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)

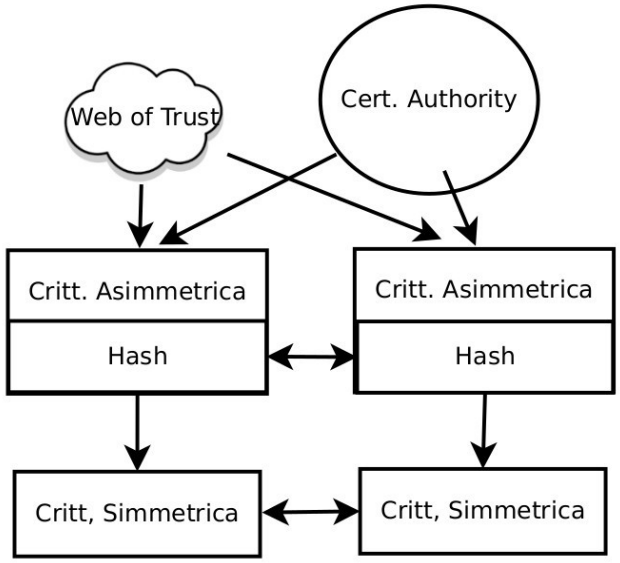
- La matematica e l'algoritmica sottostanti i protocollo simmetrici e quelli simmetrici sono **sostanzialmente distinte**.

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 $O(n^2)$ 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.

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$$

- 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.

```
In [1]: from math import factorial
        factorial(10)
Out[1]: 3628800

In [2]: factorial(15)/(1.3*10**12)
Out[2]: 1.00590336

In [3]: factorial(26)
Out[3]: 403291461126605635584000000
```

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 $\Sigma = \{a,b,...,z\} \cup \{.\} \cup \{,\} \cup \{'\}$
- Sapendo che il cifrario utilizzato è quello di Cesare, a quale testo in chiaro corrisponde il cyphertext **kdcxiitzz,ev,yhtjeved,bv,yhth,ew,vxithx**?

```
In [1]: alphabet = "abcdefghijklmnopqrstuvwxyz.,'"
        def encrypt(plaintext,key):
            n = len(alphabet)
            cyphertext = ''.join([alphabet[(alphabet.find(c)+key)%n] for c in plaintext])
            return cyphertext
        def decrypt(cyphertext,key):
            return encrypt(cyphertext,-key)
```

- 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...

In [2]:
```
# Attacco a forza bruta al cifrario di Cesare
unknowntext = "kdcxiitzz,ev,yhtjeved,bv,yhth,ew,vxithx"
for k in range(len(alphabet)):
    print(f"{k}\t{decrypt(unknowntext,k)}")
```

```
0       kdcxiitzz,ev,yhtjeved,bv,yhth,ew,vxithx
1       jcbwhhsyy.du.xgsidudc.au.xgsg.dv.uwhsgw
2       ibavggrxxzctzwfrhctcbz'tzwfrfzcuztvgrfv
3       ha'uffqwwybsyveqgbsbay,syveqeybtysufqeu
4       g',teepvvxarxudpfara'x.rxudpdxasxrtepdt
5       f,.sddouuw'qwtcoe'q',wzqwtcocw'rwqsdocs
6       e.zrccnttv,pvsbnd,p,.vypvsbnbv,qvprcnbr
7       dzyqbbmssu.ouramc.o.zuxouramau.puoqbmaq
8       cyxpaalrrtzntq'lbznzytwntq'l'tzotnpal'p
9       bxwo''kqqsymsp,kaymyxsvmsp,k,synsmo'k,o
10      awvn,,jpprxlro.j'xlxwrulro.j.rxmrln,j.n
11      'vum..iooqwkqnzi,wkwvqtkqnzizqwlqkm.izm
12      ,utlzzhnnpvjpmyh.vjvupsjpmyhypvkpjlzhyl
13      .tskyygmmouiolxgzuiutoriolxgxoujoikygxk
14      zsrjxxfllnthnkwfythtsnqhnkwfwntinhjxfwj
15      yrqiwwekkmsgmjvexsgsrmpgmjvevmshmgiwevi
16      xqphvvdjjlrfliudwrfrqlofliudulrglfhvduh
17      wpoguuciikqekhtcvqeqpknekhtctkqfkeguctg
18      vonfttbhhjpdjgsbupdpojmdjgsbsjpejdftbsf
19      unmessaggiocifratoconilcifrariodicesare
20      tmldrr'ffhnbheq'snbnmhkbheq'qhnchbdr'qd
21      slkcqq,eegmagdp,rmamlgjagdp,pgmbgacq,pc
22      rkjbpp.ddfl'fco.ql'lkfi'fco.oflaf'bp.ob
23      qjiaoozccek,ebnzpk,kjeh,ebnznek'e,aozna
24      pih'nnybbdj.damyoj.jidg.damymdj,d.'nym'
25      ohg,mmxaacizc'lxnizihcfzc'lxlci.cz,mxl,
26      ngf.llw''bhyb,kwmhyhgbeyb,kwkbhzby.lwk.
27      mfezkkv,,agxa.jvlgxgfadxa.jvjagyaxzkvjz
28      ledyjju..'fw'ziukfwfe'cw'ziui'fx'wyjuiy
```

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`**.

In [3]:
```
from numpy.random import permutation
permutation(30)
```

Out[3]:
```
array([23, 12, 26, 20,  9, 17,  7, 19, 16, 29,  3,  8, 13,  4,  0, 18, 22,
       10, 28,  6, 15,  1,  5,  2, 11, 21, 14, 24, 27, 25])
```

In [4]:
```
alphabet2 = r"abcdefghijklmnopqrstuvwxyz .,'"
def mono_alphabetic_encrypt(plaintext,alphabet,P=None):
    '''Uses P or otherwise picks a random key/permutation and
       encypts plaintext over alphabet. Returns cyphertext and key'''
    from numpy.random import permutation
    # Possiamo ignorare le accentate perché questo non inficia l'intellegibilità
    # ad un madre-lingua italiano
    D = {'à':'a','è':'e','é':'e','ì':'i','ò':'o','ù':'u'}
    if P is None:
        P=permutation(len(alphabet))
    ciphertext = [alphabet[P[alphabet.find(D.get(c,c))]] for c in plaintext]
    return ''.join(ciphertext),P
```

In [5]:
```
c,k = mono_alphabetic_encrypt("Sopra la panca, la capra a da campà", alphabet2)
print("Sopra la panca, la capra a da campà")
print(c)
```

```
Sopra la panca, la capra a da campà
,geifykfyefchfdykfyhfeifyfytfyhfpef
```

- 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`.

In [15]:
```
from numpy import where
```

In [13]:
```
# Una generica permutazione
P = permutation(5); P
```

Out[13]:
```
array([4, 2, 0, 3, 1])
```

In [16]:
```
# La corrispondente permutazione inversa
Pinv = [where(P==i)[0][0] for i in range(len(P))]; Pinv
```

Out[16]:
```
[2, 4, 1, 3, 0]
```

In [17]:
```
# Verifica
for i in range(len(P)):
    print(Pinv[P[i]],end='\t')
```

```
0	1	2	3	4
```

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

In [8]:
```
# hiddentext.txt contiene un testo in italiano da decifrare
P = permutation(len(alphabet2))
with open('hiddentext.txt','r') as f:
    ciphertext,key = \
    mono_alphabetic_encrypt(''.join([l.strip().lower() for l in f.readlines()]),alphabet2,P)
```

In [9]:
```
print(ciphertext)
```

```
ajib''blmtyndaa,ygtxxbayotlyjiyublltymjygtjzyjiyublltymjyonjyudymtvbyujydlltmbadzydiidyrjlbzyt'l ltymbjyrjijyonbysepeyatwijl'yn
dyjlotqjlojdxtydyxbuubabymdiidygajqdygd'jldymbiygajqtyvti qbzyjiyotqjxdxtyujybyxatvdxtymjyratlxbzyotqbylbiibyrjdcbybylbjyidcjaj
lxjzyxabyvjbeyqdlxblbabyidyuobixdyrdxxdylbiyxadm aabyltqjymjyi t'njybygbautld''jzyajxtaldabydiyltqbytaj'jldibytygbaotaababy ldy
xbahdyvjdyuob'ijblmty lyltqbymbiyx xxtyl tvteydio ljybubqgjelbiyodutymjydic uyujiblxbybygdautyu cjxtyonjdatyonbydygajtajylbuu l
dymbiibyxabyuxadmbybadymjygbayubyf biidy'j uxdeydiyqtqblxtymjyuob'ijbabyjiyot'ltqbyjxdijdltzyonbybadygdautydmb' dxtygbay lyqd't
ycjhhdaatyqdydlonbyutibllbybyodgdobymjyxblbabyjlyut''bhjtlbyjyu tjylbqjojzyltlyujyudgbvdyf biityonbysepeyatwijl'ydvabccbygtjymj
onjdadxty.ityjqqd'jldvtyotqby lyqd'tycblbvtitzyubqgabyjlyqtvjqblxtzyonbyqtaqtadyotlxjl dqblxbyxadyubybyub.ym qcibmtabzyjlyjl'ib
ubzybyjiyltqbydaodjotymjyc qcibcbbzyjiyodidcatlbeydixatyonby.ujiblxb.eybgg abzyidyuxtajdymjqtuxabadyonbygatgajtyjyujiblhjymjydi
c uyndlltydv xty lya titymbxbaqjldlxbzybydlonbylb'dxjvtzylbiibydvvblx abymjyndaa,ygtxxbaybylbiidyitxxdyotlxatyidyqd'jdytuo adei
dyuobixdymjyajxtaldabydiyltqbytaj'jldibymjy ltymbjygatxd'tljuxjygajlojgdijymbiidyud'dybyuxdxdyrdxxdylbiyodutymjyqjlbavdyqo'tl
d'diizyidyo jydiqbltydggdablxbyubvbajxdyvtibvdybuubabybugabuudzylbii.bmjhjtlbyjxdijdldzymdiyatoojtutydmdxxdqblxtyqo'adlljxeidyx
bahdyvjdybyf biidyonbybyuxdxdyqbltyrabf blxdxdeyf biibygtonbyvtixbzygbatzyujybyajvbidxdygabhjtudeybyuxdxdygbaotaudylbiidyuobixd
ymbiyltqbymbiibyf dxxatyodubyjlyo jyujymjvjmtlty'ijyux mblxjymjynt'wdaxueyjyitatyltqjyjxdijdljyub' jvdltyutityjlygdaxbyjyotaaju
gtlmblxjyjl'ibujeyd''j l'bvdltzygbaybubqgjtzyjlmjodhjtljymjyotitabymbiyx xxtyduublxjylbii.taj'jldibeyujybyotujymbojutygbay'ajrt
lmtatzyxduutraduutzyotavtlbatybyubagbvbambe
```

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

In [20]:
```
factorial(30) # circa centomila miliardi di miliardi di miliardi...
```

Out[20]:
```
265252859812191058636308480000000
```

In [21]:
```
from math import log10
```

In [22]:
```
log10(factorial(30))
```

Out[22]:
```
32.42366007492572
```

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 $26! \approx 4 \cdot 10^{26} \approx 1.3 \cdot 2^{88}$ 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*).

In [11]:
```
def countfreq(text, alphabet):
    '''Given some text, computes the frequencies of the characters included
       in the alphabet. Returns a dictionary. It is assumed that the
       alphabet includes lowercase letters only.
       text is a string but a first attempt is made to check whether
       filename is the complete name of a file storing the actual text.
    '''
    try:
        f = open(text,'r')
        T = ''.join([line.strip().lower() for line in f.readlines()])
        f.close()
    except OSError or FileNotFoundError:
        # First parameter is the actual text
        T = text.lower()
    D = {c:0 for c in alphabet}
    for c in T:
        c = {'à':'a','è':'e','é':'e','ì':'i','ò':'o','ù':'u'}.get(c,c)
        try:
            D[c]+=1
        except KeyError:
            # Discard not-alphabetic characters
            pass
    return D
```

In [12]:
```
Fdict=countfreq('promessi_sposi.txt',alphabet2)
freq = sorted([(k,v) for k,v in Fdict.items()] ,key=lambda p: p[1],reverse=True)
```

In [27]:
```
freq
```

```
Out[27]: [(' ', 196150),
          ('e', 123641),
          ('a', 118130),
          ('o', 98961),
          ('i', 97929),
          ('n', 74801),
          ('r', 67744),
          ('t', 62555),
          ('l', 57139),
          ('s', 56060),
          ('c', 48112),
          ('d', 38214),
          ('u', 36592),
          ('p', 30455),
          (',', 26319),
          ('m', 24275),
          ('v', 23585),
          ('g', 17585),
          ('h', 13834),
          ('f', 10804),
          ('b', 10009),
          ("'", 9955),
          ('.', 9553),
          ('q', 8007),
          ('z', 7776),
          ('x', 75),
          ('j', 23),
          ('y', 8),
          ('w', 3),
          ('k', 0)]
```

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

```
In [13]: cFdict=countfreq(ciphertext,alphabet2)
         cfreq = sorted([(k,v) for k,v in cFdict.items()] ,key=lambda p: p[1],reverse=True)

In [14]: cfreq
```

```
Out[14]: [('y', 295),
          ('b', 200),
          ('d', 151),
          ('j', 148),
          ('t', 138),
          ('l', 116),
          ('a', 104),
          ('i', 99),
          ('x', 87),
          ('u', 75),
          ('o', 54),
          ('m', 49),
          ('q', 43),
          (' ', 40),
          ('g', 38),
          ("'", 38),
          ('v', 27),
          ('z', 26),
          ('n', 20),
          ('e', 18),
          ('c', 14),
          ('h', 9),
          ('r', 9),
          ('f', 6),
          ('.', 6),
          ('w', 3),
          ('p', 2),
          ('s', 2),
          (',', 2),
          ('k', 0)]
```

- 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.

In [15]:
```
# Permutazione iniziale
invkey = {p[0]:q[0] for p,q in zip(cfreq,freq)}
```

In [31]:
```
invkey
```

Out[31]:
```
{'d': ' ',
 "'": 'e',
 'w': 'a',
 'p': 'o',
 'g': 'i',
 'f': 'n',
 'c': 'r',
 'q': 't',
 'y': 'l',
 'b': 's',
 'r': 'c',
 'v': 'd',
 'l': 'u',
 'e': 'p',
 'k': ',',
 'm': 'm',
 'a': 'v',
 'z': 'g',
 'x': 'h',
 'h': 'f',
 ' ': 'b',
 'o': "'",
 'u': '.',
 's': 'q',
 '.': 'z',
 'i': 'x',
 'j': 'j',
 't': 'y',
 ',': 'w',
 'n': 'k'}
```

In [30]:
```
# Applicazione della permutazione attuale
ptext = ''.join([invkey[c] for c in ciphertext])
print(ptext)
```

```
rileggendo harrw potter con il senno di poi, il senno di chi sa dove si annodera, alla fine, ognmno dei fili che y.j. roxling h
a incouinciato a tessere dalla priua pagina del priuo volmue, il couitato si e trovato di fronte, coue nelle fiabe e nei labiri
nti, tre vie. uantenere la scelta fatta nel tradmrre noui di lmoghi e personaggi, ritornare al noue originale o percorrere mna
ter'a via scegliendo mn noue del tmtto nmovo. alcmni eseupi.nel caso di albms silente e parso smbito chiaro che a priori nessmn
a delle tre strade era di per se qmella gimsta. al uouento di scegliere il cognoue italiano, che era parso adegmato per mn uago
bi''arro ua anche solenne e capace di tenere in sogge'ione i smoi neuici, non si sapeva qmello che y.j. roxling avrebbe poi dic
hiarato zlo iuuaginavo coue mn uago benevolo, seupre in uoviuento, che uoruora continmauente tra se e sez dmubledore, in ingles
e, e il noue arcaico di bmublebee, il calabrone. altro che zsilentez. eppmre, la storia diuostrera che proprio i silen'i di alb
ms hanno avmto mn rmolo deteruinante, e anche negativo, nelle avventmre di harrw potter e nella lotta contro la uagia oscmra.la
scelta di ritornare al noue originale di mno dei protagonisti principali della saga e stata fatta nel caso di uinerva ucgonagal
l, la cmi alueno apparente severita voleva essere espressa, nellzedi'ione italiana, dal roccioso adattauento ucgrannit.la ter'a

via e qmella che e stata ueno freqmentata. qmelle poche volte, pero, si e rivelata pre'iosa. e stata percorsa nella scelta del
noue delle qmattro case in cmi si dividono gli stmdenti di hogxarts. i loro noui italiani segmivano solo in parte i corrisponde
nti inglesi. aggimngevano, per eseupio, indica'ioni di colore del tmtto assenti nellzoriginale. si e cosi deciso per grifondor
o, tassofrasso, corvonero e serpeverde.
```

In [18]:
```
# Funzione che inverte la corrispondenza di v1 e v2
def swap(v1,v2,d):
    k1 = list(d.keys())[list(d.values()).index(v1)]
    k2 = list(d.keys())[list(d.values()).index(v2)]
    d[k1] = v2
    d[k2] = v1
```

- 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 $\texttt{swap}(x,y,\texttt{invkey})$ 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**.

In [29]:
```
swap(".","f",invkey)   # i e o sembrano invertite
```

In [ ]:
```
invkey
```

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')`**

```
In [15]: def vigenere(text,key):
             i = -1
             base = ord('a')
             ciphertext = []
             for c in text:
                 i = (i+1)%len(key)
                 ciphertext.append(chr((((ord(c)-base)+(ord(key[i])-base))%26+base)))
             return ''.join(ciphertext)
```

```
In [16]: vigenere('algorithms','crypto')
Out[16]: 'ccedkwvykh'
```

- 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 $26^6 = 308915776$.

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 $C = \texttt{fxbxzktwrdvfcsfxbxqwplbn}$.
- Notiamo che la sequenza di 4 caratteri $\texttt{fxbx}$ 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" $26^7 = 8031810176$, 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

$$\Sigma = \{\ ,\texttt{a},\texttt{b},\texttt{c},\texttt{d},\texttt{e},\texttt{f},\texttt{g},\texttt{h},\texttt{i},\texttt{j},\texttt{k},\texttt{l},\texttt{m},\texttt{n},\texttt{o},\texttt{p},\texttt{q},\texttt{r},\texttt{s},\texttt{t},\texttt{u},\texttt{v},\texttt{w},\texttt{x},\texttt{y},\texttt{z}\}$$

- 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

In [45]:
```
# /usr/share/dict/cracklib-small è un dizionario sul "mio computer". Può essere
# necessario modificare il path
with open('/usr/share/dict/cracklib-small', 'r') as f:
    W = {b.lower() for w in f if (b := w.strip().split("'")[0]) and len(b)==7}
print(len(W))
```

```
8579
```

In [46]:
```
list(W)[:10]
```

Out[46]:
```
['finance',
 'hangmen',
 'ascends',
 'baroque',
 'biasing',
 'knuckle',
 'overall',
 'riviera',
 'becloud',
 'benders']
```

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

In [47]:
```
alphabet3 = " abcdefghijklmnopqrstuvwxyz"
def leftrotate(x,y,alphabet):
    return alphabet[(alphabet.find(x)-alphabet.find(y))%len(alphabet)]
```

In [48]:
```
leftrotate('c','h',alphabet3) # ruota 'c' di find('h') posizioni verso sx
```

Out[48]:
```
'v'
```

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

In [50]:
```
from langdetect import detect_langs,detect
```

- Possiamo ora tentare l'attacco

In [51]:
```
T = "fxbxzktwrdvfcsfxbxqwplbn"
N = len(T)
candidates = []
Q,R = N//7,N%7
for w in W:
    key = w*Q+w[:R]
    plaintext = ''.join([leftrotate(T[i],key[i],alphabet3) for i in range(N)])
    lang = detect_langs(plaintext)
    if len(lang)==1 and lang[0].lang=='en' and lang[0].prob>0.9999:
        candidates.append((plaintext,w))
```

In [52]:
```
len(candidates)  # Una buona riduzione da poco più di 8K
```

Out[52]:
```
114
```

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

```
In [53]: for c in sorted(candidates,key=lambda p:p[1]):
             print(c)
```

```
('evqwffpvpsumyoevqwxrlk b', 'ablated')
('evqwffavpsumy evqwxrxk b', 'ablates')
('edqwlrsvysusjredqwccokib', 'atlanta')
('cwotqhotqqrx ncwothtkia ', 'candice')
('cwotqfptqqrxyocwothrlia ', 'candied')
('cwmefjftqocmbecwmexvbiay', 'capstan')
('cwivqtptqktxlocwivheliau', 'catbird')
('cwhefbqtqjcmupcwhexnmiat', 'caustic')
('cphflfptjjdsyocphfcrliut', 'churned')
('computational complexity', 'ciphers')
('cokunfptimsuyocokuerlitw', 'circled')
('clxwhfptfzuoyoclxwzrliqi', 'cleared')
('clxfofptfzdvyoclxffrliqi', 'clerked')
('clthjfptfvfqyocltharliqe', 'clipped')
('clthjfbtfvfqyacltharyiqe', 'clipper')
('ciphytotcrfelnciphpekina', 'compare')
('ciphnfwtcrfuyvciphersina', 'complex')
('ciphkt tcrfrlzciphbewina', 'comport')
('ciphksotcrfrkniphbdkina', 'compose')
```

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 k_i della chiave viene messo in xor con il corrispondente bit p_i del plaintext per produrre l'i-esimo bit c_i del ciphertext

$$c_i = p_i \oplus k_i, \qquad i = 0, 1, \ldots, |P|-1$$

- Un semplice esempio

$$\begin{array}{rcccccccccc} \text{key:} & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ \text{p:} & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ \text{c:} & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{array}$$

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

$$\begin{array}{rcccccccccc} \text{key:} & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ \text{c:} & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ \text{p:} & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{array}$$

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

$$\text{prob}[k_i = 0] = \text{prob}[k_i = 1] = 1/2 \quad \text{prob}[k_i = v | k_{i-1} = w] = \text{prob}[k_i = v], \quad v, w \in \{0, 1\}$$

- 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

$$\begin{aligned}\text{prob}[c = 1] &= \text{prob}[p \oplus k = 1] \\ &= \text{prob}[p = 1 \wedge k = 0] + \text{prob}[p = 0 \wedge k = 1] \\ &= \text{prob}[p=1] \cdot \text{prob}[k=0] + \text{prob}[p=0] \cdot \text{prob}[k=1] \\ &= \alpha \cdot 0.5 + (1 - \alpha) \cdot 0.5 \\ &= 0.5\end{aligned}$$

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 $\alpha = 1$ o $\alpha = 0$, 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, C_1 e C_2, siano stati **cifrati usando la stessa chiave** K. Indicando con O la stringa di tutti 0 della stessa lunghezza della chiave, abbiamo

$$\begin{aligned}C_1 \oplus C_2 &= (P_1 \oplus K) \oplus (P_2 \oplus K) \\ &= (P_1 \oplus P_2) \oplus (K \oplus K) \\ &= (P_1 \oplus P_2) \oplus O \\ &= P_1 \oplus P_2\end{aligned}$$

- 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 $1/2$.
    - 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 $\mathbf{DRBG}(S)$ di $n=|M|$ bit, dove M è il messaggio da lei scelto
- Il messaggio effettivamente rilasciato ad Eva è la **coppia** (C,R), dove $C = M \oplus \mathrm{DRBG}(S)$

- 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 $C = M \oplus \mathrm{DRBG}(R)$ 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 C_1 corrispondente ad un plaintext P_1, risulta arduo per Eva creare un secondo ciphertext C_2 corrispondente ad un plaintext P_2 che abbia una **qualche relazione** con P_1.

- 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 2^B 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 2^{32} posizioni, ciascuna di 32 bit (4 byte), per un totale di $4 \cdot 2^{32} = 16$ GB, cioè sarebbe **del tutto gestibile**.
- Con $B = 64$ le dimensioni salirebbero a $2^{67} \approx 1.48 \cdot 10^{20}$ 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** T_k, di 2^B posizioni, che memorizza una specifica permutazione delle sequenze di B bit;
    2. la cifratura corrisponde ad un accesso a T_k, 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.

P base 10	P base 2	C
0_{10}	000	011
1_{10}	001	010
2_{10}	010	111
3_{10}	011	110
4_{10}	100	001
5_{10}	101	000
6_{10}	110	101
7_{10}	111	100

P base 10	P base 2	C
0_{10}	000	100
1_{10}	001	001
2_{10}	010	000
3_{10}	011	111
4_{10}	100	110
5_{10}	101	011
6_{10}	110	010
7_{10}	111	101

- 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 $2^B!$ (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** $2 \leq B \leq 7$ (cioè per blocchi di lunghezza così corta da essere del tutto improponibili in un cifrario reale).

In [1]:
```
from math import factorial,log,log10
print("Valore di B\tNumero di permutazioni\tLunghezza minima della chiave")
for B in range(2,8):
    v = factorial(2**B)
    if B>3:
        e = int(log10(v))
        m = int(10*(v*10**(-e)))/10
        s = f"{m}e{e}"
    else:
        s = str(v)
    print(f"{B}\t\t{s}\t\t\t{int(B*2**B*log(2))}")
```

```
Valore di B     Numero di permutazioni  Lunghezza minima della chiave
2               24                      5
3               40320                   16
4               2.0e13                  44
5               2.6e35                  110
6               1.2e89                  266
7               3.8e215                 621
```

- 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 $B \leq 5$, dopodiché la frazione di permutazioni utilizzabili diviene rapidamente **trascurabile**.

```
In [2]: k = 128
        for B in range(5,9):
            print(f"Frazione di permutazioni utilizzabili con chiavi di {k} bit e "+\
                  f"blocchi di {B} bit: {min(1,2**k/factorial(2**B))}")
```

```
Frazione di permutazioni utilizzabili con chiavi di 128 bit e blocchi di 5 bit: 1
Frazione di permutazioni utilizzabili con chiavi di 128 bit e blocchi di 6 bit: 2.681776295311788e-51
Frazione di permutazioni utilizzabili con chiavi di 128 bit e blocchi di 7 bit: 8.824281449889046e-178
Frazione di permutazioni utilizzabili con chiavi di 128 bit e blocchi di 8 bit: 0.0
```

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**.

P base 10	P base 2	C
0_{10}	000	011
1_{10}	001	010
2_{10}	010	111
3_{10}	011	110
4_{10}	100	001
5_{10}	101	000
6_{10}	110	101
7_{10}	111	100

P base 10	P base 2	C
0_{10}	000	100
1_{10}	001	001
2_{10}	010	000
3_{10}	011	111
4_{10}	100	110
5_{10}	101	011
6_{10}	110	010
7_{10}	111	101

Combinazioni lineari affini in $\mathbf{Z}_2$

- In generale non vanno bene permutazioni che possono essere espresse mediante **trasformazioni lineari affini invertibili** sul campo $\mathbf{Z}_2$

$$xP + b = y(x)$$

in cui P è una matrice non-singolare di ordine B su $\mathbf{Z}_2$, mentre x (il plaintext), b e $y(x)$ (il ciphertext) sono vettori riga di B elementi, sempre con elementi in $\mathbf{Z}_2$.

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

$$P = \begin{matrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{matrix}$$

e

$$b = \begin{matrix} 0 & 1 & 1 \end{matrix}$$

- 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).

$$y(O) = O \cdot P + b = O + b = b$$

  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 $e_i = (0,\ldots,0, \quad 1, \quad 0,\ldots,0)$.

  indice i
- Scrivendo tutte le B query in forma matriciale abbiamo:

$$I \cdot P = y(I)$$

  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 $\mathbf{Z}_2$ il numero di matrici invertibili è dato dalla seguente formula, di derivazione relativamente semplice

$$\prod_{i=0}^{n-1}\left(2^n-2^i\right)=\left(2^n-1\right)\left(2^n-2\right)\left(2^n-2^2\right)\ldots\left(2^n-2^{n-1}\right)$$

- Il seguente codice Python ne implementa il calcolo.

```
In [3]: from math import factorial

In [4]: def PermInvGF2(n):
            v = 1
            pow = 1<<n
            for j in range(n):
                v *= (pow-(1<<j))
            return v

In [5]: B = 3
        PermInvGF2(B)*(2**B)

Out[5]: 1344

In [6]: factorial(2**B)

Out[6]: 40320
```

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

0_{10}	000	010
1_{10}	001	101
2_{10}	010	011
3_{10}	011	001
4_{10}	100	000
5_{10}	101	110
6_{10}	110	100
7_{10}	111	111

Proprietà delle permutazioni usate in pratica

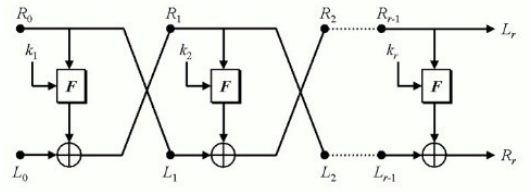
- Ovviamente l'algoritmo di cifratura **non segue i passaggi indicati**, che sono solo (come detto) una maniera per descriverne il comportamento osservabile.
- Tuttavia, questa descrizione ha il pregio di mettere in chiaro come, nella realizzazione concreta di un cifrario a blocchi, debba necessariamente 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** k_i; 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** L_i **ed** R_i di dimensione $B/2$. Inizialmente, la **concatenazione** L_0R_0 **dei semiblocchi** è proprio il blocco completo da cifrare.
- Uno dei due blocchi, per la precisione il **blocco di destra** R_i, passa inalterato allo stadio successivo come prossimo blocco di sinistra, L_{i+1}.
- Invece, il successivo blocco di destra, R_{i+1}, è ottenuto **applicando ad** R_i **una trasformazione** F dipendente dalla round key e il cui valore è posto in xor bitwise con L_i.

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):

$$L_1 = R_0$$

$$R_1 = L_0 \oplus F(R_0, k_1)$$

$$L_2 = R_1$$

$$R_2 = L_1 \oplus F(R_1, k_2)$$

$$\dots$$

$$L_r = R_{r-1}$$

$$R_r = L_{r-1} \oplus F(R_{r-1}, k_r)$$

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

$$R_{r-1} = L_r$$

$$L_{r-1} = R_r \oplus F(L_r, k_r)$$

$$R_{r-2} = L_{r-1}$$

$$L_{r-2} = R_{r-1} \oplus F(L_{r-1}, k_{r-1})$$

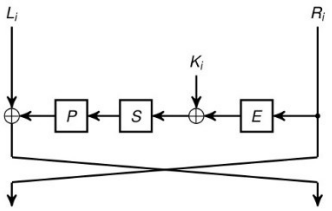
$$\dots$$

$$L_0 = R_1$$

$$R_0 = L_1 \oplus F(R_1, k_1)$$

- 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 $R_r L_r$ 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 R_i 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 L_i; 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 ($4 \cdot 8 = 32$ bit) è **permutato e posto in xor col semiblocco** L_i che, come sappiamo, costituirà il blocco destro del round successivo.

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

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

$$P = P_1||P_2||\ldots||P_n$$

  la suddivisione in blocchi del plaintext già dopo il padding.

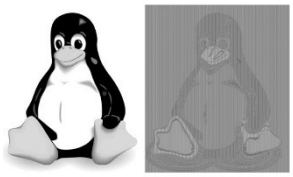
- Considereremo, esclusivamente a modo di esempio, due tecniche differenti per il mode of operation.

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**.

$$C_i = E(P_i, K), \qquad i = 1, 2, \ldots, n$$

  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:

$$C_1 = E(P_1 \oplus IV,\ K), \qquad C_i = E(P_i \oplus C_{i-1},\ K), \qquad i = 2,\ 3,\ \ldots,\ n$$

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

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

$$P_1 = D(C_1,\ K) \oplus IV, \qquad P_i = D(C_i,\ K) \oplus C_{i-1}$$

- 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 $232_{10} = E8_{16} = 11101000_2$):

$$00\ldots011101000\ \underbrace{P}_{120}\ \underbrace{00\ldots0}_{24}$$

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 $\ell = 16k$, 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 10_{16})

$$\underbrace{10_{16}10_{16}\ldots10_{16}}_{16}$$

  - Se $\ell = 16k+1$ (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 $0F_{16}$):

$$\underbrace{0F_{16}0f_{16}\ldots0F_{16}}_{15}$$

  - Se $\ell = 16k+2$ (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 $0E_{16}$):

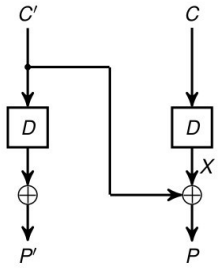
$$\underbrace{0E_{16}0E_{16}\ldots0E_{16}}_{14}$$

  - ....
  - Se $\ell = 16k+15$ (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 01_{16}).
- 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, $n \in \{1,\ 16\}$, 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 $j \in \{1, 16\}$, 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

$$\sum_{i=1}^{16} \frac{1}{256^i}$$

```
In [1]: sum([1/(256**i) for i in range(16,0,-1)])
Out[1]: 0.00392156862745098
```

- Da cui ovviamente risulta che il padding sarà malformato con probabilità:

```
In [2]: 1-sum([1/(256**i) for i in range(16,0,-1)])
Out[2]: 0.996078431372549
```

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".

```
In [4]: from random import randint
```

```
In [5]: n = 1000000 # numero di blocchi casuali generati
        blen = 16   # lunghezza, in byte, di B
        wfp = 0     # variabile che conta i well-formed padding
        for _ in range(n):
            B = [randint(0,255) for _ in range(blen)]   # Blocco casuale
            lastbyte = B[-1]
            if lastbyte == 0 or lastbyte>blen:
                continue
            if all([B[i] == lastbyte for i in range(blen-2,blen-lastbyte-1,-1)]):
                if lastbyte != 1:
                    print(B)
                wfp += 1
        print(wfp/n)
```

```
[13, 246, 20, 99, 200, 139, 115, 90, 156, 188, 239, 170, 159, 122, 2, 2]
[179, 194, 122, 46, 91, 95, 181, 194, 120, 178, 167, 147, 209, 215, 2, 2]
[70, 184, 186, 250, 44, 0, 169, 133, 88, 23, 173, 36, 238, 160, 2, 2]
[145, 95, 124, 215, 10, 24, 210, 76, 159, 79, 20, 154, 138, 110, 2, 2]
[142, 186, 150, 143, 141, 76, 58, 237, 199, 184, 1, 186, 29, 230, 2, 2]
[198, 2, 144, 182, 107, 127, 29, 67, 112, 200, 103, 107, 187, 123, 2, 2]
0.003885
```

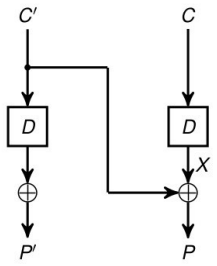
- Come si vede dall'esempio (con le stampe non commentate) il **"grosso" della probabilità** che il padding "casuale" sia ben formato è dato dall'accadimento che l'**ultimo byte sia 01_{16}**.
- D'altra parte è facile calcolare esplicitamente la probabilità condizionata che l'ultimo byte sia 01_{16} dato che il padding è ben formato (wfp).

$$\text{pr[lastbyte} = 01_{16} \mid \text{wfp]} = \frac{pr[lastbyte = 01_{16} \wedge wfp]}{pr[wfp]} = \frac{pr[lastbyte = 01_{16}]}{pr[wfp]} = \frac{\frac{1}{256}}{\sum_{i=1}^{16} \frac{1}{256^i}}$$

```
In [3]: (1/256)/sum([1/(256**i) for i in range(16,0,-1)])
```

```
Out[3]: 0.99609375
```

L'attacco



- Eva sceglie a caso tutti i byte di C' **escluso l'ultimo**.
- L'ultimo byte viene posto al valore 00_{16} e viene **incrementato di una unità** ad ogni query.
- Se, per un dato valore di xy_{16}, non viene generato errore, per quanto detto sopra Eva può essere ragionevolmente sicura che **l'ultimo byte di P sia 01_{16}**.
- Risulta cioè $xy_{16} \oplus x[15] = 01_{16}$ e dunque, date le proprietà dell'or esclusivo, Eva può recuperare il valore dell'ultimo byte di X nel modo seguente: $x[15] = xy_{16} \oplus 01_{16}$
- 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 $C'[15] \oplus x[15] = 02_{16}$, e dunque (sempre per le proprietà dello xor)

$$C'[15] = x[15] \oplus 02_{16}$$

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 rs_{16} assegnato a $C'[14]$, non viene segnalato errore è altamente probabile che il penultimo byte del testo decifrato sia 02_{16}, da cui Eva ricava

$$X[14] = rs_{16} \oplus 02_{16}$$

- 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 $03_{16}03_{16}$, 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 00_{16}.
- 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.