**Algoritmi di Crittografia**

**Nozioni preliminari**

***Scenario classico***

* La crittografia moderna include molteplici ambiti applicativi.
* Lo scenario "tipico" prevede due attori, tradizionalmente chiamati **Alice** e **Bob**, che vogliono scambiarsi informazioni su un **canale di comunicazione insicuro**, dove un terzo personaggio, **Eva**, è in grado di intercettare, leggere e modificare i messaggi.
* Assumendo che, in una data comunicazione, Alice sia il mittente e Bob il destinatario, si possono individuare i seguenti **requisiti tipici** della comunicazione stessa.
  1. La **confidenzialità** del messaggio, ovvero che Eva non sia in grado di comprenderne il contenuto;
  2. l'**autenticazione** del mittente, in base alla quale Bob è certo che il messaggio sia stato effettivamente inviato da Alice;
  3. l'**integrità** del messaggio, in base alla quale Bob è certo che Eva non lo abbia manomesso;
  4. il **non** **ripudio**, per cui Alice non può negare, in un secondo momento, di aver inviato il messaggio a Bob.

***Un po' di ulteriore terminologia***

* **Plaintext/testo in chiaro** è il messaggio che si vuole tenere segreto.
* **Encryption/cifratura** è il processo che rende il plaintext incomprensibile per Eva.
* Nella lingua inglese il testo cifrato è indicato con il termine **ciphertext**.
* Ogni processo di cifratura deve evidentemente essere **reversibile**.
* Il procedimento che dal testo cifrato restituisce il plaintext originale è detto **decryption/decifrazione**.
* Cifratura e decifrazione sono i due algoritmi che compongono un **cifrario/cipher**
* Un cifrario fa anche uso di informazione segreta, nota come **chiave/key**

***Il fondamentale principio di Kerckhoff***

In uno schema di cifratura, la sicurezza deve risiedere

solo nella segretezza della chiave

(A. Kerckhoff, La Cryptographie Militaire, 1883)

* In particolare, la segretezza non deve risiedere nell'algoritmo.

***Due "tipi" di crittografia***

* Letteralmente per "secoli" crittografia è stato solo un sinonimo di **crittografia simmetrica**.
* In un protocollo simmetrico Bob e Alice possiedono la stessa chiave ovvero hanno chiavi che possono essere **facilmente determinate** l'una a partire dall'altra.
* Solo in tempi recenti, sulla spinta della necessità di risolvere il problema cruciale dello **scambio di chiavi**, si è sviluppata la **crittografia asimmetrica**, in cui una delle due chiavi è posseduta solo da chi, fra Alice e Bob, è il destinatario di una comunicazione.
* È palese che, nel caso della crittografia asimmetrica, la determinazione della chiave privata a partire dalla chiave pubblica deve essere **computazionalmente difficile**.
* Il lavoro fondamentale, che delinea le caratteristiche della crittografia asimmetrica, è del 1976:

Whitfield Diffie, Martin Hellman  
**New directions in cryptography**  
IEEE Trans. Inf. Theory (1976)

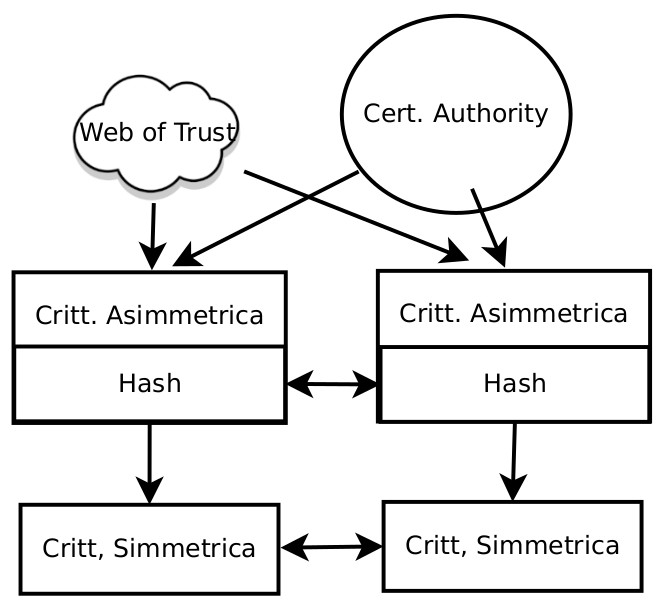
* La matematica e l'algoritmica sottostanti i protocollo simmetrici e quelli simmetrici sono **sostanzialmente distinte**.

***Sguardo d'insieme alle differenze fra i due tipi di crittografia***

|  |  |
| --- | --- |
| **Crittografia simmetrica** | **Crittografia asimmetrica** |
| Una sola chiave per cifrare e decifrare | Una chiave pubblica per cifrare e una privata per decifrare |
| Il principale ostacolo è la necessità di un accordo preliminare sulla chiave | Il mittente deve essere certo dell'identità del possessore della chiave pubblica |
| Ingestibile in applicazioni che coinvolgono più partecipanti (servono 𝑂(𝑛2) chiavi) | Gestibile con più coppie di chiavi pubblica/privata (servono 𝑛 coppie) |
| Algoritmi veloci, anche con supporto hardware | Algoritmi lenti |

***Crittografia simmetrica e asimmetrica lavorano spesso insieme***

* Vantaggi e svantaggi sono speculari e dunque, molto spesso, protocolli applicativi che utilizzano la crittografia fanno uso di **entrambe le soluzioni**.
* Al riguardo, il seguente schema esemplifica le interazioni fra i due tipi di crittografia.
* Lo scenario è relativo alla **cifratura**.

****

**Crittografia simmetrica**

* Il focus di questo corso è sulla **crittografia asimmetrica**.
* In questo primo notebook forniamo tuttavia qualche nozione anche sulla **crittografia simmetrica**.
* I concetti base verranno introdotti utilizzando principalmente **cifrari classici**, del tutto inadeguati a resistere ad attacchi portati con moderni computer, ma sufficienti per "dare un'idea".
* Vedremo tuttavia anche un cifrario moderno (anche se superato), che ci consentirà di cogliere le idee oggi alla base della crittografia simmetrica.

**Principi generali**

* Nella struttura di un cifrario simmetrico si possono individuare due distinte componenti.
* Una prima componente è l'algoritmo, o semplicemente l'insieme di operazioni che applicano una determinata **permutazione** ad una porzione del plaintext.
* Che cosa sia una porzione è evidentemente una caratteristica propria del cifrario.
* Essa può essere una **singola lettera**, un gruppo di lettere o (come nei cifrari moderni) un **gruppo di bit**, detto comunemente **blocco**.
* Una **permutazione** di un insieme di 𝑛 oggetti è uno dei possibili modi in cui gli 𝑛 oggetti si possono disporre linearmente (uno dopo l'altro); cioè è uno dei possibili **ordinamenti** degli oggetti.
* In termini matematici, una permutazione (degli elementi) di un insieme 𝐼 è una funzione invertibile di 𝐼 in sé stesso.
* Si noti che l'ovvio requisito di invertibilità della cifratura, che si precisa formalmente con l'invertibilità delle permutazioni, in termini pratici implica che plaintext e ciphertext abbiano sempre la **stessa lunghezza**.
* La seconda componente fondamentale di un cifrario è l'algoritmo che, organizzando opportunamente le permutazioni dei singoli blocchi, effettua la **cifratura di un plaintext arbitrario**.
* Questa seconda componente è detta **mode of operation**.

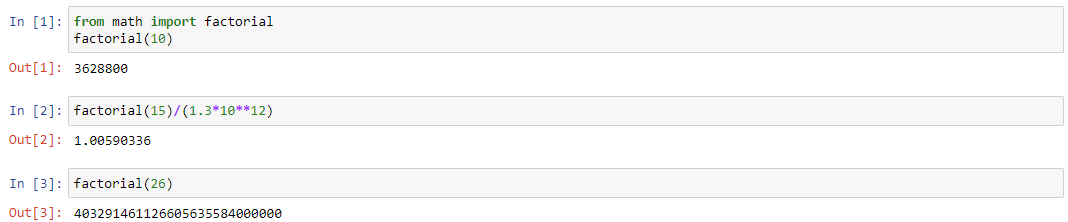
*Nota*: i cifrari che seguono l'impostazione appena delineata sono detti, non a caso, (cifrari) **a blocchi**. Esistono però cifrari simmetrici differenti, che considerano il messaggio come **flusso continuo** di bit e che utilizzano la chiave per generare un corrispondente flusso di bit (pseudo)casuali, da porre in XOR con i bit del messaggio. Tali cifrari, detti appunto ***stream cipher***, sono più adatti a, e vengono utilizzati per, cifrare (ad esempio) comunicazioni telefoniche.

***Sulle permutazioni***

* È noto che, al crescere di 𝑛, il numero di permutazioni diventa rapidamente enorme, in quanto cresce come il **fattoriale di** 𝑛.

𝑛! = 1 ⋅ 2 ⋅ 3 ⋅…⋅ (𝑛−1) ⋅ 𝑛

* Ad esempio, le possibili permutazioni di 10 oggetti sono già più di 3 milioni. Numero che sale a circa 1300 miliardi nel caso di 15 oggetti.

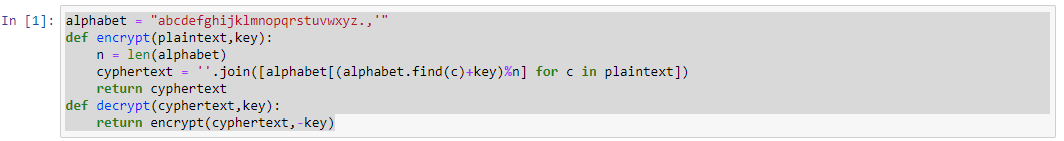


**Due cifrari classici: Cesare e Vigenère**

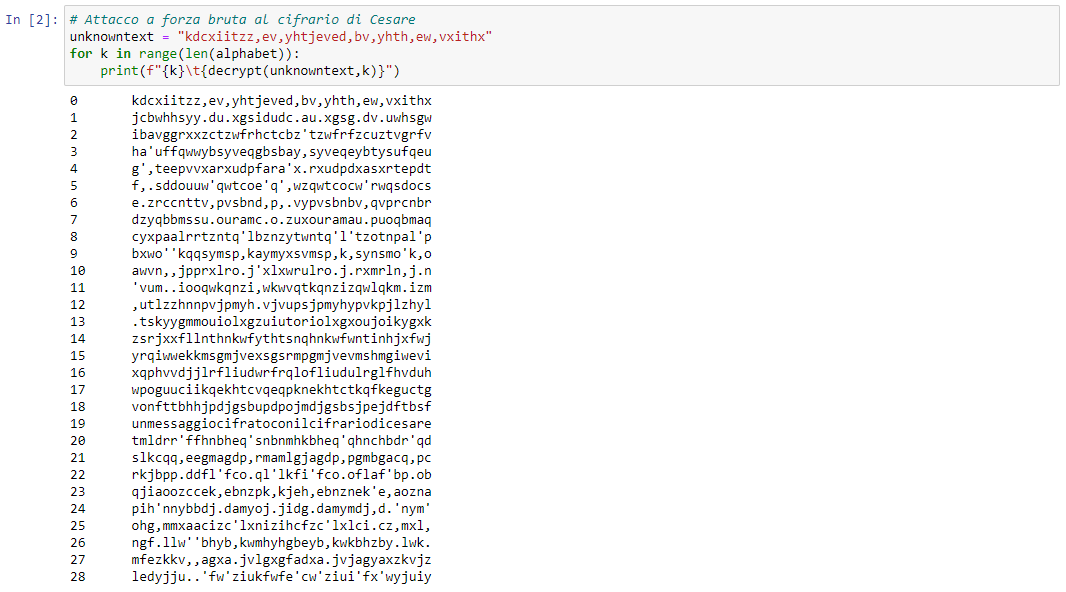
* Consideriamo messaggi scritti in un generico alfabeto Σ e consideriamo un **ordinamento circolare dei caratteri** (arbitrario ma fissato una volta per tutte).
* Circolare vuol dire che il primo carattere è considerato il successore dell'ultimo.
* Nel **cifrario di Cesare** la cifratura avviene come illustrato di seguito.
  + La permutazione applicata ad ogni carattere consiste in uno **slittamento** (**shift**) di 𝑘 posizioni in avanti.
  + Questo vuol dire che ogni carattere 𝑥 del plaintext viene sostituito con il carattere che si trova 𝑘 **posizioni più avanti** rispetto ad 𝑥 nell'ordinamento circolare dell'alfabeto.
  + Ad esempio, considerando l'alfabeto inglese e ponendo 𝑘 = 3, la 𝑎 viene sostituita con la 𝑑, la 𝑏 con la 𝑒,..., la 𝑥 con la 𝑎, la 𝑦 con 𝑏 e la 𝑧 con la 𝑐.
  + Il mode of operation del cifrario è il più semplice possibile e consiste nell'applicare la **stessa permutazione** ad ogni carattere.
* È evidente che chiave "segreta" di cifratura è proprio il valore 𝑘 dello slittamento.
* La decifrazione deve rimettere le cose a posto e questo, nel cifrario di cesare, consiste nell'effettuare uno **slittamento** di 𝑘 posizioni all'indietro, ovvero sostituire ogni carattere 𝑥x del ciphertext con un carattere nell'alfabeto che **precede** 𝑥 **di** 𝑘 **posizioni** nell'ordinamento circolare.
* Potremmo anche dire che, se 𝑘 è la chiave di cifratura, la chiave di decifrazione è il valore opposto, −𝑘.
* È chiaro che per il cifrario di Cesare il **numero** **di chiavi**, che coincide con il numero di possibili slittamenti, è limitato dalla cardinalità dell'alfabeto.

***Una semplice sfida***

* Consideriamo come alfabeto Σ = {a,b,...,z} ∪ {.} ∪ {,}∪ {'}
* Sapendo che il cifrario utilizzato è quello di Cesare, a quale testo in chiaro corrisponde il cyphertext **kdcxiitzz,ev,yhtjeved,bv,yhth,ew,vxithx**?



* Il procedimento di risoluzione di questa sfida è un caso di ***crittoanalisi***.
* In questo caso, la crittoanalisi è banale. Dato il numero limitato di chiavi, un **attacco a forza** **bruta** ha facile successo, solo con qualche necessaria sottolineatura...



***Cifrari mono-alfabetici***

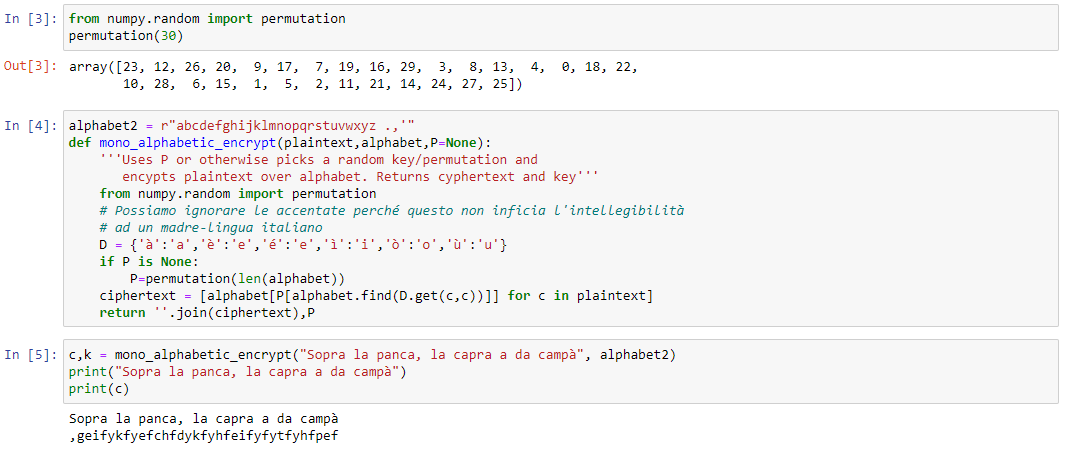
* Il **cifrario di Cesare** è forse il più semplice caso di **cifrario mono-alfabetico**.
* È tale un cifrario che utilizza una **sostituzione fissa** per ogni carattere.
* In altri termini, fissato l'alfabeto di riferimento Σ, un cifrario mono-alfabetico è definito da una **singola permutazione** ripetuta per tutti i caratteri.

***Permutazioni ideali e relazioni con la chiave***

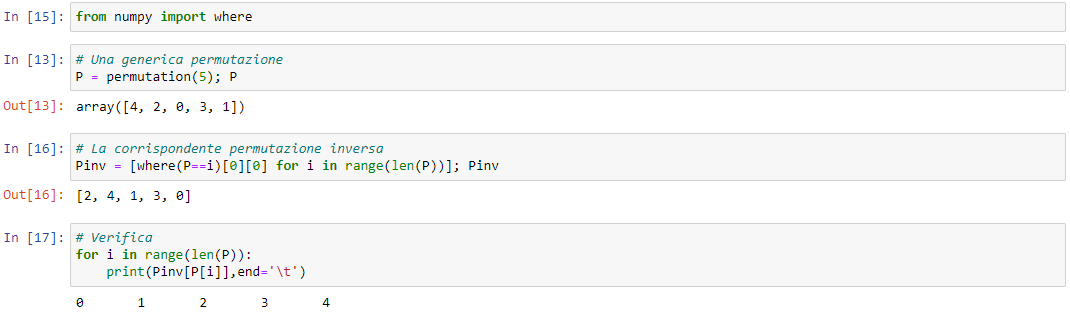
* Le permutazioni utilizzate dal cifrario di Cesare sono semplici slittamenti circolari, detti più propriamente **rotazioni**.
* Per conoscere una tale permutazione è sufficiente un singolo numero intero, compreso fra 0 e la cardinalità dell'alfabeto, che rappresenta quindi la **chiave del cifrario**.
* Il fatto che sia la chiave a determinare la permutazione è una proprietà che deve valere **sempre**, anche se il passaggio dalla chiave alla permutazione non è di norma così semplice come nel cifrario di Cesare.
* La chiave infatti è l'**unica** **informazione segreta** e dunque la permutazione deve dipendere necessariamente dalla chiave.
* Non solo, chiavi diverse devono sempre fornire permutazioni diverse. In caso contrario per un attaccante ci sarebbero **meno chiavi distinte da "provare"** e questo potrebbe agevolare sensibilmente gli attacchi.
* Infine, la permutazione dovrebbe il più possibile apparire **casuale**. Questo garantisce che la conoscenza di come viene mappata una lettera (o un gruppo di bit) non fornisce alcuna conoscenza sulla mappatura delle altre.
* Nel caso del cifrario di Cesare, la permutazione è tutto fuorché casuale. Se sappiamo che la lettera 𝑎 viene mappata nella lettera 𝑑, sappiamo immediatamente anche che 𝑏 viene mappata in 𝑒, 𝑐 in 𝑓 e così via.

***Un generico cifrario mono-alfabetico***

* Allo scopo di procurarci permutazioni arbitrarie, utilizzeremo la funzione **permutation** del modulo **numpy.random**.



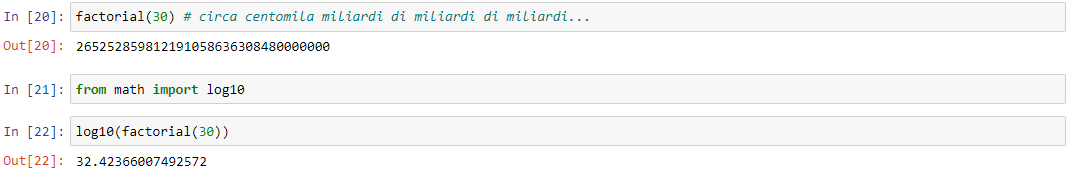
* La chiave ovviamente **non può viaggiare in chiaro**; deve in qualche modo essere preliminarmente concordata.
* Come esercizio, si scriva il semplice codice per la decifrazione.
* Al riguardo, il seguente codice calcola la **permutazione inversa** di una permutazione data come array numpy.



***Una seconda challenge: decifrare il testo cifrato che stiamo per creare***



* Sembra una challenge impossibile, visto che la permutazione è stata scelta a caso fra **30!** **permutazioni** possibili.

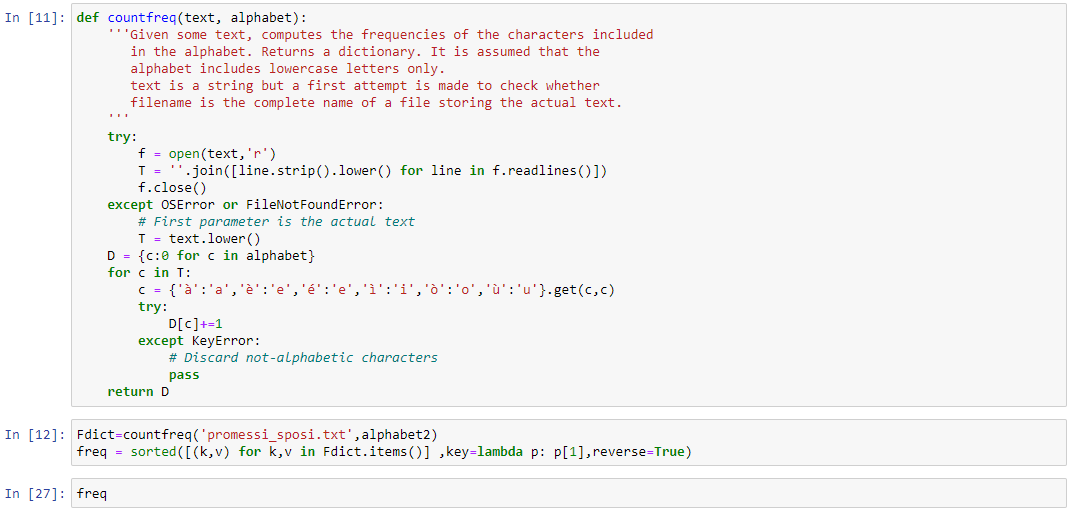


***Critto-analisi***

* Per un generico cifrario mono-alfabetico il numero di possibili chiavi è dunque pari al numero di permutazioni dell'insieme dei caratteri dell'alfabeto.
* Per l'alfabeto inglese, senza spazi ne' caratteri di punteggiatura, abbiamo quindi 26! ≈ 4⋅1026 ≈ 1.3⋅288 permutazioni. Tale numero è paragonabile al numero di atomi nel corpo umano.
* Un attacco a forza bruta è pertanto **fuori discussione**.
* Per i cifrari mono-alfabetici è però efficace una critto-analisi **basata sull'analisi delle frequenze**.

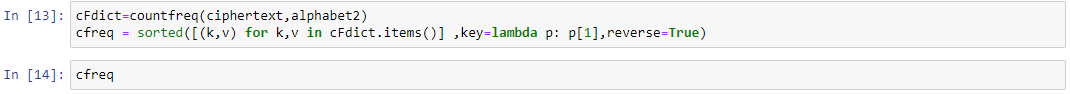
***Analisi delle frequenze***

* L'attacco è basato sull'ipotesi (non sempre verificata) che la frequenza con cui compaiono i singoli caratteri nel plaintext sia conforme alla **frequenza nella lingua** con cui il testo stesso è scritto.
* È evidente che, in generale, l'ipotesi sarà tanto più verificata quanto più il plaintext è lungo.
* Per portare questo attacco, conoscendo (o ipotizzando quale sia) la lingua con cui è scritto il plaintext, bisogna dunque disporre di una **stima delle frequenze dei caratteri** in quella lingua.
* Le celle di codice che seguono calcolano una tale stima per l'italiano, utilizzando a tale scopo un solo testo "classico" (*I promessi sposi*).



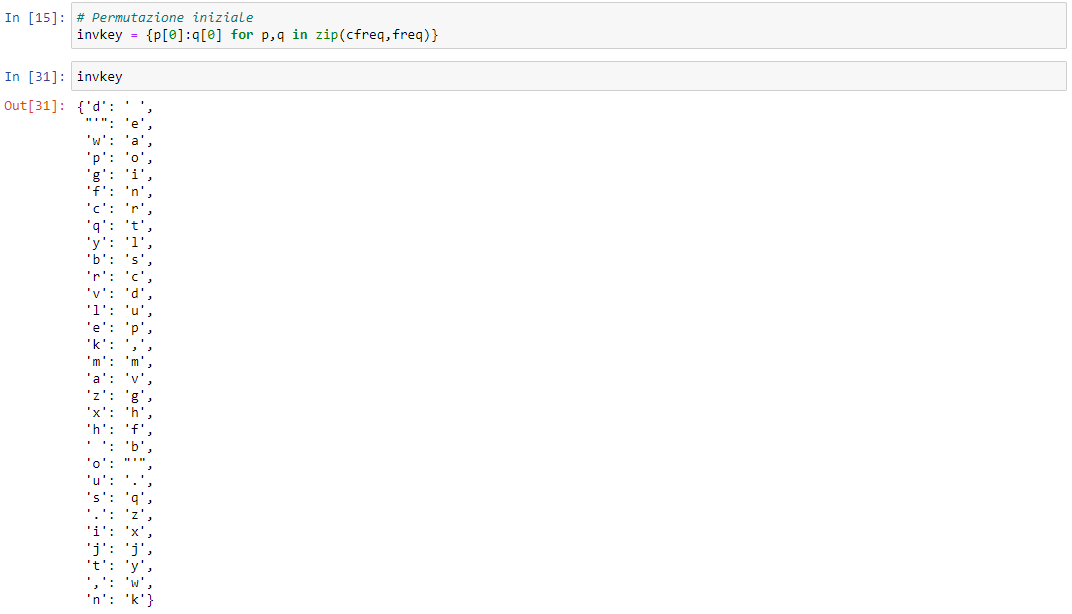


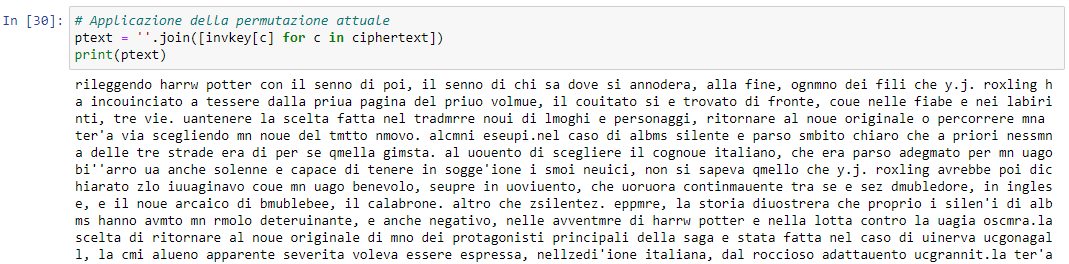
* Come secondo passo calcoliamo le **frequenze dei caratteri ciphertext** e ordiniamole in senso decrescente.

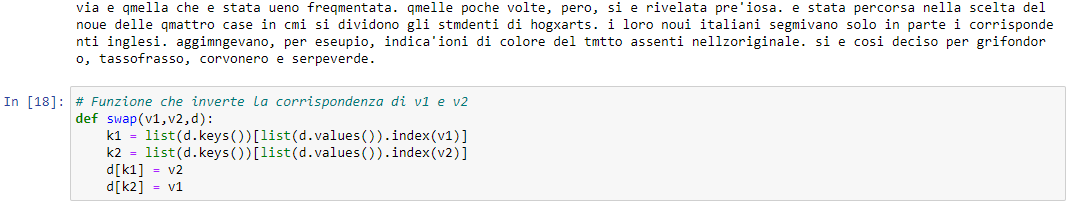




* Come terzo passo proviamo la **corrispondenza (permutazione) secca** che, quasi certamente, non funzionerà ma che, probabilmente, inizierà a far "vedere qualcosa"
* In tal caso si eseguiranno modifiche progressive fino alla convergenza alla permutazione giusta.







* Dall'**analisi visuale** si effettuano poi i seguenti cambiamenti alla permutazione iniziale
* Si noti che, se si capisce che due lettere sono semplicemente invertite, l'applicazione di 𝚜𝚠𝚊𝚙 sistema entrambe le corrispondenze.
* Se invece una lettera 𝑥 del testo cifrato codifica 𝑦 del testo in chiaro ma 𝑦 non codifica 𝑥, allora 𝚜𝚠𝚊𝚙(𝑥,𝑦,𝚒𝚗𝚟𝚔𝚎𝚢) sistema solo la corrispondenza di 𝑦.
* In questo secondo caso, dopo lo swap la lettera che era codificata da 𝑦 ora è codificata sa 𝑥.
* I passaggi sono (relativamente) tanti, ma via via **sempre più ovvi**.



***Vigenère: un cifrario poli-alfabetico***

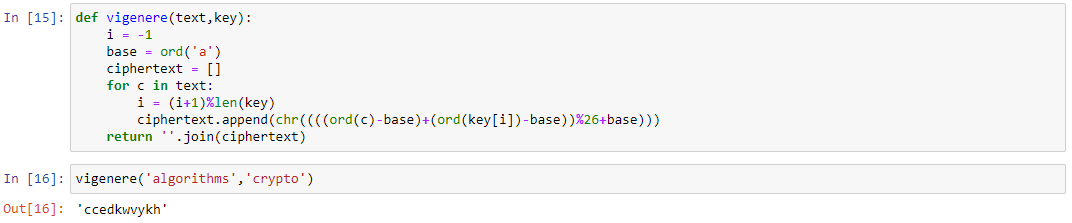
* Il cifrario di ***Vigenère*** è un esempio di cifrario poli-alfabetico.
* Questo vuol dire che uno stesso carattere non è sempre soggetto alla stessa trasformazione.
* Le chiavi sono qui **sequenze di caratteri dell'alfabeto**, ciascuno dei quali deve essere interpretato come il numero corrispondente alla sua **posizione nell'alfabeto** (partendo da 0).
* Limitandoci ancora una volta all'alfabeto inglese, possiamo dunque affermare che, ad esempio, la chiave **crypto**, vada interpretata come la sequenza di numeri **2,17,24,15,19,14** (dove abbiamo usato le virgole per evitare possibili ambiguità).
* Una volta operata questa corrispondenza, la chiave viene usata nel modo seguente:
  + Il primo carattere del testo viene fatto slittare di 2 posizioni
  + Il secondo carattere del testo viene fatto slittare di 17 posizioni
  + ...
  + Il sesto carattere del testo viene fatto slittare di 14 posizioni
  + Il settimo carattere del viene fatto slittare nuovamente di 2 posizioni
  + ...
* Ad esempio, se il plaintext fosse **algorithms** (e la chiave sempre **crypto**), il ciphertext viene determinato nel modo seguente:

key: 𝚌 𝚛 𝚢 𝚙 𝚝 𝚘 𝚌 𝚛 𝚢 𝚙

plaintext: 𝚊 𝚕 𝚐 𝚘 𝚛 𝚒 𝚝 𝚑 𝚖 𝚜

ciphertext: 𝚌 𝚌 𝚎 𝚍 𝚔 𝚠 𝚟 𝚢 𝚔 𝚑

* La decifrazione procede in modo analogo, usando la stessa chiave ma con **slittamenti all'indietro**.
* Il codice Python per eseguire la cifratura, riportato di seguito, usa le posizioni dei caratteri alfabetici nel codice ASCII
* La funzione **ord** restituisce la posizione di un carattere nel codice ASCII
* Dunque, per ottenere la posizione di un carattere c rispetto all'**inizio dell'alfabeto** è necessario calcolare la differenza **ord(c)-ord('a')**



* Il cifrario di Vigenère può essere analizzato in termini generali (e non come "sequenza" di cifrari di Cesare).
* Nel caso di Vigenère le porzioni di testo sottoposte a permutazione sono **sequenze di**𝑘 **lettere**, dove 𝑘 questa volta è la lunghezza della chiave.
* La chiave permette dunque di calcolare la cifratura univoca di qualsiasi **blocco** composto da 𝑘 lettere.
* La permutazione è dunque rappresentabile per esteso mediante una **tabella di 26**𝑘**righe**, una per ogni possibile blocco.
* L'esempio seguente mostra le righe iniziali e finali della tabella corrispondente alla chiave crypto, la cui lunghezza (numero di righe) è 266 = 308915776.

|  |  |
| --- | --- |
| **plaintext** | **ciphertext** |
| aaaaaa | crypto |
| aaaaab | cryptp |
| aaaaac | cryptq |
| ... | ... |
| yzzzzz | aqxosn |
| zzzzzz | bqxosn |

***Attacchi al cifrario di Vigenère***

* Senza l'ausilio dei computer, il cifrario era pressoché inattaccabile, in caso di chiave essenzialmente **casuale**.
* Rimane tutt'oggi difficilmente attaccabile in caso di chiave casuale, di lunghezza confrontabile con il testo e non riutilizzata più volte.
* In quest'ultimo caso, il cifrario "degenera" infatti in una sorta di **one-time pad testuale** e il one-time pad, detto anche ***cifrario di Vernam***, è l'unico matematicamente inviolabile (come vedremo subito).
* Se invece la chiave è relativamente corta rispetto al testo è possibile portare un attacco, la cui "prima fase" consiste proprio nell'individuazione della **lunghezza della chiave**.
* Si consideri, come esempio guida, il testo cifrato 𝐶=𝚏𝚡𝚋𝚡𝚣𝚔𝚝𝚠𝚛𝚍𝚟𝚏𝚌𝚜𝚏𝚡𝚋𝚡𝚚𝚠𝚙𝚕𝚋𝚗.
* Notiamo che la sequenza di 4 caratteri 𝚏𝚡𝚋𝚡 si ripete ad una distanza di 14 posizioni.
* La ripetizione può certamente essere l'identica "risultante" di differenti caratteri in testo e chiave, ma è pure possibile che essa sia dovuta ad una **stessa sequenza nel testo in chiaro**.
* Se questo è il caso, allora devono però essere uguali **anche i caratteri della chiave** in quelle posizioni.
* Ma allora, continuando con le deduzioni, o la chiave è lunga **almeno 18 caratteri**, con al suo interno delle ripetizioni, oppure la chiave ha una lunghezza che è un **divisore di 14**, il che, in questo caso, vuol dire lunghezza 2 o 7 o 14.
* Partendo da queste osservazioni, l'attaccante può tentare **attacchi esaustivi con le lunghezze ammissibili più corte** (sicuramente 2 ma anche 7, se l'alfabeto non è troppo grande) o attacchi **basati su dizionario**, nel caso delle lunghezze superiori.

***Ipotesi "di lavoro da parte di Eva": la chiave è lunga 7***

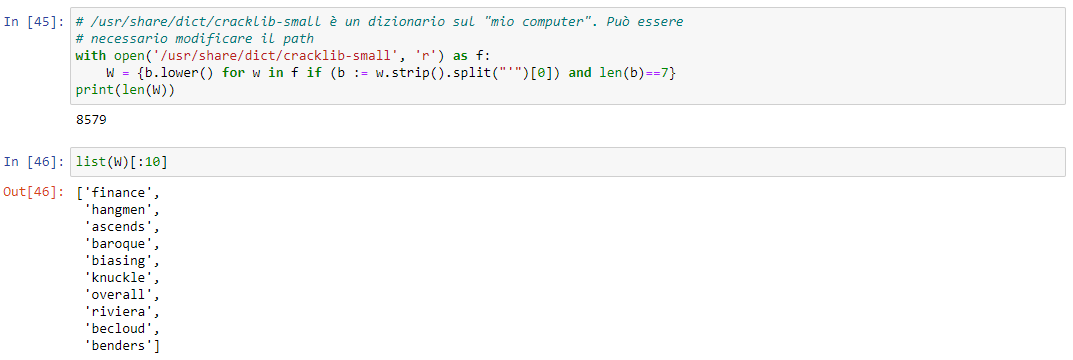
* Eva può tentare (almeno) 3 strade:
  + Attacco **brute force**; pesante ma non impossibile perché le chiavi sono "soltanto" 267 = 8031810176, cioè approssimativamente 8 miliardi.
  + **Analisi delle frequenze**: se il testo è abbastanza lungo Eva può stimare le frequenze considerando che due identici caratteri del testo cifrato corrispondono con certezza allo stesso carattere del plaintext se e solo se la loro posizione differisce di un multiplo di 7.
  + Attacco **basato sul dizionario**: ipotizzando che la chiave non sia una sequenza casuale, Eva può provare tutte le parole di 7 lettere in un dizionario inglese.
* Se l'attacco basato sul dizionario è chiaramente "parato" mediante l'utilizzo di una chiave casuale, l'analisi delle frequenze richiede una **buona dose di fortuna**.
* Ad esempio, anche il carattere più frequente potrebbe non essere individuato se esso viene trasformato in molti caratteri diversi a causa di allineamenti "sfortunati" (per Eva!)

***Attacco con l'uso del dizionario***

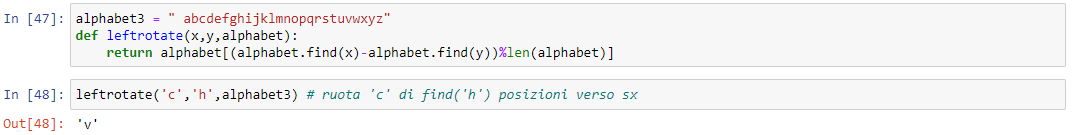
* Supponiamo di sapere che la lingua del messaggio è l'inglese e che l'alfabeto sia

Σ = { ,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}

* Usiamo l'elenco di parole presenti nel file **cracklib-small** del pacchetto Debian **cracklib-runtime**.
* Nel file ci sono circa **8K parole di 7 caratteri** e un attaccante cercherà sicuramente una chiave nel dizionario.
* Vediamo (pur nella semplicità dell'esempio) come eseguire l'attacco
* Come prima cosa recuperiamo le parole di 7 lettere dal dizionario



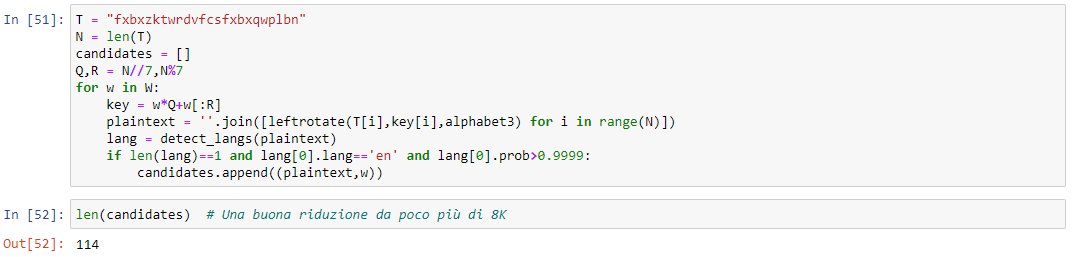
* Dopodichè definiamo solo l'operazione fondamentale per eseguire la decifrazione, ovvero lo slittamento all'indietro



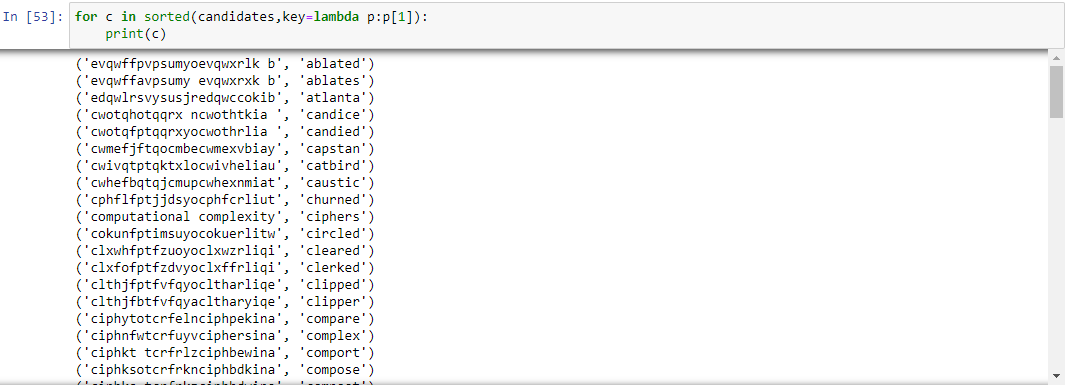
* Per evitare di dover controllare circa 8K possibili soluzioni, utilizziamo un modulo Python in grado di rilevare il linguaggio di una sequenza di testo



* Possiamo ora tentare l'attacco



* L'analisi finale dei **plaintext candidati** può essere fatta da un essere umano.



**One-time pad**

* One time pad è "teoricamente" il cifrario **più sicuro**, anche se è affetto da molteplici problemi di natura pratica che lo rendono di fatto inutilizzabile.
* Il messaggio (plaintext) 𝑃 da cifrare è una **stringa di bit** (ed è naturalmente sempre possibile considerarlo tale).
* La chiave è una sequenza di bit della **stessa lunghezza del testo in chiaro**: l'𝑖-esimo bit 𝑘𝑖 della chiave viene messo in xor con il corrispondente bit 𝑝𝑖 del plaintext per produrre l'𝑖-esimo bit 𝑐𝑖 del ciphertext

𝑐𝑖 = 𝑝𝑖 ⊕ 𝑘𝑖, 𝑖 = 0, 1,…, |𝑃|−1

* Un semplice esempio

key: 0 0 0 1 0 1 1 0 1 0

p: 1 0 1 1 0 0 1 0 0 1

c: 1 0 1 0 0 1 0 0 1 1

* Per le proprietà del'or esclusivo, la decifrazione procede allo **stesso modo**, eseguendo cioè l'operazione con chiave e ciphertext:

key: 0 0 0 1 0 1 1 0 1 0

c: 1 0 1 0 0 1 0 0 1 1

p: 1 0 1 1 0 0 1 0 0 1

***Scelta della chiave***

* Ogni bit della chiave deve avere valore 0 oppure 1 con **uguale probabilità** e in modo **indipendente** dal valore dei bit precedenti

prob[𝑘𝑖 = 0] = prob[𝑘𝑖 = 1] =prob[𝑘𝑖 = 𝑣|𝑘𝑖−1 = 𝑤] = prob[𝑘𝑖 = 𝑣], 𝑣, 𝑤 ∈ {0, 1}

* In questo modo, è immediato osservare che per i bit del messaggio cifrato **valgono le stesse proprietà**, indipendentemente da come sono scelti i bit del messaggio.
* Per la generica posizione (tralasciamo quindi gli indici per semplicità) vale infatti

prob[𝑐 = 1] = prob[𝑝 ⊕ 𝑘 = 1]

= prob[𝑝 = 1 ∧ 𝑘 = 0] + prob[𝑝 = 0 ∧ 𝑘 = 1]

= prob[𝑝=1]⋅prob[𝑘=0]+prob[𝑝=0]⋅prob[𝑘=1]

= 𝛼 ⋅ 0.5 + (1 − 𝛼) ⋅ 0.5

= 0.5

dove abbiamo indicato con 𝛼 la probabilità che il bit del messaggio sia 1.

* Si noti, in particolare, che l'uguaglianza precedente vale anche se 𝛼 = 1 o 𝛼 = 0, cioè se il bit (e più in generale il messaggio) è **deterministico**.

***Sicurezza del one-time pad***

* Se la chiave è scelta come indicato sopra ed 𝑛 è la lunghezza del plaintext, ogni messaggio cifrato ha la **stessa probabilità 2**−𝑛**di essere trasmesso**.
* Questa è ovviamente il massimo di garanzia perché, rovesciando la prospettiva, per l'attaccante il messaggio "visto" può corrispondere ad **uno qualsiasi dei 2**𝑛**possibili messaggi in chiaro**, tutti con identica probabilità 2−𝑛.
* Tuttavia, se la chiave viene utilizzata più volte, l'attaccante può **ottenere conoscenza sui messaggi** (anche se non direttamente i messaggi in chiaro).
* Supponiamo, ad esempio, che due messaggi, 𝐶1 e 𝐶2, siano stati **cifrati usando la stessa chiave**𝐾. Indicando con 𝑂 la stringa di tutti 0 della stessa lunghezza della chiave, abbiamo

𝐶1 ⊕ 𝐶2 = (𝑃1 ⊕ 𝐾) ⊕ (𝑃2 ⊕ 𝐾)

= (𝑃1 ⊕ 𝑃2) ⊕ (𝐾 ⊕ 𝐾)

= (𝑃1 ⊕ 𝑃2) ⊕ 𝑂

= 𝑃1 ⊕ 𝑃2

* In questo modo l'attaccante viene a conoscenza dello **xor dei due messaggi** e dunque, conoscendone uno dei due (o parte di uno dei due) verrebbe a conoscere anche (parte de) l'altro.

**Obiettivi generali di sicurezza**

* La precedente osservazione riguardo one-time pad ci porta a parlare degli **obiettivi generali di sicurezza**, almeno nel contesto della cifratura.
* Il primo obiettivo è, ovviamente, che Eva non sia in grado di comprendere i messaggi. Tuttavia questa è una formulazione **troppo generica**.
* Esistono **più obiettivi di attacco** da parte di Eva, di cui "capire il messaggio" non è neppure il più estremo.
* Ad esempio, **determinare la chiave** è ancora più ambizioso perché consente di mettere in chiaro uno o più messaggi.
* Un altro obiettivo consiste nel far passare per autentico un messaggio che invece Eva ha **alterato** o prodotto nella sua interezza.

***Indistinguibilità***

* L'**indistinguibilità** è la proprietà di uno schema di cifratura i cui testi cifrati per un attaccante sono essenzialmente **indistinguibili** da stringhe di bit generate casualmente.
* Più precisamente, la proprietà viene descritta in termini di un **gioco a due** (Alice e Eva).
  + Eva produce **due messaggi in chiaro** e li sottopone ad Alice;
  + Alice decide, a caso e con uguale probabilità, **quale dei due cifrare**, e lo presenta ad Eva;
  + Alice vince il gioco se Eva non riesce ad indovinare quale messaggio sia stato cifrato con probabilità significativamente maggiore di  .
  + Nel concreto questo vuol dire che Eva, dal suo punto di vista, non ha strategie più efficaci rispetto a **tirare a caso**.
* Si noti che, in caso di crittografia **asimmetrica**, se il protocollo non è randomizzato Eva può facilmente scoprire quale messaggio è stato cifrato da Alice
* Questo proprio perché la cifratura avviene con una chiave che è pubblica

***Una possibile soluzione***

* Sia nel caso simmetrico che in quello asimmetrico si utilizza un ***generatore pseudocasuale crittografico (DRBG)***
* Nel caso simmetrico, Alice produce dapprima la sequenza 𝑆 = 𝐾 || 𝑅, dove 𝐾 è la **chiave condivisa**, 𝑅 una **sequenza di bit casuali** di opportuna lunghezza mentre || indica la concatenazione
* Alice usa poi 𝑆 come **valore iniziale del generatore** per ottenere una successione **DRBG(**𝑆**)** di 𝑛=|𝑀| bit, dove 𝑀 è il messaggio da lei scelto
* Il messaggio effettivamente rilasciato ad Eva è la **coppia (**𝐶**,**𝑅**)**, dove 𝐶 = 𝑀 ⊕ DRBG(𝑆)
* Nel caso asimmetrico la precedente strategia non risolve il problema perché 𝐾 è nota anche ad Eva che, vedendo 𝑅 nel messaggio, **potrebbe ricostruire** 𝑆
* La modifica è però semplice. Poiché 𝐾 è pubblica, è inutile che essa faccia parte del seme iniziale del generatore, che dunque è **semplicemente** 𝑅
* Il messaggio effettivamente rilasciato ad Eva è la **coppia (**𝐶**,**𝐸**(**𝐾**,**𝑅**))**, dove 𝐶 = 𝑀 ⊕ DRBG(𝑅) mentre 𝐸 indica la **funzione di cifratura asimmetrica**

***Non-malleabilità***

* Un secondo obiettivo è la **non malleabilità**.
* Uno schema di cifratura si definisce **non malleabile** se, dato un ciphertext 𝐶1 corrispondente ad un plaintext 𝑃1, risulta arduo per Eva creare un secondo ciphertext 𝐶2 corrispondente ad un plaintext 𝑃2 che abbia una **qualche relazione** con 𝑃1.
* Si può intuire come la non malleabilità sia una proprietà che ha a che vedere con l'**integrità** dei messaggi.
* A riguardo del protocollo one-time pad, possiamo affermare che esso, pur essendo sicuro dal punto di vista della indistinguibilità, risulta però un cifrario **malleabile** quando la chiave venga riutilizzata.

**Cifrari a blocchi**

* I moderni cifrari simmetrici vengono detti **a blocchi** perché lavorano su blocchi di dati di lunghezza fissata.
* Indicheremo genericamente con 𝐵 il numero di bit di un blocco, un valore che tipicamente si colloca nell'**intervallo** **da 64 a 256**.
* Se il plaintext ha lunghezza minore di 𝐵, ad esso vengono preliminarmente aggiunti **bit di** **padding** (riempimento).
* Viceversa, se il plaintext ha lunghezza superiore, esso viene suddiviso in più blocchi che vengono poi **cifrati separatamente**.
* Come già detto a suo tempo, il particolare algoritmo con cui i blocchi di uno stesso messaggio vengono cifrati, di modo che si possa poi ricostruire (in sede di decifrazione) il plaintext originale, viene detto **mode of operation**.
* Entrambi gli aspetti, padding e mode of operation, sono **tutt'altro che banali** ed errori di valutazione possono dare vantaggi decisivi ad un attaccante.

***Scelta di***𝐵

* I valori di 𝐵 indicati sopra derivano dalla necessità di bilanciare due esigenze contrapposte.
* Da un lato 𝐵 deve essere un valore piccolo allo scopo di:
  + minimizzare l'overhead legato al trattamento di **messaggi corti**;
  + incrementare l'efficienza facilitando implementazioni in **hardware dedicato**.
* D'altro canto, 𝐵 non può essere troppo piccolo per non incorrere in seri **rischi di sicurezza**.
* In particolare, se 𝐵 fosse troppo piccolo, un attaccante potrebbe precompilare una tabella di 2𝐵 posizioni in cui alla generica posizione 𝑖 è memorizzato il **plaintext che ha**𝑖**come ciphertext**.
* In tal caso, l'attacco (denominato **codebook attack**) corrisponderebbe poi ad un semplice **table lookup**.
* Se, ad esempio, 𝐵 fosse 32, la tabella avrebbe 232 posizioni, ciascuna di 32 bit (4 byte), per un totale di 4 ⋅ 232 = 16 GB, cioè sarebbe **del tutto gestibile**.
* Con 𝐵 = 64 le dimensioni salirebbero a 267 ≈ 1.48 ⋅ 1020 byte, dunque senza alcuna possibilità di precomputazione e memorizzazione.

***Sull'attacco codebook***

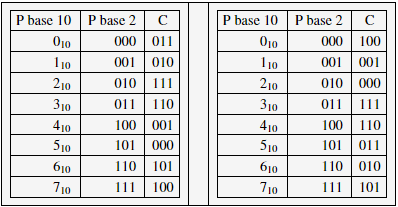
* È lecito chiedersi: Come può l'attaccante precomputare i ciphertext, visto che non dispone della chiave?
* Dopo aver ragionato sugli **obiettivi generali di sicurezza**, questa domanda ci porta a riflettere sui cosiddetti **modelli di attacco**.
* Un modello di attacco è essenzialmente un'**ipotesi** su quelle che sono le armi in mano all'attaccante.
* Qui ragioneremo solo di modelli **black box**, in cui l'attaccante non ha accesso o non utilizza informazioni diverse da input e output (plaintext e ciphertext).
* Esistono anche tipi di attacco che sfruttano **fenomeni fisici** piuttosto che proprietà matematiche.
* Per quanto possa a prima vista apparire impossibile, informazioni come il tempo di esecuzione di operazioni su una chiave privata possono essere utilizzate per violare un critto-sistema.
* Attacchi di questo tipo, che però non potremo prendere in considerazione, vengono detti (attacchi) **side-channel**.

***Attack model***

* L'assunzione più semplice è che l'attaccante **veda solo il ciphertext** e debba costruire l'attacco basandosi su quello.
* Questo modello di attaccante viene detto **Ciphertext-only attacker (COA)**
* COA è un modello in molti casi adeguato (almeno in crittografia simmetrica).
* Tuttavia, dal punto di vista del difensore, supporre che l'attaccante non conosca nulla al di fuori del testo cifrato può indurre ad **abbassare pericolosamente "la guardia"**.
* È invece prudente assumere che l'attaccante sia **sempre un passo avanti**.
* In ordine di "potenza" crescente, il modello successivo è noto come **Known-plaintext attacker** **(KPA)**, in cui l'attaccante (preliminarmente all'attacco) è stato in grado di osservare un certo numero di ciphertext con la conoscenza dei **corrispondenti plaintext**.
* Nei modelli precedenti, l'attaccante è comunque **passivo** perché non è lui/lei che determina quali testi cifrati ha possibilità di analizzare.
* Modelli più potenti prevedono che l'attaccante possa **effettuare query**.
* Il primo di tali modelli, in ordine di potenza, è il **Chosen-plaintext attacker (CPA)**, in cui l'attaccante può eseguire le cosiddette **encryption query**, può cioè scegliere un testo in chiaro e ottenere il corrispondente ciphertext.
* Si noti che questo (CPA) è il modello di riferimento nel caso della **crittografia asimmetrica**, in cui l'attaccante conosce la chiave di cifratura.
* L'ultimo modello prevede che l'attaccante possa eseguire, oltre alle encryption query, anche **decryption query**, possa cioè scegliere un ciphertext e vedere il corrispondente plaintext.
* Questo modello viene detto **Chosen-ciphertext attacker (CCA)**.

***Uno schema astratto per descrivere i block cipher***

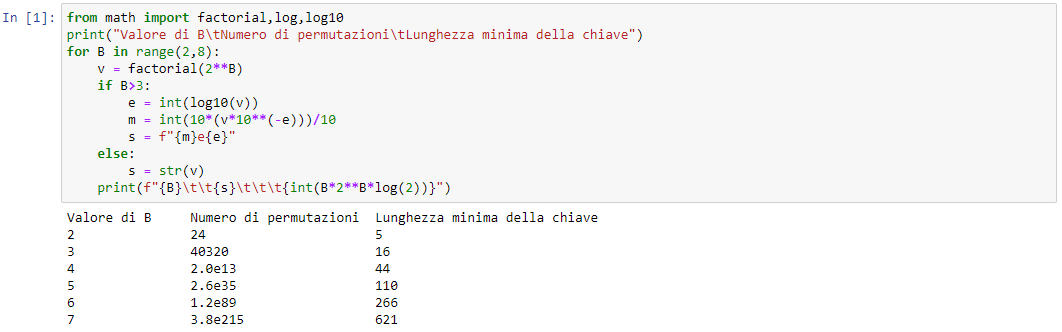
* Cifratura e decifrazione richiedono una chiave segreta 𝑘, la cui lunghezza tipicamente è nel range **fra 128 e 256 bit**.
* Sappiamo che, matematicamente, la cifratura è una **permutazione** poiché plaintext e ciphertext sono composti dallo stesso numero di bit.
* Ad ogni possibile plaintext di 𝐵 bit deve cioè corrispondere **un unico ciphertext di**𝐵**bit**, e viceversa, altrimenti il procedimento non sarebbe **reversibile**.
* Può essere istruttivo visualizzare la cifratura come un **processo in due fasi**:
  1. data la chiave 𝑘, si seleziona una specifica **lookup table**𝑇𝑘, di 2𝐵 posizioni, che memorizza una specifica permutazione delle sequenze di 𝐵 bit;
  2. la cifratura corrisponde ad un accesso a 𝑇𝑘, usando il **plaintext come chiave**, per recuperare il corrispondente ciphertext.
* Come semplice esempio, con 𝐵 = 3, a **due differenti chiavi** potrebbero corrispondere le seguenti tabelle.



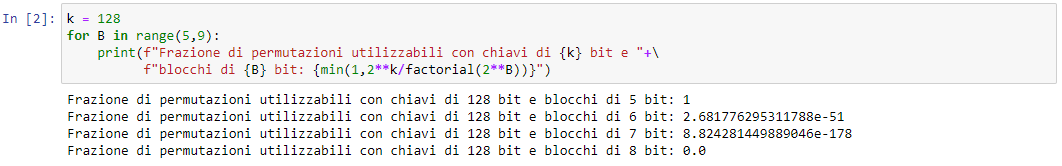
* Naturalmente l'unica informazione delle due tabelle che viene effettivamente memorizzata è la **colonna C** (ciphertext). Le prime due colonne sono gli **indici**, rispettivamente in base 10 e in base 2, e rappresentano entrambe il plaintext.

***Una riflessione su chiavi e permutazioni***

* Utilizzando il modello astratto possiamo ragionare quantitativamente sulle **permutazioni effettivamente utilizzabili** in un cifrario a blocchi e iniziare una riflessione su quali concretamente impiegare (e quali evitare).
* Sulla base del principio (che abbiamo enunciato in precedenza) in base al quale a chiavi diverse devono corrispondere permutazioni diverse, è immediato osservare che la lunghezza della chiave determina il **numero di lookup table**, e dunque di permutazioni, che possono essere utilizzate in un cifrario a blocchi con chiavi di quella lunghezza.
* Se dunque la chiave è lunga 𝑘 bit, il **numero di permutazioni usabili è** **2**𝑘.
* Questo valore, apparentemente "grande" (perché esponenziale nella lunghezza della chiave), deve essere visto sullo sfondo del **numero totale di permutazioni di** 𝐵 **bit**.
* Quest'ultima è una quantità che cresce molto più velocemente con 𝐵: essa è infatti pari a 2𝐵! (cioè il **fattoriale di 2**𝐵).
* È interessante verificare quanto dovrebbe essere lunga la chiave per poter utilizzare **tutte le permutazioni**, in funzione di valori crescenti di 𝐵.
* Con l'aiuto di Python, analizziamo i **casi 2 ≤** 𝐵 **≤ 7** (cioè per blocchi di lunghezza così corta da essere del tutto improponibili in un cifrario reale).

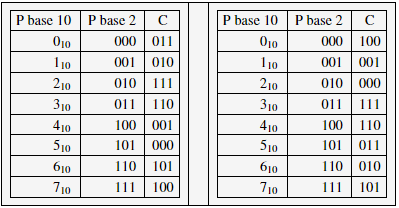


* Interessante anche l'analisi in qualche modo **"inversa"**.
* Fissiamo cioè una dimensione realistica per la chiave e valutiamo la **frazione di permutazioni** **utilizzabili** (rispetto a tutte quelle possibili) al crescere della dimensione 𝐵 dei blocchi.
* Ad esempio, con **chiavi di 128 bit** possiamo usarle tutte solo se 𝐵 ≤ 5, dopodiché la frazione di permutazioni utilizzabili diviene rapidamente **trascurabile**.



***Quali permutazioni usare?***

* Abbiamo visto che delle permutazioni possibili se ne può usare solo una **minuscola frazione**.
* D'altra parte molte permutazioni **non andrebbero bene** comunque.
* Ad esempio, le 26 possibili permutazioni usate dal cifrario di Cesare, associate alle 26 possibili chiavi, sono semplici **rotazioni**, dunque di facile inversione.
* Delle due permutazioni di sequenze di tre bit presentate nella tabella riportate sopra (che qui vengono nuovamente riprodotte per comodità) quella di destra non va sicuramente bene perché (quantomeno) **rivela sempre il bit meno significativo del plaintext**.



***Combinazioni lineari affini in***𝐙**2**

* In generale non vanno bene permutazioni che possono essere espresse mediante **trasformazioni lineari affini invertibili** sul campo 𝐙2

𝑥𝑃 + 𝑏 = 𝑦(𝑥)

in cui 𝑃 è una matrice non-singolare di ordine 𝐵 su 𝐙2, mentre 𝑥 (il plaintext), 𝑏 e 𝑦(𝑥) (il ciphertext) sono vettori riga di 𝐵 elementi, sempre con elementi in 𝐙2.

* Ad esempio, **la matrice di sinistra** nelle tabelle è rappresentabile come trasformazione lineare affine con

0 1 0

𝑃 = 1 0 0

0 0 1

e

𝑏 = 0 1 1

* Se la trasformazione è realizzata come permutazione lineare affine è infatti possibile, nel modello CPA, ricostruire 𝑏 e 𝑃 e dunque **invertire la trasformazione** stessa.
* Infatti 𝑏 semplicemente è l'immagine di 𝑂 (vettore di tutti zeri).

𝑦(𝑂) = 𝑂 ⋅ 𝑃 + 𝑏 = 𝑂 + 𝑏 = 𝑏

ed è quindi sufficiente, per determinarlo, effettuare la query che chiede il **ciphertext corrispondente al blocco di tutti 0**.

* Una volta noto 𝑏, siamo in grado di determinare esplicitamente 𝑃 con le sole ulteriori query in cui i blocchi da cifrare sono 𝑒𝑖 **= (0,…,0, 1, 0,…,0)**.

**indice**𝑖

* Scrivendo tutte le 𝐵 query in forma matriciale abbiamo:

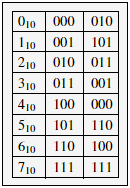
𝐼 ⋅ 𝑃 = 𝑦(𝐼)

ovvero 𝑃 = 𝑦(𝐼)

* In altri termini, la **risposta alle**𝐵**query ordinate**, è proprio la matrice di trasformazione 𝑃, che possiamo efficientemente invertire.
* Sul campo 𝐙2 il numero di matrici invertibili è dato dalla seguente formula, di derivazione relativamente semplice
* Il seguente codice Python ne implementa il calcolo.



* Ad esempio, la seguente permutazione **non è esprimibile** come combinazione lineare affine invertibile



***Proprietà delle permutazioni usate in pratica***

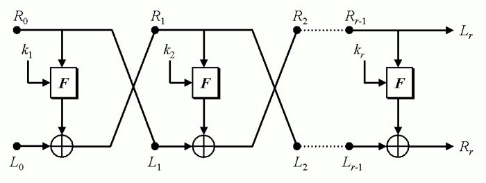
* Ovviamente l'algoritmo di cifratura **non segue i passaggi indicati**, che sono solo (come detto) una maniera per descriverne il comportamento osservabile.
* Tuttavia, questa descrizione ha il pregio di mettere in chiaro come, nella realizzazione concreta di un cifrario a blocchi, debba necessariamente esistere un qualche passaggio algoritmico per **forzare non linearità** nella permutazione realizzata.
* Altre proprietà che devono essere soddisfatte dalle permutazioni sono **diffusione** e **confusione**.
* La diffusione richiede che ogni minima modifica, localizzata in un punto del plaintext si rifletta su **tutto il testo cifrato**.
* La confusione è la caratteristica che nel ciphertext **distrugge ogni regolarità**, ogni eventuale pattern (ad esempio due caratteri consecutivi identici) presente nel plaintext.
* Come vedremo presentando il **cifrario di Feistel**, il modo per realizzare algoritmicamente permutazioni con tutte le caratteristiche suddette è realizzarlo come **rete di componenti più** **semplici**.

**Feistel networks**

* Più che un cifrario, si tratta di uno schema per realizzare cifrari a blocchi, fra i quali il più famoso è il ***Digital Encryption Standard*** (***DES***), per molti anni il "riferimento" nella cifratura simmetrica.
* Una rete di Feistel è composta da un certo numero di **stadi con identica struttura**.
* Ad ogni stadio viene utilizzata una specifica **round key**𝑘𝑖; tutte le round key sono diverse e ottenute dalla **chiave generale** utilizzata dal cifrario mediante un algoritmo noto come **key** **schedule**.
* Il generico stadio 𝑖 opera su **due semiblocchi** 𝐿𝑖 **ed** 𝑅𝑖 di dimensione 𝐵/2. Inizialmente, la **concatenazione** 𝐿0𝑅0 **dei semiblocchi** è proprio il blocco completo da cifrare.
* Uno dei due blocchi, per la precisione il **blocco di destra** 𝑅𝑖, passa inalterato allo stadio successivo come prossimo blocco di sinistra, 𝐿𝑖+1.
* Invece, il successivo blocco di destra, 𝑅𝑖+1, è ottenuto **applicando ad** 𝑅𝑖 **una trasformazione** 𝐹 dipendente dalla round key e il cui valore è posto in xor bitwise con 𝐿𝑖.

***Schema grafico e corrispondenti equazioni***

* Il seguente è un classico schema per descrivere una Feistel network



* ... e queste le corrispondenti **equazioni che definiscono i blocchi** (parametrizzate rispetto ad 𝐹):

𝐿1 = 𝑅0

𝑅1 = 𝐿0 ⊕ 𝐹(𝑅0, 𝑘1)

𝐿2 = 𝑅1

𝑅2 = 𝐿1 ⊕ 𝐹(𝑅1, 𝑘2)

…

𝐿𝑟 = 𝑅𝑟−1

𝑅𝑟 = 𝐿𝑟−1 ⊕ 𝐹(𝑅𝑟−1, 𝑘𝑟)

* Il primo problema da risolvere è verificare la **reversibilità del processo**.
* Questo è immediato, grazie alle **proprietà dell'or esclusivo**

𝑅𝑟−1 = 𝐿𝑟

𝐿𝑟−1 = 𝑅𝑟 ⊕ 𝐹(𝐿𝑟, 𝑘𝑟)

𝑅𝑟−2 = 𝐿𝑟−1

𝐿𝑟−2 = 𝑅𝑟−1 ⊕ 𝐹(𝐿𝑟−1, 𝑘𝑟−1)

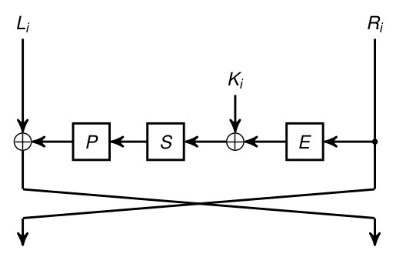
…

𝐿0 = 𝑅1

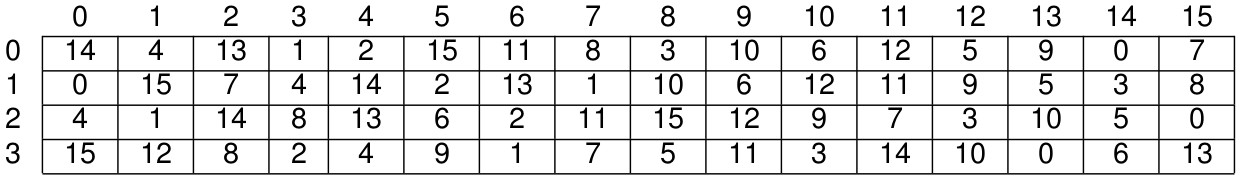
𝑅0 = 𝐿1 ⊕ 𝐹(𝑅1, 𝑘1)

* Questa seconda computazione, che calcola la decifrazione, può essere eseguita dalla rete di figura **"rovesciando" tutte le frecce** e dunque la direzione input/output dei semiblocchi.
* Tuttavia, questo **non è necessario**! È sufficiente infatti alimentare la stessa rete di figura con input 𝑅𝑟𝐿𝑟 e, naturalmente, usando le round key in ordine inverso.
* Il numero di round utilizzati in un cifrario reale deve essere tale da consentire ad una variazione nell'input di propagarsi a tutto il ciphertext (in altri termini, il numero di round è **legato alla diffusione**).

***La funzione F***



* La funzione 𝐹, i cui dettagli variano a seconda della particolare implementazione del cifrario (la figura mostra **il caso del DES**), ha comunque come scopo principale:
  1. di inserire **la chiave nel processo**, "mischiandola" con i bit del messaggio;
  2. assicurare **dipendenza non lineare** fra plaintext e ciphertext, attraverso l'S-box.
* Nel caso del DES i passaggi (da destra a sinistra in figura) sono i seguenti:
  1. I 32 bit di 𝑅𝑖 vengono **espansi** a 48 attraverso la duplicazione di alcuni bit;
  2. il risultato (48 bit) vengono messo in **xor con la round key** che, nel DES, è appunto di 48 bit;
  3. il nuovo risultato intermedio deve essere riportato a 32 bit, in vista dell'xor finale con il semiblocco 𝐿𝑖; questo è uno dei compiti dell'S-box, che è in realtà costituito da **8 tabelle, ciascuna indicizzata da 6 bit e le cui posizioni contengono 4 bit di output** (si veda la figura sotto);
  4. infine, il risultato prodotto dall'S-box (4⋅8=32 bit) è **permutato e posto in xor col semiblocco**𝐿𝑖 che, come sappiamo, costituirà il blocco destro del round successivo.



Una tabella componente l'S-box. Per selezionare la riga vengono usati 2 bit mentre 4 servono per la colonna. Il risultato è un numero fra 0 e 15 interpretato come sequenza di 4 bit.

***Key schedule***

* L'algoritmo con cui, a partire dalla chiave iniziale, vengono ottenute le varie round key è detto **key schedule**.
* Il DES usa una chiave "generale" di 56 bit (del tutto inadeguata oggi) che viene dapprima suddivisa in **due metà di 28 bit ciascuna**.
* Ad ogni round, ciascuna metà viene prima **ruotata a sinistra di 1 o 2 posizioni** (a seconda del round), dopodiché viene formata la round key prelevando **24 bit da entrambe**.

**Mode of operation**

* Il mode of operation è l'algoritmo che produce la **cifratura complessiva** di un messaggio utilizzando un cifrario a blocchi per le singole porzioni del messaggio.
* Dapprima viene effettuato il riempimento (**padding**) del messaggio iniziale 𝑀 di modo che la sua lunghezza sia un multiplo della dimensione dei blocchi
* Indicheremo con

𝑃 = 𝑃1||𝑃2||…||𝑃𝑛

la suddivisione in blocchi del plaintext già dopo il padding.

* Considereremo, esclusivamente a modo di esempio, due tecniche differenti per il mode of operation.

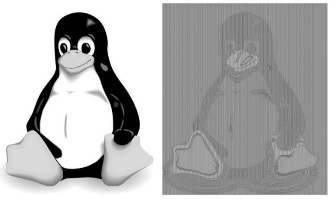
***Il modo ECB (Electronic Codebook Mode)***

* È l'algoritmo più semplice ma anche il **più vulnerabile**.
* Ogni singolo blocco viene infatti cifrato in modo **indipendente** dagli altri ma usando la **stessa** **chiave**.

𝐶𝑖 = 𝐸(𝑃𝑖, 𝐾), 𝑖 = 1, 2,…, 𝑛

dove chiaramente 𝐸 indica la funzione realizzata dal particolare cifrario a blocchi utilizzato e 𝐾 è la chiave.

* Il problema principale con ECB è che a blocchi di plaintext identici corrispondono **blocchi cifrati identici**.
* Non si pensi che questa sia una situazione molto improbabile.
* Come per il cifrario di Vigenère, se due blocchi di testo identici si **allineano alla dimensione del blocco**, i testi cifrati sono identici.
* Particolarmente problematico è il caso della cifratura di **immagini o grafici** (pensiamo agli "sfondi").
* Il seguente è un esempio famoso:



Il pinguino di Linux: immagine originale e immagine cifrata con modo ECB

Source: J.P. Aumasson, Serious Cryptography, No Starch Press (2018)

***Il modo CBC (Cipher Block Chaining)***

* La cifratura avviene nel modo seguente:

𝐶1 = 𝐸(𝑃1 ⊕ 𝐼𝑉, 𝐾), 𝐶𝑖 = 𝐸(𝑃𝑖 ⊕ 𝐶𝑖−1, 𝐾), 𝑖 = 2, 3,…, 𝑛

dove 𝐼𝑉 indica un blocco iniziale detto appunto **initialization vector**.

* La decifrazione è possibile grazie ancora alle **proprietà dell'or esclusivo**:

𝑃1 = 𝐷(𝐶1, 𝐾) ⊕ 𝐼𝑉, 𝑃𝑖 = 𝐷(𝐶𝑖, 𝐾) ⊕ 𝐶𝑖−1

* Il vettore di inizializzazione può essere scelto in vario modo ma la soluzione più comune è di utilizzare un **nonce**.

**Uno schema di padding e l'attacco "Padding oracle"**

* Poiché la lunghezza del plaintext 𝑃 da cifrare può non essere (e in generale non è) un multiplo della dimensione 𝐵 del blocco, è necessario (come già accennato) procedere ad una **estensione** del messaggio che ne porti la lunghezza ad un tale multiplo.
* In gergo tecnico, tale estensione viene chiamata ***padding*** (riempimento).
* È ovviamente necessario che il padding sia poi riconosciuto dal destinatario, cioè non confuso con il messaggio vero e proprio.
* Semplici soluzioni, come inserire una sequenza di zeri in coda, sono chiaramente destinate al fallimento.
* È necessario che mittente e destinatario seguano un vero e proprio **protocollo**.

***Lo schema di padding ISO/IEC 9797-1***

* Qui e nel resto di questa sezione sul padding supponiamo che i blocchi siano composti da 16 byte (128 bit), come nel caso dell'***American******Encryption Standard*** (AES).
* Lo standard ISO/IEC 9797-1 per il padding prevede di operare come segue.
* Come prima cosa si calcola la lunghezza ℓ del plaintext 𝑃 (lunghezza **espressa in bit**).
* Si prepara quindi il plaintext modificato 𝑃′ inserendo, nell'ordine:
  1. Un primo blocco in cui viene scritta la lunghezza ℓ di 𝑃 utilizzando tutti i 16 byte del blocco (quindi inserendo eventuali **zeri non significativi** all'inizio).
  2. Il plaintext originale 𝑃.
  3. Tanti bit quanto serve per completare l'ultimo blocco (eventualmente nessuno, nel caso in cui il messaggio abbia lunghezza che è multiplo di 128).
* Ad esempio, per un plaintext 𝑃 lungo **232 bit**, la codifica sarebbe come segue (si tenga presente che **23210=E816=111010002**):

00…011101000 𝑃 00…0

120 24

***Lo schema di padding oggetto dell'attacco***

* Un differente schema di padding, utilzzato per portare l'attacco di cui parleremo subito dopo, prevede invece di **non inserire** il blocco iniziale con la lunghezza ma di utilizzare **differenti sequenze di riempimento**, a seconda della lunghezza del plaintext originale 𝑃.
* Più precisamente, lo schema prevede quanto segue, dove la lunghezza ℓ di 𝑃 è espressa non in bit bensì in **byte**.
  + Se ℓ=16𝑘, per un qualche 𝑘>0, si appende un intero blocco in fondo a 𝑃, formato da 16 byte identici, il cui valore numerico è precisamente 16 (ovvero 1016)

10161016…1016

16

* + Se ℓ=16𝑘+1 (cioè, nel caso in cui l'ultimo blocco di 𝑃 occupi un solo byte dei 16 disponibili), in coda all'ultimo blocco si aggiungono 15 byte identici, il cui valore numerico è 15 (ovvero 0𝐹16):

0𝐹160𝑓16…0𝐹16

15

* + Se ℓ=16𝑘+2 (cioè, nel caso in cui l'ultimo blocco di 𝑃 occupi due byte dei 16 disponibili), in coda all'ultimo blocco si aggiungono 14 byte identici, il cui valore numerico è 14 (ovvero 0𝐸16):

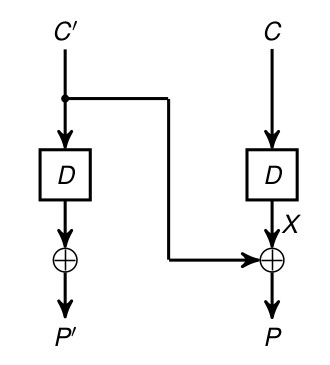
0𝐸160𝐸16…0𝐸16

14

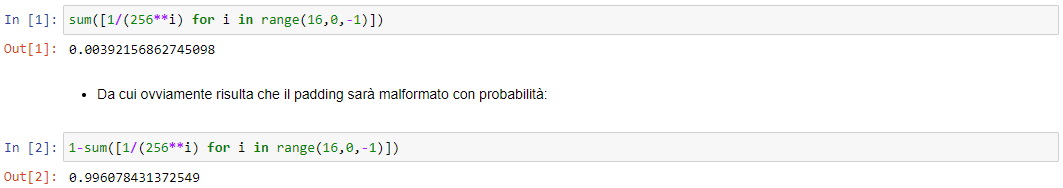
* + ....
  + Se ℓ=16𝑘+15 (cioè, nel caso in cui l'ultimo blocco di 𝑃 occupi tutti i byte disponibili eccetto uno), in coda all'ultimo blocco si aggiunge il byte di valore numerico 1 (ovvero 0116).
* Il vantaggio di questo schema è che utilizza un blocco in più **solo se la lunghezza del plaintext (in byte) è già un multiplo della lunghezza del blocco**. E chiaramente è necessario farlo in questo caso.
* In sede di decifrazione, eliminare il padding è molto semplice.
* Si guarda l'**ultimo** **byte dell'ultimo blocco decifrato** e se ne interpreta il valore numerico.
* Se tale valore è 𝑛, 𝑛 ∈ {1, 16}, il padding è formato precisamente dagli **ultimi**𝑛**byte**, che vengono eliminati.
* In particolare, se 𝑛 = 16, si elimina del tutto l'**ultimo** **blocco**.

***Condizioni per l'attacco***

* L'attacco funziona con lo schema di padding appena illustrato e con qualsiasi cifrario blocchi che operi unitamente al **mode of operation CBC**.
* Il modello di attacco è "di fatto" debole; Eva non necessità altro che la conoscenza del cypertext. Eva deve poter sì effettuare query di decifrazione, ma l'attacco è verso un server che tipicamente risponde...
* È però necessario che il software che effettua la decifrazione sollevi un'eccezione (cioè, segnali un errore) nel caso di **padding malformato** (cosa che faceva, ad esempio, SSL/TLS).
* Un padding può essere malformato se l'ultimo byte ha valore numerico **0 oppure maggiore di 16**.
* Se invece ha valore 𝑗 ∈ {1, 16}, è malformato nel caso in cui almeno uno dei **precedenti**𝑗**−1 byte**non ha valore 𝑗
* Per comodità, schematizziamo gli ultimi due round della decifrazione con CBC. Non usiamo gli indici ma indichiamo con 𝐶′ e 𝐶 rispettivamente il penultimo e l'ultimo blocco cifrato cui corrispondono i blocchi in chiaro 𝑃′ e 𝑃.

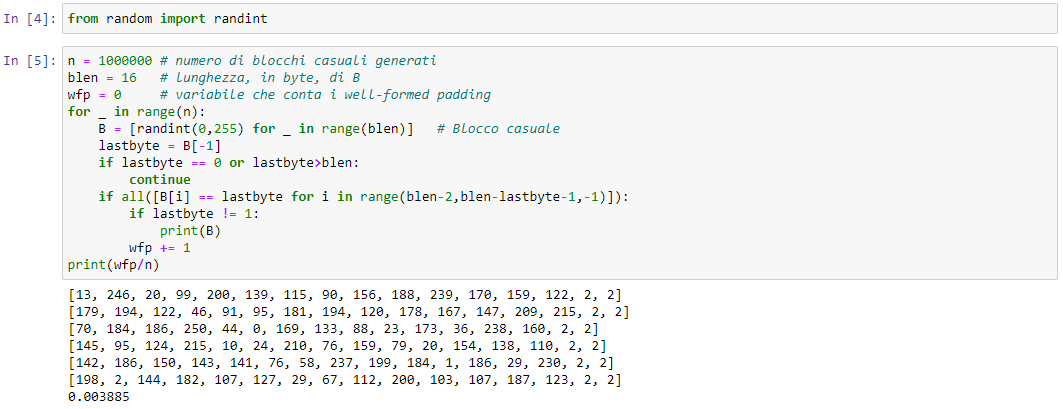


* Notiamo che se modifichiamo arbitrariamente il blocco 𝐶′, con alta probabilità il blocco 𝑃 che verrà messo in chiaro avrà un padding malformato.
* Si può facilmente calcolare la probabilità dell'evento complementare, e cioè che il padding sia ben formato



***Un algoritmo Monte Carlo come esercizio***

* Come esercizio è anche istruttivo **stimare** la probabilità con un **algoritmo Monte Carlo**.
* Il programma potrebbe essere meglio comprensibile dopo aver letto il successivo sulla randomness. L'algoritmo è tuttavia molto semplice da poter essere facilmente descritto.
  1. Viene generato un certo numero di sequenze composte da **16 numeri casuali** compresi fra 0 e 255 (i numeri interi positivi rappresentabili in binario su 8 bit).
  2. Si verifica che la sequenza includa un **padding ben formato** (ultimi 𝑘 numeri uguali a 𝑘, per un qualche valore di 𝑘 compreso fra 1 e 16).
  3. In caso affermativo, si conta **un caso favorevole**.
  4. Al termine, la probabilità è approssimata dal rapporto fra casi favorevoli e sequenze generate.
  5. Questo valore sarà un'approssimazione tanto migliore quanto più è alto il numero di sequenze "testate".

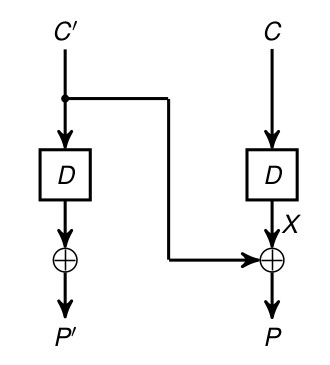


* Come si vede dall'esempio (con le stampe non commentate) il **"grosso" della probabilità** che il padding "casuale" sia ben formato è dato dall'accadimento che l'**ultimo byte sia 0116**.
* D'altra parte è facile calcolare esplicitamente la probabilità condizionata che l'ultimo byte sia 0116 dato che il padding è ben formato (wfp).

pr[lastbyte = 0116 | wfp] =  = =



***L'attacco***



* Eva sceglie a caso tutti i byte di 𝐶′ **escluso l'ultimo**.
* L'ultimo byte viene posto al valore 0016 e viene **incrementato di una unità** ad ogni query.
* Se, per un dato valore di 𝑥𝑦16, non viene generato errore, per quanto detto sopra Eva può essere ragionevolmente sicura che **l'ultimo byte di**𝑃**sia 0116**.
* Risulta cioè 𝑥𝑦**16** ⊕ 𝑥**[15] = 0116** e dunque, date le proprietà dell'or esclusivo, Eva può recuperare il valore dell'ultimo byte di 𝑋 nel modo seguente: 𝑥**[15] =** 𝑥𝑦**16** ⊕ **0116**
* A questo punto Eva ripete il trucco alla luce della conoscenza di 𝑥[15].
* Fissa cioè il valore di 𝐶′[15] in modo tale che 𝐶′[15] ⊕ 𝑥[15] = 0216, e dunque (sempre per le proprietà dello xor)

𝐶**′[15] =** 𝑥**[15]** ⊕ **0216**

sceglie poi i primi quattordici byte di 𝐶′ in modo random e inizia a far variare il byte 𝐶′[14] a partire da 0 con incrementi unitari.

* Per lo stesso argomento di sopra, quando, per un certo valore 𝑟𝑠16 assegnato a 𝐶′[14], non viene segnalato errore è altamente probabile che il penultimo byte del testo decifrato sia 0216, da cui Eva ricava

𝑋**[14] =** 𝑟𝑠**16** ⊕ **0216**

* A questo punto si può immaginare il seguito. Conoscendo i due byte 𝑋[14] e 𝑋[15], Eva fissa i byte 𝐶′[14] e 𝐶′[15] in modo tale che gli ultimi due byte del messaggio decodificato siano **03160316**, sceglie a caso i primi tredici byte di 𝐶′ e comincia ad eseguire richieste facendo variare il terz'ultimo byte (𝐶′[13]) a partire da 0016.
* Procedendo in questo modo Eva riesce a conoscere **interamente**𝑋 e, usando il valore 𝐶′ "vero", è in grado di ottenere il plaintext 𝑃
* Il procedimento può essere tranquillamente **iterato** per conoscere tutti i blocchi del plaintext.
* Per il penultimo blocco (la cui cifratura dipende dai blocchi precedenti ma **non dall'ultimo**) Eva non deve fare altro che considerare i blocchi cifrati escluso l'ultimo.
* Deve cioè fare **"come se"** il messaggio cifrato fosse composto solo dai primi 𝑛−1 blocchi e sottoporre questi (con le modifiche indicate in precedenza) al software ricevente (es. un server TLS).
* Per gli altri il discorso è analogo.