressor)

Un altre matemàtic molt important en aquesta branca fou Pierre de Fermat. Fermat és conegut per introduir per primer cop el càlcul infinitesimal i la seva recerca en la teoria dels nombres. En aquest àmbit, Fermat postulà diverses conjectures. El seu teorema més famós és el *Darrer Teorema de Fermat,* un teorema que ell reclamà haver provat.

Mentre Fermat llegia l'obra d'un altre matemàtic destacat en la teoria dels nombres, Fermat escrigué en el marge d'una pàgina el següent: "És impossible per qualsevol número que sigui una potència major de la segona ser escrit com la suma de dos altres enters de la mateixa potència.". I afegeix, "Tinc una demostració meravellosa i vertadera d'aquesta proposició, però aquest marge és massa estret per escriure".

Dit en altres paraules, que no existeixen tres enters positius a, b, c que satisfacin l'equació $a^p + b^p = c^p$ per qualsevol enter p major de 2. S'ha comprovat que existeixen infinites solucions pels casos $p = 1$ i $p = 2$. El famós teorema no fou demostrat fins al 1995 pel matemàtic britànic Andrew Wiles.

El teorema més important de Pierre de Fermat per la branca de la criptografia moderna és el *Petit Teorema de Fermat,* el qual més endavant estudiarem. Aquest teorema ens ajudarà a determinar si un nombre natural p és primer o no.

Avui dia, la teoria dels nombres és una branca fonamental per la criptografia moderna. Els nombres primers i l'aritmètica modular són presents en gairebé tots els algorismes que anirem estudiant.

1.4 L'aritmètica modular

Quan dividim dos enters positius A i B, obtenim la següent equació:

$$\frac{A}{B} = Q + \frac{R}{B} \Rightarrow A = QB + R$$

On A és el dividend, B el divisor, Q el quocient i R el residu. $A, B, Q, R \in \mathbb{R}$ i $B \neq 0$ i $0 \leq r < B$.

A vegades només ens interessa quin és el residu quan dividim A i B. Quan és així, emprem l'operador mòdul:

$$A \bmod B = R$$

Amb *Python* i la majoria d'altres llenguatges de programació, l'operador mòdul se simbolitza amb un signe de tant per cent (%).

Proposo un exemple per familiaritzar-nos amb l'aritmètica modular. Sigui $A = 34$ i $B = 5$,

$$\frac{34}{5} = 6 + \frac{4}{5} \Leftrightarrow 34\ mod\ 5 = 4$$

Si observem com es comporta el residu quan incrementem el dividend, veurem perquè l'aritmètica modular és un sistema aritmètic cíclic. Sigui $B = 4$,

$$0\ mod\ 4 = 0$$

$$1\ mod\ 4 = 1$$

$$2\ mod\ 4 = 2$$

$$3\ mod\ 4 = 3$$

$$4\ mod\ 4 = 0$$

$$5\ mod\ 4 = 1$$

I així successivament. Si examinem els residus, veiem que van incrementant per 1 fins que arribem a un múltiple de 4. Llavors, la seqüència es repeteix. El fet que un múltiple de 4 tingui com a residu 0 és senzill de demostrar. Donat que un múltiple de 4 es pot expressar com $4k$ on $k \in \mathbb{N}$,

$$\frac{4k}{4} = k + \frac{0}{4} \Rightarrow R = 0 \Rightarrow 4k\ mod\ 4 = 0$$

I generalitzant per un divisor B,

$$Bk\ mod\ B = 0, \qquad k \in \mathbb{N}$$

Si emprem aquesta identitat, podem calcular mòduls d'una manera més ràpida: trobem el múltiple de B més proper al dividend A i la diferència entre el dividend i aquest múltiple és el residu de la divisió.

$$(Bk + c)\ mod\ B = c, \;\; 0 \leq c < B$$

Una altra manera d'entendre com l'aritmètica modular funciona és amb un rellotge. Si tornem a l'exemple anterior on $B = 4$, podem dibuixar un rellotge amb quatre valors. Com que els valors d'un rellotge es repeteixen de manera cíclica, l'analogia serveix.

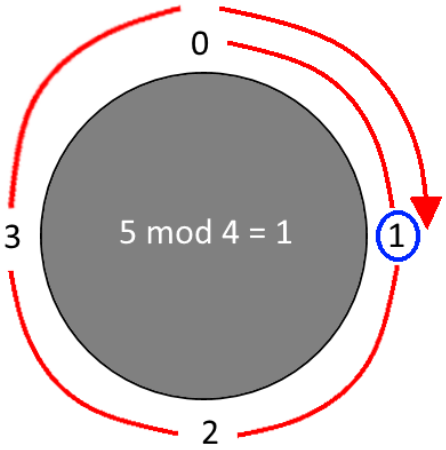


Figura IV – Rellotge Modular – Elaboració pròpia

En l'exemple proposat en la il·lustració de la Figura IV, s'intenta avaluar l'expressió $5\ mod\ 4$ emprant un rellotge. S'escriuen els quatre valors que pot prendre i, començant pel 0, es compten cinc passes en direcció horària. En aquest cas, el cicle es completarà una vegada (un múltiple de 4) i sobrarà una passa. En acabar el procés, el valor en el qual haguem "caigut" serà el resultat de l'operació. $5\ mod\ 4 = 1$.

És possible que vegem una altra notació quan estiguem treballant amb l'aritmètica modular. Si dos valors M i N són de la mateixa equivalència, escrivim:

$$M \equiv N\ (mod\ B)$$

Que dos valors siguin de la mateixa equivalència d'un mòdul significa que els dos valors tenen el mateix residu quan són dividits per aquest mòdul. És a dir, l'expressió de dalt és equivalent a:

$$M\ mod\ B = N\ mod\ B$$

L'expressió $M \equiv N\ (mod\ B)$ es llegeix com "M és congruent a N amb un mòdul de B".

Emprant l'analogia del rellotge, dos valors són congruents si, quan se'ls aplica l'operació $mod\ B$, cauen en el mateix valor. Diem que aquests dos valors pertanyen en la mateixa classe d'equivalència o són congruents.

Més endavant, i sobretot quan ens endinsem més en la criptografia moderna, veurem perquè ens és útil l'aritmètica modular. Parlarem de funcions unidireccionals, tests de primeritat i d'exponenciació modular.

(Khan Academy 2016)

2 La criptografia clàssica

Els usos més antics de la criptografia es remunten a les primeres civilitzacions. El reemplaçament de símbols, una de les formes més bàsiques de la criptografia, apareix a l'escriptura de l'antic Egipte i Mesopotàmia. L'exemple més antic es troba en la tomba de Khnumhotep II, que visqué fa 3.900 anys.

(Binance Academy 2019)

Tot i això, alguns historiadors conclouen que el propòsit d'aquesta pràctica en les tombes egípcies era embellir el llenguatge i donar-li un toc místic. En la civilització mesopotàmica, el reemplaçament d'alguns símbols servia per a ocultar informació en els textos quan eren transportats. Aquest ús és més similar al d'avui en dia, la protecció d'informació quan s'ha de transmetre. És aquí, doncs, on la criptografia nasquí.

Més endavant, la criptografia s'utilitzaria per protegir informació militar. A l'antiga Grècia, els espartans encriptaven missatges enrotllant una tira de pergamí al voltant d'un objecte cilíndric. El missatge s'escrivia i llavors el paper es desenrotllava; només si tenies un objecte cilíndric del mateix diàmetre podies desxifrar el pergamí. A aquest sistema criptogràfic s'hi ha donat el nom d'*Escítala* o *Scytale* en anglès.

Figura V – Una escítala – Lliure de drets d'autor

L'escítala és un exemple de xifratge de transposició, on els caràcters del missatge no canvien; el que canvia és l'ordre d'aquests.

2.1 Xifratge d'Atbash

Aquest xifratge de substitució apareix en textos jueus molt antics. Apareix en el llibre de Jeremies, el segon llibre profètic de la Bíblia i és un dels més coneguts, malgrat no ser gens segur. El xifratge d'Atbash no utilitza cap clau, cosa que el fa molt vulnerable i fàcil de desxifrar.

Aquest xifratge consisteix a canviar la primera lletra de l'alfabet per l'última, la segona per la penúltima i així successivament:

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

Taula 1 – El xifratge d'Atbash en lletres

Com a exemple, si encriptem el pangrama en anglès

"THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG",

obtenim:

"GSV JFRXP YILDM ULC QFNKH LEVI GSV OZAB WLT".

El *ciphertext* és completament illegible, però tan fàcil és encriptar que desencriptar. Com que l'únic que hem fet ha sigut voltejar l'alfabet, tornar a encriptar el *ciphertext* el desencriptaria. Si ens fixem en les caselles de color groc, la lletra *L* es converteix en *O* quan encriptem. Simultàniament, la lletra *O* es converteix en *L*.

Per a entendre millor per què ocorre això, substituirem cada lletra pel seu índex a l'alfabet, és a dir, $A = 0; B = 1; C = 2; \dots; Z = 25$:

0	1	2	3	4	5	6	7	8	9	10	...	25
25	24	23	22	21	20	19	18	17	16	15	...	0

Taula 2 – El xifratge d'Atbash numèric

La suma de cada columna sempre és igual a 25.

Podríem haver començat enumerant pel número u, però com veurem més endavant, les llistes en els llenguatges de programació comencen pel zero i l'aritmètica modular també comença amb el zero.

A continuació definirem la funció f que acceptarà com a entrada una lletra de l'alfabet llatí modern i retornarà la lletra encriptada amb Atbash.[2]

Com que el domini i el codomini de la funció f és l'alfabet llatí modern, escrivim $f: A \to A$ on $A = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{G}, \mathrm{H}, \mathrm{I}, \mathrm{J}, \mathrm{K}, \mathrm{L}, \mathrm{M}, \mathrm{N}, \mathrm{O}, \mathrm{P}, \mathrm{Q}, \mathrm{R}, \mathrm{S}, \mathrm{T}, \mathrm{U}, \mathrm{V}, \mathrm{W}, \mathrm{X}, \mathrm{Y}, \mathrm{Z}\}$. Diem que la funció f és una endofunció, ja que el domini és el mateix conjunt que el codomini. En aquest cas, el codomini és el mateix conjunt que el rang de la funció perquè tots els membres del codomini tenen una antiimatge.

Com que a cada lletra li correspon una única lletra del mateix conjunt i el conjunt A és finit (la cardinalitat és de vint-i-sis), la funció f és bijectiva. A més a més, volem demostrar que la funció és involutiva ($(f \circ f)(x) = x$). És a dir, que si apliquem la funció dues vegades, tornarem al valor inicial.

Abans que res, per a construir aquesta funció, necessitarem assignar a cada lletra un valor numèric. Utilitzarem la mateixa substitució d'abans ($A = 0; B = 1; \dots; Z = 25$).

Un cop fet això, definir la funció f és molt senzill si en comptes de lletres treballem amb nombres naturals. Ara, el domini de la funció és el conjunt de nombres naturals i el zero entre el 0 i el 25 (inclosos), o sigui, $D_f = \{x \in \{0\} \cup \mathbb{N} \mid x \leq 25\}$. L'observació que hem fet anteriorment és clau: si $f(x) = y, \quad x + y = 25$.

Si resolem per $f(x)$,

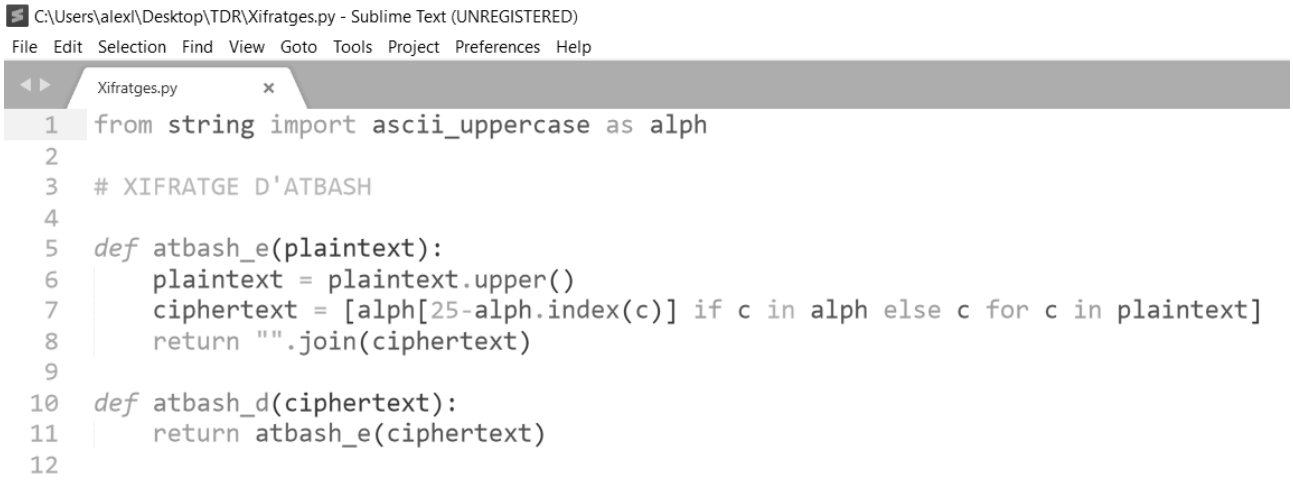
$$f(x) = 25 - x$$

Seguidament, podem comprovar que aquesta funció és involutiva:

$$(f \circ f)(x) = f\big(f(x)\big) = f(25 - x) = 25 - (25 - x) = x$$

Per a acabar, codificarem tot el procés amb *Python*. No us preocupeu si no en sabeu, ja que aniré descrivint tot el procés detalladament. Definirem una funció que tingui com a única entrada una *string* o cadena de caràcters (una paraula, frase o missatge) i que retorni la cadena de caràcters encriptada amb Atbash.

[2] Per a qualsevol dubte amb relació a l'estudi de les funcions, dirigiu-vos a l'apèndix final.



Fragment de codi 1 – El xifratge d'Atbash

En la primera línia de codi estem important l'alfabet llatí en majúscules (en una *string*) i l'hem guardat a la variable *alph*. La tercera línia és només un comentari.

En la línia cinc definim la funció que he anomenat *atbash_e*. Com podem veure, per a declarar una funció utilitzem la declaració *def* amb *Python.* La funció demana una variable que he anomenat *plaintext*.

En la línia sis, la funció *str.upper* simplement converteix totes les lletres de la variable *plaintext* en majúscules; per si l'usuari introdueix cap lletra minúscula.

En la setena línia de codi estem construint el *ciphertext.* La manera més pràctica i curta de construir-lo és mitjançant una comprensió de llista. Aquí estem dient que:

Per cada caràcter (guardat a la variable temporal *c*) del *plaintext*, si *c* no està a l'alfabet (*alph),* s'afegirà el caràcter *c* al *ciphertext* (per si la *string* conté números, espais o altres símbols que no siguin lletres). Si, en efecte, *c* és una lletra de l'alfabet (*c in alph*), s'afegirà la lletra amb l'índex $25 - alph.index(c)$ de l'alfabet; on *alph.index(c)* és l'índex del caràcter *c* a l'alfabet.

Com que hem utilitzat una comprensió de llista, *ciphertext* és una llista. La vuitena línia de codi retorna la *string* que s'obté a l'unir tots els caràcters d'aquesta llista.

Finalment, també he definit la funció de desxifratge anomenada *atbash_d*. Com que la funció és involutiva, només cal retornar el mateix que retornaria la funció *atbash_e.*

2.2 Xifratge de Cèsar

Sens dubte aquest xifratge és un dels més coneguts i un dels més segurs de la seva època. El famós xifratge fou inventat pels romans i, com el nom indica, porta el cognom del líder polític de la República romana Juli Cèsar, qui utilitzava aquest xifratge per a comunicar-se privadament. El xifratge admet una clau, però aquesta només pot prendre vint-i-cinc valors, cosa que el fa bastant senzill de desxifrar sense saber-ne la clau.

En el xifratge de Cèsar, també conegut com a xifratge per decalatge, cada lletra del *plaintext* se substitueix per la lletra que ocupa k llocs més endavant a l'alfabet, on k és el *keyword*. En el següent exemple, $k = 7$.

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

Taula 3 – El xifratge de Cèsar amb un decalatge de 7 posicions

En l'exemple de color groc, la lletra *A* es desplaça set llocs a l'alfabet i se substitueix per la lletra *H*. En el cas que no tinguem prou lletres a l'alfabet, com és el cas de color salmó, saltem de la *Z* a la *A.* Fem ús de l'alfabet d'una manera **cíclica**. En l'exemple següent encriptarem el pangrama amb el xifratge de Cèsar i un decalatge $k = 7$:

"THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG" esdevé:

"AOL XBPJR IYVDU MVE QBTWZ VCLY AOL SHGF KVN".

Per desencriptar, retrocedim k llocs a l'alfabet (on k és la clau o el decalatge). D'aquesta manera estem desfent l'encriptació. Com que el xifratge de Cèsar és un xifratge de substitució mono-alfabètica (cada lletra se substitueix per una de diferent), és pràctic escriure una taula de dues files on a la primera surti l'alfabet del *plaintext* i a la segona l'alfabet del *ciphertext*.

En el cas que no sapiguem quina és la clau que s'ha emprat, com que aquesta només pot prendre vint-i-cinc valors, un simple atac per força bruta (anar provant claus fins que el *ciphertext* cobri sentit) seria suficient per a desxifrar el xifratge de Cèsar.

Seguidament definirem la funció f_k (on k és el decalatge) que acceptarà com a entrada una lletra de l'alfabet i retornarà la lletra encriptada amb el xifratge de Cèsar.

Com el xifratge d'Atbash, $f_k: A \to A$ on A és el conjunt de lletres de l'alfabet llatí modern. A més a més, també comprovarem per què la clau només pot prendre vint-i-cinc valors. Abans, però, substituirem cada lletra pel seu índex a l'alfabet ($A = 0; B = 0; \dots; Z = 25$).

Com que l'índex de la lletra resultant és el mateix que la suma de l'índex de la lletra del *plaintext* i el decalatge, escrivim:

$$f_k(x) = x + k$$

Això, però, no és del tot correcte. No hem tingut en compte el cas quan la suma de l'índex i la clau és superior a 25. Tornem a l'exemple de la lletra *W* (índex 22) i el decalatge de 7:

$$f_7(22) = 29$$

Com hem vist a la pàgina anterior, el resultat hauria de ser la lletra *D* (índex 3) ja que saltem de la lletra *Z* a la *A*. Això equivaldria a restar 26 unitats al resultat si, i només si, el resultat fos major o igual a 26. Afortunadament, podem utilitzar l'aritmètica modular per a mantenir el rang de la funció dintre del seu codomini. $29\ mod\ 26 = 3$.

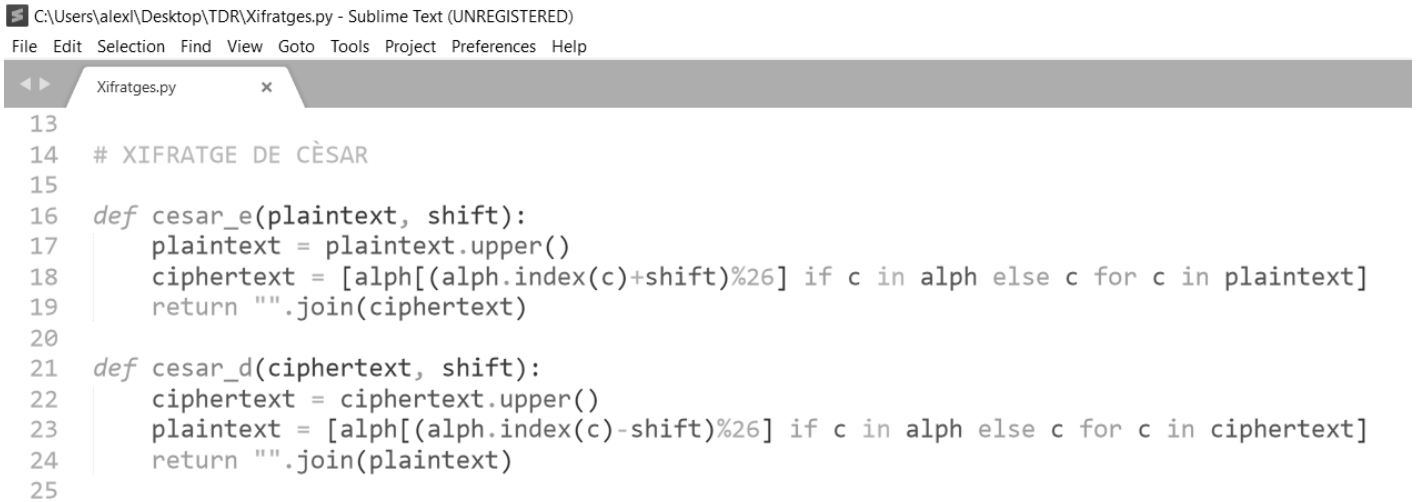
$$f_k(x) = (x + k)\ mod\ 26$$

A més a més, si $k \notin \{x \in \{0\} \cup \mathbb{N} |\ x \leq 25\}$, k pot ser expressada com $a + 26b$ on a sí que és membre del dit conjunt. Com que $(a + 26b)\ mod\ 26 = a$, deduïm que si el decalatge és superior a 25, el decalatge resultant pot ser expressat per un altre inferior a 26.

Per exemple, si prenem una clau $k = 31$, el *ciphertext* resultant serà el mateix que obtindríem si haguéssim escollit k ser 5. Intuïtivament, una clau de vint-i-sis significa que cada lletra de l'alfabet es desplaça vint-i-sis llocs a l'alfabet i retorna a la mateixa posició inicial (equivaldria a ser un decalatge de 0).

$$31\ mod\ 26 = 5$$

$$31 = 26 \cdot 1 + 5$$



Fragment de codi 2 – El xifratge de Cèsar

Procedirem a la programació del xifratge de Cèsar amb *Python*. Definirem les funcions d'encriptació *cesar_e* i desencriptació *cesar_d*. La primera acceptarà com a entrada el *plaintext* i el decalatge o *shift* en anglès. A la segona, en canvi, l'usuari introdueix el *ciphertext* i el decalatge i la funció retorna el missatge original.

Com en el xifratge d'Atbash, en les línies disset i vint-i-dos de codi cal convertir tot el text en majúscules del *plaintext* i del *ciphertext* respectivament. A les línies 18 i 23 de codi és on construïm el que la funció retornarà mitjançant una comprensió de llista.

Les comprensions de llista, encara que puguin semblar bastant complexes, són la manera més eficaç per a dur a terme tasques que necessiten iteració. Descriuré en detall la primera comprensió de llista, però convido el lector a meditar sobre la iteració i la sintaxi d'aquestes. En la línia divuit de codi, estem dient que:

Per cada caràcter *c* del *plaintext*, si *c* no està a l'alfabet, ens saltem aquest caràcter i l'afegim directament al *ciphertext*. Si, en efecte, *c* és una lletra de l'alfabet, s'afegirà al *ciphertext* la lletra amb l'índex $(alph.index(c) + shift)\ mod\ 26$ de l'alfabet, on *alph.index(c)* és l'índex de la lletra *c* a l'alfabet. L'operador mòdul se simbolitza amb el signe de tant per cent *%*.

Com que la comprensió de llista genera una llista, en la línia 19 i 24 retornem la unió dels caràcters de les llistes *ciphertext* i *plaintext* respectivament. A més a més, observem que en el desxifratge restem el decalatge a l'índex de la lletra del *ciphertext.* D'aquesta manera estem desfent el procés d'encriptació.

2.3 Quadrat de Polibi

El quadrat de Polibi, inventat per l'historiador grec Polibi que visqué durant el segle II aC, és un xifratge de substitució mono-alfabètica el qual originalment no necessita cap clau o *keyword*. Com veurem més endavant, podem modificar-lo lleugerament perquè admeti una clau. Hi ha més xifratges que estudiarem que utilitzen el quadrat de Polibi per a encriptar, com per exemple el xifratge de Playfair o el xifratge ADFGVX.

El xifratge en si és bastant fàcil d'entendre; les vint-i-sis lletres de l'alfabet són col·locades en un quadrat o taula de valors de cinc fileres i cinc columnes:

	1	2	3	4	5
1	A	B	C	D	E
2	F	G	H	I / J	K
3	L	M	N	O	P
4	Q	R	S	T	U
5	V	W	X	Y	Z

Taula 4 – El quadrat de Polibi sense clau

Com que necessitem col·locar vint-i-sis lletres en vint-i-cinc coordenades, dues lletres han de compartir el mateix espai. Per conveni, aquestes dues són la I i la J.

Per a encriptar, cada lletra del *plaintext* se substitueix per un nombre natural de dues xifres: el número de la seva filera i el número de la seva columna respectivament. Si encriptem

"POLYBIUS SQUARE" obtenim

"35 34 31 54 12 24 45 43 43 41 45 11 42 15".

El xifratge pot ser modificat perquè en *Bob* i l'*Alice* puguin comunicar-se amb una clau. Si canviem la posició de les lletres al quadrat de Polibi, el *ciphertext* resultant també canvia. Podem definir la clau, doncs, com el posicionament de les lletres al quadrat o l'ordre de les lletres a l'alfabet.

Si la clau és "YTGQWXPIMVULNHFEDBAZKORSC", construirem el següent quadrat de Polibi:

	1	2	3	4	5
1	Y	T	G	Q	W
2	X	P	I / J	M	V
3	U	L	N	H	F
4	E	D	B	A	Z
5	K	O	R	S	C

Taula 5 – El quadrat de Polibi amb un alfabet com a clau

Encara podem simplificar la clau una mica més si així ho desitgem. Si deixem que emissor i receptor es posin d'acord en una paraula o *keyword*, podem reorganitzar les lletres de la següent manera:

Si el mot clau és "CRYPTOGRAPHY", comencem emplenant el quadrat amb les primeres lletres de la paraula. Si ens trobem amb una lletra que ja hem escrit, la saltem. Quan acabem d'escriure el mot, continuem emplenant amb l'alfabet com en el primer cas. Si seguim el procés, obtenim el quadrat de Polibi

	1	2	3	4	5
1	C	R	Y	P	T
2	O	G	A	H	B
3	D	E	F	I / J	K
4	L	M	N	Q	S
5	U	V	W	X	Z

Taula 6 – El quadrat de Polibi amb una paraula com a clau

Per a definir d'una manera més formal aquest xifratge, podem introduir els valors en una matriu de cinc per cinc la qual anomenarem M. Si encriptem l'element m_{ij} de la matriu, obtenim el valor numèric de dues xifres $\overline{ij}$ o $10i + j$.

$$M = \begin{bmatrix} C & R & Y & P & T \\ O & G & A & H & B \\ D & E & F & I & K \\ L & M & N & Q & S \\ U & V & W & X & Z \end{bmatrix}$$

Xifratges.py

```
28  # QUADRAT DE POLIBI
29
30  def matriu(keyword):
31      matriu = [[0 for y in range(5)] for x in range(5)]
32      i = j = 0
33      for c in keyword.upper().replace("J", "I") + alph.replace("J", ""):
34          if any([c in m for m in matriu]): continue
35          matriu[i][j] = c
36          j = (j+1)%5
37          if j == 0: i += 1
38      return matriu
39
40  def polibi_e(keyword, plaintext):
41      M = matriu(keyword)
42      ciphertext = ""
43      for c in plaintext.upper().replace("J", "I"):
44          if c not in alph:
45              if c != " ": ciphertext += c + " "
46              continue
47          fila = [n for n in M if c in n][0]
48          ciphertext += str(M.index(fila)+1) + str(fila.index(c)+1) + " "
49      return ciphertext[:-1]
50
51  def polibi_d(keyword, ciphertext):
52      M = matriu(keyword)
53      plaintext = ""
54      for n in ciphertext.split(" "):
55          if len(n) != 2:
56              plaintext += n + " "
57              continue
58          plaintext += M[int(n[0])-1][int(n[1])-1] + " "
59      return plaintext[:-1]
60
```

Fragment de codi 3 – El quadrat de Polibi amb una paraula com a clau

He dividit el codi en tres funcions. La primera, anomenada *matriu*, té com a entrada la paraula clau que s'utilitzarà per a construir el quadrat de Polibi. Aquesta funció retorna el quadrat de Polibi en forma de llista la qual conté cinc llistes (les cinc files) amb cinc elements cadascuna (els cinc elements per fila). D'aquesta manera aconseguim crear una matriu del quadrat de Polibi que ens servirà per a encriptar i desencriptar.

Les dues altres funcions són la d'encriptació i la de desencriptació. La d'encriptació retornarà els valors de dues xifres separats per un espai. Els caràcters que no siguin membres del nostre alfabet de vint-i-sis lletres no seran encriptats i apareixeran al *ciphertext* en majúscules. Tots els espais seran ignorats.

La funció de desencriptació accepta com a entrada la *string* del *ciphertext*. Aquest mateix ha de tenir separats els números de dues xifres per un espai i pot contenir símbols no reconeguts, els quals també hauran d'anar separats per un espai. La funció retornarà el *plaintext* amb un espai per cada lletra o caràcter. A més a més, les lletres I del *plaintext* poden ser originalment o bé jotes o bé is.

Primer de tot, en la primera funció anomenada *matriu*, creem una matriu nul·la de cinc per cinc. Seguidament, declarem les variables *i* i *j*, les quals es referiran a quina fila i quina columna respectivament guardar cada element. La funció *str.replace* reemplaça qualsevol instància de la lletra jota per una i. Després, iterem primer amb les lletres del *keyword* i llavors amb les de l'alfabet. A la línia 32, si una d'aquestes lletres ja es troba a la matriu, aquesta serà omesa i es continuarà amb la iteració.

Dintre la funció d'encriptació *polibi_e* executem la funció *matriu* i guardem la matriu en la variable *M.* Com sempre, iterem amb les lletres del *plaintext* en majúscules i reemplacem totes les jotes per is. Si el caràcter no està a l'alfabet, l'afegim directament al *ciphertext* amb un espai. A la línia 46 identifiquem la fila de la matriu que conté la lletra en qüestió i la guardem en la variable *fila*. Seguidament, afegim al *ciphertext* l'índex de la fila més u (a causa de l'enumeració que comença amb el zero), l'índex de la lletra dintre la fila (és a dir, l'índex de la columna) i un espai. El *ciphertext* resultant és la sortida de la funció, el qual és una *string*.

A la funció de desencriptació el procés és similar: creem la matriu *M* i iterem cada dos caràcters (sense els espais entremig). Si no són dos caràcters (és a dir, un caràcter sol que no és una lletra), aquest s'afegirà directament al *plaintext*. Si, en efecte, la *string* conté dos dígits, s'afegirà al *plaintext* l'element de la matriu m_{ij} on *i* és el primer dígit i *j* el segon. Finalment, la funció retorna la cadena de caràcters resultant excepte l'últim, ja que aquest és un espai de més que hem afegit automàticament mentre iteràvem.

2.4 Xifratge Afí

El xifratge afí és una generalització del més conegut xifratge de Cèsar. És un xifratge de substitució mono-alfabètica, és a dir, cada lletra del *paintext* se substitueix per una altra única i, per tant, les funcions d'encriptació i desencriptació són bijectives.

El xifratge consisteix a elegir dues variables a i b les quals seran el nostre *keyword.* Com en el xifratge de Cèsar, aquest utilitza l'aritmètica modular per a poder encriptar totes les lletres. Hi ha només tres requisits a l'hora d'escollir a i b:

1. $1 \leq a < 26$
2. $MCD(a, 26) = 1 \Leftrightarrow a$ i 26 han de ser coprimers entre si
3. $1 \leq b \leq 26$

Com hem fet amb el xifratge de Cèsar i amb el d'Atbash, substituirem cada lletra pel seu índex corresponent a l'alfabet $(A = 0; B = 0; \dots; Z = 25)$. La funció d'encriptació, doncs, és la funció afina

$$f_{a,b}(x) = (ax + b)\ mod\ 26, \qquad x \in [0, 25].$$

Vegem-ho amb un exemple per a entendre-ho millor. Sigui $a = 5$ i $b = 16$,

$$f(x) = (5x + 16)\ mod\ 26$$

Seguidament he creat un full d'Excel on l'usuari pot escollir qualssevol valors per a i b. El programari encriptarà cada lletra del *plaintext* amb el xifratge afí:

B5 =RESIDUO(B2*C4+E4; 26)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
2		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
3		A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
4		a	5	b	16																						
5		16	21	0	5	10	15	20	25	4	9	14	19	24	3	8	13	18	23	2	7	12	17	22	1	6	11
6		Q	V	A	F	K	P	U	Z	E	J	O	T	Y	D	I	N	S	X	C	H	M	R	W	B	G	L

Taula 7 – Full d'Excel del xifratge afí

A continuació podem veure què passa quan escollim un valor a que no sigui coprimer amb 26. Si a és coprimer amb un nombre m, el màxim comú divisor d'aquests dos ha de ser igual a 1. És a dir, $a \notin \{2, 4, 6, 8, 10, 12, 13, 14, 16, 18, 20, 22, 24, 26\}$ i $1 \leq a < 26$.

Sigui $a = 13$ i $b = 3$,

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
a	13	b	3																						
3	16	3	16	3	16	3	16	3	16	3	16	3	16	3	16	3	16	3	16	3	16	3	16	3	16
D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q	D	Q

Taula 8 – Full d'Excel del xifratge afí on el paràmetre a no compleix els requisits

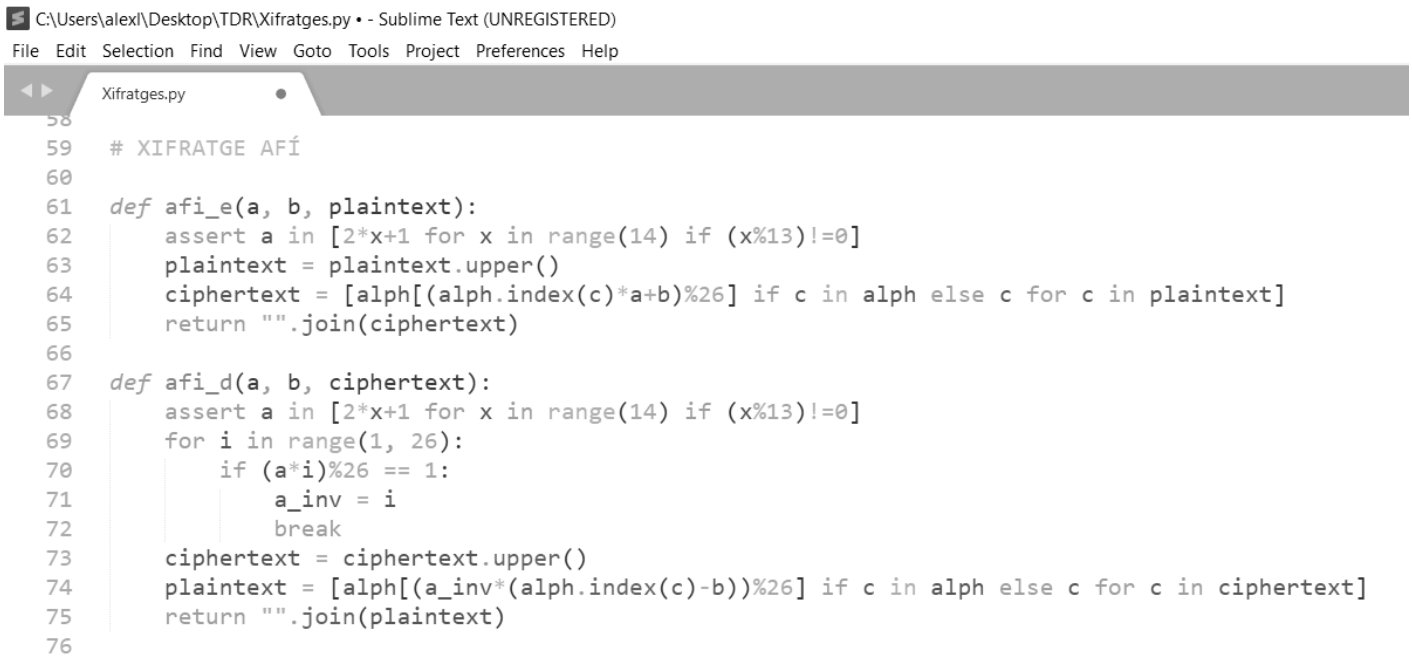
Veiem que cada lletra se substitueix per un 16 (lletra Q) o un 3 (lletra B), per tant, les nostres prediccions són certes: $1 \leq a < 26$ i $MCD(a, 26) = 1$. Tenint en compte aquestes dues condicions, $a \in \{1, 3, 5, 7, 9, 11, 15, 17, 19, 21, 23, 25\}$.

La funció de desencriptació (anomenada g), però, presenta una complicació. Si calculem la funció inversa de la d'encriptació sense tenir cura que estem operant amb aritmètica modular, deduiríem que:

$$g_{a,b}(x) = a^{-1}(x - b)\ mod\ 26.$$

El problema apareix aquí: a^{-1} no pot ser un número invers com $\frac{1}{5}$ perquè estem treballant sempre amb nombres naturals i una fracció irreductible no ho és. Com a conseqüència, hem de definir un nou terme: l'invers modular.

L'invers modular és aquell que compleix que $a \cdot a^{-1} = 1\ (mod\ m)$. Existeix un mètode per a esbrinar-lo d'una manera més formal, però el que farem serà provar valors per a a^{-1} fins que aquesta equació es compleixi. Per tant, a^{-1} no té res a veure amb les fraccions i les divisions convencionals.




```
59  # XIFRATGE AFÍ
60
61  def afi_e(a, b, plaintext):
62      assert a in [2*x+1 for x in range(14) if (x%13)!=0]
63      plaintext = plaintext.upper()
64      ciphertext = [alph[(alph.index(c)*a+b)%26] if c in alph else c for c in plaintext]
65      return "".join(ciphertext)
66
67  def afi_d(a, b, ciphertext):
68      assert a in [2*x+1 for x in range(14) if (x%13)!=0]
69      for i in range(1, 26):
70          if (a*i)%26 == 1:
71              a_inv = i
72              break
73      ciphertext = ciphertext.upper()
74      plaintext = [alph[(a_inv*(alph.index(c)-b))%26] if c in alph else c for c in ciphertext]
75      return "".join(plaintext)
76
```

Fragment de codi 4 – El xifratge afí

He definit la funció d'encriptació *afi_e* i la de desencriptació *afi_d*. Les dues tenen com a entrada els paràmetres a i b discutits anteriorment, a més del missatge a encriptar o encriptat respectivament.

En la línia 62, si a no està dintre dels valors que pot prendre discutits en la pàgina anterior, el codi s'aturarà immediatament i retornarà un error de tipus *AssertionError.* A continuació, canviem qualsevol lletra minúscula que aparegui en el *plaintext* a majúscula. En la línia 64, fem ús d'una comprensió de llista similar a la emprada pel xifratge de Cèsar.

Dintre de la funció de desencriptació, després d'assegurar-nos que hem pres un valor permès per a la variable a, busquem l'invers modular d'a. Per cada valor que pot prendre l'invers modular (entre 1 i 25), si $a \cdot a^{-1} = 1\ (mod\ 26)$, aquest valor quedarà guardat dintre la variable *a_inv* i la declaració *break* acabarà immediatament amb la cerca. Les següents línies de codi són ben similars a les de la funció d'encriptació, amb l'única diferència a quina operació dur a terme en la comprensió de llista (línia 74).

3 La criptografia medieval

A mesura que les societats van desenvolupant-se, adquirint nous coneixements i compartint-los amb altres civilitzacions, la criptografia cada cop va prenent un rol més important. Una de les regions que més innovacions aportà, no només a la criptografia però també a la filosofia, ciències i arts, fou el món islàmic durant l'edat d'or islàmica.

En el segle IX, un matemàtic àrab anomenat Al-Kindí descobrí una tècnica per a desxifrar sense clau qualsevol xifratge que es basés en la substitució mono-alfabètica. Aquí és, doncs, on va néixer el que anomenem avui dia criptoanàlisi.

La criptoanàlisi és l'estudi del conjunt de tècniques emprades per a intentar desxifrar un missatge encriptat sense saber-ne la clau. Si l'*Alice* i en *Bob* utilitzen un algorisme o protocol criptogràfic no molt segur (com tots els que hem vist fins ara), l'*Eve* podria desxifrar-lo ràpidament amb l'ajuda d'aquestes tècniques. La criptoanàlisi, doncs, juga un rol molt important per a la intel·ligència militar d'avui dia.

A Europa, la criptografia també començava a guanyar pes i importància. Durant el Renaixement italià noves tècniques criptogràfiques van aparèixer en diversos punts de la Itàlia moderna. Investigacions recents de la Universitat d'Arizona (EUA) suggereixen que el manuscrit de Voynich fou escrit durant el Renaixement italià.

El manuscrit de Voynich és un manuscrit de 240 pàgines d'autoria anònima, alfabet no identificat i idioma desconegut (probablement inventat), del qual no s'ha pogut extreure cap informació amb èxit. Roman un dels misteris sense resoldre més intrigants de la història de la criptografia.

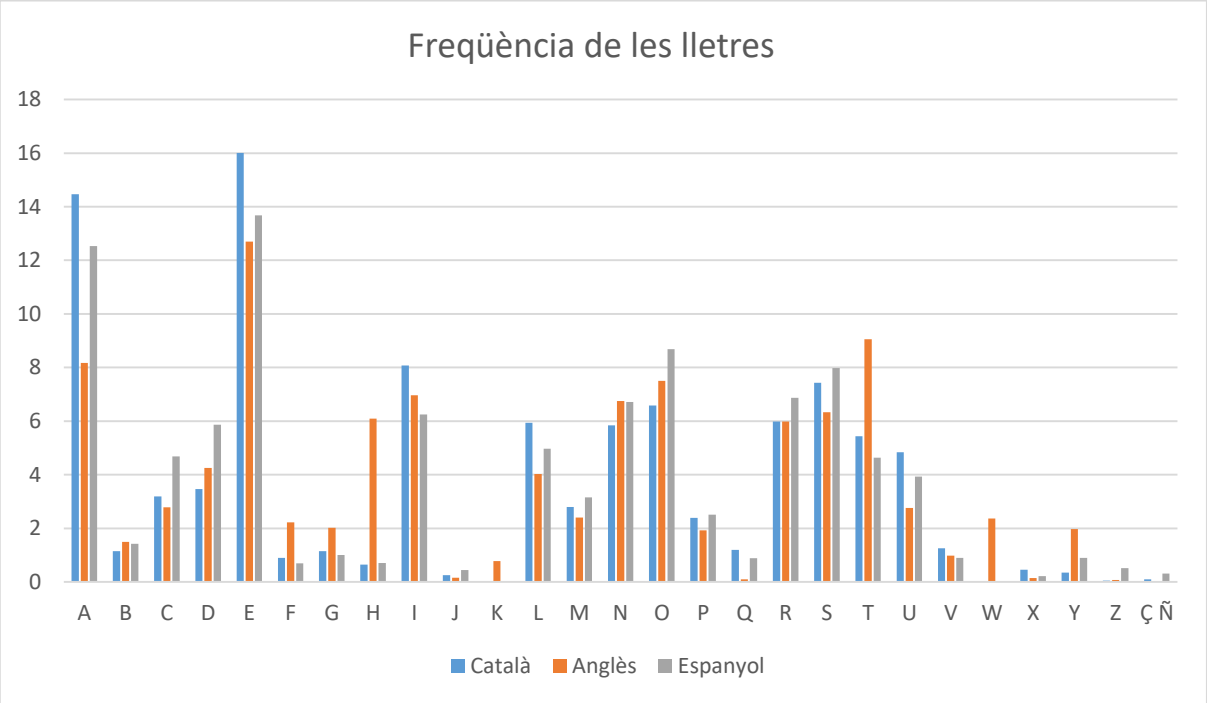
Figura VI – El manuscrit de Voynich – Beinecke Rare Book & Manuscript Library, Yale University

3.1 Anàlisi de les freqüències

Al segle IX, Al-Kindí descobrí un mètode per a desxifrar tots els xifratges que utilitzessin la substitució mono-alfabètica: l'anàlisi de les freqüències. La primera explicació d'aquest descobriment es trobà en un llibre seu anomenat *El Manuscrit per al Desxifratge de Missatges Criptogràfics.*

L'anàlisi de les freqüències es basa en l'empremta que tots els idiomes que utilitzen alfabet deixen. Per a cada idioma, hi ha una sèrie de lletres i combinacions de lletres que s'empren més sovint que d'altres. Per exemple, en català, la lletra E apareix molt més sovint que no pas la lletra Z.

La següent gràfica mostra el percentatge d'ús de lletres en català[3], anglès[4] i espanyol[5]:



Gràfic 1 – Distribució de la freqüència de les lletres en català, anglès i espanyol

Com podem veure, la lletra més utilitzada en els tres idiomes és la lletra E.

[3] *Enciclopèdia de la Llengua Catalana.* Barcelona: Edicions 62, 2001.
[4] *Oxford Dictionary.* Oxford University Press. 29 de desembre de 2012
[5] Fletcher Pratt, Secret and Urgent: the Story of Codes and Ciphers Blue Ribbon Books, 1939.

Com que en una substitució mono-alfabètica cada lletra se substitueix per una i només una altra, les freqüències de les lletres encara són visibles i fàcils d'analitzar. Si descobrim un patró similar al de l'idioma emprat, podrem concloure amb bastant certesa que hem pogut desxifrar un caràcter. Això sí: com més llarg el missatge és, més s'assemblarà a la freqüència de les lletres en l'idioma emprat.

En el següent exemple he encriptat un missatge en català (sense accents) amb el xifratge afí. El nostre objectiu serà desxifrar el *ciphertext* sense saber els valors de a i b que el xifratge afí requereix.

Ciphertext: "XOXQ WTQ WQQWJQ REAUHQ HWYZWH TTYEJWQ Y YKEUTQ WH PYKHYXUX Y WH PJWXQ. QOH POXUXQ PW JUO Y PW IOHQIYWHIYU, Y RUH PW IOAVOJXUJ-QW DJUXWJHUTAWHX WTQ EHQ UAB WTQ UTXJWQ."

Comencem analitzant el *ciphertext* i observant si hi ha patrons de lletres que s'assemblin als del català. Per exemple, veiem que la lletra "Y" s'empra molt sovint entre paraules, cosa que suggereix que la lletra "Y" és, en efecte, la substitució de la lletra "I".

Si comptem les lletres, veurem que les que més s'empren són la "W" amb 19 ocurrències, la "Q" amb 17, la "H" amb 13, la "U" amb 12 i la "X" i la "Y" amb 11 ocurrències. Podem assegurar, doncs, que la lletra "E" s'encripta per la "W", ja que la "E" és la lletra més emprada amb diferència i la "W" apareix en la majoria de mots de dues lletres (digrames). Similarment, la lletra E s'utilitza en les preposicions "DE" i "EN", digrames que apareixen amb bastant freqüència en català. Podem concloure, doncs, que "EN" s'encripta per "WH" i "DE" per "PW".

Dit això, ja coneixem dues lletres més: la "D" i la "N". Anem a veure com queda el nostre *ciphertext* si substituïm les lletres que ja sabem en minúscules:

"XOXQ eTQ eQQeJQ REAUnQ neiZen TTiEJeQ i iKEUTQ en diKniXUX i en dJeXQ. QOn dOXUXQ de JUO i de IOnQIienIiU, i RUn de IOAVOJXUJ-Qe DJUXeJnUTAenX eTQ EnQ UAB eTQ UTXJeQ.".

Continuem deduint altres lletres tenint en compte la seva freqüència al *ciphertext*. Per exemple, és probable que la lletra "Q" del *ciphertext* sigui la "S" en el missatge, ja que aquesta apareix bastant i s'empra molt sovint a final de mot. A partir d'aquí, podem també deduir que la "T" és una "L", ja que en tres ocasions apareix "eTs" seguit d'un mot acabat en "s" (probablement plural). Això ens fa pensar que "WTQ" significa "els".

"XOXs els esseJs REAUns neiZen lliEJes i iKEUls en diKniXUX i en dJeXs. sOn dOXUXs de JUO i de IOnsIienIiU, i RUn de IOAVOJXUJ-se DJUXeJnUlAenX els Ens UAB els UlXJes.".

Estem fent progrés. A partir d'aquí podem esbrinar altres caràcters mitjançant mots que puguem deduir sense cap dificultat:

- "J" és una "R" perquè el mot "esseJs" ha de ser per força "éssers".
- "Z" és una "X" perquè el mot "neiZen" ha de ser per força "neixen".
- "O" és una "O" perquè el mot "sOn" ha de ser per força "són".

"XoXs els essers REAUns neixen lliEres i iKEUls en diKniXUX i en dreXs. son doXUXs de rUo i de IonsIienIiU, i RUn de IoAVorXUr-se DrUXernUlAenX els Ens UAB els UlXres.".

- "E" és una "U" perquè el mot "lliEres" ha de ser per força "lliures".
- "X" és una "T" perquè el mot "dreXs" ha de ser per força "drets".
- "U" és una "A" perquè el mot "rUo" ha de ser per força "raó".

Com més avancem, més a prop estem de desxifrar el *ciphertext*. Si seguíssim així, arribaríem a un punt on el missatge podria ser llegit sense cap mena de dificultat:

"Tots els éssers humans neixen lliures i iguals en dignitat i en drets. Són dotats de raó i de consciència, i han de comportar-se fraternalment els uns amb els altres."

– Article 1 de la Declaració Universal dels Drets Humans.

3.2 Xifratge de Vigenère

El xifratge de Vigenère, atribuït erròniament al diplomàtic francès Blaise de Vigenère durant el segle IX, fou descrit per primera vegada per un criptòleg italià anomenat Giovan Battista Bellaso l'any 1553 en el seu llibre *La cifra del. Sig. Giovan Battista Bellaso.*

(Rodriguez-Clark 2012)

El xifratge es basa en la substitució polialfabètica: substituïm utilitzant dos alfabets de substitució. El xifratge de Vigenère i la màquina Enigma són exemples de xifratges de substitució polialfabètica. El de Vigenère no és tan senzill de desxifrar sense clau com els mono-alfabètics, ja que un caràcter es pot substituir per qualsevol altre. Tot i que va rebre el sobrenom de "le chiffre indéchiffrable" (francès: "el xifratge indesxifrable"), existeixen mètodes com l'eliminació de Kasiski que el fan vulnerable.

El xifratge de Vigenère utilitza el que anomenem com a "Tabula Recta":

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

Taula 9 – Tabula recta del xifratge de Vigenère

Abans que res, l'*Alice* i en *Bob* han de triar la paraula clau o *keyword.* Aquesta pot ser de qualsevol llargada. Posteriorment, l'*Alice* escriu el missatge i, a sota, el *keyword* repetidament fins que cada lletra del *plaintext* estigui emparellada amb una del *keyword.* En l'exemple següent, el missatge és "Xifratge de Vigenère" i la clau és "Poli":

X	I	F	R	A	T	G	E	D	E	V	I	G	E	N	E	R	E
P	O	L	I	P	O	L	I	P	O	L	I	P	O	L	I	P	O

Taula 10 – Plaintext i Keyword del xifratge de Vigenère emparellats

El següent pas és utilitzar la Tabula Recta. Per a cada lletra del missatge i del *keyword*, la lletra encriptada serà la que es trobi en les coordenades corresponents a la Tabula. Si, per exemple, la lletra a encriptar és la "L" i la del *keyword* és la "C", la lletra del *ciphertext* seria la "N". Aquest esquema simbolitza aquest darrer exemple:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**
A	A	B	C	D	E	F	G	H	I	J	K	L	M
B	B	C	D	E	F	G	H	I	J	K	L	M	N
C	C	D	E	F	G	H	I	J	K	L	M	N	O
D	D	E	F	G	H	I	J	K	L	M	N	O	P

Taula 11 – Encriptació d'una lletra amb Vigenère

I, tornant a l'exemple anterior, obtenim el *ciphertext*:

X	I	F	R	A	T	G	E	D	E	V	I	G	E	N	E	R	E
P	O	L	I	P	O	L	I	P	O	L	I	P	O	L	I	P	O
M	W	Q	Z	P	H	R	M	S	S	G	Q	V	S	Y	M	G	S

Taula 12 – Encriptació final amb el xifratge de Vigenère

Com acabem de veure, en el xifratge de Vigenère hi intervenen dues cadenes de caràcters i, d'aquestes dues, se n'extreu una altra. A més a més, si investiguem més a fons la Tabula Recta, ens adonaríem que és el mateix encriptar una lletra amb una del *keyword* que a la inversa. És a dir, sigui la funció $E(p,k)$ la que encripta una lletra del *plaintext* p amb una del *keyword* k mitjançant Vigenère, $E(p,k) = E(k,p)$.

Sembla impossible definir d'una manera més formal aquest xifratge, però no ho és.

Com hem fet amb uns quants xifratges en el passat, substituïm cada lletra per la seva posició a l'alfabet ($A = 0; B = 1; \dots; Z = 25$).

Vegem un tros de la Tabula Recta un cop modificat l'alfabet:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	0
2	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	0	1
3	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	0	1	2
4	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	0	1	2	3
5	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	0	1	2	3	4
6	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	0	1	2	3	4	5
7	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	0	1	2	3	4	5	6
8	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	0	1	2	3	4	5	6	7

Taula 13 – Tabula Recta numèrica del xifratge de Vigenère

Observem que, a excepció dels nombres ombrejats en verd, $E(p,k) = p + k$.

El problema apareix quan avancem més i la suma de p i k és major o igual a 26. En aquest cas, tornem a contar des del zero i, per tant, comencem un nou cicle. Per exemple, si $p = 20$ i $k = 6$, com que $p + k = 26$, $E(p,k) = 0$. Això significa que haurem d'utilitzar un cop més l'aritmètica modular. Només d'aquesta manera podem definir la funció:

$$E(p,k) = (p + k)\ mod\ 26$$

Per a desencriptar, $D(c,k) = (c - k)\ mod\ 26$ (on c és l'índex de la lletra del *ciphertext* a l'alfabet). Gràficament, per a desencriptar, cerquem la lletra del *ciphertext* en la fila de la lletra del nostre *keyword* i, un cop l'hàgim trobat, la lletra del *plaintext* és la coordenada horitzontal de la posició descrita anteriorment.

Evidentment, per a codificar el xifratge utilitzaré la definició més formal, ja que és més senzilla i no requereix la creació de tota la Tabula Recta.

Abans, però, he volgut fer una altra petita pràctica. Veurem com, utilitzant el xifratge de Vigenère, l'anàlisi de les freqüències no serveix per a intentar desxifrar un missatge sense clau comparant-lo amb un xifratge de substitució mono-alfabètica com l'afí.

Com a *plaintext* he pres la síntesi en català d'aquest Treball de Recerca.

- Claus del xifratge afí: $a = 17; b = 9$
- Clau del xifratge Vigenère: "VIGENERE"

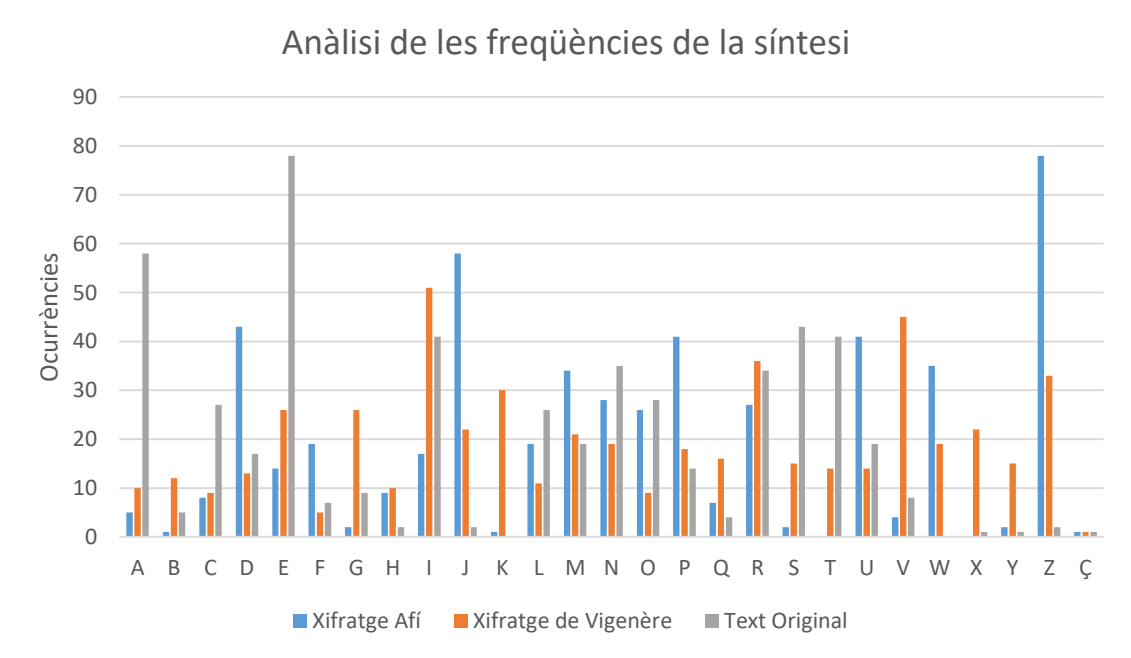
Ciphertext utilitzant el xifratge afí:

"ZW ZO IPJ I'JCLP, VLJW OJ DNRPZUJU JCJWÇJ RJIJ RNE FZD RJE J LW FNW IPHPUJOPUSJU P IZEZW FZD IZ OZD WNCZD UZRWNONHPZD, OJ RMPEUNHMJQPJ P ZO DZL QLULM DNW LW ZOZFZWU ZDDZWRPJO EZM OJ WNDUMJ DZHLMZUJU P EMPCJIZDJ. ZO FZL UMZAJOO IZ MZRZMRJ ZD RZWUMJ ZW JVLZDU JMU; DNAMZUNU ZW PWCZDUPHJM ZOD FZUNIZD RMPEUNHMJQPRD VLZ OZD RPCPOPUSJRPNWD JWUPHLZD ZFEMJCZW. CPJUGJMZF ZW ZO EJDDJU, EMZDZWU P QLULM, ZWD DPULJMZF ZW ZOD RNWUZKUD YPDUNMPRD P IZQPWPMZF FJUZFJUPRJFZWU JVLZDUD FZUNIZD. ZOD RNFEJMJMZF P, EZM J JRJAJM I'ZWUZWIMZ ZOD JOHNMPDFZD, ZOD RNIPQPRJMZF ZW EBUYNW. ZO FZL NAGZRUPL ZD, INWRD, INWJM LWJ WNRPN AJDPRJ IZ OJ RMPEUNHMJQPJ JO OZRUNM."

Ciphertext utilitzant el xifratge Vigenère:

"ZV KP QMR H'VDAM, DYRR GI YSPMVXVB GZNRÇR GVLG GBT DIN KGT N YE QJV JMTMKEGQZDNX Z HZXKR ZIJ HZ TKW ASMIN BKGASCSBQKW, YE TVDXZSTVRJDI O IY WVY ACZYE WFR PV KPRQVRO MYWRRTMVT VIE PR RJAZVN WVKPZKXNX Z TMQBEQIJE. ZT SIH XIIWIRP QI IIXMXGN IJ GZVZVN IE ELCKWG EIX; NWHVRXFX ZV ORIIJXDOGV RPJ QZBUHRW TVDXZSTVRJDKY UHI CIN KOZVPZXUIIMBRJ EIBOKHIJ IHXXEIIE. ZDIZNNVVQ ZV KP CEJWVB, VVRWVRO Q LYGYI, IIA YMGYRVZU KR RPJ GJVZIKXJ LDAZSEMTW D LKJVRZVZU SEGIDEOQIEZIEX VYAIFXJ QZBUHRW. VPN KUQCEIEMMS M, CII E VKGFNV U'IIBKRQVV IGA GPTSIMNUKW, RPJ GJLOJVGRVZU KR CCKLJV. KP ZIL SWRKGGML IN, LURPW, USIIX YAE ESXQU FNWZGV LK PN GIMKBUKEEWMV IR PRGKSM."

Anàlisi de les freqüències amb el xifratge afí i el xifratge Vigenère:



Gràfic 2 – Anàlisi de les freqüències de la síntesi amb el xifratge de Vigenère, l'afí i el plaintext original

Podem observar com el xifratge de Vigenère no conserva cap semblança amb el text original mentre que en el xifratge afí l'anàlisi de les freqüències és el mateix. Per exemple, la lletra "E" en el text original correspon a la lletra "Z" en el xifratge afí, però a simple vista no podríem dir el mateix amb un xifratge polialfabètic com el de Vigenère.

```
163
164 # XIFRATGE DE VIGENÈRE
165
166 def mapped_key(modal, key):
167     assert all(i in alph for i in key)
168     modal = modal.upper()
169     map_key = []
170     i = 0
171     for c in modal:
172         if c not in alph:
173             map_key.append(c)
174         else:
175             map_key.append(key[i%len(key)])
176             i += 1
177     return map_key
178
179 def vigenere_e(plaintext, key):
180     plaintext = plaintext.upper()
181     key = mapped_key(plaintext, key.upper())
182     ciphertext = [alph[(alph.index(a) + alph.index(b))%26] if a in alph else a for a, b in zip(plaintext, key)]
183     return "".join(ciphertext)
184
185 def vigenere_d(ciphertext, key):
186     ciphertext = ciphertext.upper()
187     key = mapped_key(ciphertext, key.upper())
188     plaintext = [alph[(alph.index(a)-alph.index(b))%26] if a in alph else a for a, b in zip(ciphertext, key)]
189     return "".join(plaintext)
190
```

Fragment de codi 5 – El xifratge de Vigenère

He decidit dividir la programació del xifratge de Vigenère en tres funcions. La primera, anomenada *mapped_key*, farà que la clau tingui el mateix nombre de lletres que una *string* la qual he anomenat *modal*. La segona i la tercera, anomenades *vigenere_e* i *vigenere_d*, són les encarregades d'encriptar un missatge i desencriptar-lo respectivament. Com podem veure en les línies 115 i 121 de codi, aquestes dues canviaran la clau *key* per la clau modificada amb la funció *mapped_key*, on la variable *modal* és el nostre *plaintext* o *ciphertext*.

No aniré amb més detall sobre la programació, ja que l'estructura és molt similar a les dels xifratges que ja hem comentat. El que sí que farem serà veure com podem fer servir el programari. Sigui el missatge a encriptar "XIFRATGEDEVIGENERE" amb la clau "POLI", declarem la funció amb les variables que hem pres, utilitzem la funció *print* perquè la consola "imprimeixi" el valor i executem el codi:

```
190
191 print(vigenere_e("XIFRATGEDEVIGENERE", "POLIS"))
192
```

```
MWQZSIUPLWKWRMFTFP
[Finished in 1.4s]
```

Fragment de codi 6 – Exemple del xifratge de Vigenère

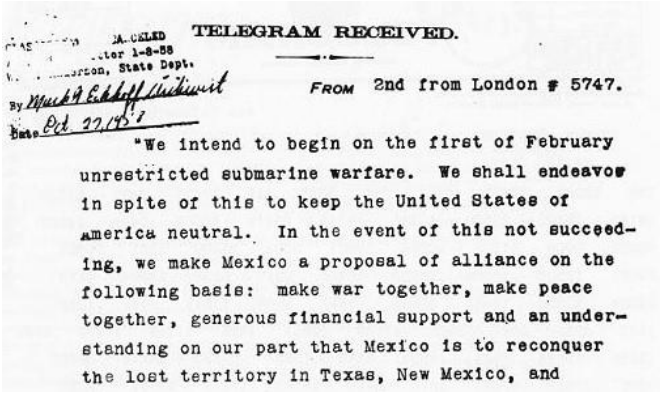
4 La criptografia dels segles XIX i XX

No va ser fins a principis del segle XX que la criptografia començà a guanyar un pes important en la societat. Fins llavors, la majoria d'avenços criptogràfics van ser noves tècniques per a la criptoanàlisi. Per exemple, en l'any 1863, l'alemany, criptògraf i arqueòleg Friedrich Kasiski descobrí un mètode per a atacar els xifratges de substitució polialfabètica: l'anomenat avui dia "Eliminació de Kasiski"[6].

Durant la Primera Guerra mundial, gràcies a la intel·ligència britànica, es detectà un telegrama anomenat telegrama de Zimmermann enviat pel Ministeri d'Assumptes Exteriors alemany que oferia una aliança militar entre Mèxic i Alemanya. Com a recompensa, després de la guerra, Mèxic obtindria els estats de Texas, Nou Mèxic i Arizona. Després del seu desxifratge, desgraciadament pels alemanys, els Estats Units declararen la guerra contra Alemanya.

Un dels avenços més destacables fou l'invent del bloc d'un sol ús (anglès: "one time pad") o "OTP" l'any 1917 per l'anglès Gilbert Vernam. Aquest xifratge resulta ser pràcticament impossible de desxifrar sense la mateixa clau. Tot i això, l'*Alice* i en *Bob* han de compartir una clau més gran o igual de llargada que el missatge en si.

A mitjans del segle XX es desenvoluparen màquines de xifratge mecàniques, les quals mitjançant un procés mecànic encriptaven un missatge. Durant la Segona Guerra mundial, els alemanys inventaren una màquina de xifratge electromecànica amb cinc rotors anomenada màquina Enigma. L'aparell, però, resultà ser vulnerable als atacs criptoanalítics dels francesos, polonesos i britànics.



TELEGRAM RECEIVED.

FROM 2nd from London # 5747.

"We intend to begin on the first of February unrestricted submarine warfare. We shall endeavor in spite of this to keep the United States of America neutral. In the event of this not succeeding, we make Mexico a proposal of alliance on the following basis: make war together, make peace together, generous financial support and an understanding on our part that Mexico is to reconquer the lost territory in Texas, New Mexico, and

Figura VII – El telegrama de Zimmermann – National Archives

[6] Si voleu més informació sobre l'eliminació de Kasiski, visiteu https://crypto.interactive-maths.com/kasiski-analysis-breaking-the-code.html

4.1 Xifratge de Playfair

El xifratge de Playfair fou desenvolupat l'any 1854 per l'inventor britànic Charles Wheatstone (1802-1875), més conegut pel circuit elèctric per a mesurar resistències: el pont de Wheatstone. Playfair és un xifratge de blocs que utilitza el quadrat de Polibi com a suport per a encriptar un missatge.

Es comença creant un quadrat de Polibi amb una clau simètrica que l'*Alice* i en *Bob* hagin decidit amb anterioritat mitjançant el procés descrit en la pàgina vint-i-tres. Per exemple, si la clau és "CRYPTOGRAPHY", el quadrat resultant serà el següent:

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

Taula 14 – El quadrat de Polibi amb una clau

Seguidament, amb el missatge, formem parelles de dues lletres començant pel principi. Si hi ha una parella on les dues lletres coincideixen, hi afegim una lletra "X" addicional. Si ens falta una lletra al final per a formar una parella, afegim una "X" al final.

Per exemple, si el missatge és "HIDETHEGOLDINTHETREESTUMPS", el missatge preparat per a l'encriptació serà "HI DE TH EG OL DI NT HE TR EX ES TU MP SX". Podem veure com entra en joc la primera condició (en color groc) i la segona (en blau).

Després d'aquesta modificació, el xifratge encripta cada parella de lletres per separat i en retorna una nova parella. Segons com les lletres se situïn en el quadrat, diferenciarem entre parelles de lletres que formin un rectangle, que estiguin en la mateixa columna o que se situïn en la mateixa fila.

Comencem comentant el cas més comú: si la parella forma un rectangle. Per exemple, considerem la parella "NT".

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

Taula 15 – Rectangle en el quadrat format per N i T

Observem com la parella de lletres proposada forma un rectangle on "N" i "T" són dos vèrtexs oposats. Com a conseqüència, apareixen dos nous vèrtexs, "Y" i "S". Aquesta és la nova parella encriptada. Comencem amb la primera lletra de la parella ("N") i ens movem en direcció horitzontal fins a trobar el nou vèrtex ("S"). Si fem el mateix amb la segona lletra, obtenim la parella encriptada "SY".

Ara bé, què fem si les dues lletres es troben en la mateixa columna com, per exemple, la parella "OL"? En aquest cas, hem de baixar una posició cada lletra en la columna respectivament:

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

Taula 16 – Columna en el quadrat formada per O i L

La "O" s'encripta per la lletra que té baix seu ("D") i la "L" per la "U". Si haguéssim d'encriptar la parella "OU", com que la "U" se situa al final de la columna i no té un element baix seu, aquesta s'encriptaria pel primer element de la columna (la "C").

Si resulta que les dues lletres es troben en la mateixa fila, canviem cada lletra de la parella per la qual es troba adjacent a aquesta per la dreta. Si no en té (com és el cas de la parella "TR", canviem l'última lletra de la fila per la primera.

El *ciphertext* resultant és `"IQEFPBMEDUEKSYGICYIvKMCZQRQz"`, on les minúscules simbolitzen les "X" suplementàries que hem hagut d'afegir.

```
# XIFRATGE DE PLAYFAIR

def modif_miss(text): ...

# Funció que afegeix X's quan calgui

def playfair_e(plaintext, keyword):
    M = matriu(keyword)  # Reutilitzant la funció de Polibi
    modf = modif_miss(plaintext)
    modf = [(modf[i:i+2]) for i in range(0, len(modf), 2)]
    ciphertext = ""
    for p in modf:
        for r in M:
            if p[0] in r and p[1] in r: # Mateixa fila
                e = [r[(r.index(p[0])+1)%5], r[(r.index(p[1])+1)%5]]
                break
            elif p[0] in r:
                i = r.index(p[0])
                for s in M:             # Mateixa columna
                    if p[1] in s and s.index(p[1]) == r.index(p[0]):
                        e = [M[(M.index(r)+1)%5][i], M[(M.index(s)+1)%5][i]]
                        break
                    elif p[1] in s:     # Diferent Fila i Columna
                        e = [r[s.index(p[1])], s[r.index(p[0])]]
                        break
        ciphertext += "".join(e)
    return ciphertext

def playfair_d(ciphertext, keyword): ...

print("Plaintext :", playfair_e("HIDETHEGOLDINTHETREESTUMPS", "CRYPTOGRAPHY"))
print("Ciphertext:",playfair_d("IQEFPBMEDUEKSYGICYIVKMCZQRQZ", "CRYPTOGRAPHY"))
```

```
Plaintext : IQEFPBMEDUEKSYGICYIVKMCZQRQZ
Ciphertext: HIDETHEGOLDINTHETREXESTUMPSX
[Finished in 0.8s]
```

Fragment de codi 7 – El xifratge de Playfair

Per motius d'espai, he hagut d'amagar la funció de desencriptació, que és gairebé idèntica a la d'encriptació, i la funció *modif_miss*, que afegeix una "X" quan calgui en el *plaintext*. En el codi presentat en el llapis de memòria USB podreu veure totes les funcions i encriptar els vostres missatges amb qualsevol xifratge comentat.

Com podem veure, el codi és bastant més llarg que no pas el d'altres xifratges que hem vist. Això és degut al fet que he hagut de recórrer a un codi més robust per a poder iterar dins del quadrat de Polibi i dins de cada fila. A més a més, he reutilitzat la funció (*matriu*) que hem definit en l'apartat del quadrat de Polibi.

Finalment, encriptem el missatge que hem proposat al principi i, per a acabar de comprovar que no hi hagi cap error, desencriptem el *ciphertext*. Els arguments de la funció *print* i el temps que ha trigat la consola apareixen en la consola.

4.2 One Time Pad

Inventat per Gilbert Vernam l'any 1917, el bloc d'un sol ús o "One Time Pad" és un dels xifratges més segurs. És molt semblant al xifratge de Cèsar o al xifratge de Vigenère, però la clau que s'empra ha de ser igual o més llarga que el missatge a encriptar.

L'*Alice* genera una llista de nombres (entre el 0 i el 25) completament **aleatòria**. Llavors, comparteix aquesta llista amb en *Bob*. Quan l'*Alice* vulgui enviar un missatge a en *Bob*, el que ha de fer és encriptar cada lletra amb el decalatge de cada nombre en la llista. Posem per cas que la clau compartida sigui "13 5 9 22 21 1 29 4 22 0 8" i el missatge a encriptar sigui "ONE TIME PAD":

O	N	E	T	I	M	E	P	A	D
13	5	9	22	21	1	29	4	22	0
B	S	N	P	D	N	H	T	W	D

Taula 17 – El funcionament del bloc d'un sol ús

Abans que res, l'*Alice* transcriu cada lletra del missatge en l'índex que ocupa a l'alfabet. Després, suma aquest índex amb el decalatge corresponent i aplica l'aritmètica modular amb un mòdul $m = 26$. Finalment, converteix aquest nombre natural en lletra.

Si estiguéssim fent servir el xifratge de Cèsar, només faríem servir un mateix decalatge per a totes les lletres del *plaintext*. A més a més, com que els decalatges són aleatoris, qualsevol lletra té les mateixes possibilitats a sortir que qualsevol altra. És a dir, només un atac per força bruta podria desxifrar el missatge sense clau.

Amb el xifratge de Cèsar, un atac per força bruta és òptim, ja que només hem de provar 26 possibles claus. Amb el bloc d'un sol ús, com que cada lletra s'encripta amb un decalatge aleatori, haurem de provar 26^n (on n és la llargada del missatge) posicions per estar un cent per cent segurs que hem provat totes les possibles claus.

En el missatge anterior, haurem de provar $26^{10} = 1.41167 \cdot 10^{14}$ claus (més de 141 bilions de possibles claus).

Encara que teòricament aquest mètode d'encriptació té una seguretat perfecta, en la pràctica sorgeixen dos problemes que hem d'atacar:

1. Com s'intercanvien les claus l'*Alice* i en *Bob* de manera segura?
2. Com podem generar una llista de nombres completament aleatòria?

Amb relació a la primera qüestió, existeixen mètodes avui dia per a poder intercanviar claus de manera segura com l'intercanvi de claus Diffie-Hellman o, fins i tot, una distribució de claus quàntica. Tot i això, la clau encara ha de ser més llarga o igual que el missatge posterior i no pot ser utilitzada més d'una vegada, ja que la clau podria ser vulnerable. Això el fa poc pràctic.

Envers el segon problema, un nombre aleatori no és gens fàcil de generar. Volem un generador de nombres veritablement aleatoris que no siguin pseudoaleatoris, és a dir, que els nombres no estiguin decidits amb anterioritat.

Per exemple, quan tirem un dau, el resultat sembla aleatori, però només ho és a causa de la nostra ignorància, ja que el resultat ja està decidit quan tirem el dau. Tècnicament, si poguéssim determinar tots els factors que entren en joc (força de llançament, distància, rotació, ...), podríem esbrinar quin nombre obtindrem.

```
193  # BLOC D'UN SOL ÚS - OTP
194
195  def arr_dec(llarg):
196      return [random.randint(1, 25) for _ in range(llarg)]
197
198  def otp_e(arr_dec, plaintext):
199      plaintext = plaintext.upper()
200      j = 0
201      ciphertext = ""
202      for i, c in enumerate(plaintext):
203          if c not in alph:
204              ciphertext += c
205              j += 1
206              continue
207          ciphertext += alph[(arr_dec[i-j]+alph.index(c))%26]
208      return "".join(ciphertext)
209
210  def otp_d(arr_dec, ciphertext): ⋯
```

Fragment de codi 8 – El bloc d'un sol ús o One Time Pad

En *Python*, existeix una llibreria anomenada *random* la qual s'encarrega de treballar amb l'atzar. Aquí l'estem utilitzant dintre de la funció *arr_dec*, la qual s'encarrega de generar una llista d'un determinat nombre de nombres aleatoris entre el 0 i el 25.

```
166
167   print(arr_dec(9))
```

```
[25, 3, 22, 19, 3, 21, 3, 24, 17]
[Finished in 0.1s]
```

Fragment de codi 9 – Funció arr_dec

En les dues altres funcions (d'encriptació i de desencriptació), com que requereixen una iteració especial, no he emprat una comprensió de llista. El codi és bastant més llarg comparat amb els altres xifratges que hem vist. Per tant, he decidit amagar la funció de desencriptació. Així i tot, la podeu veure en el codi.

Vegem el mateix exemple que hem posat anteriorment:

```
167   clau = [13, 5, 9, 22, 21, 1, 29, 4, 22, 0, 8]
168   missatge = "ONE TIME PAD"
169
170   ciphertext = otp_e(clau, missatge)
171   print(ciphertext)
172
173   print(missatge == otp_d(clau, ciphertext))
```

```
BSN PDNH TWD
True
[Finished in 0.2s]
```

Fragment de codi 10 – Exemple del bloc d'un sol ús

En les línies 167 i 168 estem guardant el llistat de decalatges dintre la variable *clau* i el missatge `"ONE TIME PAD"` dintre de la variable *missatge*.

En les línies 170 i 171, guardem el *ciphertext* resultant dintre la variable *ciphertext* i ensenyem aquest valor en pantalla (primera línia de la consola).

Llavors, en la línia 173, si el missatge original i el resultant després d'encriptar i desencriptar són idèntics, la consola respondrà amb el booleà `True`. Si tenim un problema amb la programació, la consola respondrà amb un `False`.

4.3 Màquina Enigma

La màquina enigma és el nom d'una màquina d'encriptació desenvolupada pels alemanys durant la Segona Guerra mundial per a transmetre missatges de la manera més pràctica i segura possible. Els criptoanalistes aliats, així i tot, van aconseguir desxifrar-la i, gràcies a això, es van poder prevenir alguns atacs enemics. En resum, el desxifratge de la màquina Enigma ajudà considerablement les forces aliades vèncer l'exèrcit alemany.

La màquina Enigma és un mecanisme format per rotors que desordena les vint-i-sis lletres de l'alfabet. L'operari comença a escriure el missatge en el teclat mecànic i, per a cada lletra, s'il·lumina la lletra encriptada en un panell. El gir dels rotors canvia les connexions elèctriques entre les tecles i els llums.

Figura VIII – Màquina Enigma – Museu Nacional de la Ciència i la Tecnologia Leonardo da Vinci, Milà

Com podem veure, la màquina Enigma està composta per tres rotors giratoris amb 26 possibles combinacions, un teclat, un panell d'il·luminació i, davant de la màquina, l'operari pot augmentar la seguretat del seu xifratge mitjançant un panell on pot canviar les connexions elèctriques de cada lletra manualment.

La configuració dels rotors amb la pitjada d'una tecla crea un circuit elèctric que il·luminarà la lletra del *ciphertext*. Seguidament, el primer rotor gira una posició. Els altres dos també giraran depenent de la seva posició.

Com que hi ha tres rotors amb vint-i-sis possibles posicions cadascun, podríem calcular el nombre de configuracions que la màquina Enigma té:

$$26^3 = 17576$$

Els tres rotors, però, poden ser intercanviats entre si. A més a més, s'han d'escollir tres dels cinc rotors disponibles en una màquina Enigma. Si ho tenim en compte, tenim cinc opcions per al primer rotor, quatre pel segon i tres pel tercer:

$$5 * 4 * 3 = 60$$

I això fa pujar el nombre de configuracions fins a un màxim de $60 * 17576 = 1054560$ possibles configuracions. Nosaltres, però, no utilitzarem el panel conegut com a *plugboard* que canvia les connexions elèctriques de cada lletra per individual. Si l'utilitzéssim, però, el nombre de possibles configuracions s'alçaria per sobre de les $158 \cdot 10^{18}$ possibles configuracions.

Els rotors en la màquina Enigma actuen com xifratges de substitució mono-alfabètica: cada rotor està definit per un alfabet desordenat. A part dels rotors, existeix un últim rotor anomenat UKW (de l'alemany: "Umkehrwalze") o simplement "reflector" que té la funció de retornar la connexió elèctrica de nou als rotors. El següent diagrama ho representa visualment:

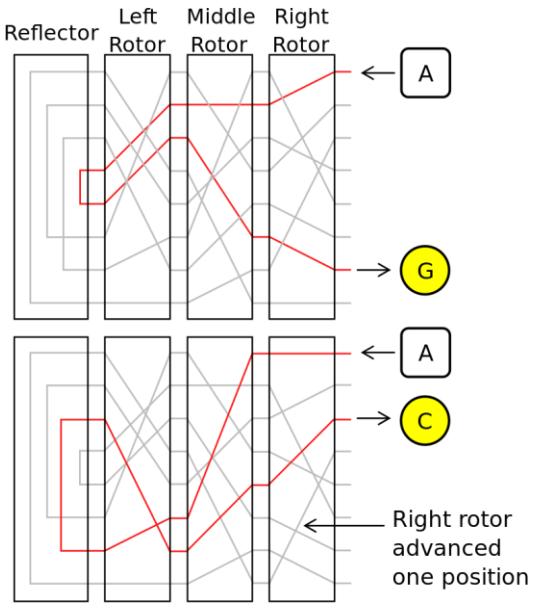


Figura IX – Les connexions elèctriques de la màquina Enigma – Matt Crypto (Usuari Viquipèdia)

Les configuracions elèctriques de cada rotor depenen de la màquina Enigma que estiguem emprant. Abans de la Segona Guerra mundial ja s'havien patentat diverses màquines d'encriptació similars a la que empraren els alemanys durant la guerra. Nosaltres ens centrarem en la "màquina Enigma I" o "Wehrmacht Enigma".

Els rotors, com hem mencionat abans, actuen com a xifratges de substitució mono-alfabètica. A partir de les connexions elèctriques del rotor, l'alfabet es convertia en un altre de molt diferent. El disseny dels rotors depèn[7] de la màquina Enigma. Per al model de la "Wehrmacht Enigma", els rotors i el reflector (rotor IV) són els següents:

I. "EKMFLGDQVZNTOWYHXUSPAIBRCJ"
II. "AJDKSIRUXBLHWTMCQGZNPYFVOE"
III. "BDFHJLCPRTXVZNYEIWGAKMUSQO"
IV. "YRUHQSLDPXNGOKMIEBFZCWVJAT"

Per a entendre-ho, posem un exemple on tots els rotors es troben en les seves posicions inicials i cada rotor ocupa la seva posició en ordre (primer el rotor I, després el II ...). Pitgem en el teclat mecànic la lletra "E":

Rotor III	Rotor II	Rotor I	Reflector	R.inv I	R.inv II	R.inv III
E → J	J → B	B → K	K → N	N → K	K → D	D → B

Taula 18 – El viatge de la senyal elèctrica pels rotors de la màquina Enigma

El primer rotor (rotor III) connecta la lletra "E" amb la "J". Això és així, ja que la lletra "J" ocupa el lloc de la "E" en l'alfabet del rotor (el disseny del primer rotor). Després de "rebotar" en el reflector, el senyal viatja en sentit invers (vegeu en la Figura IX de la pàgina anterior).

Aquest exemple il·lustratiu no és del tot correcte, però, ja que el rotor gira posició abans d'encriptar la lletra pitjada. Quan pitgem una lletra, el primer rotor (el de la dreta) sempre gira una posició. Les vint-i-sis posicions en un rotor estan identificades per una lletra de la "A" a la "Z". Si, per exemple, emprem el rotor II en la posició "D", el nou alfabet de substitució es veuria decalat per 3 lletres (posició de la lletra "D" a l'alfabet).

[7] Llistat amb el disseny dels rotors de les diferents màquines Enigmes: https://en.wikipedia.org/wiki/Enigma_rotor_details

Disseny del rotor II en la posició "D": "KSIRUXBLHWTMCQGZNPYFVOEAJD".

A més a més, cada rotor té una osca que permetrà girar en el pròxim pas el rotor de la seva esquerra si el rotor està en la posició de l'osca. Per a la màquina Enigma I, les posicions "clau" de cada rotor són les següents:

- Rotor I: "Posició Q"
- Rotor II: "Posició E"
- Rotor III: "Posició V"

Posem un exemple:

I	II	III
T	M	T
T	M	U
T	M	V
T	N	W

Taula 19 – Avanç d'un rotor de la màquina Enigma

Observem que el rotor de la dreta, rotor III o també anomenat "rotor ràpid" gira una posició per cada tecla pitjada. Quan el rotor III arriba a la seva posició clau (ombrejada en verd), permet que en el següent pas el rotor de la seva dreta (rotor II) pugui avançar.

A més a més, si el rotor lent (rotor I) avança, aquest forçarà el rotor del mig a avançar també, ja que el segon rotor es troba en la seva posició clau i, com que el segon rotor no avança sempre, aquest permetria que el rotor I avancés vint-i-sis vegades. Vegem aquest esdeveniment anomenat "*Double Stepping*" amb més detall:

I	II	III
S	D	U
S	D	V
S	E	W
T	F	X
T	F	Y
T	F	Z

Taula 20 – Un exemple del "double stepping"

Observem que el rotor ràpid arriba a la posició clau (ombrejada en roig) i, en el següent pas, permetrà que el segon rotor avanci. Quan el segon rotor avança, resulta que ha arribat a la seva posició clau, és a dir, permetrà que el rotor lent avanci una posició. Quan aquest ho fa, però, el rotor II també avança una posició. Si això no passés, el segon rotor seguiria en la posició clau i, fins que el rotor ràpid no arribés a la seva posició clau, el rotor I seguiria avançant.

A causa de l'arquitectura de la màquina Enigma, l'encriptació segueix el mateix procés que la desencriptació. Això sí, les configuracions dels *rotors* i del *plugboard* han de ser les mateixes sempre.

(Lyons, Practical Cryptography 2009-2012)

En la programació de la màquina Enigma, no he incorporat el *plugboard* perquè considero que no cal afegir-hi un grau més de complexitat.

La funció principal, *enigma*, té quatre variables:

- *ordre_rotors:* Demana l'ordre dels tres rotors d'esquerra a dreta en una string o en una llista. Per exemple, "312".
- *pos:* Demana la posició inicial de cada rotor en ordre d'esquerra a dreta. Per exemple, "TWA".
- *osques:* Les posicions clau de cada rotor en el seu ordre. Per exemple, "EQV".
- *plaintext:* El missatge en una *string* que la màquina Enigma encriptarà o desencriptarà.

També he definit una funció secundària anomenada *pas* la qual requereix tres variables (els rotors, les lletres d'osca i la posició de cada rotor) i retornarà els rotors després d'un cicle i la posició d'aquests.

```python
# MÀQUINA ENIGMA

RI = "EKMFLGDQVZNTOWYHXUSPAIBRCJ"
RII = "AJDKSIRUXBLHWTMCQGZNPYFVOE"
RIII = "BDFHJLCPRTXVZNYEIWGAKMUSQO"
UKW = "YRUHQSLDPXNGOKMIEBFZCWVJAT"

d = {"1":RI, "2":RII, "3":RIII}

def pas(rotors, osques, pos):                       # Funció per a avançar una posició
    gir = False
    if pos[1] == osques[1]:                         # Si el segon rotor està en posició clau,
        rotors[0] = rotors[0][1:]+rotors[0][0]      # Moure el segon i el lent una posició
        pos[0] = alph[(alph.index(pos[0])+1)%26]
        rotors[1] = rotors[1][1:]+rotors[1][0]
        pos[1] = alph[(alph.index(pos[1])+1)%26]
        gir = True                                  # var que confirma que el segon ha avançat
    if pos[2] == osques[2] and not gir:             # Si el primer està en posició clau,
        rotors[1] = rotors[1][1:]+rotors[1][0]      # El segon avança si encara no ho ha fet
        pos[1] = alph[(alph.index(pos[1])+1)%26]
    rotors[2] = rotors[2][1:]+rotors[2][0]          # El primer rotor avança sempre
    pos[2] = alph[(alph.index(pos[2])+1)%26]
    return rotors, pos


def enigma(ordre_rotors, pos, osques, plaintext):   # MÀQUINA ENIGMA I
    plaintext = plaintext.upper()                   # plaintext en majúscules
    pos = list(pos)
    rotors = [d[ordre_rotors[0]], d[ordre_rotors[1]], d[ordre_rotors[2]]]  # Ordre dels rotors
    for i in range(3):                                                     # Posicions inicials
        rotors[i] = rotors[i][alph.index(pos[i]):]+rotors[i][:alph.index(pos[i])]
    ciphertext = ""
    for c in plaintext:                         # Si el caràcter no és una lletra,
        if c not in alph:
            ciphertext += c                     # ignora'l.
            continue
        rotors, pos = pas(rotors, osques, pos)  # Funció per a avançar una posició
        for i in [2, 1, 0]:                     # Anada de la senyal fins el reflector
            c = alph[(alph.index(rotors[i][alph.index(c)])-alph.index(pos[i]))%26]
        c = UKW[alph.index(c)]                  # Rebot del reflector UKW-B
        for i in range(3):                      # Tornada de la senyal al panell
            c = alph[rotors[i].index(alph[(alph.index(c)+alph.index(pos[i]))%26])]
        ciphertext += c                         # Recull de totes les lletres
    return ciphertext
```

Fragment de codi 11 – La màquina Enigma

Vegem un exemple on encriptem una cita del criptoanalista britànic, qui desenvolupà tècniques per a desxifrar els codis alemanys, Alan Turing (1912-1954):

```python
ordre =        "312"
pos_inicials = "GMT"
osques =       "VQE"
plaintext =    """\nSOMETIMES IT IS THE PEOPLE NO ONE IMAGINES ANYTHING
OF WHO DO THE THINGS THAT NO ONE CAN IMAGINE.\n\t- Alan Turing"""

m = enigma(ordre, pos_inicials, osques, plaintext)
print(m)
print(enigma(ordre, pos_inicials, osques, m))
```

```
WERJRPPTG HQ PQ ANZ QJAYRU BG IDJ VAEAZTYV OFLNXXWE
RO YND QJ HNL EUSWRM WTGP TU HEK FKJ KDVLCZZ.
    - KKUW DVPYUS

SOMETIMES IT IS THE PEOPLE NO ONE IMAGINES ANYTHING
OF WHO DO THE THINGS THAT NO ONE CAN IMAGINE.
    - ALAN TURING
```

Fragment de codi 12 – Exemple de la màquina Enigma

5 La criptografia moderna

Durant la dècada del 1970, amb l'auge de la informàtica gràcies a corporacions tecnològiques com l'IBM (International Business Machines) i la necessitat de crear algorismes criptogràfics més sofisticats, aparegueren xifratges que utilitzen el llenguatge dels ordinadors, és a dir, la lògica binària. Van aparèixer algorismes de xifratge per blocs com el DES (Data Encryption Standard) l'any 1976 per a l'Agència de Seguretat Nacional (NSA) o l'AES (Advanced Encryption Standard) després dels atemptats de l'11-S.

També es va desenvolupar un nou sistema d'encriptació: la criptografia de clau pública o asimètrica. Tots els xifratges que hem vist fins ara són de clau privada o simètrica: l'*Alice* i en *Bob* han de tenir necessàriament la mateixa clau per a encriptar i desencriptar respectivament. Això, però, no és necessari amb la criptografia de clau pública. Veurem amb més detall el protocol Diffie-Hellman, l'encriptació RSA (utilitzada arreu avui dia) i la signatura digital, amb la que no caldrà signar mai més físicament.

També explicarem per sobre què són les criptomonedes, quina és la seva arquitectura de *blockchain* (anglès: "cadena de blocs") i quin serà el possible futur de la criptografia amb l'arribada d'ordinadors quàntics. El gegant nord-americà Google afirmà a finals d'octubre del 2019 que el seu ordinador quàntic preeminent aconseguí realitzar una tasca en un parell de minuts que a un ordinador ordinari li costaria més d'un miler d'anys. Podria l'arribada de la computació quàntica revolucionar la criptografia tal com la coneixem?

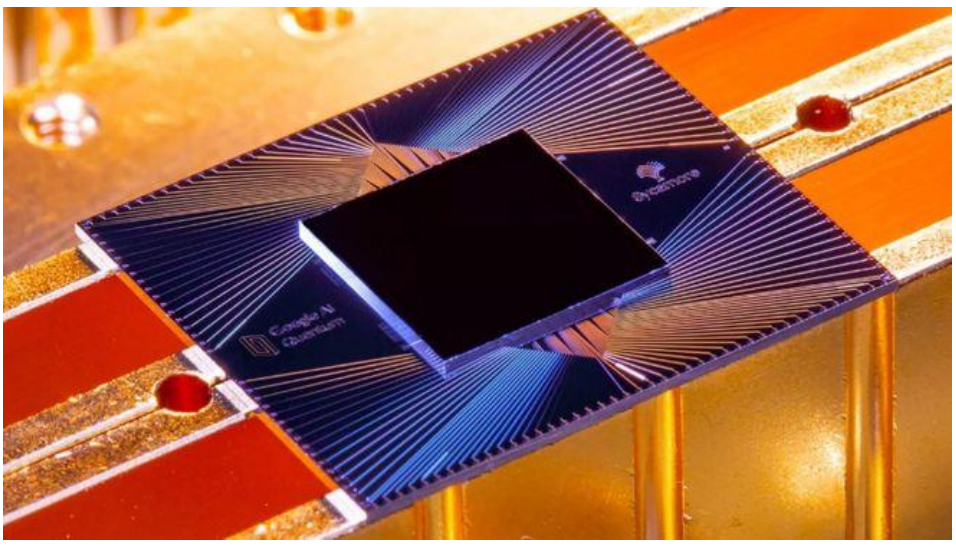


Figura X – El processador quàntic Sycamore del gegant tecnològic Google – Google Inc.

5.1 Algorisme d'Euclides

En aquest apartat no estudiem un xifratge, sinó eines que ens serviran per a entendre algorismes que emprarem més endavant.

Com he mencionat amb més detall en la pàgina 10, Euclides fou un matemàtic grec qui, en el seu llibre més famós anomenat *Els Elements,* desenvolupà un mètode ràpid per a calcular el màxim comú divisor de dos nombres: el conegut Algorisme d'Euclides.

Com bé sabem tots, el màxim comú divisor o M.C.D. de dos nombres és el nombre natural màxim el qual divideix exactament els dos nombres, és a dir, que la divisió no deixa cap residu.

No demostrarem l'algorisme d'Euclides, sinó que simplement explicarem el seu procediment. Per a entendre-ho fàcilment, l'explicaré amb un exemple: busquem el màxim comú divisor de 1305 i 195.

Comencem amb la següent equació:

$$1305 = 195 \cdot q + r \qquad r, q \in \mathbb{N}$$

On q és el quocient i r és el residu de la divisió. Si calculem q i r, obtenim que $q = 6$ i $r = 135$. A continuació, el divisor esdevé el nou dividend i el residu el nou divisor. És a dir, en el següent pas obtenim:

$$195 = 135 \cdot q + r \qquad r, q \in \mathbb{N}$$

$$q = 1, \qquad r = 60$$

I continuem així successivament fins que el residu sigui zero:

$$135 = 60 \cdot q + r \rightarrow q = 2, \;\; r = 15$$

$$60 = 15 \cdot q + r \rightarrow q = 4, \;\; r = 0$$

Si el residu és zero, el màxim comú divisor dels dos nombres inicials (1305 i 195) és el residu de l'última equació, o sigui, $MCD(1305, 195) = 15$.

```
89
90  # ALGORISME D'EUCLIDES
91
92  def mcd(a, b):
93      if b > a:
94          return mcd(b, a)
95      if a % b == 0:
96          return b
97      return mcd(b, a % b)
98
```

Fragment de codi 13 – L'algorisme d'Euclides

Com de costum, he codificat aquest algorisme amb *Python*. Amb aquest algorisme he fet ús de la recursivitat, és a dir, executar la funció dintre de la mateixa funció.

A la línia 62, simplement estem assegurant-nos que el dividend és el nombre més gran dels dos. Si no és el cas, la funció retorna el que retornaria la funció si a passa a ser b i viceversa. Seguidament, a la línia 64, estem creant la condició per a sortir d'una recursivitat infinita. Si el residu és zero, retornem l'últim valor de b.

Finalment, si no arribem a un residu de zero, retornem el que retornaria la funció *mcd* quan el divisor passa a ser el dividend i el residu ($a \ mod \ b$) passa a ser el divisor. Com que sempre arribarem a un punt on el residu doni zero, no obtenim un error de recursivitat.

5.2 Data Encryption Standard (DES)

El DES (Data Encryption Standard) és un algorisme de xifratge de clau simètrica desenvolupat durant la dècada del 1970 i escollit com a estàndard FIPS (Federal Information Processing Standard) als Estats Units l'any 1976. Avui dia, el DES és bastant insegur, vulnerable a atacs per força bruta a causa de la seva relativa senzillesa. Des de fa pocs anys un algorisme similar però de més complexitat, l'AES (Advanced Encryption Standard), ha substituït el DES.

El DES és un xifratge de blocs que té com a *plaintext* una seqüència de 64 bits i retorna com a *ciphertext* 64 bits. A més a més, necessitem una clau de 64 bits de la qual en traurem les *sub-keys* o claus secundàries de totes les setze rondes d'encriptació que el DES té.

Els següents diagrames de blocs d'elaboració pròpia[8] expliquen tots els processos del DES pels quals el *plaintext* i la clau passaran. Com podem veure, el DES consisteix en setze rondes on la clau i el *plaintext* de cada ronda pateixen una sèrie de canvis. En el primer diagrama podem veure el funcionament principal del DES. Posteriorment explicaré què significa cada procés i automatitzarem tot el procés amb *Python.*

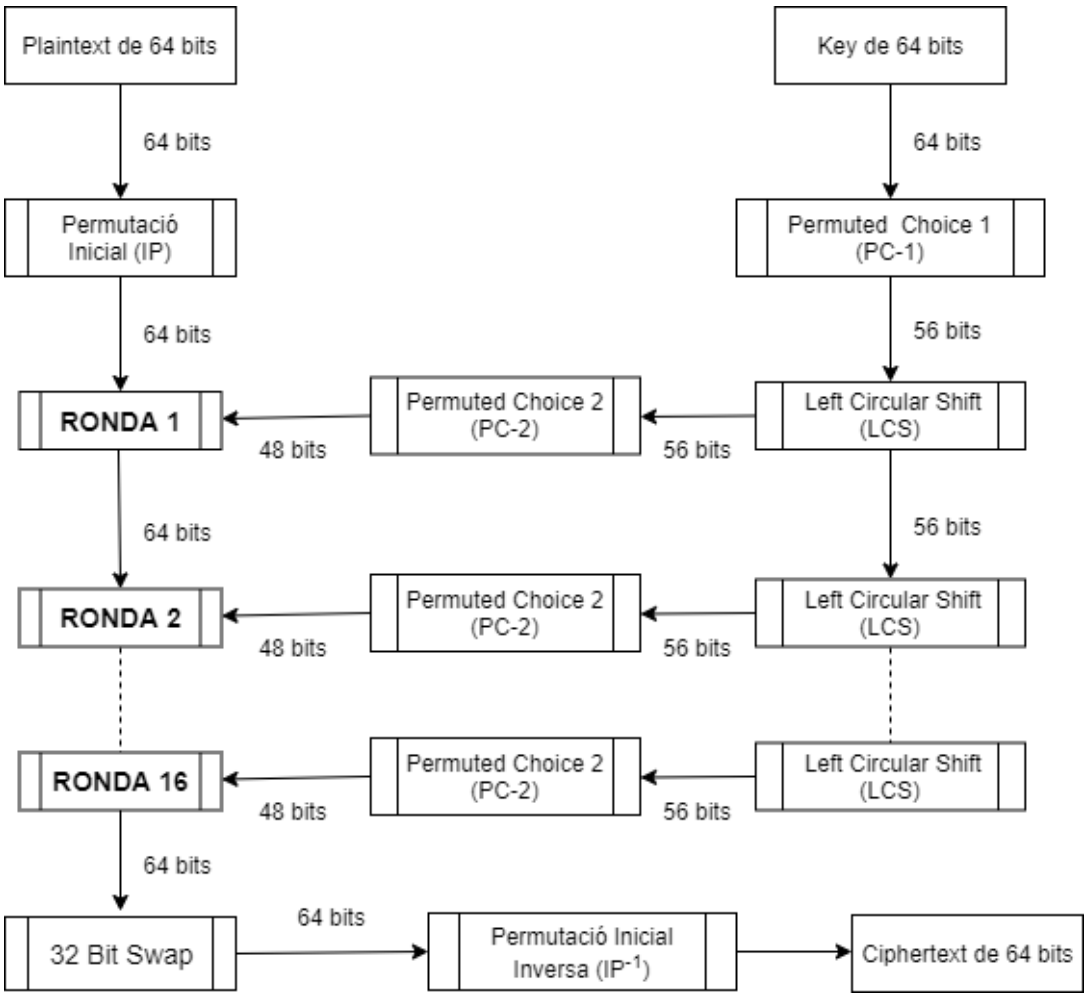


Diagrama 1 – Diagrama de blocs del funcionament general del DES

[8] Elaborat a partir del software gratuït draw.io i rèplica del diagrama de blocs del vídeo educatiu sobre el DES a YouTube: https://youtu.be/SaZGjQBItBc de l'usuari "Sundeep Saradhi Kanthety".

En aquest segon diagrama de blocs podem veure com funciona una ronda del DES. Com hem vist en el primer diagrama, el DES consisteix en setze rondes on intervé un *plaintext* i una *sub-key* o clau secundària. Aquestes dues se separen en dos blocs: els bits de l'esquerra (E) i els bits de la dreta (D). A partir de la següent pàgina començaré a explicar com funciona cada procés.

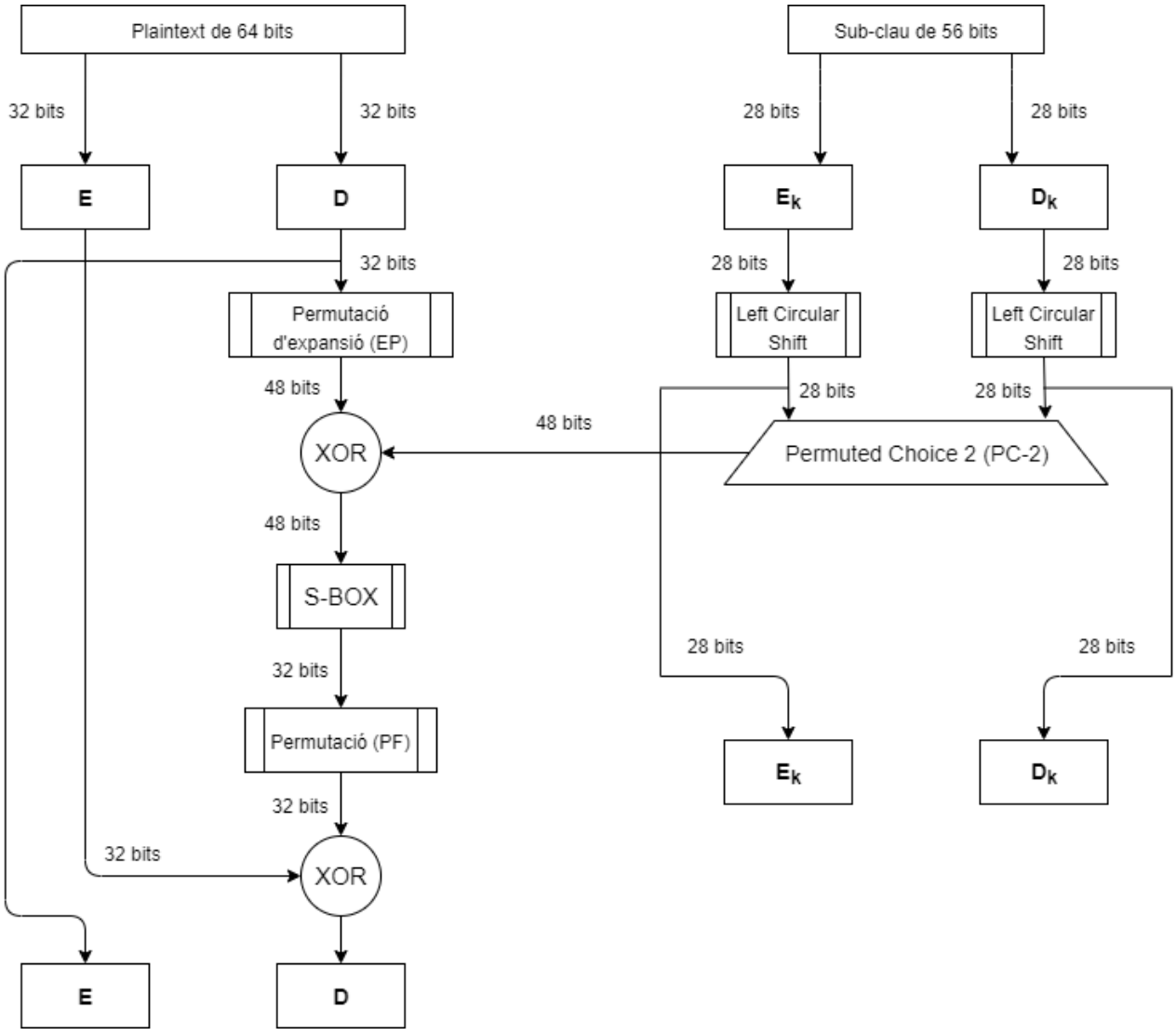


Diagrama 2 – Diagrama de blocs del funcionament d'una ronda en DES – Elaboració pròpia[8]

El DES és un xifratge de blocs, és a dir, la informació (en aquest cas una cadena de bits) s'organitza en una sèrie de "blocs" individuals que, a partir de canvis i permutacions, el *ciphertext* que obtindrem serà molt diferent de la informació inicial.

Començarem amb la primera permutació, Permutació Inicial o IP. Aquesta funció té com a entrada el *plaintext* original de 8 bytes (64 bits) i com a sortida una nova cadena de 64 bits (8 bytes).

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Taula 21 – Permutació Inicial del DES

Els bits estan configurats en una taula de valors 8x8: el bit que ocupa la posició 58 de l'entrada passa a ocupar la primera posició a la sortida, el bit número 60 passa a ser el novè bit i el bit que ocupa la posició 7 passa a ser l'últim. L'ordre de bits en una taula, doncs, es llegeix d'esquerra a dreta i, a continuació, de dalt a baix.

La següent funció que comentarem és la "Permuted Choice 1" o (PC-1). Aquesta permutació rep la nostra clau original de 8 bytes (64 bits) i en treu la primera *sub-key* de 56 bits o 7 bytes.

Esquerra (28 bits)							***Dreta (28 bits)***						
57	49	41	33	25	17	9	63	55	47	39	31	23	15
1	58	50	42	34	26	18	7	62	54	46	38	30	22
10	2	59	51	43	35	27	14	6	61	53	45	37	29
19	11	3	60	52	44	36	21	13	5	28	20	12	4

Taula 22 – Permuted Choice 1 (PC-1) del DES

Els bits de les *sub-keys* se separen en dos blocs: els bits de l'esquerra i els de la dreta.

Després de les dues funcions inicials, s'inicia un cicle on es repetiran les mateixes accions setze vegades. A partir d'aquí, ens podem fixar en el *Diagrama 2*: el *sub-plaintext* i la *sub-key* es divideixen en dos blocs: el bloc esquerre i el bloc dret.

Per a la funció "Left Circular Shift" (LCS) necessitem 28 bits i el número de la ronda, ja que aquest determinarà quants bits caldrà decalar amb la següent taula:

R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**	**R10**	**R11**	**R12**	**R13**	**R14**	**R15**	**R16**
1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Taula 23 – Left Circular Shift (LCS) del DES

Cal decalar un o dos bits per l'esquerra i de manera circular, és a dir, de manera cíclica. Si tinc per exemple una seqüència no-binària "ABCDEFGH" i he de decalar dues lletres, obtindré "CDEFGHAB".

Després d'aplicar un LCS als 28 bits de l'esquerra de la *sub-key* (E_k) i als 28 de la dreta (D_k), hem d'ajuntar els dos blocs i aplicar la "Permutation Choice 2" (PC-2). Observem que té com a entrada 56 bits i en retorna només 48, és a dir, en perd 9.

La PC-2 funciona igual que la permutació inicial IP, però hi haurà bits que no utilitza:

14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

Taula 24 – Permuted Choice 2 (PC-2) del DES

Abans de prosseguir, però, els 32 bits de la dreta del nostre *sub-plaintext* han de convertir-se en 48 per a poder aplicar l'operador binari XOR. Per a fer-ho, apliquem la permutació d'expansió (EP).

La funció d'expansió es comporta de la mateixa manera que la permutació inicial IP. Aquí, però, entren 32 bits i en surten 48, és a dir, allarguem la cadena de bits.

32	1	2	3	4	5
4	5	6	7	8	9
8	9	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	1

Taula 25 – Permutació d'expansió (EP) del DES

La taula de la veritat de la funció lògica XOR (disjuntiva exclusiva) mostra com el resultat serà vertader (valor binari d'1) si els dos *inputs* són diferents entre si. L'operador s'aplicarà per cada parella de bits en les dues cadenes corresponents.

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Taula 26 – Taula de la veritat de la funció XOR

Després d'aplicar l'operador XOR per a cada parell de bits, haurem d'aplicar al resultat de 48 bits (8 parelles de 6 bits) el que anomenem caixes de substitució o simplement "S-BOX". Cada grup de 6 bits ens donarà un valor o un altre depenent de la S-BOX que estiguem emprant. En total n'hi ha vuit: S_1 pel primer grup de sis bits, S_2 pel segon, S_n pel grup de sis bits que ocupa la posició n.

Cada S-BOX, doncs, prendrà com a entrada una cadena de sis bits i en retornarà quatre. Per tant, després d'unir un altre cop tots els resultats de totes les vuit caixes de substitució obtindrem una cadena de 32 bits.

Quan rebem la cadena de sis bits, el primer i l'últim defineixen quina filera (del 0 al 3) hem de prendre i els quatre bits centrals defineixen la columna (del 0 al 15) que haurem de prendre. La coordenada en la S-BOX serà un nombre decimal del 0 al 15, és a dir, un nombre binari de quatre xifres del 0000 al 1111.

S_1	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
00	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
01	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
10	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
11	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Taula 27 – La primera S-BOX de l'algorisme DES

Si , per exemple, prenem la cadena "110110" amb la caixa S_1, obtindrem com a resultat el nombre "0111" (el nombre 7 en binari).

1	1	0	1	1	0

Taula 28 – Funcionament amb detall d'una S-BOX del DES

Després de les capses de substitució, com podem veure en el diagrama de la pàgina 55, ve una última funció (PF) abans d'aplicar una altra vegada l'operació XOR. Aquesta funció no té cap misteri, ja que es comporta com la primera que hem explicat:

16	7	20	21	29	12	28	17
1	15	23	26	5	18	31	10
2	8	24	14	32	27	3	9
19	13	30	6	22	11	4	25

Taula 29 – Funció de permutació (PF) del DES

Després de finalitzar amb totes les setze rondes, capgirem el bloc dret amb el bloc esquerre (32 Bit Swap) i apliquem la funció inicial inversa. Si tornem a la mateixa taula de la Permutació Inicial (IP) però invertim el procés, el primer bit passarà a ocupar la posició número 58, el novè bit passarà a ocupar la posició número 60 i l'últim bit passa a ocupar la setena posició al *ciphertext* final de 64 bits.

(U.S. Dept. of Commerce / National Institute of Standards and Technology 1999)

```
308
309 def xor(a, b):
310     return [str(int(x==y)) for x, y in zip(a, b)]
311
312 def LCS(a, ronda):
313     if ronda in (1, 2, 9, 16):
314         n = 1
315     else:
316         n = 2
317     return a[n:]+a[:n]
318
319 def dec_a_bin(n):
320     # Decimal de l'1 al 16
321     b = bin(n)[2:]
322     return (4-len(b))*"0"+b
323
324 def DES(plaintext, clau):
325     plaintext = [plaintext[x-1] for x in IP]                    # Permutació Inicial (IP)
326     E = plaintext[:len(plaintext)//2]
327     D = plaintext[len(plaintext)//2:]                           # Dividir en esq. i dreta
328     Ek = [key[x-1] for x in PC1e]                               # Permuted Choice 1 (PC-1)
329     Dk = [key[x-1] for x in PC1d]
330     for r in range(1, 17):                                      # Iterem amb 'r' (la ronda)
331         Ek = LCS(Ek, r)
332         Dk = LCS(Ek, r)
333         c = [D[x-1] for x in EP]                                # Permutació d'Expansió (EP)
334         c = xor(c, [list(Ek+Dk)[x-1] for x in PC2])             # Permuted Choice 2 i XOR
335         s = []
336         for i in range(0, len(c), 6):                           # S-BOX
337             t = c[i:(i+6)]
338             fila = int(t[0]+t[-1], 2)
339             col  = int("".join(t[1:5]), 2)
340             t = dec_a_bin(S_BOX[i//6][fila][col])
341             s += t
342         s = [s[x-1] for x in PF]                                # Permutació (PF)
343         E, D = D, xor(s, E)
344     ciphertext = D + E                                          # 32 Bit Swap
345     return "".join([ciphertext[x-1] for x in inv_IP])           # Permutació Inicial Inversa
346
```

Fragment de codi 14 – L'algorisme DES

Les variables de cada permutació les podreu trobar el codi. El *plaintext* i la clau han de ser de 64 bits. El *ciphertext* que la funció DES retorna serà, doncs, de 8 bytes.

El principi de la fi del DES arribà l'any 1999 quan la màquina "Deep-Crack" dissenyada per a atacar a la força bruta el DES aconseguí desxifrar una clau en 22 hores i 15 minuts. L'AES, publicat el novembre del 2001, fou adoptat com a FIPS oficialment el dia 26 de maig de 2002. Aquest té una estructura i funcionament similar al DES, però és molt més difícil d'atacar.

5.3 L'intercanvi de claus Diffie-Hellman

L'intercanvi de claus Diffie-Hellman és un protocol criptogràfic de clau pública que permet que l'*Alice* i en *Bob* puguin arribar a una clau privada sense que l'*Eve* pugui descobrir-la. Aquest protocol innovador fou inventat pels criptòlegs estatunidencs Whitfield Diffie, Martin Hellman i Ralph Merkle l'any 1976.

La idea principal contextualitzada és la següent:

1. L'*Alice* i en *Bob* escullen públicament un color.
2. L'*Alice* barreja el color públic amb el seu color privat. En *Bob* fa el mateix.
3. L'*Alice* i en *Bob* s'envien mútuament les seves mescles.
4. L'*Alice* afegeix la seva pintura privada a la mescla enviada pel *Bob*. En *Bob* fa el mateix que l'*Alice*.
5. L'*Alice* i en *Bob* ara tenen el mateix color davant seu.

Si l'*Eve* és capaç d'interceptar tots els colors enviats pel *Bob* i l'*Alice*, obté el color públic inicial i les dues mescles inicials del pas 2. Suposant que la separació d'una mescla és impossible, l'*Eve* no pot fer-hi res.

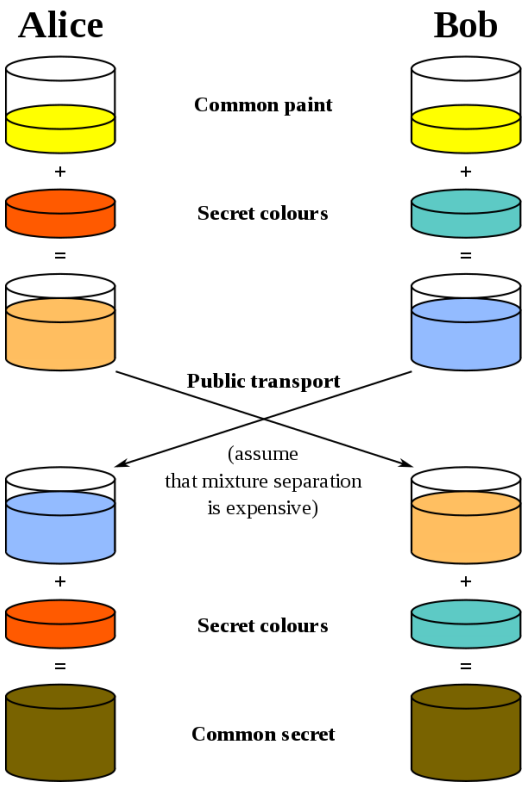


Diagrama 3 – Contextualització del protocol Diffie-Hellman – A.J. Han Vinck, University of Duisburg-Essen

Mesclar dos colors és senzill, però esbrinar un dels colors primitius és una tasca complicada. Si ho volem definir matemàticament, haurem de trobar una funció que sigui fàcil d'aplicar en una direcció, però difícil saber-ne l'antiimatge. Aquests tipus de funcions s'anomenen "unidireccionals" i són d'extrema necessitat perquè el protocol de clau pública funcioni. Els creadors del protocol Diffie-Hellman van recórrer a l'aritmètica modular per a trobar una funció unidireccional.

S'escull un nombre primer p perquè sigui el mòdul i un nombre g que sigui una arrel primitiva mòdul p. Un nombre g és arrel primitiva mòdul p si, per a qualsevol enter a coprimer a p, existeix un enter k tal que $g^k \equiv a \ (mod\ p)$. Per exemple, si $p = 17$, podríem escollir com a $g = 3$.

$$3^n \ mod\ 17 = a$$

D'aquesta manera, per a qualsevol exponent n, el valor a és igual de probable que sigui qualsevol nombre entre el 0 i el 16. Si, per exemple, $n = 29$, $3^{29}\ mod\ 17 = 12$. El problema sorgeix quan hem de fer el contrari: esbrinar quin valor n hem emprat donada la congruència a:

$$3^n \ mod\ 17 = 12$$

En aquest cas, només podem recórrer a prova i error. Si utilitzem nombres petits com el 17, trobar l'exponent és bastant senzill. Si utilitzem, però, un nombre primer llarguíssim, la potència computacional de tots els ordinadors del planeta avui dia necessitaria milers d'anys per a poder resoldre-ho. Aquesta és, doncs, la nostra funció unidireccional.

1. L'*Alice* i en *Bob* escullen públicament el nombre primer i la base (també anomenada generador). En aquest cas, $p = 17$ i $g = 3$.
2. L'*Alice* i en *Bob* escullen la seva clau privada n i calculen $3^n\ mod\ 17$. Si l'*Alice* escull el número 15 i en *Bob* el 13, l'*Alice* obtindrà $3^{15}\ mod\ 17 = 6$ i en *Bob* obtindrà $3^{13}\ mod\ 17 = 12$.
3. L'*Alice* i en *Bob* s'envien públicament els seus resultats.
4. L'*Alice*, amb el resultat d'en *Bob*, calcula $12^{15}\ mod\ 17 = 10$. Si en *Bob* fa el mateix amb el resultat de l'*Alice*, obté $6^{13}\ mod\ 17 = 10$. L'*Alice* i en *Bob* ara tenen davant seu la mateixa clau.

Si observem atentament per què l'*Alice* i en *Bob* obtenen el mateix número, haurem de trobar la manera d'afirmar que

$$12^{15} \equiv 6^{13} \ (mod\ 17)$$

L'*Alice* ha rebut la seva base d'en *Bob* quan aquest ha calculat $3^{13}\ mod\ 17$ i en *Bob* ha rebut la seva base de l'*Alice* quan ha calculat $3^{15}\ mod\ 17$. Per tant,

$$(3^{13})^{15} \equiv (3^{15})^{13} \ (mod\ 17)$$

Donada la propietat de les potències $(a^b)^c = a^{bc}$, $(3^{13})^{15} = (3^{15})^{13} = 3^{15 \cdot 13}$.

L'*Eve*, amb tota la informació que s'ha enviat ($g = 3; p = 17; a = 6; b = 12$), sense cap de les claus privades no pot fer-hi res. Si parlem de nombres molt llargs, és tècnicament impossible esbrinar la clau. En el següent apartat ampliarem els nostres coneixements sobre la teoria dels nombres i, més endavant, ensenyaré de quina manera podem arribar a un nombre primer molt llarg.

5.4 RSA (primera part)

Tot i que l'intercanvi de claus Diffie-Hellman funciona, en la vida real no resulta molt pràctic. Si, per exemple, l'*Alice* és un banc, ella hauria d'aplicar el protocol Diffie-Hellman per a cada *Bob* i, a més a més, hauria de guardar totes les claus que manté amb cada *Bob*. A causa d'això, un nou algorisme fou desenvolupat l'any 1977 pels criptògrafs Ron Rivest, Adi Shamir i Leonard Adleman, de l'Institut Tecnològic de Massachusetts (MIT): el sistema criptogràfic de clau pública RSA; que pren les inicials dels cognoms dels seus creadors.

El sistema que van idear, RSA, permet a l'*Alice* establir una comunicació segura amb cada *Bob* sense haver de guardar múltiples claus. Si, posem per cas, que l'*Alice* comprés un cadenat, l'obrís, es quedés la clau i el fes públic perquè algú li trametés un missatge encriptat amb el cadenat, ella podria llegir el missatge amb la clau. D'aquesta manera, si l'*Alice* fes públics múltiples cadenats amb la mateixa clau, tothom que li volgués escriure podria fer-ho d'una manera segura.

Per a definir aquest procés d'una manera més formal, els creadors de l'RSA van dividir la clau en dues claus: la clau d'encriptació i la clau de desencriptació. La clau

d'encriptació seria pública per a tothom, però la de desencriptació, que fa el procés invers que la d'encriptació, seria privada per a l'*Alice*. De la mateixa manera que en l'intercanvi de claus Diffie-Hellman, RSA necessita una funció unidireccional que sigui fàcil d'aplicar en un sentit (encriptació), però difícil sense la clau de desencriptació. Si l'*Eve* no té la clau de desencriptació, serà impossible per a ella esbrinar els missatges que en *Bob* envia a l'*Alice*.

La solució passa, com abans, per utilitzar l'aritmètica modular. Si m és el missatge que en *Bob* envia a l'*Alice*, e és la clau d'encriptació i N un mòdul públic per a tothom, en *Bob* crea el *ciphertext* c:

$$m^e \ mod \ N \equiv c$$

Si l'*Eve* intercepta c, esbrinar la base de l'exponent, és a dir, el missatge m, és pràcticament impossible i hauria d'emprar la prova-error. Per a desencriptar-ho, l'*Alice* elevarà el *ciphertext* c a un exponent d que haurà calculat prèviament i que desencriptarà:

$$c^d \ mod \ N \equiv m$$

Per tant, si substituïm m, obtenim:

$$m^{ed} \ mod \ N \equiv m$$

Abans de continuar amb la generació de les claus e i d, però, haurem d'ampliar els nostres coneixements sobre la teoria dels nombres. Repassarem què són els nombres primers i aprendrem sobre el matemàtic més prolífic de la història: Leonhard Euler.

5.5 La factorització en nombres primers

Els nombres primers són tots aquells nombres naturals $(1, 2, 3 \dots)$ que només es poden dividir exactament (residu nul) amb el mateix nombre i l'1. A més a més, per definició, el nombre 1 no és primer. No obstant això, tots els nombres naturals que no són primers tenen una connexió inevitable amb els nombres primers: el teorema fonamental de l'aritmètica.

El teorema fonamental de l'aritmètica afirma que, sigui k un nombre natural, aquest és o bé un nombre primer o un nombre que pot ser representat com la multiplicació de nombres primers i, a més a més, aquesta representació és **única** llevat de l'ordre dels factors. Aquesta representació es coneix popularment com la factorització d'un nombre en nombres primers. Per exemple,

$$5040 = 2^4 \cdot 3^2 \cdot 5 \cdot 7$$

$$157 = 157\ (nombre\ primer)$$

Tot i que la factorització en nombres primers pot semblar una tasca relativament senzilla a fer, resulta que no ho és. Factoritzar 5040 en nombres primers, com que aquest és divisible per primers com el 2, 3, 5 o 7, no és gens difícil. Si demano factoritzar 8051 en nombres primers sense l'ajut dels ordinadors, però, més d'un (incloent-m'hi) es cansaria de provar nombres primers i tiraria la tovallola, ja que aquest és compost per dos primers no tan comuns: el 97 i el 83.

Si demanem a un ordinador factoritzar en nombres primers un número N llarguíssim i que, a sobre, aquest només té dos factors, com que no existeix un mètode òptim però la prova-error en l'interval $[2, \sqrt{N}]$, aquest trigarà anys fins a trobar un dels factors.

Com veurem més endavant, desgraciadament per l'*Eve*, sense els dos factors primers del mòdul N, ella no podrà esbrinar mai el missatge.

5.6 La funció ϕ d'Euler

El matemàtic suís Leonhard Euler (1707-1783), a més de les seves innumerables aportacions en el càlcul infinitesimal, la teoria de grafs, la topologia, en la notació matemàtica i la dinàmica de fluids, també treballà en la teoria dels nombres.

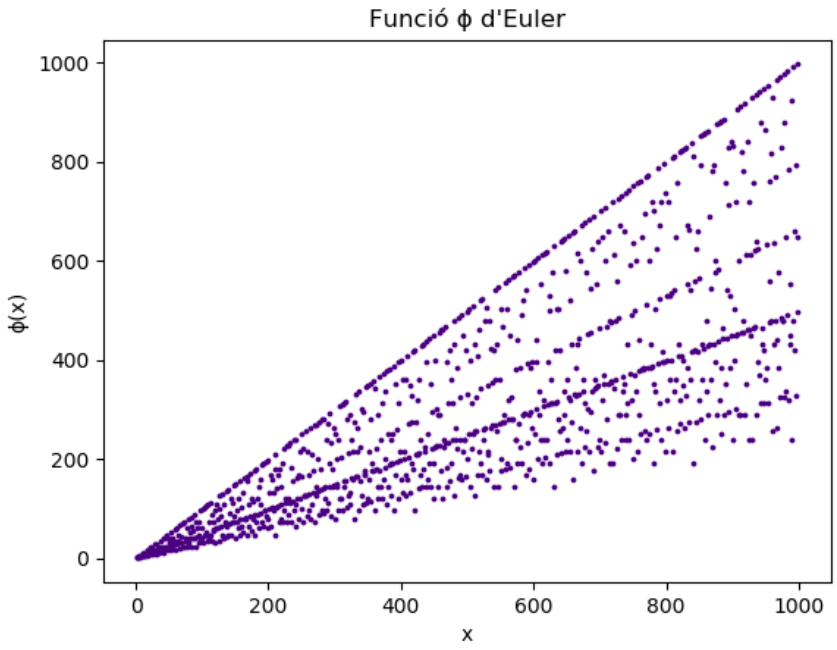
En aquest camp, Euler se centrà en els estudis de Pierre de Fermat. A més a més, descobrí relacions entre els nombres primers i l'anàlisi. Per exemple, Euler demostrà que la suma dels recíprocs dels nombres primers divergeix:

$$\sum_{p\ primer} \frac{1}{p} = \frac{1}{2} + \frac{1}{3} + \frac{1}{5} + \frac{1}{7} + \frac{1}{11} + \frac{1}{13} + \cdots = \infty$$

Euler també demostrà el Petit Teorema de Fermat, el Teorema de la suma dels dos quadrats, inventà la funció ϕ (lletra de l'alfabet grec: "phi") i també demostrà que el número $2^{31} - 1 = 2.147.483.647$ és un nombre primer. El número ostentà el títol del nombre primer més gran conegut fins a l'any 1867, vuit dècades després de la seva mort.

Una de les funcions primordials en RSA és la funció ϕ d'Euler. Es defineix com el nombre de nombres naturals més petits i coprimers a l'argument donat. Per exemple, si volem calcular $\Phi(10)$, com que només hi ha quatre naturals més petits o iguals a deu que no comparteixen cap factor comú $(1;\ 3;\ 7;\ 9)$, $\Phi(10) = 4$.

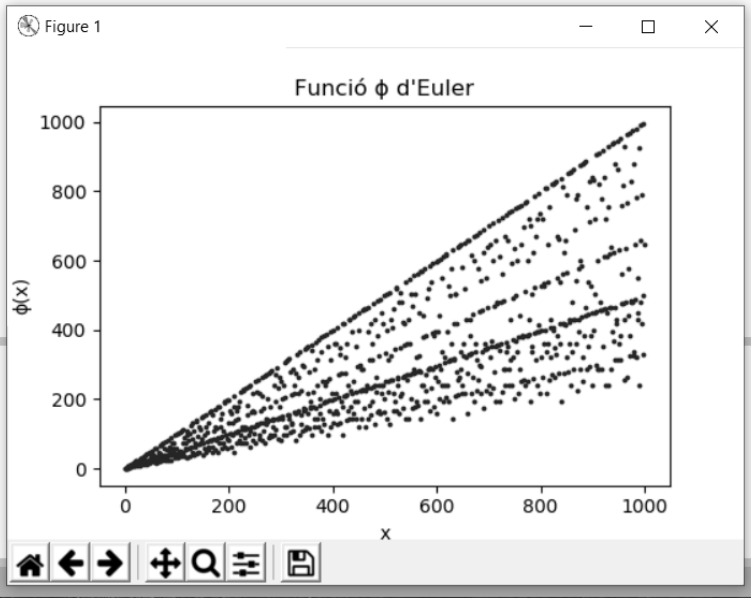
El següent gràfic d'elaboració pròpia mostra la funció ϕ en els eixos cartesians:



Gràfic 3 – Gràfica de la funció ϕ d'Euler

```
368  # FUNCIÓ PHI D'EULER
369
370  def phi(n):
371      # FUNCIÓ PHI D'EULER
372      return len([x for x in range(1, n+1) if mcd(x, n) == 1])
373
374  def grafic_phi(n):
375      # import matplotlib.pyplot as plt
376      # GRÀFICA DE LA FUNCIÓ PHI AMB LA LLIBRERIA MATPLOTLIB.PYPLOT
377      x = [i for i in range(1, n)]
378      y = [phi(i) for i in x]
379      plt.scatter(x, y, color="indigo", s=3)
380      plt.xlabel("x")
381      plt.ylabel("φ(x)")
382      plt.title("Funció φ d'Euler")
383      plt.show()
384
385  print(phi(10))
386  grafic_phi(1000)
```

```
4
```



Line 385, Column 15

Fragment de codi 15 – La funció ϕ d'Euler i la generació de la seva gràfica

Per a graficar amb Python, recomano la llibreria gratuïta *matplotlib.pyplot*. Després d'executar la funció *plt.show()*, apareix una finestra emergent amb la gràfica i un menú d'opcions per a ajustar paràmetres, guardar el gràfic o moure i ampliar el gràfic a una regió concreta dels eixos cartesians.

És inevitable veure un patró que la funció ϕ segueix. Sembla que, per a un determinat conjunt de naturals, la imatge és molt propera a l'argument. Aquests són els nombres primers. Si ens ho parem a pensar, quants naturals més petits que un nombre primer són coprimers amb ell? Per definició, com que són més petits i no tenen com a factor un primer més gran, $\Phi(p) = p - 1$ on p pertany al conjunt dels nombres primers.

A més a més, la funció ϕ compleix la propietat multiplicativa:

$$\Phi(pq) = \Phi(p) \cdot \Phi(q), \qquad mcd(p, q) = 1$$

Per tant, si p i q són dos nombres primers, $\Phi(pq) = (p-1)(q-1)$.

5.7 El Teorema d'Euler

El Teorema d'Euler és una generalització del Petit Teorema de Fermat. Aquest teorema, demostrat per Leonhard Euler, estableix una relació entre la funció ϕ i l'exponenciació modular. Si a i n són dos nombres naturals coprimers, llavors

$$a^{\varphi(n)} \equiv 1 \ (mod\ n)$$

On $\varphi(n)$ és la funció ϕ d'Euler.

El Petit Teorema de Fermat, demostrat per Euler i conjecturat per Pierre de Fermat, és una variant del Teorema d'Euler. Si p és un nombre primer i a és coprimer a p, llavors

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

Assumint que hem demostrat el Teorema d'Euler i emprant una propietat de la funció ϕ que hem demostrat en la pàgina anterior, demostrar el Petit Teorema de Fermat és trivial. Una vegada hem entès aquests conceptes bàsics sobre l'aritmètica modular, ja podem continuar amb el protocol RSA.

5.8 RSA (segona part)

Segons el Teorema d'Euler, sabem que si un natural m és coprimer a N, llavors es compleix que $m^{\varphi(N)} \equiv 1 \ (mod\ N)$. A més a més, sabem que calcular $\Phi(N)$ és fàcil a condició que N sigui un nombre primer o la multiplicació de dos primers. Aquests dos últims teoremes ens ajudaran, com a *Alice*, a determinar quina clau privada d haurem de generar per a desencriptar. Només ens cal modificar una mica el Teorema d'Euler.

Si elevem 1 a qualsevol exponent k, obtenim 1. Per tant, si elevem a un valor k la congruència modular, obtenim que:

$$m^{k\cdot\varphi(N)} \equiv 1 \ (mod\ N)$$

A més a més, si multipliquem per m, obtenim:

$$m^{1+k\cdot\varphi(N)} \equiv m \ (mod\ N)$$

I, com hem explicat en la primera part de l'RSA en la pàgina 64,

$$m^{ed} \equiv m \ (mod\ N)$$

Per tant, igualant els exponents,

$$1 + k \cdot \Phi(N) = ed \rightarrow d = \frac{1 + k \cdot \Phi(N)}{e}$$

Ja hem trobat la manera de calcular d. Per al valor N, l'*Alice* generarà dos nombres primers llargs i els multiplicarà. L'*Alice* sap com calcular $\Phi(N)$, però sense saber-ne la factorització, és pràcticament impossible per a l'*Eve* calcular d.

El valor e serà la clau d'encriptació pública. Aquest ha de ser un nombre imparell i no pot compartir cap factor comú amb $\Phi(N)$.

El valor k serà una constant qualsevol que faci que el valor d sigui un nombre natural.

Finalment, si sabem que $N = p \cdot q$ on p i q són primers, $\Phi(N) = (p-1)(q-1)$.

Posem un exemple il·lustratiu. Abans que res, l'*Alice* haurà de generar dos nombres primers, multiplicar-los i generar les claus e i d.

1. $p = 71; \quad q = 51; \quad N = p \cdot q = 3621; \quad \Phi(N) = (p-1)(q-1) = 3500$
2. L'*Alice* escull un nombre e qualsevol sempre que sigui imparell i no comparteixi cap factor amb $\Phi(N)$. En aquest cas, podem elegir $e = 3$.
3. $d = \frac{1+3500k}{3}$. Busquem un valor k que faci que d sigui un natural. Per exemple, podríem elegir $k = 4$. La clau de l'*Alice*, doncs, seria $d = 4667$.
4. L'*Alice* mostra a tothom els valors N i e.

Després, és el torn d'en *Bob* si vol enviar un missatge m. Sigui $m = 2019$, en *Bob* calcula $m^e \ mod \ N = 2019^3 \ mod \ 3621 = c = 1959$ i li envia aquest *ciphertext* a l'*Alice.* L'*Alice*, per a desfer l'encriptació, calcula $c^d \ mod \ N = 1959^{4667} \ mod \ 3621 = 2019 = m$.

És clar que aquest exemple és tan sols il·lustratiu, ja que el nombre N, com he dit, ha de ser llarguíssim (normalment de 4096 bits o més). Per tant, els seus factors (p, q) també ho han de ser. El següent dubte raonable que podem tenir és com generar un nombre primer de, per exemple, 2048 bits. Això ens condueix, doncs, als tests de primalitat.

He escrit el codi de RSA a continuació del test de primalitat de Fermat, ja que emprem aquest test per a generar p i q.

5.9 Test de primalitat de Fermat

Per a poder generar un nombre primer prou llarg perquè els nostres xifratges siguin prou segurs davant d'un possible atac de l'*Eve*, necessitem generar **aleatòriament** un nombre tan llarg com nosaltres vulguem. Després, mitjançant un test de primalitat, podrem afirmar si aquest és un nombre primer o no. En cas que no ho sigui, haurem de generar-ne un altre.

El test que nosaltres emprarem es basa en el Petit Teorema de Fermat que hem vist anteriorment. Segons el Petit Teorema de Fermat, sigui p un nombre primer i a un coprimer a p,

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

Per tant, si volem determinar si un nombre p és primer o no, generem un valor a que sigui coprimer a p. Llavors, si $a^{p-1}\ mod\ p \neq 1$, podem afirmar que p no és primer, ja que si ho fos, el resultat violaria el Petit Teorema de Fermat. Tot i això, si la congruència és igual a 1, no podem estar completament segurs que p sigui primer.

Per exemple, si $p = 341$ i $a = 2$,

$$2^{340} \equiv 1 \ (mod\ 341)$$

Però 341 no és primer, ja que té dos factors primers $(341 = 11 \cdot 31)$. Per tant, quan apliquem el test de primalitat de Fermat, haurem d'aplicar-lo múltiples vegades per a estar gairebé al cent per cent segurs que p és primer.

Si deixem que el valor a estigui comprès en l'interval $[2,\ p-1]$, com que a serà definitivament coprimer amb p si aquest és primer, el condicionant es complirà. Si p no és coprimer i a comparteix un factor, ens és absolutament igual. És més, si a i p comparteixen un factor comú, podrem descartar els nombres de Carmichael.

Un nombre de Carmichael h compleix que, per a qualsevol valor a coprimer a h,

$$a^{h-1} \equiv 1 \ (mod\ h)$$

La congruència es complirà sempre que a sigui coprimer a h i h sigui un nombre de Carmichael. El número de Carmichael més petit és el 561. Per tant, si a no és coprimer amb h, la congruència pot no complir-se.

```
385  # TEST DE PRIMALITAT DE FERMAT
386
387  def fermat(p, tests = 100):
388      if (p % 2 == 0 and p != 2) or p < 2:
389          return False
390      if p == 2:
391          return True
392      for _ in range(tests):
393          a = random.randint(2, p-1)
394          if p % a == 0:
395              return False
396          else:
397              if pow(a, p-1, p) == 1:
398                  continue
399              else:
400                  return False
401      return True
402
403  def generar_primer(llargada=1024): # Nombre de bits
404      p = 0
405      while not fermat(p):
406          p = random.getrandbits(llargada)
407      return p
408
409  print(fermat(9973), "\n")
410  print(generar_primer(1000))
```

```
True

203121052128349984241077508445858927071982407328650460666840745005282210768281978513641201112188824197732834073106792998626757688107941824039855511533903197544147122956170648925036706400578123966885636875653304207437894993998757078355208985079767909405915978854145788688145600731585959162866601099175 1
[Finished in 4.9s]
```

Fragment de codi 16 – Test de primalitat de Fermat amb exemples

Abans de començar amb els tests de primalitat, comprovo si el nombre és divisible per dos. Si ho és i aquest no és igual a dos, ja podem retornar el booleà `False.` Després, en la línia 394, aprofito per a comprovar que p no és divisible per a. Si resulta que ho és, retornem el booleà negatiu. Finalment, utilitzo la funció *pow* per a l'aritmètica modular. El primer argument és la base, el segon l'exponent i l'últim el mòdul.

La funció *generar_primer* genera un nombre primer d'una longitud determinada de bits. Si no proporcionem la llargada, el nombre prendrà la llargada premeditada (1024 bits).

Finalment, en les línies 409 i 410, posem a prova les dues funcions. Comprovem que el número 9973 és primer i, després, generem un nombre primer de mil bits. M'agradaria destacar el fet que l'ordinador ha trigat 4,9 segons a realitzar tota la tasca.

```
408
409  # RSA
410
411  def generar_claus(llarg=1024): # Nombre de bits
412      p = generar_primer(llarg)  # Generació primers
413      q = generar_primer(llarg)
414      N = p * q
415      phi = (p-1)*(q-1)
416      e = 3
417      while mcd(phi, e) != 1:  # MCD amb Algor. Euclides
418          e += 2
419      k = 1
420      d = 1
421      while not d == 0:        # Trobem el valor k
422          k += 1
423          d = (1+phi*k)%e
424      d = (1+phi*k)//e         # Càlcul de d
425      print("\nMÒDUL PÚBLIC N:\n"+str(N))
426      print("\nCLAU PÚBLICA D'ENCRIPTACIÓ e: "+str(e))
427      print("\nCLAU PRIVADA DE DESENCRIPTACIÓ d:\n"+str(d))
428
429  def rsa_e(m, e, N):
430      return pow(m, e, N)   # Exponenciació Modular
431
432  def rsa_d(c, d, N):
433      return pow(c, d, N)   # Exponenciació Modular
434
```

Fragment de codi 17 – RSA

Aquest és el codi de l'encriptació en RSA. Com podem veure, la funció principal és la que genera les claus d'encriptació i de desencriptació. Aquí, he hagut d'optimitzar la manera de calcular d, ja que Python no permet dividir amb decimals directament un nombre tan gran. Per tant, he anat augmentant k fins que el residu de $1 + k \cdot \Phi(N)$ dividit per e doni zero. Quan això passi, calcularé el quocient de la divisió i ho guardaré en d.

A part del codi, he creat un petit programa en el mateix directori que els altres xifratges on l'usuari pot interaccionar amb el protocol RSA. Recomano variar la longitud en bits dels nombres primers p i q i observar quant de temps triga l'ordinador a completar la tasca. A vegades, per a generar les claus a partir de dos nombres primers de 2048 bits, aquest pot trigar més de mig minut.

5.10 Les criptomonedes

Una criptomoneda o criptodivisa és un recurs digital monetari d'intercanvi que utilitza la criptografia per a assegurar transaccions i verificar la transferència d'actius. La principal característica que les diferencia d'una moneda corrent com el dòlar estatunidenc (USD) o l'Euro (EUR) és que el sistema de les criptomonedes no requereix una autoritat central com, per exemple, el Banc Central Europeu amb l'Euro.

La primera criptomoneda, el bitcoin (BTC), fou creada l'any 2009 per Satoshi Nakamoto, pseudònim del seu creador anònim. Des de llavors, n'han aparegut d'altres com Litecoin, Etherium o Ripple. Deixant de banda les possibles controvèrsies que poden sorgir, hom pot començar a invertir en qualsevol criptomoneda sense uns coneixements amplis de la criptografia rere aquesta.

L'arquitectura de les criptodivises permet enviar pagaments directament entre les parts i sense passar a través d'una institució financera. Ho aconsegueixen mitjançant el que anomenem signatura digital i una xarxa d'igual a igual. La signatura digital és un recurs que mostra validesa i certifica la veracitat d'una transacció. La xarxa d'igual a igual, en canvi, permet l'intercanvi de dades en la xarxa sense un servidor central.

El sistema que les monedes electròniques utilitzen s'anomena *blockchain* o "cadena de blocs". Cada bloc d'aquesta cadena té una certa informació emmagatzemada. En el cas de les criptomonedes, aquesta és l'emissor, el receptor, la quantitat, la data, etc. En segon lloc, cada bloc conté el que anomenem el "Hash" o el número d'identificació del bloc. Finalment, cada bloc també guarda el *hash* del bloc anterior. D'aquesta manera, els blocs s'uneixen formant cadenes.

La seguretat del *blockchain* ve donada pel *hash* i per la resta d'usuaris que la componen. El *hash* sempre té la mateixa llargada (256 bits en el cas del bitcoin) i es genera en funció de la informació que conté el bloc. D'aquesta manera, si algú canvia quelcom del contingut, el *hash* variarà. Com que el següent bloc no "encaixarà" amb l'anterior, la cadena de blocs queda invàlida. A més a més, com que tothom conserva la seva pròpia còpia de la cadena, la cadena d'aquell usuari queda anul·lada. Aquí és on rau la seguretat de les criptomonedes i del sistema *blockchain*.

Alguns usuaris, però, s'uneixen al sistema per a crear nous blocs. Els anomenem "miners" i són els encarregats de, una vegada es necessiti un nou bloc per a emmagatzemar més informació, resoldre un problema que requereix bastant potència computacional i, una vegada resolt, crear el nou bloc. A canvi, aquests reben una recompensa en BTC.

Actualment, aquesta recompensa és de 12,5 BTC[9]. Cada 210.000 blocs, aquesta recompensa es redueix a la meitat. D'aquesta manera, mai existiran més de vint-i-un milions de BTC.

Per a fer-nos una idea de com difícil és generar un nou bloc, s'estima que per a crear-ne un avui en dia, tots els miners alhora necessiten deu minuts a un ritme de 100 *exahashes* per segon. És a dir, $10^{20}\ \frac{hashes}{s}$. Aquesta velocitat s'anomena taxa de *hash* i és la unitat de mesura de potència de processament de la xarxa Bitcoin.

(Nakamoto 2008)

Figura XI – Logotip del protocol Bitcoin – Lliure de drets d'autor

[9] Informació i dades extretes del web https://www.bitcoinblockhalf.com/

5.11 La computació quàntica

Un ordinador quàntic és aquell que empra fenòmens de la mecànica quàntica per a calcular i operar amb dades. En un ordinador clàssic, la memòria només pot estar en un estat i, normalment, ho representem amb una cadena de bits (0 i 1). En un ordinador quàntic, però, la memòria està superposada quànticament en diferents estats. La unitat de memòria fonamental en aquest tipus d'ordinadors s'anomena "bit quàntic" o *qubit*.

Perquè ho entenguem d'una manera més física, imaginem-nos un escenari hipotètic on tenim un gat viu dins d'una caixa d'acer tancada i opaca amb un comptador de Geiger-Müller (detector de radiació ionitzant) i una certa quantitat de substància radioactiva. En el transcurs d'una hora, hi ha una probabilitat d'un 50% que un dels àtoms es desintegri, resultant en l'activació del comptador Geiger i, llavors, l'accionament d'un mecanisme que trenca un flascó de cianur d'hidrogen i mata el gat.

Aquest escenari s'anomena la paradoxa del gat de Schrödinger. Mentre no obrim la caixa en el transcurs d'una hora, el gat estarà alhora viu i mort, un estat que anomenem superposició quàntica, ja que la vida del gat depèn d'un esdeveniment subatòmic. Quan obrim la caixa, podrem determinar amb certesa quin és l'estat del gat.

En un *qubit*, no només tenim els dos estats d'un bit, sinó que el *qubit* pot trobar-se en una superposició d'aquests dos. Tot aquest conjunt d'estats es pot visualitzar amb l'esfera de Bloch, on el vèrtex superior d'aquesta representa l'estat 0 i l'inferior l'1. Un punt d'aquesta esfera és, doncs, un dels estats que un *qubit* pot prendre.

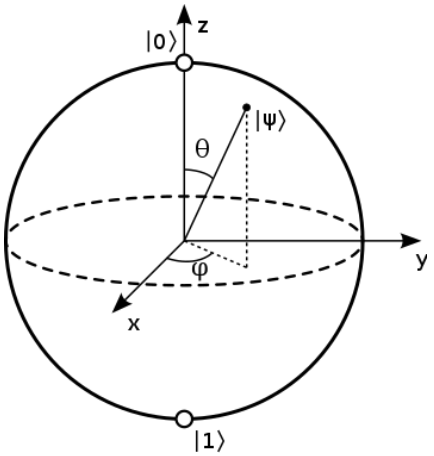


Figura XII – L'esfera de Bloch – Smite-Meister

Quan més a prop del $|0\rangle$ es trobi la posició del *qubit*, més important serà l'estat 0 i el mateix amb l'estat $|1\rangle$. Quan nosaltres volem observar quin és l'estat del *qubit*, com en l'estat del gat en la paradoxa del gat de Schrödinger, obtenim o bé el valor 0 o bé l'1. D'aquesta manera, quan llegim el *qubit*, l'estat d'aquest es pertorba i el *qubit* s'alinearà en vertical dins l'esfera de Bloch. De la mateixa manera, quan s'obre la caixa del gat de Schrödinger i es comprova si està viu o mort, l'estat de l'animal no variarà, encara que el tornem a posar dintre de la caixa.

Quan llegim el *qubit*, aquest apuntarà amunt o avall depenent del seu estat de superposició. Si està més a prop de l'extrem nord, té més possibilitats de ser llegit com a 0 que com a 1. Per tant, després de la lectura del *qubit*, aquest perdrà el seu estat de superposició i es projectarà en un sol eix.

Totes aquestes propietats tan abstractes i pràctiques del món quàntic, però, poden posar en perill tots els protocols de clau pública que existeixen actualment, incloent-hi RSA i l'intercanvi de claus Diffie-Hellman. Per exemple, l'any 1994, el professor de matemàtiques aplicades en el MIT Peter Shor desenvolupà un algorisme quàntic capaç de factoritzar un nombre en nombres primers molt més ràpidament que no pas un ordinador convencional. Això posa en perill tot el protocol RSA, ja que aquest assumeix que la factorització en nombres primers d'un nombre molt llarg és pràcticament impossible de realitzar.

A més a més del protocol RSA i l'intercanvi de claus Diffie-Hellman, la computació quàntica redueix a la meitat el temps necessari per a atacar un xifratge de clau privada o simètrica respecte a un ordinador clàssic. D'aquesta manera, la clau d'un xifratge de clau simètrica com AES-256 té la mateixa seguretat que una clau d'AES-128 utilitzant en el primer cas un ordinador quàntic.

No tot són inconvenients, però. L'arribada de la computació quàntica en la criptografia pot oferir més seguretat contra qualsevol atac que un ordinador clàssic. Efectivament, aquest nou tipus de computació podria deixar obsolets als sistemes digitals i iniciar una nova revolució: la revolució quàntica. La criptografia, sens dubte, canviarà radicalment en el futur si la computació quàntica aconsegueix dominar-la.

De moment, així i tot, la criptografia quàntica es troba en els seus inicis.

Conclusions

En aquest treball, he pogut compartir la meva passió per la criptografia i aquests coneixements amb vostès. A més a més, en aquest document, he pogut ordenar cronològicament tots aquests xifratges i codificar-los. Encara que n'hàgim vist molts, però, aquests només són els més destacables de la història de la criptografia.

En el transcurs d'aquest treball, he hagut de cercar, equiparar i examinar tots els recursos i fonts al meu abast per a poder aprendre i transmetre tot aquest aprenentatge. Molts conceptes de la criptografia ja els tenia ben assimilats i apresos abans de triar el meu Treball de Recerca, però he assolit la majoria dels meus coneixements sobre la criptografia i la matemàtica aplicada a la criptografia gràcies a aquesta recerca exhaustiva.

Amb referència a Python, he descrit el meu procés d'aprenentatge en l'apèndix d'aquest treball. Finalment, pel que fa als coneixements matemàtics i computacionals que he ofert en aquesta memòria, aquest Treball de Recerca no hauria estat viable sense el previ estudi d'aquests coneixements mitjançant recursos d'aprenentatge en línia com, per exemple, *Khan Academy,* o llibres d'aquests àmbits com, per exemple, *The Mathematics of Secrets* de l'autor Joshua Holden.

Hem arribat al final del nostre trajecte i hem vist com la criptografia ha evolucionat en el transcurs de l'existència humana. Espero i desitjo haver-me explicat d'una manera clara, entenedora, intel·ligible i precisa alhora. També espero haver-vos tramès tots els coneixements possibles sobre la criptografia i, si mai en sentiu parlar, que no sembli quelcom complicat o difícil d'entendre. Un dels meus entrebancs principals ha estat, precisament, saber com compartir tots aquests coneixements amb vostès.

Dit això, tan sols espero i desitjo que el meu objectiu s'hagi assolit i que vostès hagin pogut acostar-se al fascinant món de la criptografia. Espero que amb les línies de codi el lector pugui reproduir els diferents mètodes d'encriptació.

Referències Bibliogràfiques

Planes web consultades

Binance Academy. (2019, 26 agost). *History of Cryptography*. Recuperat 1 de setembre de 2019, de https://www.binance.vision/security/history-of-cryptography

Brilliant.org. (s/d). *Enigma Machine | Brilliant Math & Science Wiki*. Recuperat 2 de novembre de 2019, de https://brilliant.org/wiki/enigma-machine/

Cypher Research Laboratories Pty. Ltd. (2013, 17 abril). *A Brief History of Cryptography*. Recuperat 16 de juny de 2019, de http://www.cypher.com.au/crypto_history.htm

Dunham, William. (2013, 10 de juliol). *Number Theory.* Editat per inc. Encyclopædia Britannica. Recuperat 8 d'agost de 2019, de https://www.britannica.com/science/number-theory

Khan Academy. (2016, 2 de desembre). *What is modular arithmetic?*. Recuperat 27 d'agost de 2019, de https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic

Lyons, James. (s/d). *Practical Cryptography | Affine Cipher.* Recuperat 15 d'octubre de 2019, de http://practicalcryptography.com/ciphers/classical-era/affine/

Lyons, James. (s/d). *Practical Cryptography | Enigma Cipher.* Recuperat 10 de novembre de 2019, de http://practicalcryptography.com/ciphers/mechanical-era/enigma/#javascript-example

Nakamoto, Satoshi. (2008, 31 d'octubre). «Bitcoin: A Peer-to-Peer Electronic Cash System.». Recuperat 1 de desembre de 2019, de https://bitcoin.org/bitcoin.pdf

Rodriguez-Clark, Daniel. (2012, 23 de març). *Crypto Corner.* Recuperat 27 d'octubre de 2019, de https://crypto.interactive-maths.com/vigenegravere-cipher.html

U.S. Dept. of Commerce / National Institute of Standards and Technology. (1999, 25 d'octubre). «Data Encryption Standard (DES).» (Federal Information Processing Standards Publication 46-3). Recuperat 18 de novembre de 2019, de https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf

Llibres consultats

Tom M. Apostol. (1973). *Calculus Volumen 1* (2a edició). Barcelona, Espanya: Reverté.

Holden, Joshua (2017). *The mathematics of secret.* Nova Jersey, EUA: Princeton Uni. Press.

Imatges

Figura III. Euclides (autor) - Ratdolt, Erhald (impressor). (1482 imprès, s.III aC). *Elements d'Euclides.* Recuperat 13 de juliol de 2019, de https://commons.wikimedia.org/wiki/File:Euclid%27s_Elements,_1482.jpg

Figura VI. Autor desconegut (Conservat en Beinecke Rare Book & Manuscript Library, Yale University). (s.XV-XVI dC). *El manuscrit de Voynich.* Recuperat 22 d'agost de 2019, de https://commons.wikimedia.org/wiki/File:Voynich_Manuscript_(170).jpg

Figura VII. National Archives. (1917, 16 de gener). *Telegrama de Zimmermann.* Recuperat 21 d'octubre de 2019, de https://en.wikipedia.org/wiki/File:Zimmermann-telegramm-offen.jpg

Figura VIII. Museu Nacional de la Ciència i la Tecnologia Leonardo da Vinci, Milà. *Màquina Enigma.* Recuperat 15 de novembre de 2019, de https://commons.wikimedia.org/wiki/File:Enigma_(crittografia)_-_Museo_scienza_e_tecnologia_Milano.jpg

Figura IX. Usuari Matt Crypto, Wikimedia Commons. (2007, 16 de març). *Connexions elèctriques de la màquina Enigma.* Recuperat 16 de novembre de 2019, de https://commons.wikimedia.org/wiki/File:Enigma-action.svg

Figura X. Erik Lucero / Google Inc. (2019, 23 d'octubre). *L'ordinador quàntic de Google.* Recuperat 29 de novembre de 2019, de https://www.nytimes.com/2019/10/30/opinion/google-quantum-computer-sycamore.html

Diagrama 3. A.J. Han Vinck, University of Duisburg-Essen. (2011). *Contextualització de l'intercanvi de claus Diffie-Hellmann.* Recuperat 30 de novembre de 2019, de https://commons.wikimedia.org/wiki/File:Diffie-Hellman_Key_Exchange.svg

Figura XII. Usuari Smite-Meister, Wikimedia Commons. (2009, 30 de gener). *L'esfera de Bloch.* Recuperat 5 de desembre de 2019, de https://commons.wikimedia.org/wiki/File:Bloch_sphere.svg

Apèndix

Sobre les funcions

Una funció f és un conjunt de parells ordenats (x, y) cap dels quals té el mateix primer element. És a dir, una correspondència que associa cada objecte d'un conjunt X que anomenem "domini" a un i només un objecte del conjunt Y o "codomini". Quan escrivim $f: X \to Y$, estem dient que la funció f té com a domini el conjunt X i el conjunt Y com a codomini.

(Apostol, 1973)

En l'exemple de la pàgina disset, el domini i el codomini de la funció f és l'alfabet llatí modern, ja que cada lletra del *plaintext* se substituirà per una i només una altra lletra del mateix alfabet.

En l'expressió $f(x)$, x és l'argument de la funció i $f(x)$ el valor o la imatge d'aquest. El rang o recorregut de la funció f és el conjunt de tots els valors que pot prendre f per a totes les $x \in X$. Això implica, doncs, que el rang és un subconjunt de el codomini. Escrivim, doncs, $R_f \subseteq Y$. El rang pot ser un subconjunt propi del codomini (no totes les $y \in Y$ tenen una $x \in X$ tal que $f(x) = y$) o el mateix conjunt. En aquest últim cas, diem que la funció f és una "endofunció". Si resulta que per a cada $y \in Y$ hi ha exactament una $x \in X$ tal que $f(x) = y$, diem que la funció f és "bijectiva".

La cardinalitat d'un conjunt A es refereix al nombre d'elements del conjunt i se simbolitza amb dues barres verticals $|A|$. Si ens referim al conjunt de lletres de l'alfabet llatí modern, la seva cardinalitat és de 26.

Sobre Python i l'aprenentatge personal

Python és un llenguatge de programació d'alt nivell dinàmic que posa èmfasi en la llegibilitat del codi. És un dels llenguatges de programació més populars del món i alhora un dels més fàcils d'aprendre. Avui dia, l'índex TIOBE (mesura de popularitat dels diferents llenguatges de programació) suggereix que Python se situa en tercer lloc seguit del llenguatge C i de Java. La seva popularitat creixent es dona en gran manera gràcies a la seva relativa senzillesa respecte als altres llenguatges de programació d'alt nivell.

Aprendre'n no requereix ni un esforç molt gran ni uns coneixements avançats sobre la programació. Personalment, Python ha sigut i segueix essent l'únic llenguatge de programació que he après i emprat. Gràcies a llocs web d'aprenentatge gratuït com *Sololearn*[10] o *Stack Overflow*[11], hom pot aprendre qualsevol llenguatge de programació avui dia gratuïtament.

Vaig començar a aprendre HTML i CSS l'any 2017 amb *Sololearn*. Un any més tard, vaig decidir que era hora d'aprendre un llenguatge de programació. Després de visitar diferents pàgines web, vaig arribar a la conclusió que Python era l'opció ideal. A més a més, mentre aprenia Python, creava projectes personals com, per exemple, un programa que calcula les possibilitats que al tirar cinc daus tots cinc siguin el mateix nombre (popularment conegut com a "Yahtzee") o un programa que, si li introdueixes els resultats d'unes possibles eleccions en una circumscripció, et retorna el nombre d'escons que cada partit obtindrà (mitjançant el sistema de representació proporcional d'Hondt) i les seves desviacions corresponents.

A mesura que passa el temps, hom es dona compte dels seus errors, cerca una solució a internet i no els comet mai més. Aquest és el procés d'aprenentatge: qui aprèn dels seus errors no s'equivoca. L'error és part del camí d'aprenentatge en qualsevol estudi.

[10] Lloc web oficial: https://www.sololearn.com/
[11] Lloc web oficial: https://stackoverflow.com/