Crittografia (*cryptography*)

https://github.com/Cyofanni/high-school-cs-class/tree/main/python/cryptography

Liceo G.B. Brocchi
Classi seconde Scientifico – opzione scienze applicate
Bassano del Grappa, Dicembre 2022

Perché nascondere le informazioni

- Alcuni dati e lo scambio di alcune informazioni devono rimanere private, personali
  - e.g. in ambito militare, le informazioni sui prossimi spostamenti di un esercito non devono essere scoperte dal nemico
  - e.g. in ambito industriale, le comunicazioni interne ad un'azienda relative ad un nuovo prodotto non devono essere conosciute da aziende concorrenti
  - e.g. in ambito politico, in un paese che non rispetta i diritti umani, un dissidente deve avere un modo per comunicare segretamente senza che il governo lo scopra

Il Cifrario di Cesare (*Caesar Cipher*)

- È un **cifrario a sostituzione semplice monoalfabetico**. Opera sui caratteri (i cifrari moderni, realizzati sui computer, operano sui **bit**)
- Ogni carattere del *testo in chiaro* viene sostituito dal carattere in posizione **key** a destra, **modulo 26** (l'alfabeto inglese ha 26 lettere)
- Vediamo un esempio che rende più chiara la definizione data sopra

Il Cifrario di Cesare

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

rotazione di 1 in avanti

b c d e f g h i j k l m n o p q r s t u v w x y z a

rotazione di 2 in avanti

c d e f g h i j k l m n o p q r s t u v w x y z a b

rotazione di 3 in avanti

d e f g h i j k l m n o p q r s t u v w x y z a b c

rotazione di 4 in avanti

e f g h i j k l m n o p q r s t u v w x y z a b c d

Il Cifrario di Cesare

rotazione di 25 in avanti

z a b c d e f g h i j k l m n o p q r s t u v w x y

rotazione di 26 in avanti

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

nell'aritmetica modulare, detta anche *aritmetica dell'orologio*:
sommare 26 equivale a sommare 0
sommare 27 equivale a sommare 1
sommare 28 equivale a sommare 2
sommare 29 equivale a sommare 3
sommare 30 equivale a sommare 4
...
sommare 52 equivale a sommare 0
sommare 53 equivale a sommare 1
etc...

Il Cifrario di Cesare

- **Compito**:

  scrivere un programma C++ che ruota l'alfabeto inglese un numero di volte pari ad un intero positivo ricevuto da *standard input*

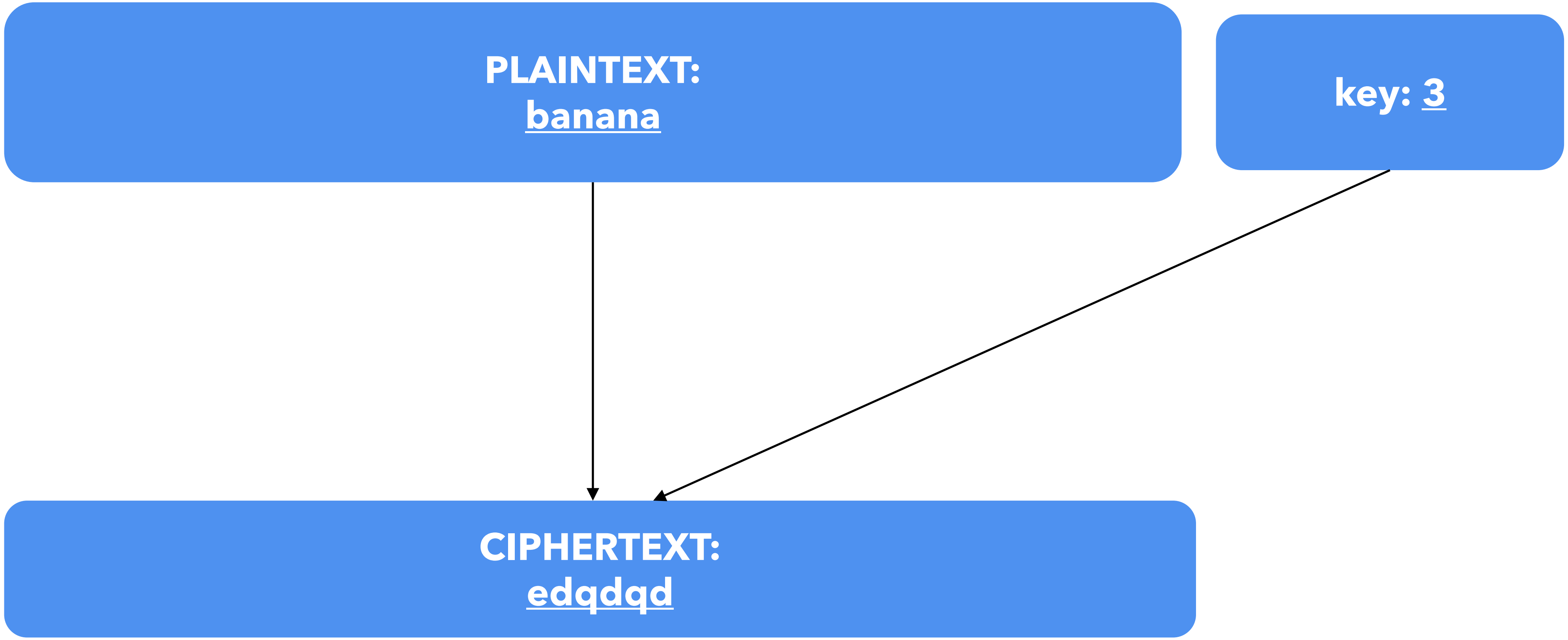
Il Cifrario di Cesare

PLAINTEXT:
banana

key: 1

CIPHERTEXT:
cbobob

Il Cifrario di Cesare



Il Cifrario di Cesare

- Per decrittare basta effettuare la rotazione nel verso opposto

- Mittente e ricevente devono aver condiviso la chiave in anticipo prima di poter comunicare

- Questo tipo di crittografia viene detto **crittografia simmetrica e a chiave segreta**

- Con il Cifrario di Cesare, a lettere uguali nel testo in chiaro corrispondono lettere uguali nel testo cifrato. Questo rende facile trovare la chiave analizzando la frequenza delle lettere, perché sappiamo che in un testo scritto in una lingua naturale, le lettere compaiono con frequenze diverse. Ad esempio, in inglese, la vocale ‘e’ compare molto più frequentemente della ‘q’

Il codice **ASCII** (*American Standard Code for Information Interchange*), *versione non estesa*

```python
print("ASCII CODE (decimal)\t", "CHARACTER")

i = 32
while i < 127:
    print(i, '\t\t\t', chr(i))
    i += 1

print()
```

Ad ogni carattere corrisponde un codice numerico (qui lo stampiamo in decimale).
A noi interessano soltanto i caratteri stampabili della lingua inglese, che vanno dal codice ASCII 32 al 126

Usage: `python3 ascii_table.py`

https://en.wikipedia.org/wiki/ASCII

Il Cifrario di Cesare in Python

```
#!/usr/bin/python3

#Usage example: python3 caesar_cipher.py "hello world" 3

import sys

alphabet_size = 26
ascii_base = 97

def shift_character(plain_c, key):
   return chr(((ord(plain_c) + key - ascii_base) % alphabet_size) +
ascii_base)

plain_text = sys.argv[1]
key = sys.argv[2]

for pc in plain_text:
   print(shift_character(pc, int(key)), end = '')

print()
```

le operazioni sono mod 26, che è la dimensione dell'alfabeto (lettere minuscole dell'alfabeto inglese

Usage: `python3 caesar_cipher.py "hello world" 3`

Il Cifrario di Vigenère (XVI secolo)

- Questo cifrario prevede che i caratteri di un testo **non vengano traslati tutti dello stesso valore n**
- La 'lunghezza' della traslazione di ogni carattere viene stabilita da una chiave alfabetica sovrapposta più volte al testo in chiaro, in questo modo. Il plaintext è *wewillattackatfiveoclockpm* e la chiave è *qwerty*

plaintext: wewillattackatfiveoclockpm
repeated key: qwertyqwertyqwertyqwertyqw

Funzionamento: se una lettera p del testo in chiaro è sovrapposta alla lettera k della chiave, allora la lettera p va traslata della posizione di k nell'alfabeto

Ad esempio, da sinistra: la prima 'w' va traslata di 17 posizioni, perché17 è la posizione di 'q' nell'alfabeto,
la prima 'e' va traslata di 23 posizioni perché la 'w' è in posizione 23 nell'alfabeto etc…

Il Cifrario di Vigenère (XVI secolo)

PLAINTEXT	CHIAVE	TRASLAZIONI	CIPHERTEXT
C	B	2	E
O	E	5	T
M	E	5	R
P	F	6	V
U	B	2	W
T	E	5	Y
E	E	5	J
R	F	6	X

La chiave è **BEEF**;
‘B’ corrisponde alla traslazione 2, perché è la seconda lettera dell’alfabeto inglese (partendo da 1);
‘E’ corrisponde alla traslazione 5 perché la quinta lettera dell’alfabeto inglese etc…
Infatti: ‘C + ‘2’ = ‘E’; ‘O’ + 5 = ‘T’; ‘M’ + 5 = ‘R’ etc…

Il Cifrario di Vigenère (XVI secolo)

```
#!usr/bin/python3
#Usage example: python3 vigenere_cipher.py helloworld banana

import sys

alphabet_size = 26
ascii_base = 97

def shift_character(plain_c, shift):
   return chr(((ord(plain_c) + shift - ascii_base) % alphabet_size) +
ascii_base)

plain_text = sys.argv[1]
key = sys.argv[2]
key_index = 0

for pc in plain_text:
   if key_index == len(key):
      key_index = 0
   print(shift_character(pc, ord(key[key_index]) - ascii_base), end = '')
   key_index = key_index + 1

print()
```

le operazioni sono mod 26, che è la dimensione dell'alfabeto inglese minuscolo.
Per semplicità, lavoriamo solo con le lettere minuscole.
Al carattere 'a' facciamo corrispondere uno spostamento nullo

Il Cifrario di Vigenère (XVI secolo)

```
#!usr/bin/python3
#Usage example: python3 invert_vigenere_key.py banana

import sys

alphabet_size = 26
ascii_base = 97

def invert_character(c):
    if (ord(c) == ascii_base):
        return c
    return chr(alphabet_size - (ord(c) - ascii_base) + ascii_base)

key = sys.argv[1]

for c in key:
    print(invert_character(c), end = '')

print()
```

**La chiave per decrittare il testo si ottiene da quella per criptare, «invertendola» carattere per carattere.
E.g. 'b' diventa 'z', 'c' diventa 'y' etc...**

La crittografia simmetrica

- I cifrari di Cesare e Vigenère si basano su una **chiave segreta**, perché la segretezza dei messaggi criptati si basa sulla segretezza della chiave
- Questi metodi crittografici vengono detti **simmetrici** perché se **k** è la chiave utilizzata per criptare i messaggi, **-k** è quella usata per decriptarli
- Siano **Alice** e **Bob** due persone che vogliono comunicare in modo sicuro: con uno schema a chiave simmetrica, sarebbero costretti a mettersi d'accordo in anticipo sulla chiave da utilizzare per comunicare
- Ovviamente, al giorno d'oggi non si usano più i cifrari di Cesare e Vigènere
- Se Alice e Bob comunicano tramite Internet, secondo voi si scambiano la chiave segreta in chiaro tramite Internet stessa?
- **È chiaramente insicuro comunicare utilizzando un algoritmo simmetrico la cui chiave è stata condivisa su un canale insicuro**

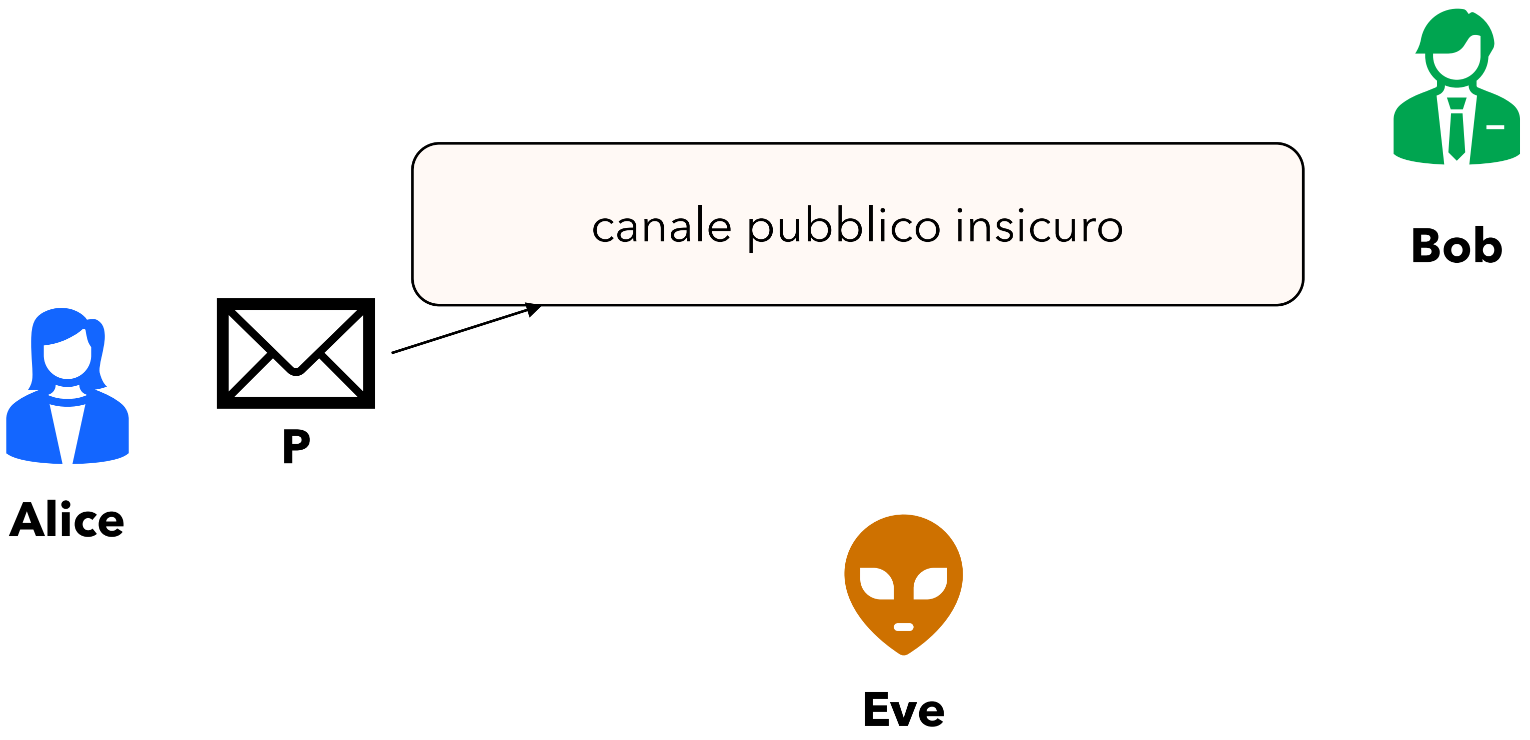
La crittografia asimmetrica

- La **crittografia asimmetrica/a chiave pubblica** risolve i problemi descritti sopra, eliminando la necessità di una chiave segreta condivisa e utilizzata sia per criptare, sia per decriptare i messaggi

- In uno schema a chiave pubblica, ad un utente vengono attribuite (da una *Certification Authority*) una **chiave pubblica**, consultabile da chiunque in una sorta di elenco telefonico, e una **chiave privata**, che va mantenuta segreta

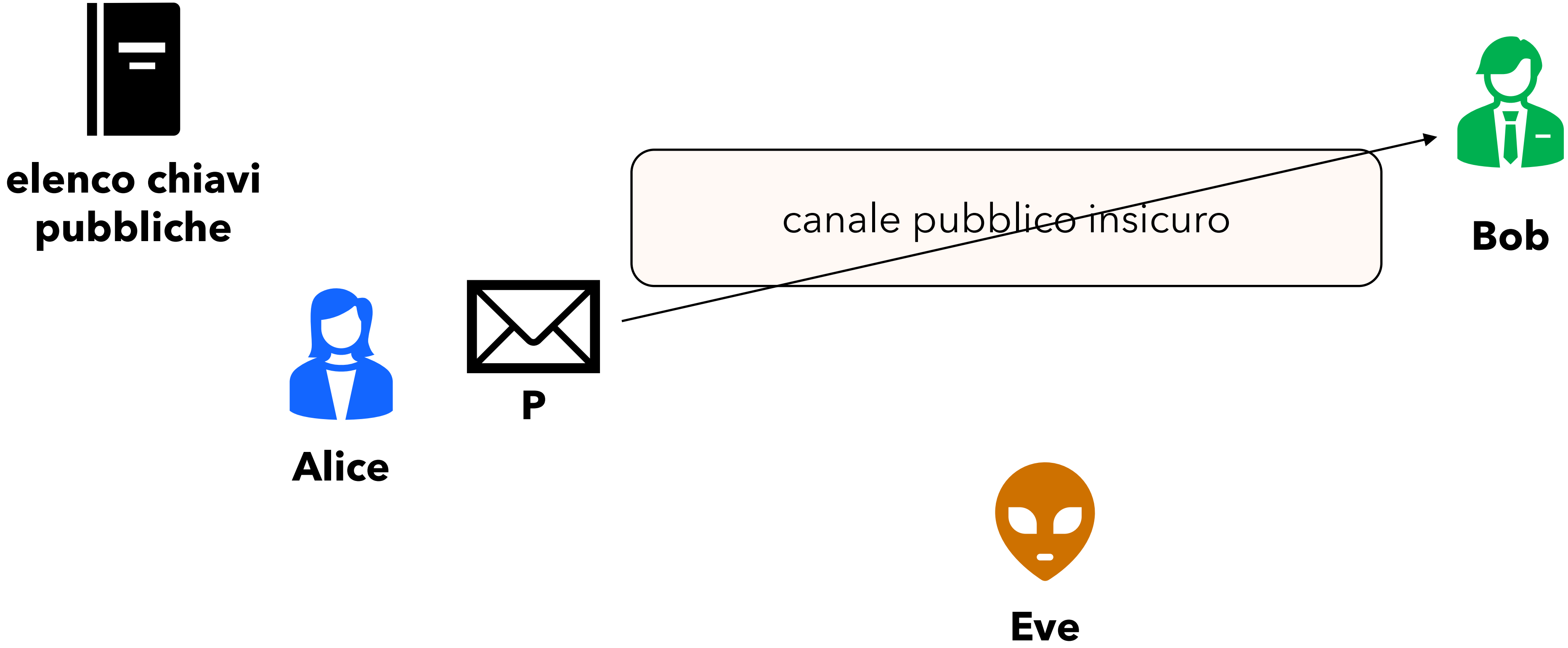
La crittografia asimmetrica

Se *Alice* vuole inviare un messaggio P a *Bob*, senza che *Eve* sia in grado di leggerlo, su un canale pubblico come Internet



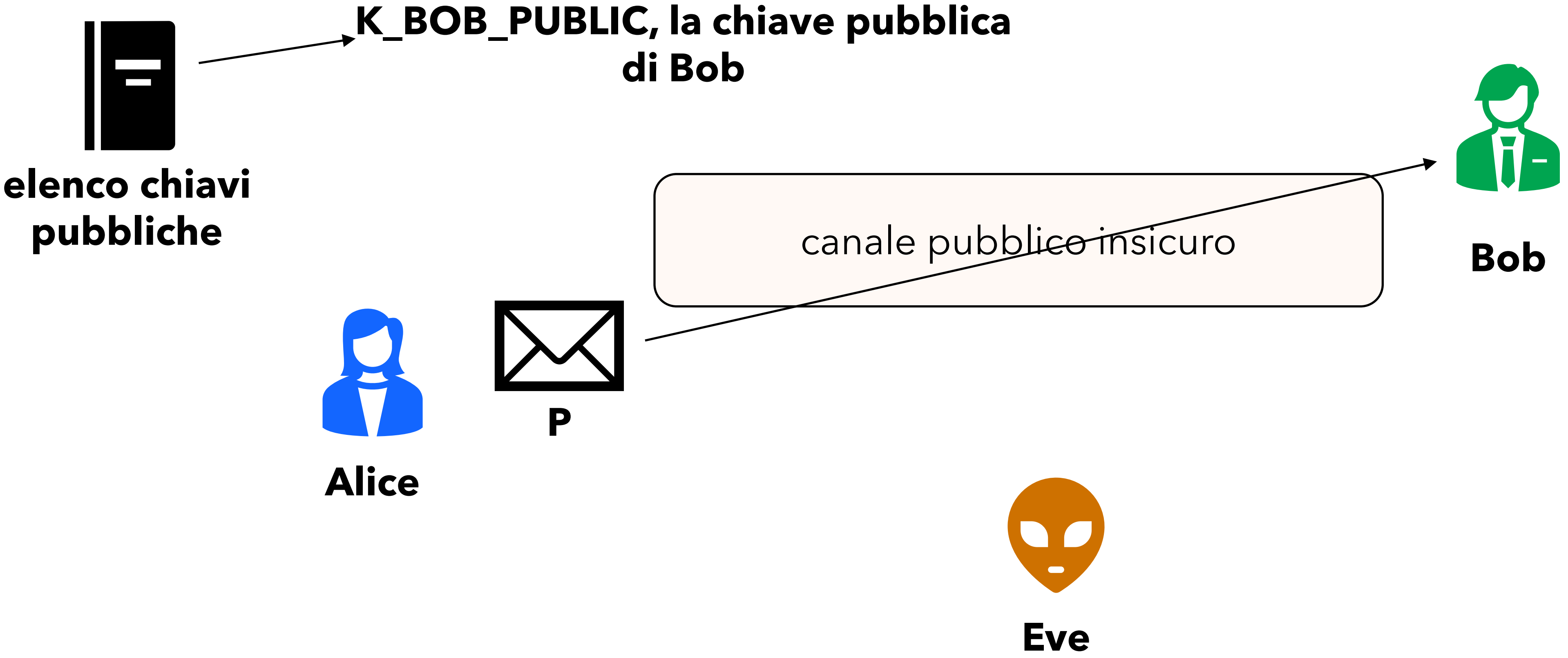
La crittografia asimmetrica

deve prima ottenere la chiave pubblica di Bob, che è facilmente reperibile da tutti in una specie di *elenco telefonico* pubblicato da qualche parte



La crittografia asimmetrica

Alice cripta P utilizzando un algoritmo **E**, noto e parametrizzato sulla chiave pubblica di Bob, *K_BOB_PUBLIC*



La crittografia asimmetrica

Alice invia *C = E_K_BOB_PUBLIC(P)* a Bob.

Scriviamo E_K_BOB_PUBLIC(P) come *f(P)*, ossia l'applicazione di una funzione *f* sull'input *P*

La crittografia asimmetrica

Bob, oltre alla chiave pubblica, possiede anche una chiave privata (che conosce solo lui) associata alla chiave pubblica: K_BOB_PRIVATE.

È importante ricordare che le chiavi vengono generate ed utilizzate a coppie: *(chiave pubblica, chiave privata)*

La crittografia asimmetrica

Bob decripta C utilizzando un algoritmo **D**, noto e parametrizzato sulla chiave privata di Bob, *K_BOB_PRIVATE*: Bob calcola *D_K_BOB_PRIVATE(C)*, che possiamo scrivere come $f^{-1}(C)$, dove f^{-1} è l'inversa di *f*

La crittografia asimmetrica

- Ora Bob può leggere il messaggio originale inviato da Alice, in quanto: $f^{-1}(C) = f^{-1}(f(P)) = P$

La crittografia asimmetrica

- Tutto molto interessante in teoria, ma è realizzabile?
- Per realizzarlo. è necessario che da f sia praticamente impossibile ricavare f^{-1}
  - f è costituito dall'algoritmo noto e dalla chiave pubblica
  - se f^{-1} fosse facilmente ricavabile anche da chi non ha la chiave privata, allora lo schema non sarebbe in alcun modo sicuro

- Serve quindi un algoritmo crittografico per il quale non sia possibile ricavare f^{-1} a partire da f

- Schemi del genere sono stati realizzati e si basano sulla **Teoria dei Numeri**

La crittografia asimmetrica

- Uno degli schemi crittografici a chiave pubblica più importanti è **RSA**, messo a punto e descritto pubblicamente nel 1977 da *Ron Rivest, Adi Shamir* e *Leonard Adleman*

- Nello schema RSA, la funzione facile da calcolare e praticamente impossibile da invertire è la moltiplicazione di due numeri primi **p** e **q** molto grandi (di centinaia di cifre, spesso 1024, 2048 o 4096 bit):

n = p*q

- Se trovare numeri primi molto grandi e moltiplicarli è computazionalmente facile, l'operazione inversa, ossia ricavare *p* e q partendo da n, ossia *fattorizzarlo*, è computazionalmente molto difficile

La crittografia asimmetrica

- I tre inventori di RSA hanno quindi trovato una funzione *f* facile da calcolare in una direzione, ma la cui inversa f^{-1} *non è ricavabile facilmente*
- Siamo quindi in presenza di uno schema perfetto per realizzare un *public key cryptosystem*, descritto sopra solo astrattamente
- Quando diciamo che una funzione è *computazionalmente difficile da calcolare*, intendiamo che anche una rete di supercomputer che lavorano in parallelo non riuscirebbe a calcolarla in tempi ragionevoli
- **Non è quindi una questione di potenza di calcolo, ma di *assenza di un algoritmo efficiente* che calcoli la funzione. In pratica, finora, nessuno ha mai elaborato un algoritmo efficiente per la fattorizzazione di numeri molto grandi**

La crittografia asimmetrica

- In informatica si incontrano spesso delle coppie di problemi (P1, P2), con P1 e P2 *apparentemente simili*, per cui però vale che:
    - P1 può essere risolto con un algoritmo efficiente
    - per P2 nessuno ha mai trovato un algoritmo efficiente. Forse questo algoritmo non esiste nemmeno, ma nessuno lo ha mai dimostrato.

- Nel caso di RSA:
    - **P1**: verificare se un numero naturale, anche molto grande, è primo, moltiplicare 2 numeri primi molto grandi (problema risolto)
    - **P2**: dato un numero naturale, fattorizzarlo (problema *intrattabile*)

Test di primalità

- Il test di primalità detto **trial division** per un numero naturale n, consiste nel cercare i divisori tra compresi tra 2 e $\sqrt{n}$
- Chiaramente, se si trova un divisore, allora n non è primo

```
#!usr/bin/python3
import sys
import math
def is_prime(n):
   is_prime = True
   if n % 2 == 0 and n != 2 or n == 1:
      is_prime = False

   elif n % 2 != 0 and n > 2:
      trial_div = 3
      found_div = False
      while trial_div <= int(math.sqrt(n)) and found_div == False:
         if n % trial_div == 0:
            is_prime = False
            found_div = True
         trial_div += 2
   return is_prime

for number in range(300):
   if is_prime(number):
      print(number, 'is prime')
   else:
      print(number, 'is not prime')
```

Ripasso: un numero n è primo se e solo se non possiede altri divisori oltre a 1 e n

Test di primalità

- Sembra tutto molto semplice… il problema è che questo algoritmo è pessimo su numeri molto grandi. Formalmente, si dice che il suo tempo di esecuzione è esponenziale sulla dimensione del numero da testare. Ovviamente se il numero ha divisori piccoli l'algoritmo risponde immediatamente. *Ma l'analisi di un algoritmo si fa sempre sul caso peggiore, non su quello migliore*

- Provate l'algoritmo trial division implementato sopra in Python con questi numeri:

```
345986722333095487230548293657986723611
```

```
3459867223330954872305482936579867236115459867223991954845
3054829365734592387867236231
```

Test di primalità

```
start_time = time.time()
number = 345986722333095487230548293657986723611
if is_prime(number):
   print(number, 'is prime')
else:
   print(number, 'is not prime')
print("------ trial division took %s seconds to produce a result ------" %
      (time.time() - start_time))
```

Sulla mia macchina ci mette circa 1 minuto. Potreste obiettare: *'è scritto in Python, che è un linguaggio ad alto livello e interpretato, se lo scrivi in C o in Assembly e utilizzi un supercomputer della NASA o di Google ottieni risultati migliori'.*
È vero che si otterrebbero risultati migliori, ma sarebbero miglioramenti del tutto irrilevanti. Il problema di fondo è che questo algoritmo è inefficiente o lo sarà sempre, indipendentemente dall'implementazione software e dall'hardware.
NB: in C++ potete lavorare facilmente con numeri così grandi?

Test di primalità

- Esistono algoritmi molto efficienti per il test di primalità, anche di numeri spaventosamente grandi

- Alcuni esempi: l'algoritmo basato sul Piccolo Teorema di Fermat e l'algoritmo di Miller-Rabin

- Accontentiamoci di utilizzare il test di primalità efficiente offerto da **sympy**, una libreria utilizzabile da Python. Lanciate:

```
sudo pip install sympy
```

Test di primalità

- Aprite una shell Python e provate il test di primalità con uno dei numeri che hanno messo in crisi l'algoritmo *trial division*. Vi risponderà immediatamente:

```
cyofanni@LAPTOP-I0S1RKRC:~$ python3
Python 3.10.6 (main, Nov  2 2022, 18:53:38) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import sympy
>>> sympy.isprime(34598672233309548723054829365798672361154598672239919548453054829365734592387867236231)
True
>>>
```

Utilizzo di sympy

```python
import sympy
#primality test
print(sympy.isprime(3457309489867986172311123))
#greatest common divisor
print(sympy.gcd(252, 105))
#get next prime number after parameter
print(sympy.nextprime(1334937654386587625342))
'''prime factorization
  12009876938645223429834286375872313457639485673965798237462375
'''
print(sympy.factorint(1200))
'''benchmark
   12009876938645223429834
   1200987693864522342983445
   12009876938645223429834450985749286723434587681723

1200987693864522342983445098574928672343458768172398672386452865487651625381625387253186253
787231098248713114
'''
#euler's phi
print(sympy.totient(10))
```