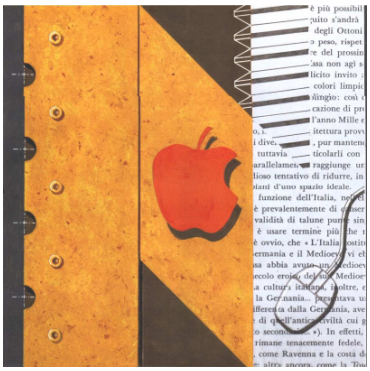


Newton-Pertini di Camposampiero

Crittografia

Student :
Alex Gasparini

Teacher :
Gianfranco Lamon

16 settembre 2025

Indice

1 Introduzione **2**
1.0.1 Cifratura . . . 3
1.0.2 Decifratura . . . 3
1.0.3 Chiave . . . 4
1.0.4 Rotto . . . 4
1.1 Principio di Kerckhoffs . . . 5
1.2 Il problema dello scambio della Chiave . . . 5
1.3 Le prime Classificazioni . . . 6
1.3.1 Sistemi Simmetrici - OTP . . . 7
1.3.2 Sistemi Simmetrici - PRNG / Stream . . . 7
1.3.3 Sistemi Simmetrici - A Blocchi . . . 8
1.3.4 Sistemi Asimmetrici - RSA . . . 9
1.3.5 Sistemi Asimmetrici - Scambio di Chiavi . . . 10
1.3.6 Post-Quantum . . . 11

2 Cifrari Antichi - Cesare **12**
2.1 Cifrari Monoalfabetici . . . 14

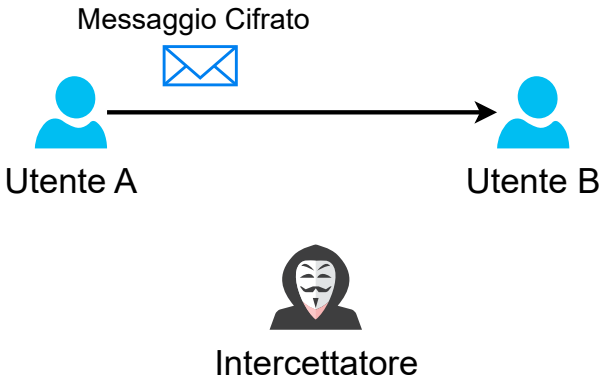
3 Cifrati Antichi - Enigma **18**
3.1 Preambolo Storico . . . 18
3.2 Funzionamento . . . 18
3.3 Struttura . . . 19
3.3.1 Plugboard . . . 19
3.3.2 Rotori . . . 19
3.3.3 Riflettore . . . 20
3.4 Funzionamento . . . 21

4 One Time Pad **22**
4.1 XOR . . . 22
4.2 Funzionamento . . . 23
4.2.1 Considerazioni . . . 23
4.3 Crackabilità . . . 24

5 Stream **25**
5.1 PRNG . . . 25
5.2 LFSR . . . 27
5.3 Stream . . . 29

1 Introduzione

In questo corso andremo a vedere le basi della **Crittografia moderna**. In primis, dobbiamo capire cosa vuol dire "*Crittografia*". Partiamo dall'etimologia: dal greco ***kryptós*** (nascosto) – ***graphía*** (scrittura). In sostanza, la Crittografia è la disciplina che studia e analizza come inviare e ricevere messaggi **nascosti**. Con il termine "nascosti" si intende che solo ed esclusivamente la sorgente e il destinatario possono leggere il contenuto del messaggio, mentre qualsiasi altra persona non può.



In questa immagine l'utente A manda un messaggio criptato all'utente B; in questo modo solo A e B potranno leggere il contenuto del messaggio, mentre l'intercettatore, anche se riuscisse ad averne una copia, non sarebbe in grado di leggerlo (dato che è criptato).

La crittografia la usiamo tutti i giorni (anche involontariamente) con i nostri dispositivi elettronici. Un esempio è **WhatsApp**, che tramite una crittografia **End-to-End** (che avremo tempo di approfondire) permette di inviare messaggi in maniera sicura, in modo che nessun altro (nemmeno WhatsApp stesso!) possa leggere il messaggio che hai mandato al tuo amico. Ha anche utilità nell'autenticazione digitale e nei documenti elettronici: infatti, tutti i sistemi come **SPID** oppure **CIE** sfruttano la crittografia per funzionare. La crittografia viene utilizzata anche dalle case produttrici di console (come **Sony** per la **PlayStation**) per impedire di crackare le loro console. Di esempi ce ne sono a centinaia e avremo tempo per scoprirli tutti.

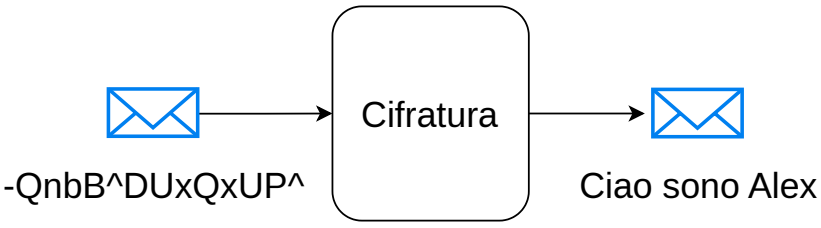
Anche se ho elencato molti esempi **digitali**, la crittografia è una disciplina basata sulla **matematica**. Infatti, tutti i sistemi crittografici si fondano su prove matematiche (come il **Logaritmo Discreto** e la **Fattorizzazione di numeri composti**) per funzionare. Mi piace definire la crittografia come una branca a metà strada tra matematica e informatica, poiché utilizza concetti matematici applicati a contesti informatici.

Un altro punto fondamentale da chiarire è che, anche se parleremo di sistemi moderni come **AES**, **RSA**, **Diffie-Hellman** e **ECC**, inventati tra il 1960 e il 1990 circa, in realtà la crittografia è molto più antica. Già dall'**Impero Romano** (753 a.C. – 476 d.C.) se ne parlava: chiaramente i sistemi erano molto più semplici di quelli odierni, ma all'epoca servivano per inviare messaggi all'esercito. In questo coro di cifrari "antichi" ne vedremo due, forse i più impattanti nella storia: il **Cifrario di Cesare**, che possiamo definire il primo sistema crittografico, e la macchina **Enigma**, che durante la Seconda Guerra Mondiale fu di fondamentale importanza per le truppe dell'Asse; gli alleati, grazie a **Alan Turing**, riuscirono a decifrarla, aiutando così la loro vittoria.

Fatte tutte le premesse del caso, iniziamo a parlare di crittografia; come prima cosa, vediamo e definiamo tutti i termini utilizzati in questo ambito.

1.0.1 Cifratura

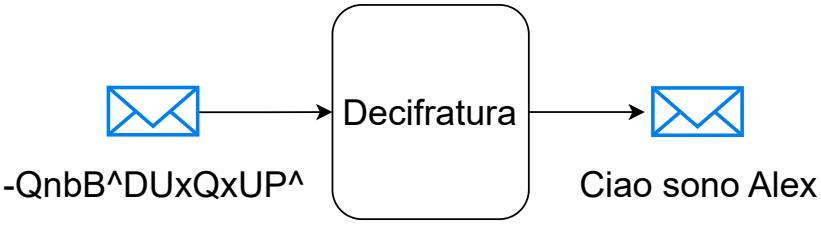
La cifratura di un messaggio è il processo che permette di **alterare** un messaggio in modo che nessun altro (eccetto chi lo invia e il destinatario) possa leggere il contenuto originario. La cifratura deve avvenire tramite un **algoritmo di cifratura** e con l'ausilio di una (o più) **chiavi**.



In questo caso, il messaggio "**Ciao sono Alex**" tramite una cifratura è diventato **-QnbBˆDUxQxUPˆ**; se qualcuno riuscisse a intercettare il messaggio cifrato, non capirebbe nulla.

1.0.2 Decifratura

La decifratura è il passaggio **inverso** della cifratura, nel senso che permette di tornare al messaggio originale partendo dal messaggio cifrato. Chiaramente, bisogna usare lo **stesso algoritmo di cifratura** e, soprattutto, la **stessa chiave**, che deve essere conosciuta solo da chi invia il messaggio e da chi lo deve ricevere.



Iniziamo a utilizzare un po' di nozionismo matematico: il meccanismo di cifratura e decifratura può essere paragonato a una **funzione**, poiché entrambe prendono una variabile in input e restituiscono un valore in output. Possiamo quindi definire la cifratura come

$$c = f(m)$$

Dove c è il messaggio cifrato, m è il messaggio originale e f è la "funzione di cifratura". Dato ciò, la "funzione di decifratura" sarà definita come

$$m = f^{-1}(c)$$

Questo può essere dedotto dalla seguente equazione:

$$m = f^{-1}(f(m))$$

Tranquilli, per ora abbiamo terminato con il nozionismo matematico.

1.0.3 Chiave

Una chiave è una qualsiasi **stringa** o, più semplicemente, un **numero**; la caratteristica principale è che una chiave deve **rimanere privata**, perché permette di cifrare e decifrare i messaggi. Se qualcuno riuscisse a rintracciare la vostra chiave privata, potrebbe leggere tutti i messaggi che inviate e ricevete. L'idea della chiave in crittografia è simile a quella di una **password**: allo stesso modo, se qualcuno vi ruba la password, può accedere al vostro account. In realtà, la chiave può anche essere qualcosa di più complesso, come un **punto nel piano cartesiano** (usato nella *Elliptic Curve Cryptography*).

Quindi, per evitare confusione, correggiamo la definizione precedente dicendo che una chiave è un qualsiasi **dato**, oppure un **insieme di dati**, che deve rimanere **segreto**.

In realtà, vedremo verso metà corso che esiste anche una cosiddetta **chiave pubblica**, ovvero una chiave come quella che abbiamo definito fino a ora, ma che **chiunque può conoscere**. Se vi sembra strano o controintuitivo, quando lo vedremo tutto sarà chiaro.

1.0.4 Rotto

Un sistema crittografico si definisce **rotto** qualora sia possibile decifrare un messaggio criptato senza conoscere la chiave. Un sistema rotto, chiaramente, non può essere utilizzato, perché chiunque riuscirebbe a decifrare il messaggio. Un esempio di sistema rotto è il **DES** (che vedremo nel capitolo "Cifrari a Blocchi"). Il DES è stato inventato nel 1976 e, all'inizio, era molto usato; il problema era che utilizzava una chiave a lunghezza fissa: 54 bit. Ad oggi, purtroppo, una chiave a 54 bit è soggetta ad attacchi **brute-force**[1], e per questo oggi il DES non può più essere usato per cifrare ed è stato sostituito dall'**AES**.

[1] Attacchi in cui si provano tutte le possibili combinazioni di una chiave; richiedono molto tempo, ma per chiavi molto piccole (come il DES) possono funzionare

1.1 Principio di Kerckhoffs

Ora che abbiamo iniziato a familiarizzare con i primi termini della crittografia, possiamo comprendere il principio fondante della disciplina: il **Principio di Kerckhoffs**.

Teorema 1: Principio di Kerckhoffs

La sicurezza di un sistema crittografico deve dipendere unicamente dalla chiave segreta e non dalla segretezza dell'algoritmo stesso.

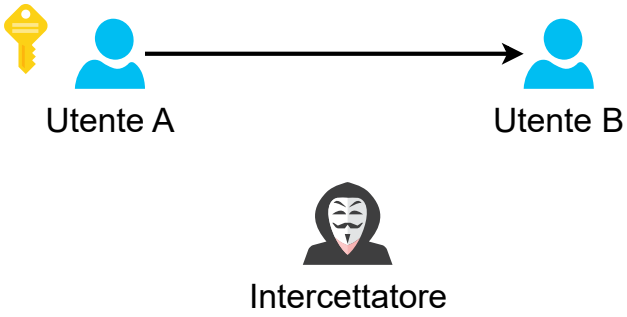
Sostanzialmente, Kerckhoffs afferma che non deve essere segreto **l'algoritmo di cifratura**; la forza di un sistema crittografico deriva invece dalla difficoltà di romperlo, e non dalla segretezza dell'algoritmo.

Per questo motivo, oggi sappiamo perfettamente quali algoritmi utilizzano i vari siti e le app; non è un rischio conoscere il metodo con cui viene criptato un messaggio, perché la sicurezza risiede nella segretezza della chiave, che, chiaramente, deve rimanere privata.

Per esempio, secondo Kerckhoffs, se tu e un tuo amico volete creare un sistema per scambiarvi messaggi segreti, non potete semplicemente utilizzare un sistema debole mantenendolo segreto a tutti gli altri; se qualcuno riuscisse a scoprire l'algoritmo, potrebbe leggere tutti i vostri messaggi. i messaggi.

1.2 Il problema dello scambio della Chiave

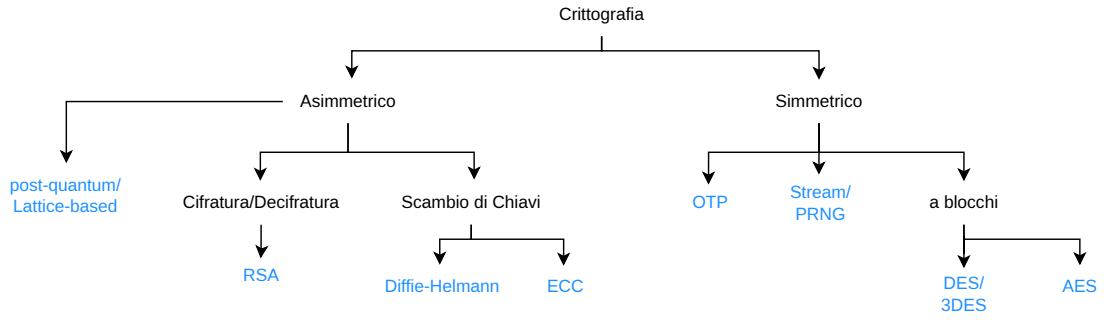
Prima di cominciare a parlare della classificazione dei sistemi crittografici, è necessario affrontare il problema dello scambio della chiave. Ripetendo quanto visto finora, la crittografia studia come due utenti possano scambiarsi messaggi in modo che nessun altro ne possa leggere il contenuto. Abbiamo capito che il messaggio viene cifrato e poi decifrato tramite un algoritmo. Inoltre, un algoritmo ha bisogno di una chiave per cifrare i messaggi, e la chiave deve essere posseduta solo da chi invia il messaggio e da chi lo deve ricevere; nessun altro dovrebbe conoscerla, altrimenti anche altri utenti potrebbero decifrare i messaggi. Tuttavia, non abbiamo ancora considerato come i due utenti possano scambiarsi una chiave comune o, comunque, concordare su una chiave da usare nel sistema.



In questa immagine, l'utente A ha generato una chiave da usare per cifrare i messaggi, ma deve trovare un modo per inviarla a B (così che lui possa decifrare i messaggi di A) senza che l'intercettatore riesca a intercettarla. Questo problema è stato risolto tramite i sistemi **asimmetrici**; come ciò sia possibile lo vedremo quando li studieremo nel dettaglio. Per ora, vi basta sapere che lo scambio della chiave è risolto da questi tipi di sistemi.

1.3 Le prime Classificazioni

I sistemi crittografici si dividono in diverse sottocategorie, ognuna con le proprie caratteristiche. Per comprenderle meglio, vediamo subito una mappa riassuntiva di tutte le categorie e poi le commenteremo una ad una. Perciò, ecco a voi la mappa:



La prima grande distinzione nella crittografia moderna è quella tra sistemi **simmetrici** e **asimmetrici**. Un sistema simmetrico utilizza **una sola chiave**, che deve rimanere sempre privata, mentre i sistemi asimmetrici hanno **due chiavi: una privata e una pubblica**. I due sistemi sono complementari l'uno all'altro e, ora, vedremo i principali vantaggi e svantaggi di entrambi.

Sistemi Simmetrici:

- I sistemi simmetrici sono veloci dal punto di vista computazionale, nel senso che i computer possono eseguire rapidamente gli algoritmi di cifratura e decifratura.

  Questi algoritmi utilizzano combinazioni di operazioni booleane (come lo **XOR**) e operazioni su matrici, calcoli che i computer sanno eseguire in maniera eccellente. In alcuni casi, è anche possibile parallelizzare alcune fasi per velocizzare ulteriormente il processo.

- Tuttavia, i sistemi simmetrici non risolvono il problema dello scambio della chiave.

Sistemi Asimmetrici:

- I sistemi asimmetrici risolvono il problema dello scambio della chiave, nel senso che non è necessario che i due utenti abbiano precedentemente condiviso una chiave segreta.

- I sistemi asimmetrici sono più lenti dal punto di vista computazionale, perché devono eseguire operazioni su numeri molto grandi (parliamo di numeri con fino a 600 cifre!).

Queste sono, intanto, le prime differenze tra sistemi asimmetrici e simmetrici, e si nota che sono complementari: difficilmente, infatti, nei progetti si utilizza solo uno dei due, perché è preferibile impiegare entrambi. Per esempio, il protocollo **HTTPS**, che serve per inviare le pagine web in modo cifrato, crea una chiave per un sistema **simmetrico**, ma la chiave viene cifrata tramite un sistema **asimmetrico**. In questo modo, i due utenti ottengono la chiave in **maniera sicura** (perché è stata inviata tramite un sistema asimmetrico), mentre la comunicazione vera e propria utilizza un sistema simmetrico, poiché è più **veloce**.

1.3.1 Sistemi Simmetrici - OTP

Tra le due categorie, i sistemi simmetrici sono i primi che vedremo, poiché sono tendenzialmente più semplici. Come si nota dal grafico, i simmetrici si suddividono in altre tre categorie: **OTP**, **Stream** e **a Blocchi**. Il primo che analizzeremo è **OTP** (*One Time Pad*), l'unico sistema definito **perfettamente sicuro**, ovvero tale per cui, partendo dal messaggio cifrato senza la chiave, è impossibile risalire al messaggio originale. Tutti gli altri sistemi, se sottoposti ad attacchi **brute-force**, permettono di risalire al messaggio originale senza la chiave di cifratura. Chiaramente, tali attacchi sono praticamente infattibili perché richiederebbero anni per essere completati; tuttavia, nell'ipotesi di disporre di un computer infinitamente potente, il cifrario OTP sarebbe l'unico **impossibile** da decifrare senza la chiave.

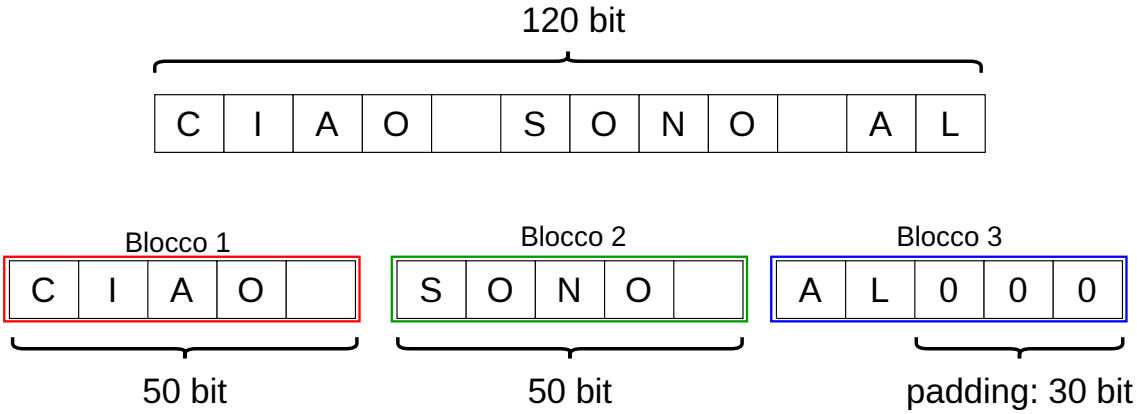
Dopo questa definizione, potreste pensare che si potrebbe usare sempre e solo l'**OTP**, ma purtroppo ha un limite che lo rende poco praticabile: è necessario cambiare la chiave dopo ogni cifratura. Infatti, se due messaggi vengono cifrati con la stessa chiave, tramite delle **criptoanalisi** è possibile risalire sia alla chiave sia ai messaggi originali. Se qualcuno pensasse di generare sempre nuove chiavi, la gestione diventerebbe eccessivamente complessa e insostenibile per la comunicazione.

1.3.2 Sistemi Simmetrici - PRNG / Stream

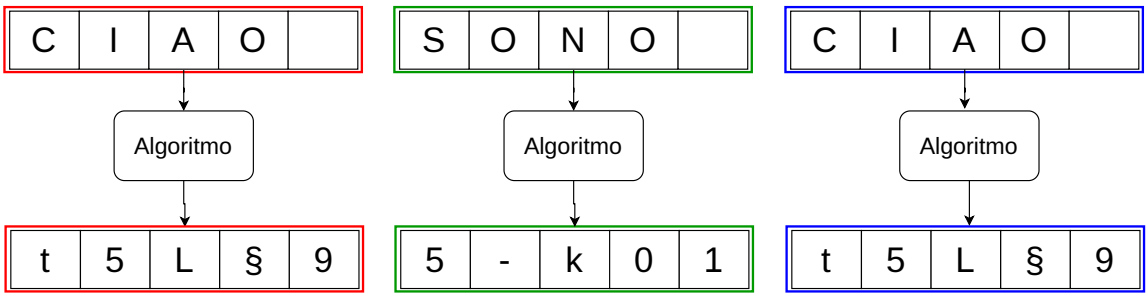
La categoria successiva è quella degli **Stream**, che però vedremo solo brevemente, poiché non sono molto utilizzati. Gli Stream funzionano generando un **flusso** (da cui il nome *Stream*) di **numeri casuali**, che vengono poi utilizzati per cifrare i messaggi tramite l'**OTP**. Il problema è che la generazione di numeri casuali per i computer è **impossibile**, poiché i computer sono sistemi **deterministici**. Per produrre numeri per questo sistema, si utilizzano algoritmi detti **PRNG** (*Pseudo Random Number Generator*), che cercano di generare numeri apparentemente casuali ma che presentano comunque correlazioni tra loro; con determinati attacchi, è possibile prevedere i numeri successivi dello stream anche senza conoscere la chiave. Per questo motivo, questa tipologia di cifrari oggi non viene quasi mai utilizzata.

1.3.3 Sistemi Simmetrici - A Blocchi

Finalmente arriviamo ai veri sistemi simmetrici: quelli a blocchi. Questi cifrari sono quelli utilizzati oggigiorno per la loro sicurezza e **velocità**. I cifrari a blocchi funzionano suddividendo il messaggio in blocchi di n bit. Supponiamo di voler cifrare un messaggio di 120 bit e di avere un cifrario a blocchi che opera su blocchi da 50 bit. In questo caso, il nostro messaggio **sarà diviso in 3 blocchi:** il primo da 50 bit, il secondo da 50 bit e l'ultimo da 20 bit. Se l'ultimo blocco non raggiunge la dimensione prevista dal cifrario, vengono aggiunti degli zeri (o un qualsiasi **padding**) in modo che raggiunga la lunghezza di 50 bit.



Questo meccanismo di creare blocchi è necessario perché ogni blocco viene poi trasformato in una **matrice**. Per questo motivo, è importante che ogni blocco abbia una dimensione definita, poiché gli algoritmi sfruttano operazioni su matrici per cifrare il messaggio. Allo stesso modo, il messaggio cifrato sarà anch'esso una matrice della stessa dimensione, che verrà poi riconvertita in messaggio. È importante comprendere che, poiché l'algoritmo cifra un blocco alla volta, lo stesso blocco in input produce sempre lo stesso blocco cifrato. Ciò significa che, se due blocchi all'interno del messaggio originale sono identici, avranno lo stesso blocco cifrato.



I blocchi cifrati vengono poi riuniti per formare il messaggio cifrato. Per evitare che due blocchi identici producano lo stesso output (cosa che potrebbe aiutare a risalire al messaggio originale), sono stati ideati i **modi di funzionamento dei cifrari a blocchi**, che vedremo con calma e in dettaglio più avanti.

La forza di questi cifrari è che utilizzano soltanto operazioni **booleane** (**and**, **or**, **xor** e **not**) e operazioni **su matrici**, il che li rende molto veloci, poiché queste operazioni sono altamente ottimizzate nei computer odierni.

I principali algoritmi a blocchi sono **DES** (*Data Encryption Standard*) e **AES** (*Advanced Encryption Standard*). Il DES nacque nel 1976 e, per i vent'anni successivi, fu lo standard per i cifrari a blocchi. Nel 1999, però, dei ricercatori riuscirono a **rompere** il cifrario, rendendolo insicuro a causa della breve lunghezza della chiave, che permetteva un attacco brute-force. Al suo posto arrivò **AES** nel 1998, che ad oggi è ancora considerato un sistema sicuro. Inoltre, fu creato il **3DES** (*Triple DES*), che consiste nel cifrare un messaggio tre volte con il DES; questo stratagemma permette ancora oggi di usare il DES in forma sicura, anche se oggi si utilizza quasi esclusivamente il 3DES e non più il DES singolo.

1.3.4 Sistemi Asimmetrici - RSA

Passando all'altro lato del nostro schema arriviamo ai sistemi **asimmetrici**. Come abbiamo gia detto, questi sistemi hanno come caratteristica principale che non hanno solamente una chiave ma ne hanno **una privata e una pubblica**. RSA (che è l'acronimo dei 3 crittografi che lo hanno inventato) è un algoritmo che permette di creare le due chiavi in maniera sicura, e permette anche ci **cifrare e decifrare** (cosa che negli algoritmi a **Scambi di chiave** non è contemplata). RSA sfrtutta la difficoltà di **Fattorizzare numeri grandi** (che come dicevamo stiamo parlando di numeri a 600 cifre circa). Infatti ad oggi, l'unico modo che abbiamo per fattorizzare un numero è provare a dividere tutti i numeri minori di esso. Quindi per i computer odierni non è sostenibile un calcolo così impegnativo.

Algorithm 1: Fattorizzazione di un numero n

```
Input: n
Output: list
list ← [];
for k ← 2 to n − 1 do
    if n mod k = 0 then
        list.append(k);
return list;
```

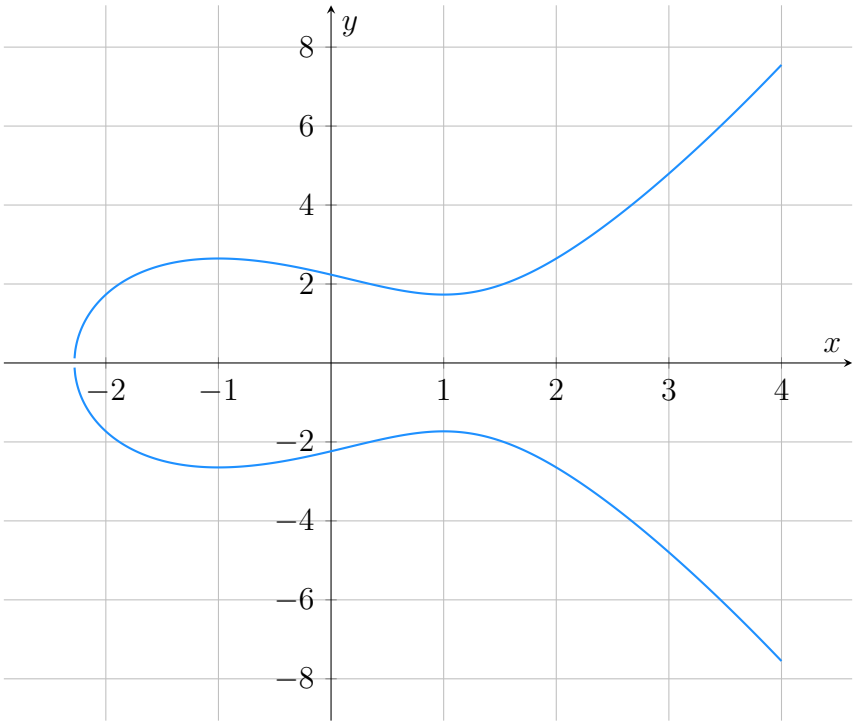
1.3.5 Sistemi Asimmetrici - Scambio di Chiavi

Tra gli algoritmi asimmetrici abbiamo anche quelli che io ho chiamato **Scambio di Chiave**, i quali si occupano esclusivamente di **generare una chiave condivisa tra due utenti**, che poi sarà utilizzata con un algoritmo simmetrico come l'**AES** o il **DES**. Questo approccio consente di combinare i vantaggi dei sistemi simmetrici (**velocità**) con quelli dei sistemi asimmetrici (**sicurezza**).

Tra questi algoritmi, i due principali sono: **Diffie-Hellman** (dal nome dei ricercatori che lo hanno inventato) e **ECC** (*Elliptic Curve Cryptography*). Diffie-Hellman, come vedremo, è molto semplice e sfrutta un meccanismo di potenze e moduli, mentre ECC è più particolare: utilizza infatti una curva ellittica definita da un'equazione del tipo seguente:

$$y^2 = x^3 + ax + b$$

che ha come grafico:



L'idea alla base di ECC è quella di **modulare la curva secondo un numero primo** p e di scegliere un punto casuale su di essa, sul quale eseguire una particolare operazione, la **somma tra punti**, che vedremo con calma, poiché risulta particolarmente complessa.

Ad ogni modo, sia Diffie-Hellman che ECC sfruttano la difficoltà di calcolare il **logaritmo discreto**. In altre parole, per noi e per i computer è semplice risolvere la seguente equazione:

$$2^x = 9$$

Infatti, per risolverlo basta calcolare il logaritmo.

$$x = log_2 9$$

Il discorso cambia però quando applichiamo questo ragionamento ai moduli (che vedremo meglio in seguito; se non li avete mai visti, potete saltare al prossimo paragrafo). Infatti, se consideriamo un' equazione del genere:

$$2^x \equiv 9 \mod 17$$

È complicatissimo risolverla, perché non è più possibile applicare il logaritmo a causa del modulo. Come per la fattorizzazione, ad oggi non esistono algoritmi che permettano di risolvere queste equazioni in tempi ragionevoli.

1.3.6 Post-Quantum

Ultimo appunto prima di partire con i cifrari antichi: voglio fare una piccola introduzione sui **Cifrari Post-Quantum**, perché gli algoritmi **RSA**, **Diffie-Hellman** e **ECC** sono attualmente molto difficili da rompere con i computer moderni, ma è possibile che, con i computer **quantistici**, possano essere violati anche in pochi secondi. Infatti, i computer quantistici sono particolarmente adatti a risolvere problemi su cui si basano questi algoritmi: ad esempio, possono fattorizzare numeri enormi o calcolare il logaritmo discreto in tempi brevissimi.

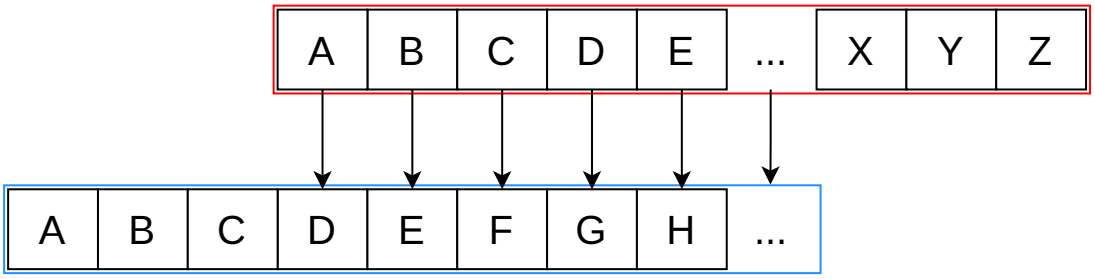
Tuttavia, ad oggi i computer quantistici sono ancora in fase di sviluppo e c'è ancora molta strada da fare: per esempio, il numero più grande mai fattorizzato con un computer quantistico è **21**. Non si può sapere come saranno i computer quantistici in futuro; per questo motivo, i maggiori ricercatori stanno lavorando per creare nuovi cifrari **difficili da rompere anche per computer quantistici**.

Tra questi, quelli che stanno ricevendo maggiore attenzione dalla comunità scientifica sono i **Lattice-Based**. Si tratta di algoritmi che sfruttano i **reticoli** (in inglese *Lattice*), ovvero "piani cartesiani" a n dimensioni, ad esempio $\mathbb{Z}^n$.

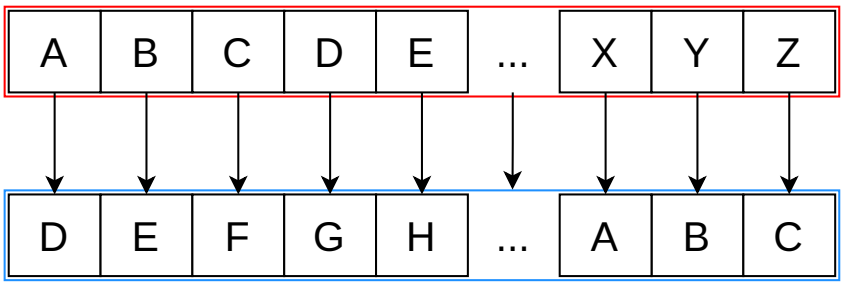
Si possono generare dei vettori linearmente indipendenti a n dimensioni. A partire da questi, è possibile definire un punto e determinare il percorso più semplice per raggiungerlo usando i nuovi vettori. Ad ogni modo, questi cifrari sono particolarmente complessi, ma rappresentano anche il futuro della crittografia. Per questo motivo, in questo corso non li tratteremo, ma consiglio a tutti di studiarli autonomamente.

2 Cifrari Antichi - Cesare

Cominciamo ad analizzare i primi cifrari, iniziando dal più antico mai inventato: infatti, come avevamo già detto nell'introduzione, il **Cifrario di Cesare** risale all'Impero romano. Il cifrario fu ideato per inviare ai legionari informazioni per proseguire la battaglia. Chiaramente non potevano mandare i messaggi in chiaro, anche perché, se gli avversari ne venivano a conoscenza, potevano rispondere con una contromossa. Per questo **Giulio Cesare** inventò un cifrario che si basava sullo spostamento di un determinato numero di posizioni delle lettere nell'alfabeto. Un classico esempio del cifrario di Cesare è quello di **spostare di 3 posizioni indietro**.

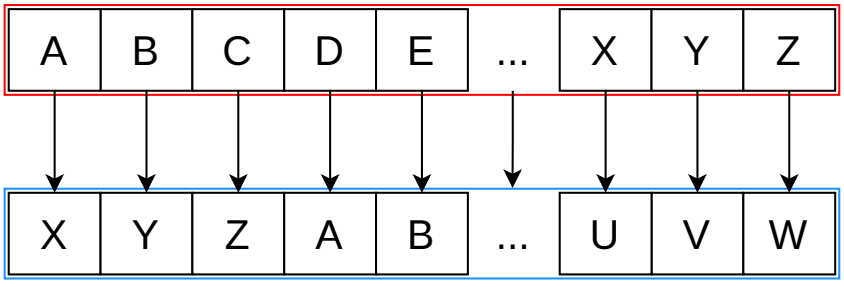


Le lettere che "escono a sinistra" le associamo alle ultime tre lettere.



Per cifrare un messaggio basta prendere ogni lettera del testo in chiaro e sostituirla con la lettera corrispondente nella tabella. Ad esempio, se volessimo cifrare la parola **Abbecce**, con il cifrario di Cesare diventerebbe **Deehffh**.

Per decifrare, invece, basta applicare lo stesso procedimento, ma spostando l'alfabeto verso destra. Con questo esempio, il testo decifrato sarà:



In questo esempio abbiamo spostato l'alfabeto di tre posizioni, ma questo è un numero che possiamo scegliere liberamente. Secondo la tradizione, Giulio Cesare utilizzava il numero tre, ma si può scegliere qualsiasi valore di spostamento. Poiché questo numero è decisivo per la cifratura, il numero di spostamenti non è altro che la **chiave** del sistema. Chiaramente, a chiavi diverse corrispondono cifrari diversi e, di conseguenza, differenti cifrature dei messaggi.

Una possibile implementazione in Python del cifrario di Cesare con 3 spostamenti:

```
def Cifratura_Cesare(mess):
    return "".join([chr(((ord(char) - ord('a') + 3) % 26) + ord('a'
        )) for a in mess])

def Decifratura_Cesare(mess):
    return "".join([chr(((ord(char) - ord('a') - 3) % 26) + ord('a'
        )) for a in mess])
```

Online, questa cifratura viene anche chiamata **ROT** (abbreviazione di *ROTation*), seguita dal numero di spostamenti. Quindi, **ROT3** corrisponde a 3 spostamenti (come negli esempi precedenti), mentre **ROT15** sposterà di 15 posizioni. In Python, un'implementazione per un generico **ROT-n** potrebbe essere:

```
def Cifratura_ROTn(mess, n):
    return "".join([chr(((ord(char) - ord('a') + n) % 26) + ord('a'
        )) for a in mess])

def Decifratura_ROTn(mess, n):
    return "".join([chr(((ord(char) - ord('a') - n) % 26) + ord('a'
        )) for a in mess])
```

Quindi, se qualcuno vuole usare questo cifrario, basta scegliere un numero tra 1 e 25 e tenerlo segreto. Tuttavia, questo metodo è soggetto ad attacchi **brute-force**: come abbiamo detto, questo cifrario ha al massimo **25 chiavi possibili**, quindi qualcuno potrebbe creare un programma che provi tutte le combinazioni. Un possibile programma in Python:

```
def attacco_rotn(messDaDecifrare):
    for i in range(1, 26):
        print(Decifratura_ROTn(messDaDecifrare, i))
```

2.1 Cifrari Monoalfabetici

In realtà, l'attacco brute-force non è l'unico al quale è soggetto questo cifrario. Questo tipo di cifrario è definito **monoalfabetico** perché la stessa lettera viene sempre cifrata con la stessa lettera corrispondente. Tutti i cifrari monoalfabetici sono soggetti ad attacchi detti **Letter Frequency**. Ciò significa che la **percentuale** di una lettera nel testo originale è la stessa della lettera corrispondente nel testo cifrato. Facciamo un esempio con **Abbcccdddd**, che, cifrato con ROT3, diventa **Deefffgggg**.

Abbcccdddd $\Longrightarrow$ Deeffgggg

Lettera	**percentuale**
A	10%(1/10)
B	20%(2/10)
C	30%(3/10)
D	40%(4/10)

Lettera	**percentuale**
D	10%(1/10)
E	20%(2/10)
F	30%(3/10)
G	40%(4/10)

Questa caratteristica dei sistemi monoalfabetici può essere sfruttata per decifrare, anche parzialmente, il messaggio. Questo perché possiamo creare delle tabelle con la frequenza di ogni lettera per ciascuna lingua. Ad esempio, la **tabella di frequenza** della lingua italiana è:

Lettera	Frequenza
E	11.49 %
A	10.85 %
I	10.18 %
O	9.97 %
N	7.02 %

...

link alla tabella completa : https://www.sttmedia.com/characterfrequency-italian

Questa tabella indica che, in media, un messaggio o una parola in italiano ha quelle probabilità di contenere le rispettive lettere. Quindi, perché stiamo facendo tutto questo discorso? Perché, se facciamo la stessa analisi anche sul messaggio cifrato e confrontiamo le lettere più frequenti nel messaggio cifrato con la tabella sopra, è possibile identificare alcune lettere. Chiaramente, più è lungo il messaggio, più è probabile indovinare le lettere.

N.B. Chiaramente, esistono tabelle di frequenza per ogni lingua; qui ho riportato quella italiana come esempio.

Facciamo un esempio pratico: supponiamo di avere il seguente messaggio:

hlvc irdf uvc crxf uz Tfdf, tyv mfcxv r dvqqfxzfief, kir ulv trkvev efe zekviifkkv uz dfekz, klkkf r jvez v r xfcwz, r jvtfeur uvccf jgfixviv v uvc izvekiriv uz hlvccz, mzve, hlrjz r le kirkkf, r izjkizexvijz, v r giveuvi tfijf v wzxlir uz wzldv, kir le gifdfekfizf r uvjkir, v le'rdgzr tfjkzvir urcc'rckir grikv; v zc gfekv, tyv zmz tfexzlexv cv ulv izmv, gri tyv iveur retfi gzu jvejzszcv rcc'fttyzf hlvjkr kirjwfidrqzfev, v jvxez zc glekf ze tlz zc crxf tvjjr, v c'Ruur izetfdzetzr, gvi izgzxczri gfz efdv uz crxf ufmv cv izmv, rccfekrereufjz uz elfmf, crjtzre c'rthlr uzjkveuvijz v irccvekrijz ze elfmz xfcwz v ze elfmz jvez.

Possiamo fare un'analisi di frequenza su questo messaggio e confrontarla con la **tabella di frequenza** della lingua italiana.

Messaggio cifrato

Lettera	Frequenza
V	12.12%
Z	11.52%
R	9.90%
F	9.09%
E	8.28%
I	7.88%
C	6.46%
K	5.45%
U	4.44%
J	4.44%
L	4.04%
T	3.64%
X	2.83%
G	2.42%
D	2.02%
M	1.82%
H	1.01%
W	1.01%
Y	0.81%
Q	0.61%
S	0.20%

Tabella di frequenza

Lettera	Frequenza
E	11.49%
A	10.85%
I	10.18%
O	9.97%
N	7.02%
T	6.97%
R	6.19%
L	5.70%
S	5.48%
C	4.30%
D	3.39%
U	3.16%
P	2.96%
M	2.87%
V	1.75%
G	1.65%
H	1.43%
B	1.05%
F	1.01%
Z	0.85%
Q	0.45%

Con questo, capiamo che molto probabilmente una lettera tra **V**, **Z** oppure **R** nel testo cifrato corrisponderà alla **E** nel messaggio in chiaro. Chiaramente, questo ragionamento si può applicare a tutte le altre lettere, le quali, con buona probabilità, corrisponderanno a una delle lettere con percentuali simili.

N.B. Usando l'alfabeto italiano, le lettere straniere (**K**, **J**, **W**, **X** e **Y**) non sono presenti nella tabella di frequenza. Nel messaggio cifrato che ho creato, queste lettere possono apparire, ma mancheranno 5 lettere dell'alfabeto italiano perché non hanno corrispondenza con le lettere straniere mancanti.

Inoltre, si possono fare delle **osservazioni linguistiche**. Per esempio, in italiano solamente le lettere **A**, **E** e **O** possono stare da sole (per indicare le relative funzioni di preposizione). Quindi, nel messaggio cifrato possiamo sicuramente dire che le lettere **V** e **R** possono corrispondere solamente a **A**, **E** oppure **O**, il che combacia anche con le percentuali delle tabelle. Ad ogni modo, proviamo a sostituire nel messaggio cifrato le lettere con la **percentuale** più simile:

hder tivo ser ripo sa tovo, ufe gorpe i vezzopaotno, lti sde uilene non anlettolle sa vonla, ldllo i cena e i porba, i ceuonsi serro cmotpete e ser taenltite sa hderra, gaen, hdica i dn ltillo, i tacltanpetca, e i mtenset uotco e bapdti sa badve, lti dn mtovonlotao i seclti, e dn'ivmai uoclaeti sirr'irlti mitle; e ar monle, ufe aga uonpadnpe re sde tage, mit ufe tensi inuot mas cencaqare irr'ouufao hdecli lticbotvizaone, e cepna ar mdnlo an uda ar ripo uecci, e r'rssi tanuovanuai, met tamaprait moa nove sa ripo soge re tage, irronlininsoca sa ndogo, ricuain r'iuhdi saclensetca e tirrenlitca an ndoga porba e an ndoga cena.

Ora, come ora, il testo non sembra molto chiaro, ma possiamo iniziare a sistemarlo. Per esempio, come dicevamo prima, le lettere singole possono essere solamente **E**, **A** o **O**. Attualmente, però, abbiamo **I** e **E**, quindi dobbiamo sostituire la **I**. Poiché la **I** è molto vicina in percentuale (secondo la tabella di frequenza) alla **A**, proviamo a invertire le due lettere. In questo modo, il testo diventa:

hder tavo ser rapo si tovo, ufe gorpe a vezzopiotno, lta sde ualene non inlettolle si vonli, ldllo a ceni e a porbi, a ceuonsa serro cmotpete e ser tienltate si hderri, [...]

Ora possiamo notare che ci sono molti **SI**, che molto probabilmente possono essere **DI**. Quindi, proviamo a invertire **D** e **S**.

hser tavo der rapo di tovo, ufe gorpe a vezzopiotno, lta dse ualene non inlettolle di vonli, lsllo a ceni e a porbi, a ceuonda derro cmotpete e der tienltate di hserri, [...]

Ora siamo abbastanza sicuri che le lettere **D**, **A**, **I** e **E** siano al posto corretto. Possiamo continuare a fare delle analisi e, man mano, dedurre il testo originale. Per esempio, nella prima riga c'è **DSE**, che molto probabilmente potrebbe essere **DUE** (visto che la **D** e la **E** siamo sicuri che siano corrette). Oppure, la parola **vezzopiotno** potrebbe essere **mezzogiorno**, e così via. Se vuoi, puoi provare a risolvere questo enigma da solo, ho creato un codice Python:

link: https://github.com/AlexBro98LoVero/Dispense/blob/main/Giochi/1_1_giocoParole.py

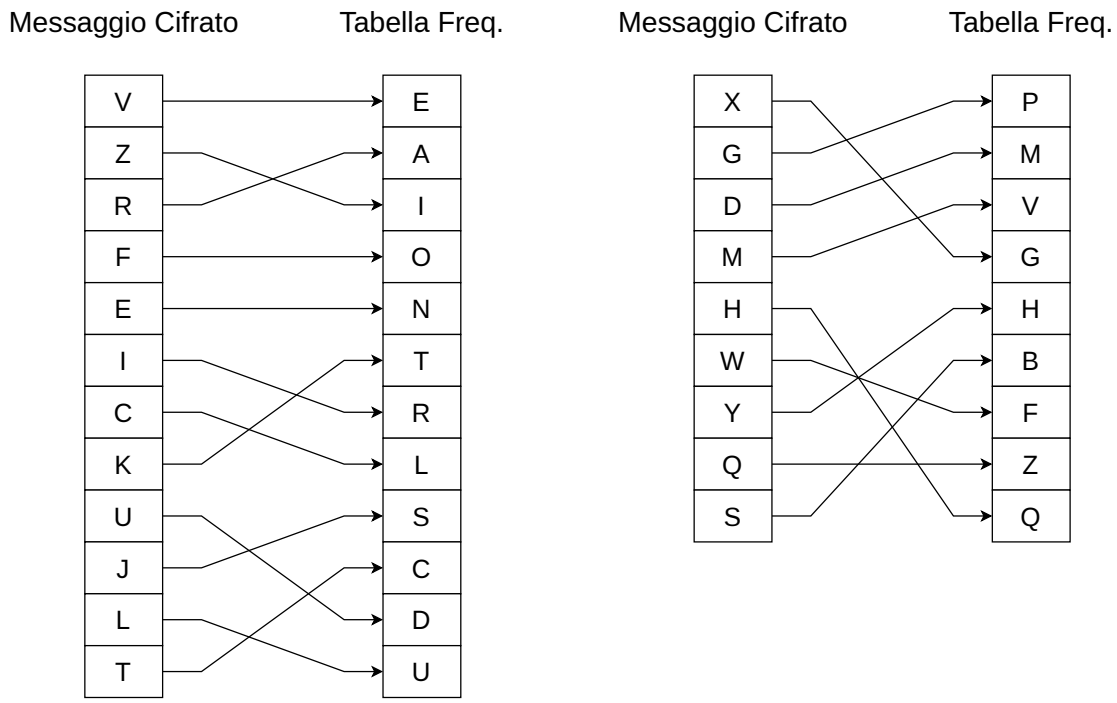
altrimenti, continua che ora c'è la soluzione.

Ad ogni modo, con un po' di pazienza potreste vedere che il testo cifrato qui sopra non è altro che i primi versi dei **Promessi Sposi**.

quel ramo del lago di como, che volge a mezzogiorno, tra due catene non interrotte di monti, tutto a seni e a golfi, a seconda dello sporgere e del rientrare di quelli, vien, quasi a un tratto, a ristringersi, e a prender corso e figura di fiume, tra un promontorio a destra, e un'ampia costiera dall'altra parte; e il ponte, che ivi congiunge le due rive, par che renda ancor piu sensibile all'occhio questa trasformazione, e segni il punto in cui il lago cessa, e l'adda rincomincia, per ripigliar poi nome di lago dove le rive, allontanandosi di nuovo, lascian l'acqua distendersi e rallentarsi in nuovi golfi e in nuovi seni.

N.B. Per semplicità, ho messo tutto in minuscolo e ho rimosso le lettere accentate, ma chiaramente, se contemplate, possono essere d'aiuto per scoprire parole all'interno del testo.

Ora vediamo quanto erano corrette le corrispondenze.



Si può notare che, nel complesso, questa tecnica ha indovinato quasi tutte le lettere; infatti, molte erano sbagliate di una sola posizione.

- Indovinate: $4/21 \approx 19.0$ %
- Sbagliato di una casella: $10/21 \approx 47.6$ %
- Sbagliato di due caselle: $5/21 \approx 23.8$ %
- sbagliato di tre caselle: $1/21 \approx 4.8$ %
- sbagliato di quattro caselle: $1/21 \approx 4.8$ %

3 Cifrati Antichi - Enigma

La tecnica **OTP**, per come l'abbiamo studiata fino ad ora, è utilizzabile solo tramite i computer, ma in realtà è molto più antica e, durante la **Seconda Guerra Mondiale**, ha giocato un ruolo fondamentale. Infatti, i tedeschi hanno utilizzato la **Macchina Enigma**, che permetteva loro di comunicare inviando messaggi cifrati. La macchina Enigma, per l'appunto, usa una tecnica di crittografia molto simile, se non uguale, all'**OTP**.

Foto della macchina Enigma

3.1 Preambolo Storico

La macchina Enigma è stata brevettata da **Arthur Scherbius**, un ingegnere tedesco, nel **1918**. Al contrario di quanto si pensa, Enigma veniva utilizzata anche prima della Seconda Guerra Mondiale, soprattutto dalla **marina militare tedesca**. Tuttavia, Enigma ha avuto un ruolo importantissimo durante la Seconda Guerra Mondiale, perché ha permesso ai tedeschi di inviare messaggi sicuri, impedendo agli Alleati di comprendere le decisioni militari tedesche.

La macchina Enigma aveva una **chiave privata** (che vedremo in dettaglio più avanti) che veniva cambiata **ogni giorno a mezzanotte**. La chiave veniva inviata soltanto agli ufficiali tedeschi di grado più alto per evitare fughe di informazioni riservate. Infatti, verso mezzanotte, ai generali tedeschi arrivava un messaggio chiamato **Schlüsselheft**, che conteneva la chiave del giorno. In realtà, alcuni reparti tedeschi, come la marina militare, che richiedevano maggiore sicurezza, cambiavano chiave ogni **8 ore**. Inoltre, erano previsti protocolli di sicurezza, come il fatto che il **Schlüsselheft** dovesse essere conservato in un luogo sicuro sotto chiave, con accesso riservato solo a poche persone autorizzate. Se vi era il rischio che gli Alleati potessero entrarne in possesso, il **Schlüsselheft** doveva essere immediatamente distrutto tramite il fuoco.

3.2 Funzionamento

La macchina Enigma serviva per crittografare e decrittografare i messaggi. Era dotata di una tastiera e, alla pressione di un tasto (dove ogni tasto rappresentava una lettera dell'alfabeto tedesco), si accendeva un led che indicava un'altra lettera dell'alfabeto. Per utilizzare una macchina Enigma, di solito erano necessarie due persone: una che scriveva il messaggio da cifrare o decifrare e un'altra che annotava le lettere che si illuminavano.

Nell'immagine possiamo notare che, alla pressione del tasto **I** sulla tastiera, si illumina la lettera **U** nella parte superiore.

3.3 Struttura

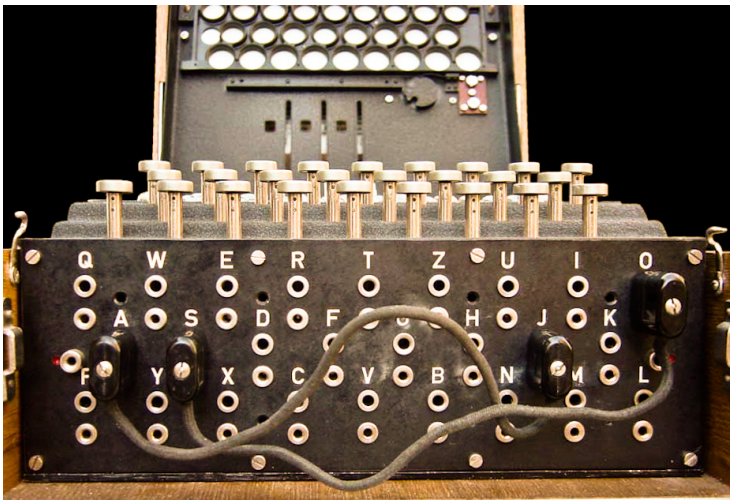
Ogni macchina Enigma era composta da sei parti principali: **Tastiera**, **Rotori**, **Riflettore**, **Lampboard**, **Plugboard** e la **Batteria**.

Partiamo dagli elementi più semplici: la batteria serviva ad alimentare la parte elettrica della macchina, mentre la tastiera conteneva le 26 lettere dell'alfabeto sotto forma di pulsanti. La tastiera funzionava in modo simile a quelle odierne, nel senso che, alla pressione di un tasto, si chiudeva il circuito elettrico.

La **Lampboard** (detta **Lampenbrett** in tedesco) era la parte in cui si trovavano le lettere che si illuminavano alla pressione dei tasti sulla tastiera.

3.3.1 Plugboard

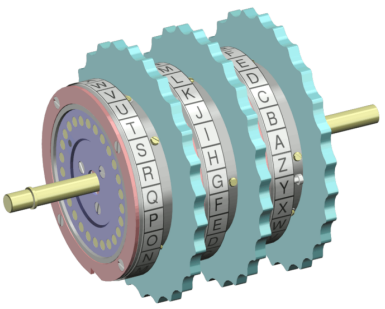
La **Plugboard** (detta anche **Steckerbrett**) serviva per invertire una coppia di lettere. Questa inversione avveniva sia in input (quindi prima di eseguire l'algoritmo di cifratura) sia in output (dopo aver cifrato il messaggio). Questa caratteristica serviva esclusivamente ad aggiungere un ulteriore livello di sicurezza, ma non era il punto focale della macchina.



Nell'immagine possiamo notare come siano collegate le coppie (**A**, **J**) e (**S**, **O**). Quindi, se veniva premuto il tasto **A** sulla tastiera, esso veniva prima trasformato in **J**, che poi seguiva il resto dell'algoritmo. Allo stesso modo, se la lettera crittata era **S**, questa veniva cambiata in **O**, e quindi si illuminava la lettera **O** sulla lampboard.

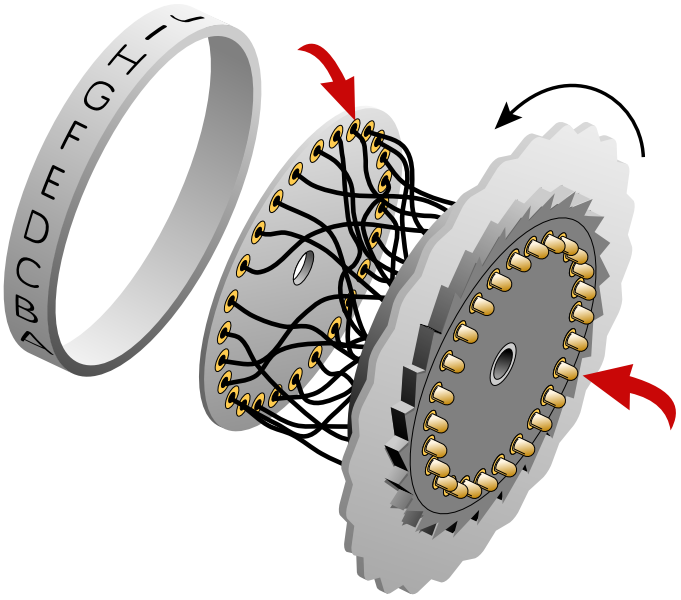
3.3.2 Rotori

La parte principale dell'algoritmo di cifratura sono **i rotori**, che permettono di trasformare le lettere per crittarle. Un rotore è un componente metallico circolare collegato a un ingranaggio che gli permette di ruotare e cambiare lettera, come vedremo in seguito. Inoltre, è montato su un asse che ne consente la rotazione e, su entrambi i lati, presenta 26 punti di contatto metallici che permettono di trasmettere la corrente tra un rotore e l'altro.



Render 3D di tre rotori collegati

Per comprendere meglio il funzionamento di un rotore, è utile analizzarne la struttura interna:



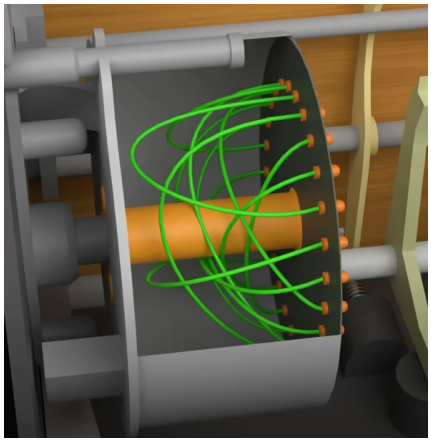
Come accennato in precedenza, entrambi i lati del rotore presentano piastre metalliche per la conduzione della corrente (indicate dalla freccia rossa nell'immagine). Inoltre, ogni punto di contatto su un lato è collegato a un punto sull'altro lato, ma in modo disordinato. Questo collegamento permette di mescolare le lettere: una lettera che "entra" nel rotore ne "uscirà" come un'altra, a causa di questo rimescolamento.

Di norma, una macchina Enigma contiene tre rotori collegati in serie per aumentare la complessità della cifratura. Tuttavia, alcune versioni della macchina Enigma utilizzano quattro o più rotori, semplicemente per incrementare la sicurezza e rendere ancora più difficile la decrittazione.

3.3.3 Riflettore

Per aumentare ulteriormente la sicurezza, i tre rotori sono collegati a un **Riflettore**, il quale fa "rientrare" il segnale nei rotori, permettendo un'ulteriore trasformazione delle lettere. Questo avviene perché la corrente attraversa nuovamente tutti e tre i rotori in senso inverso.

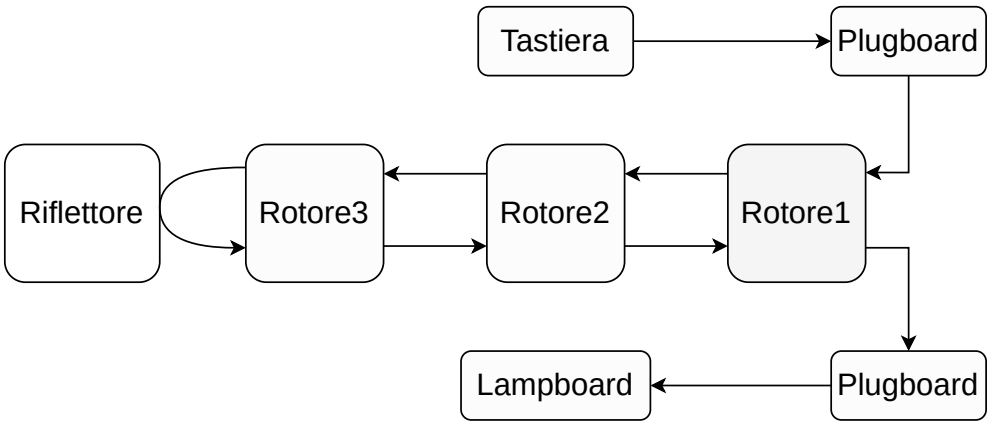
In questo modo, una lettera viene cifrata **sei volte** prima di essere visualizzata sulla lampboard.



Render 3D di un Riflettore

3.4 Funzionamento

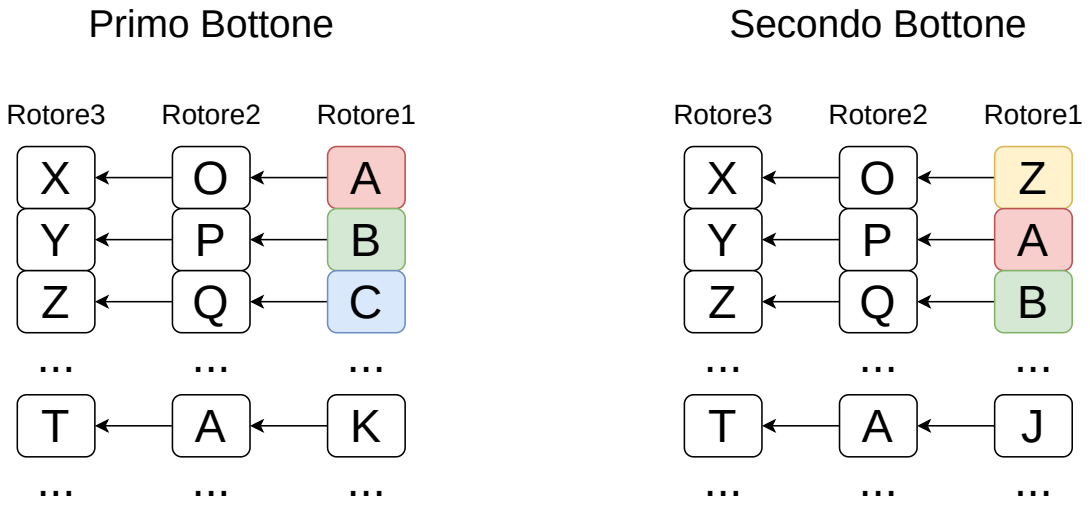
Ripercorriamo il funzionamento della macchina Enigma.



Quando viene premuto un tasto sulla **Tastiera**, il segnale passa attraverso la **Plugboard**, che sostituisce la lettera con quella associata se un cavo è collegato; altrimenti, il segnale prosegue direttamente verso i **Rotori**. Successivamente, attraversa i tre rotori, modificando la lettera tre volte. Grazie al **Riflettore**, il segnale viene reinviato attraverso i rotori, che trasformano nuovamente la lettera per altre tre volte. Infine, il segnale passa nuovamente per la **Plugboard**, che può applicare un'ulteriore sostituzione se la lettera è collegata a un'altra. Dopo tutte queste trasformazioni, la lettera finale viene visualizzata sulla **Lampboard**.

Durante questo processo, la lettera viene trasformata un minimo di **6 volte** e fino a un massimo di **8 volte** se la **Plugboard** modifica sia l'input che l'output.

Non appena il tasto premuto viene rilasciato, il primo rotore ruota di una posizione grazie a un ingranaggio, modificando così la configurazione delle lettere. Questo meccanismo assicura che, anche premendo due volte di seguito la stessa lettera, vengano visualizzate lettere completamente diverse, poiché la rotazione del rotore cambia continuamente le combinazioni possibili.



4 One Time Pad

La prima forma di crittografia che vediamo è **One Time Pad**, che è una crittografia **Simmetrica**. Con il termine crittografia simmetrica si intendono tutte le tecnite crittografiche che usano una sola chiave per criptare e decriptare, al contrario della crittografia asimmetrica che ne usa due, ma questo lo vedremo bene più avanti. Prima di capire bene il funzionamento di questa tecnica è meglio rivedere cos'è lo **XOR** e come funziona.

4.1 XOR

Lo **XOR** (*eXclusive OR*) è un operatore booleano binario la cui tabella di verità è la seguente:

A	B	XOR(A, B)
0	0	0
0	1	1
1	0	1
1	1	0

Per ricordarla a memoria, basta sapere che, se A e B sono diversi, lo XOR restituisce 1; altrimenti, se sono uguali, restituisce 0.

Ci interessano in particolare le seguenti proprietà dello XOR:

$$(A \oplus B) \oplus B = A$$

$$A \oplus 0 = A$$

dove $\oplus$ è il simbolo dello XOR. In realtà, queste proprietà sono tutto ciò che ci serve per comprendere il nostro algoritmo crittografico, quindi passiamo ad analizzarne il funzionamento.

4.2 Funzionamento

L'algoritmo **OTP** inizia generando una chiave lunga almeno quanto il messaggio da cifrare. La chiave può assumere qualsiasi forma, purché sia privata e nessuno la conosca. Dopo la generazione della chiave, è sufficiente XORarla con il messaggio che vogliamo inviare:

$$C = M \oplus K$$

dove C è il messaggio cifrato (*Crypted message*), M è il messaggio originale (*Message*) e K è la chiave (*Key*).

Una volta inviato il messaggio, il destinatario può semplicemente applicare l'operazione XOR tra il messaggio ricevuto e la chiave per recuperare il messaggio originale:

$$C \oplus K$$

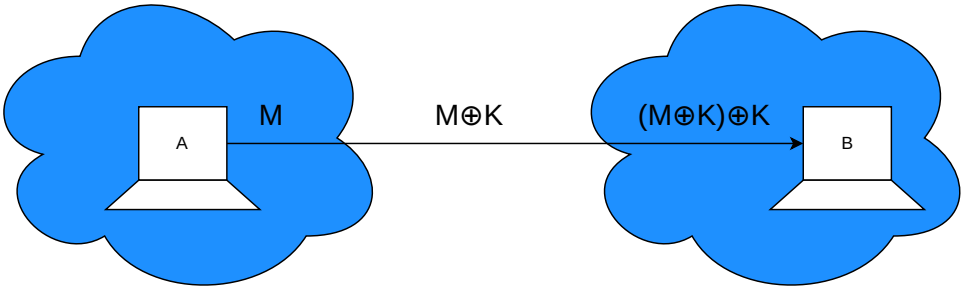
Per capire perché il metodo funziona, sostituiamo l'equazione precedente ($C = M \oplus K$):

$$(M \oplus K) \oplus K$$

Utilizzando la proprietà dello XOR, otteniamo:

$$M = (M \oplus K) \oplus K$$

In questo modo, applicando l'operazione XOR, recuperiamo esattamente il messaggio originale.



4.2.1 Considerazioni

Questo algoritmo, però, non si occupa della condivisione sicura della chiave. Per questo scopo si può utilizzare l'algoritmo **Diffie-Hellman**, che è progettato proprio per condividere chiavi private in modo sicuro.

La forza e la bellezza di questo algoritmo risiedono nel fatto che è **completamente sicuro**: è stato infatti dimostrato matematicamente che è impossibile decifrare il messaggio cifrato ($M \oplus K$) senza conoscere la chiave. L'**OTP** è, infatti, l'unico algoritmo **impossibile da decifrare** e completamente sicuro che conosciamo al momento. L'unico metodo per indovinare il messaggio è tentare casualmente e sperare in un colpo di fortuna.

Ovviamente, non è tutto oro ciò che luccica: se fosse davvero perfetto, useremmo solo questa tecnica e saremmo al 100% sicuri. Tuttavia, come vedremo, questo algoritmo presenta un problema non da poco.

4.3 Crackabilità

Il problema di questa tecnica risiede nel nome: **ONE TIME Pad**. Ovvero, questa tecnica si può usare **una sola volta** con la stessa chiave. Questo è un requisito fondamentale, perché qualora una chiave venga usata più di una volta, tramite varie tecniche si può estrapolare la chiave e i messaggi, almeno parzialmente.

Questo accade perché, dati due messaggi in chiaro ($\mathbf{M_1}$, $\mathbf{M_2}$), una chiave privata ($\mathbf{K}$) e le loro combinazioni cifrate ($\mathbf{C_1}$, $\mathbf{C_2}$):

$$C_1 = M_1 \oplus K$$
$$C_2 = M_2 \oplus K$$

Supponiamo di aver intercettato i messaggi cifrati: possiamo applicare l'operazione XOR tra di loro:

$$C_1 \oplus C_2$$

Riscriviamo l'espressione sostituendo i valori di C_1 e C_2:

$$C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_2 \oplus K)$$

Ora, grazie alla proprietà associativa dello XOR, possiamo riorganizzare i termini:

$$C_1 \oplus C_2 = (M_1 \oplus M_2) \oplus (K \oplus K)$$

Poiché $K \oplus K = 0$, otteniamo:

$$C_1 \oplus C_2 = (M_1 \oplus M_2) \oplus 0$$

Infine, ricordando che $X \oplus 0 = X$, il risultato finale è:

$$C_1 \oplus C_2 = M_1 \oplus M_2$$

Dopo tutti questi passaggi, abbiamo scoperto che la XOR tra i du.

e messaggi cifrati è uguale alla XOR dei due messaggi in chiaro. In questo modo, siamo riusciti a "rimuovere" la chiave. Chiaramente, con la XOR dei messaggi in chiaro, comunque non riusciamo a decriptarli, ma possiamo dedurre alcune parti. Infatti, esistono alcuni algoritmi, come **crib drag**, che, dati alcuni messaggi cifrati, possono decifrare parti dei messaggi e della chiave. Questi algoritmi sono particolarmente complessi e sfruttano certi pattern nelle frasi (come banalmente gli spazi, oppure gli articoli nelle varie lingue) e anche degli **attacchi con dizionario** (dictionary attack). Per capirli bene e in maniera pratica, consiglio di scaricare i seguenti script Python: https://github.com/CameronLonsdale/MTP. Inoltre, per gli amanti del rap, vi lascio un esempio divertente per cercare di capire quale canzone è: https://github.com/AlexBro98LoVero/Dispense/blob/main/Giochi/3_1_mtptxt

5 Stream

Il prossimo algoritmo che vedremo è quello degli **Stream**. Prima di parlarne, però, dobbiamo capire cosa sono i **Pseudo-Random Number Generator**, perché sono la base degli stream, anche se inizialmente possono sembrare non avere nulla a che fare con essi.

5.1 PRNG

Per chi ha già creato qualche programma in **C/C++**, può darsi che abbia dovuto **generare dei numeri casuali**. Tuttavia, è difficile crearli perché i computer sono sistemi **deterministici**, cioè ad ogni input corrisponde sempre lo stesso output. Quindi, per i computer è impossibile generare numeri realmente casuali; per questo motivo sono stati inventati gli **Pseudo-Random Number Generator**, ovvero dei generatori di numeri **pseudo-casuali**, che a noi **sembrano** casuali, ma che in realtà sono deterministici.

Un primo esempio è il **Linear Congruential Generator** (LCG), che parte da alcuni valori iniziali X_0, a, c e m (X_0 è detto **seme** o **seed**), i quali possono essere scelti a piacere. A partire da questi valori, possiamo generare nuovi numeri pseudo-casuali con la seguente formula:

$$X_{n+1} = aX_n + c \pmod{m}$$

Per chi non lo sapesse, il simbolo $\pmod{n}$ indica il **resto della divisione** tra due numeri. Per esempio, $9 \pmod{4}$ significa che devo fare la divisione $9/4$ e considerare solo il resto. In questo caso, $9/4 = 2$ con resto 1, quindi $9 \pmod{4} = 1$. Riprenderemo questi concetti nei capitoli dedicati a **RSA** e **Diffie-Hellman**.

Per esempio, se scegliamo come valori iniziali $m = 9$, $a = 4$, $c = 1$ e come seme $X_0 = 3$, otteniamo la seguente sequenza di numeri pseudo-casuali:

$$\begin{aligned} X_1 &= 4 \cdot 3 + 1 \pmod{9} \\ &= 13 \pmod{9} \\ X_1 &= 4 \end{aligned}$$

$$\begin{aligned} X_2 &= 4 \cdot X_1 + 1 \pmod{9} \\ &= 4 \cdot 4 + 1 \pmod{9} \\ &= 17 \pmod{9} \\ X_2 &= 8 \end{aligned}$$

$$\begin{aligned} X_3 &= 4 \cdot 8 + 1 \pmod{9} \\ &= 33 \pmod{9} \\ X_3 &= 6 \end{aligned}$$

Per questa configurazione, i primi valori generati dalla sequenza sono:

$$4, 8, 6, 7, 2, 1, 5, 3, 4...$$

Si nota che, all'apparenza, sembra una sequenza casuale di numeri, ma in realtà deriva da una formula e da alcune variabili fissate inizialmente (a, m, c e X_0).

Un problema di questi generatori è che sono **ciclici**, nel senso che, prima o poi, la sequenza si ripete. Per esempio, se continuiamo la sequenza di prima:

$$\underline{4, 8, 6}, 7, 2, 0, 1, 5, 3, \underline{4, 8, 6}, 7, 2, 0, 1, 5, 3, \underline{4, 8}$$

Notiamo che questa sequenza è lunga 8 numeri, perché poi si ripete. Questo problema è dovuto al fatto che abbiamo scelto numeri **piccoli**; se invece scegliamo numeri più grandi, ad esempio a 10 o 15 cifre, la ciclicità sarà molto più ampia.

Questo sistema di generazione, però, ha un altro problema: dati alcuni output, è possibile ricavare i parametri (a, m e c). Infatti, se prendiamo quattro numeri **successivi**, che chiamiamo X_α, $X_{\alpha+1}$, $X_{\alpha+2}$ e $X_{\alpha+3}$, possiamo metterli a sistema.

$$\begin{cases} X_{\alpha+1} = aX_\alpha + c \pmod m \\ X_{\alpha+2} = aX_{\alpha+1} + c \pmod m \\ X_{\alpha+3} = aX_{\alpha+2} + c \pmod m \end{cases}$$

In questo caso, notiamo che abbiamo un sistema di 3 equazioni con 3 incognite, quindi possiamo risolverlo e calcolare a, m e c. In realtà, il fatto che ci sia il modulo rende tutto più complicato, ma comunque è possibile risolverlo.

Se vi state chiedendo perché è un enorme problema poter dedurre i parametri, la risposta è la seguente: per quanto vedremo tra poco con gli **stream**, è obbligatorio che la sequenza sia imprevedibile, perché quei numeri saranno le **chiavi private** per gli stream. Pertanto, è necessario che sia impossibile generare le chiavi successive (ovvero i numeri successivi nella sequenza).

Per giocare un po' con il **LCG**, vi fornisco due link: uno per generare la sequenza di numeri e un altro link che, dati 4 output, calcola a, m e c.

link sequenza: https://github.com/AlexBro98LoVero/Dispense/blob/main/Giochi/4_1_generaSequenza.py

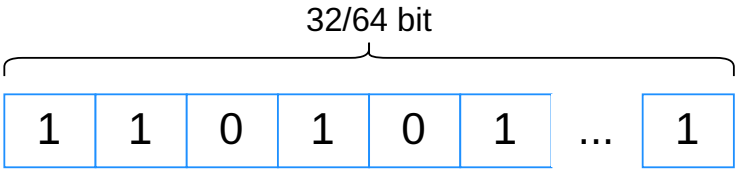
link 4 output: https://github.com/AlexBro98LoVero/Dispense/blob/main/Giochi/4_2_trovareParametri.py

N.B. Può succedere che si trovino più parametri validi; questo è dovuto al fatto che si tratta di equazioni modulari. Chiaramente, per scoprire quali sono i parametri corretti, bisognerebbe avere altri output e verificare per quali parametri funzionano.

Il **LCG** era utilizzato fino a **Java 17** (anno di pubblicazione 2020) come metodo per generare numeri nella libreria standard.

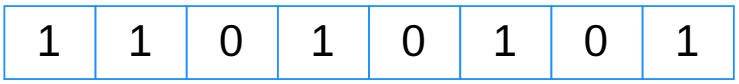
5.2 LFSR

Un'altra tecnica per generare numeri casuali è quella del **Linear Feedback Shift Register** (*LFSR*). Questo metodo parte definendo un **registro**, ovvero un **array** o **buffer** di bit, inizializzati a un **valore iniziale**, che fungerà da **chiave privata** del nostro sistema. La lunghezza del registro è arbitraria, ma di solito è di **32 bit** oppure **64 bit**.



Poi si scelgono dei **tap**, ovvero delle posizioni del buffer. Ad esempio, supponiamo di scegliere le posizioni **2**, **3** e **5**. Il numero di tap può variare. Queste posizioni saranno usate perché i bit in quelle posizioni verranno **xorati** tra loro; il risultato andrà in testa al registro e tutti i bit verranno spostati verso destra (e l'**ultimo** bit andrà perso).

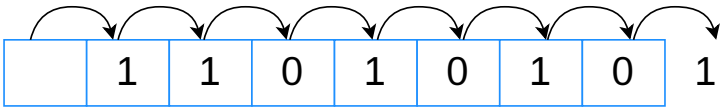
Per esempio, supponiamo di avere un registro a 8 bit con il seguente stato iniziale:



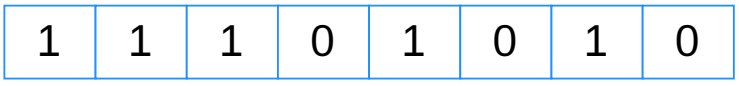
Supponiamo che i **tap** siano: **2**, **3** e **5**. Allora, il numero successivo sarà dato dallo **xor** di questi tre bit.

$$\text{nuovo bit} = \text{bit posizione } 2 \oplus \text{bit posizione } 3 \oplus \text{bit posizione } 5$$
$$\text{nuovo bit} = 1 \oplus 0 \oplus 0$$
$$\text{nuovo bit} = 1$$

Ora, spostiamo tutti i bit di una posizione verso destra nel registro.



Il bit che "sporge" a destra può essere escluso, e in prima posizione inseriamo il bit che abbiamo calcolato in precedenza.



E questo sarà il nuovo numero casuale, se ne volessi altri basta rieseguire il procedimento su questo nuovo stato del registro.

Una cosa interessante è notare che i **tap** possono essere espressi come **polinomi**. Rimanendo con l'esempio di prima dei tap (**2**, **3**, **5**), possiamo esprimere questa configurazione tramite il seguente polinomio:

$$p(x) = x^2 + x^3 + x^5 + 1$$

Qualsiasi configurazione è esprimibile tramite un polinomio, dove i gradi della variabile x corrispondono ai tap, con l'aggiunta di un termine noto (+1), e tutti i monomi hanno coefficiente 1.

Questo polinomio ha permesso di studiare le configurazioni ottimali per ottenere una generazione di numeri migliore. Infatti, grazie alla rappresentazione polinomiale, si è scoperto che se il polinomio è **primitivo modulo 2**, la generazione di numeri pseudo-casuali sarà massima. Non entrerò nei dettagli, ma considerate che un polinomio primitivo modulo 2 è simile a un **numero primo**, nel senso che non esistono prodotti di polinomi modulo 2 che diano come risultato quel polinomio primitivo.

Per esempio, il polinomio $p(x) = x^2 + x + 1$ è primitivo, perché non può essere scritto come prodotto di altri polinomi. Invece, il polinomio $p(x) = x^2 + 1$ non è primitivo, perché, in modulo 2, può essere scritto come:

$$\begin{aligned} p(x) &= (x+1)^2 \pmod 2 \\ &= x^2 + 2x + 1 \pmod 2 \\ &= x^2 + 1 \pmod 2 \end{aligned}$$

Inoltre, sempre con questo meccanismo, la generazione sarà massima anche quando le posizioni sono **numeri primi tra loro** e quando il **numero di posti è pari**.

Se non avete compreso bene questa parte, non preoccupatevi: il bello di questo concetto è vedere come una rappresentazione polinomiale di un sistema possa aiutare a migliorarlo e a massimizzarne l'efficienza.

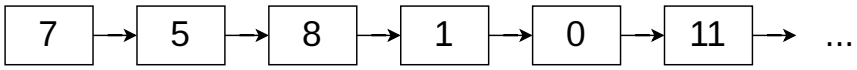
Grazie a questi accorgimenti, si è scoperto che il polinomio più sicuro per un LFSR a 16 bit è:

$$p(x) = x^{11} + x^{13} + x^{14} + x^{16} + 1$$

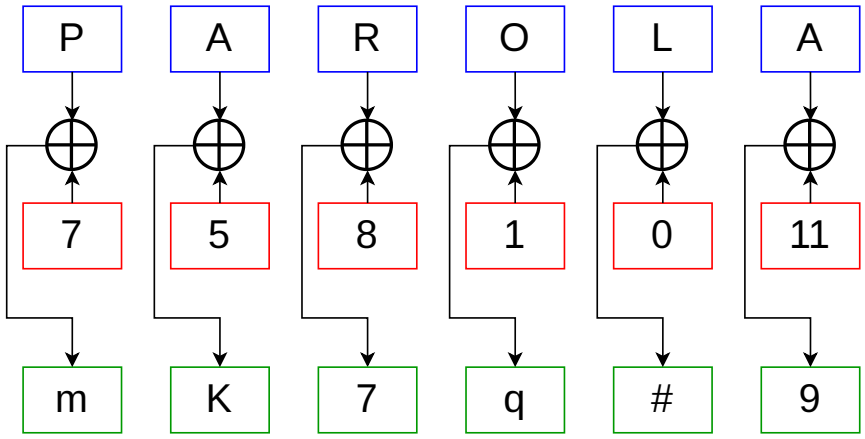
5.3 Stream

Fatta la premessa sui **PRNG**, capiamo cosa sono gli stream. Gli stream cercano di replicare la forza della cifratura **OTP**, che, come abbiamo già visto, è l'unico sistema teoricamente impossibile da decifrare. In pratica, gli stream utilizzano un algoritmo **PRNG**, come **LCG** o **LFSR** che abbiamo visto prima, e ogni lettera del messaggio viene cifrata tramite una **cifratura OTP** (cioè mediante un'operazione **XOR**) con uno dei numeri generati dal PRNG.

Quindi, per cifrare con gli stream, si inizia generando una sequenza di numeri:



Successivamente, ogni lettera del messaggio da cifrare viene combinata tramite un'operazione **XOR** con uno dei numeri generati.



N.B. Ricordo che il simbolo $\oplus$ indica l'operazione di **XOR**.

Cosi la stringa "**parola**" è diventata "**mK7q#9**".

Per decifrarlo, sarà sufficiente rigenerare i numeri usando gli stessi parametri e lo stesso seed della cifratura; successivamente, basta eseguire nuovamente l'operazione **XOR** tra il messaggio cifrato e la chiave, proprio come avviene nella cifratura **OTP**.

La **chiave privata** di questo sistema è quindi il **seed** e i **parametri** del PRNG, perchè avendoli si può generare la sequenza di numeri e poi cifrare il messaggio. Anche se in realtà si possono rendere pubblici i parametri e lasciare **solamente il seed come chiave privata**, tanto con valori di paramentri molto altri (10/15 cifre) non rendono facilmente la rottura del cifrario

L'idea alla base degli stream è di replicare la sicurezza dell'**OTP** e di usare chiavi diverse per ogni cifratura, poiché ricordiamo che il problema dell'OTP è che non si può riutilizzare la stessa chiave, altrimenti si può risalire ai messaggi originali.

Ad oggi, però, i cifrari a **stream** non sono molto diffusi, a causa della difficoltà di generare numeri veramente casuali e perché, se si scopre il **seed** (che costituisce la chiave privata del cifrario), è possibile ricavare l'intero messaggio.