Algoritmi di Crittografia (2023/24)

Notebook 4

In [ ]:
```
from IPython.display import HTML
HTML('<style>{}</style>'.format(open('/home/mauro/.jupyter/custom/custom.css').read()))
```

Latex definitions

Casualità e algoritmi probabilistici

Randomness (casualità)

- Una risorsa di straordinaria importanza in molti algoritmi e applicazioni, in particolare proprio le applicazioni crittografiche, è rappresentata dalla disponibilità di bit casuali.
- In realtà questo è un modo sbrigativo ma comunemente utilizzato per definire sequenze di bit generati casualmente, indipendenti e con probabilità uniforme.
- Se $\{X_i\}_{i=1,2,..}$ è una tale sequenza, per ogni $i \geq 0$,

$$\text{prob}[X_i = 0] = \text{prob}[X_i = 1] = \frac{1}{2}$$

  e

$$\text{prob}[X_{i+1} = 0|X_i = 0] = \text{prob}[X_{i+1} = 0|X_i = 1] = \text{prob}[X_i = 0]$$
$$\text{prob}[X_{i+1} = 1|X_i = 0] = \text{prob}[X_{i+1} = 1|X_i = 1] = \text{prob}[X_i = 1]$$

- Una sequenza di bit casuali è il punto terminale di un procedimento che ha inizio da una qualche sorgente di casualità, cioè un dispositivo logico o fisico in grado di produrre eventi casuali (misurabili) in un opportuno spazio.

Generatori hardware

- Non è facile avere a disposizione sorgenti di "vera" casualità.
- Generatori veramente casuali sono tipicamente ottenibili mediante dispositivi esterni al computer, che sfruttano fenomeni fisici stocastici.
- In generale, tali dispositivi sono realizzati intorno a tre componenti principali:
  1. un *trasduttore*, che trasforma il rumore stocastico del fenomeno fisico utilizzato (ad esempio, rumore termico in un resistore o variazioni nell'ampiezza o nella fase di un oscillatore) in segnali elettrici
  2. un *amplificatore*, il cui scopo è di aumentare l'ampiezza dei segnali in modo che siano misurabili
  3. un *convertitore analogico-digitale* che produce in output sequenze di bit
- La generazione attraverso questa via è però costosa e la quantità di bit casuali non è sempre adeguata alle esigenze delle applicazioni

Generatori pseudocasuali per applicazioni crittografiche (CSPRNG)

- La disponibilità di sequenze di bit casuali in crittografia è richiesta in molteplici casi.
- Per gli scopi di questo corso è fondamentale comprendere l'importanza di generare numeri casuali che possano poi servire per la determinazione di chiavi in protocolli asimmetrici.
- I bit casuali necessari in applicazioni crittografiche vengono tipicamente ottenuti, nelle quantità richieste, mediante procedimenti deterministici a partire da sequenze casuali (in generale più corte).
- Le sequenze messe in questo modo a disposizione delle applicazioni sono dette *pseudocasuali*.

CSPRNG in ambiente Linux

- Le sequenze casuali sono ottenute, internamente al computer, da fenomeni difficilmente predicibili, anche se non sempre casuali.
- Fra questo possiamo ricordare i movimenti del mouse, la frequenza di digitazione sulla tastiera, il tempo che intercorre fra alcuni tipi di interrupt.
- Quanto più le sorgenti sono di natura distinta e le sequenze opportunamente combinate, tanto più i bit soddisfano le esigenze richieste dalle applicazioni crittografiche.
- I bit casuali raccolti (collected) dal sistema sono inseriti nel c.d. entropy pool, composto da max 4096 bit.
- Dal pool vengono prelevali bit casuali su richiesta delle applicazioni e in esso vengono inseriti bit casuali dal kernel.
- Il kernel tiene traccia del numero di bit casuali (o bit di entropia) presenti nel kernel.
- Tale quantità è conoscibile attraverso il comando indicato di seguito.

In [ ]:
```
!cat /proc/sys/kernel/random/entropy_avail
```

- Linux mette a disposizione interfacce per due generatori, che sono equivalenti quando l'entropia è elevata ma che con poca entropia si comportano differentemente.
- I generatori sono accessibili come device logici:

  1. /dev/random, che blocca il processo richiedente se l'entropia richiesta non è suffciente;
  2. /dev/urandom, che invece è non blocking.

In [ ]:
```
%%bash
dd if=/dev/random bs=8 count=1000 2>/dev/null  | base64
```

CSPRNG in Python

- In sistemi Linux, /dev/urandom è accessibile dal comando os.urandom di Python. La chiamata di os.urandom restituisce il numero di byte casuali specificati come parametro.

In [ ]:
```
from os import urandom
```

In [ ]:
```
urandom(4)
```

In [ ]:
```
R = urandom(4)
for b in R:
    print(bytes([b]))      # bytes(iterabile-di-interi) -> lista con altrettanti byte
```

- Possiamo anche visualizzare più convenientemente i byte come sequenze esadecimali.

In [ ]:
```
from binascii import b2a_hex
```

In [ ]:
```
b2a_hex(R)
```

- Mettendo insieme i pezzi, possiamo creare un generatore casuale in $[0,1)$, che è poi il generatore casuale fondamentale

In [ ]:
```
def myrand():
    from os import urandom
    D = 2**(-64)
    return int.from_bytes(urandom(8),'big')*D
```

In [ ]:
```
myrand()
```

Algoritmi probabilistici

- Un algoritmo probabilistico è tale perché utilizza una qualche sorgente di casualità, che possiamo immaginare "tipo" /dev/urandom, cioè in grado di fornire una sequenza (teoricamente) illimitata di bit casuali, indipendenti e distribuiti in maniera uniforme.
- Algoritmi di questo tipo giocano un ruolo importante in crittografia e, in particolare (cioè in relazione a ciò che qui ci interessa), in crittografia asimmetrica.
- In questo caso, però, è conveniente immaginare che la sorgente produca direttamente numeri casuali, visto che le basi matematiche della crittografia asimmetrica risiedono proprio in teoria dai numeri.
- Per valutare la complessità degli algoritmi probabilistici che prenderemo in considerazione, si può ancora utilizzare il ben noto modello *bit cost*, in cui ogni operazione aritmetico/logica ha un peso che dipende dal numero di bit degli operandi.
- Nella valutazione della bontà di un algoritmo è in generale necessario tenere conto anche della quantità di bit casuali richiesti, proprio perché i bit casuali non sono una risorsa illimitata.
- Se la casualità viene fornita in termini di numeri, piuttosto che di bit casuali, dobbiamo comunque valutare la lunghezza in bit dei numeri stessi.

Algoritmi decisionali

- È detto *decisionale* un algoritmo il cui output è binario: 0/1, True/False, Sì/No.
- Gli algoritmo probabilistici che vedremo saranno solo di tipo decisionale.
- L'elemento di casualità, nel comportamento visibile dell'algoritmo, può risiedere nel risultato, che può essere errato con una certa probabilità, o nel tempo di calcolo.
- A seconda dei due casi, gli algoritmi vengono definiti (di tipo) *Monte Carlo* o (di tipo) *Las Vegas*.
- Un algoritmo decisionale può essere visto come un classificatore binario (come avviene, ad esempio, nel contesto del machine learning).
- Ad esempio, dato un messaggio di posta elettronica, decidere se è spam (ovvero se appartiene all'insieme dei messaggi spam) o non-spam.

Algoritmi Monde Carlo

- Un algoritmo probabilistico di tipo Monte Carlo per un problema (decisionale) P, ha le caratteristiche indicate di seguito.

  1. Su input la cui risposta (esatta) è True (es. la mail è spam), restituisce True con probabilità fissa (indipendente dalla lunghezza dell'input) $\epsilon > 0$.
  2. Su input $x \in P$ la cui risposta (esatta) è False, restituisce False.

- Per amore di precisione, ciò che abbiamo definito è un algoritmo denominato (con terminologia inglese, decisamente più sintetica) Probability-bounded one-sided error Monte Carlo .
  - Probability-bounded (sottinteso, "away from zero") perché si richiede che la probabilità di errore sia non solo positiva, ma che non tenda a zero al crescere della dimensione dell'input.
  - One-sided error perché l'algoritmo può sbagliare solo quando risponde False (e la risposta esatta è True).

Un algoritmo non "probability-bounded"

- Un semplice algoritmo Monte Carlo one-sided error è la seguente procedura per determinare se un numero intero n dispari è composto, cioè se può essere espresso come prodotto di due numeri diversi da 1 e da n stesso.
- Dato n, scegli a caso un numero intero p dispari nell'intervallo $I = [1, n/2]$.
- Calcola il resto r della divisione di n per p
- Se $r = 0$ restituisci True (con ciò intendendo che il numero è Composto), altrimenti restituisci False (ovvero forse Primo)

- L'algoritmo è one-sider error perché può sbagliare solo se restituisce False. Se invece restituisce True il resto è 0, dunque p è un divisore di n,che quindi è un numero composto.
- Tuttavia esso non è probability-bounded.
- Per dimostrarlo basta un argomento debole.
- Sappiamo infatti che i divisori di un numero n vanno in coppia, x e $\frac{n}{x}$, di cui uno non maggiore di $\sqrt{n}$.
- Ne consegue che i divisori di n sono al più $2\sqrt{n}$.
- Scegliendo p dispari uniformemente a caso in I abbiamo una probabilità che esso sia un divisore limitata da $\frac{2\sqrt{n}}{n/4} = 8\frac{\sqrt{n}}{n} = \frac{8}{\sqrt{n}}$, una quantità che tende a zero quando n tende a infinito.

Perché è importante limitare (anche debolmente) l'errore probabilistico

- Sia A un algoritmo Monte Carlo per stabilire se un dato numero è composto.
- Per ogni $x \in \mathcal{Z}$, poniamo $y_x = \text{True}$ se il numero x è composto e $y_x = \text{False}$ se invece x è un numero primo.
- Indichiamo poi con $A(x)$ la risposta fornita da A su input x.
- La "qualifica" di A come algoritmo Monte Carlo per la "compositeness" implica che:

$$\text{prob}[A(x) = \text{False}|y_x = \text{False}] = 1 \text{ e } \text{prob}[A(x) = \text{True}|y_x = \text{True}] \geq \epsilon$$

  per un qualche valore $\epsilon > 0$ opportuno.
- La probabilità che l'algoritmo sia corretto quando il numero è composto è dunque limitata inferiormente, indipedentemente dalla grandezza del numero in input.
- Supponiamo ora che risulti $\epsilon = 0.01$.
- Si può certamente obiettare che avere una probabilità di successo (su numeri composti) garantita dell'ordine di 10^{-2} non è poi un grande risultato.
- Tuttavia se la sorgente produce bit indipendenti possiamo aumentare la probabilità di successo in modo arbitrario, come indichiamo di seguito

Run indipendenti

- A partire da A si costruisce un algoritmo A' nel modo seguente.
- Si eseguono N run identici di A.
- Non appena uno di tali run restituisce valore True, anche A' termina restituendo il valore True.
- Se invece, in tutti gli N run, A restituisce False, allora anche A' termina restituendo il valore False.
- Ci chiediamo quindi che cosa si può dire della correttezza di A' in funzione del valore N.
- Se l'input è primo, ovviamente A produrrà sempre il valore False, e dunque (indipendentemente da come si è scelto N) anche A' restituisce False.
- Se invece il numero è composto, poiché i run sono eseguiti utilizzando sequenze di bit indipendenti, la probabilità che l'algoritmo restituisca N volte False è limitata superiormente dal prodotto delle singole probabilità: $(1 - 0.01)^N = 0.99^N$.
- Per $N = 2$ risulta $0.99^2 = 0.98$
- Se il procedimento viene ripetuto 69 volte, con input un numero composto, la probabilità che A sbagli tutte le volte è limitata dal valore $0.99^{69} < 0.4998$,
- L'algoritmo A' restituirà dunque la risposta corretta con probabilità almeno $\frac{1}{2}$ se $N \geq 69$.
- Con valori ancora maggiori di N possiamo rendere la probabilità di successo, anche nel caso in cui l'input sia un numero composto, arbitrariamente vicina a 1.

Algoritmi Las Vegas

- Un algoritmo probabilistico di tipo Las vegas restituisce sempre la risposta corretta in tempo determinato solo con una data probabilità.
- Un semplice esempio di algoritmo Las Vegas è la procedura con la quale si può generare una sorgente $\{x_i\}_{i\geq 1}$ di bit uniformi (cioè tali che $\text{prob}[x_i = 0] = \text{prob}[x_i = 1]$) e indipendenti, a partire da una sorgente $\{z_i\}_{i\geq 1}$ di bit indipendenti ma distribuiti con probabilità non uniforme: $\text{prob}[z_i = 0] = p, \text{prob}[z_i = 1] = 1 - p$, per un qualche valore $0 < p < 1$
- L'algoritmo è il seguente:
  1. Poni $i = 1$
  2. Considera i bit z_i e z_{i+1}
  3. Se $z_i = 0$ e $z_{i+1} = 1$ restituisci 0
  4. Se $z_i = 1$ e $z_{i+1} = 0$ restituisci 1
  5. Poni $i = i + 2$ e ritorna al passo 2

Studio del tempo atteso

- Poiché la sorgente $\{z_i\}_{i\geq 1}$ fornisce bit i.i.d, la probabilità che l'algoritmo restituisca 0 è data da $p(1-p)$ e questa ovviamente è uguale a $(1-p)p$, che è la probabilità che l'algoritmo restituisca 1
- Quando dunque l'algoritmo restituisce un risultato, questo è sempre corretto (nel senso che la probabilità delle due risposte è identica)
- Tuttavia finché le coppie di bit z_i e z_{i+1}, $i = 0, 2, \ldots$, sono uguali, l'algoritmo non restituisce nulla
- Quanti "cicli" dell'algoritmo dobbiamo attendere per avere il risultato?
- Se indichiamo con s il valore $1 - 2p(1-p)$, che è la probabilità che z_i e z_{i+1} siano uguali, la probabilità che il risultato sia restituito al k-esimo tentativo è data da $s^{k-1}(1-s)$
- Siamo dunque in presenza della ben nota distribuzione geometrica il cui valore atteso è:

$$\begin{aligned}
\sum_{k=1}^{\infty} ks^{k-1}(1-s) &= (1-s)\sum_{k=1}^{\infty} ks^{k-1} \\
&= (1-s)\frac{d}{ds}\int_0^s \left(\sum_{k=1}^{\infty} kt^{k-1}\right) dt \\
&= (1-s)\frac{d}{ds}\sum_{k=1}^{\infty}\int_0^s kt^{k-1} dt \\
&= (1-s)\frac{d}{ds}\sum_{k=1}^{\infty} s^k \\
&= (1-s)\frac{d}{ds}\left(\frac{1}{1-s} - 1\right) \\
&= (1-s)\frac{1}{(1-s)^2} \\
&= \frac{1}{1-s}
\end{aligned}$$

- Chiaramente, se $p = \frac{1}{2}$, e dunque anche $s = \frac{1}{2}$, ritroviamo il valore atteso 2 che è chiaramente corretto: se $\{z_i\}_{i\geq 1}$ è anche uniforme, bastano mediamente due tentativi
- Se la sorgente è invece molto "biased" verso 0 o 1, ad esempio se $p = \frac{1}{4}$, allora $s = \frac{5}{8}$ e il valore atteso è $\frac{8}{3} \approx 2.67$. Infine se $p = \frac{1}{10}$ il valor atteso diviene circa 5.56

Bit casuali e probabilità di terminazione

- È possibile anche un'analisi "complementare" del nostro semplice algoritmo Las Vegas.
- Possiamo cioè studiare la relazione fra bit casuali utilizzati (della sorgente originale $\{z_i\}_{i\geq 1}$ ovviamente) e la probabilità che l'algoritmo termini entro un certo numero di tentativi.
- Ci chiediamo quindi: qual è la probabilità che $2k$ bit siano sufficienti per ottenere il risultato, cioè il "bit uniforme"?
- Il conto è molto semplice: con $2k$ bit casuali possiamo eseguire k tentativi e la probabilità che l'algoritmo termini entro k tentativi è:

$$\sum_{i=1}^{k} s^{i-1}(1-s) = (1-s) + (s-s^2) + (s^2-s^3) + \ldots + s^{k-1} - s^k = 1 - s^k$$

dove, ricordiamo ancora una volta, s è la probabilità che i due bit della sorgente considerati al generico tentativo siano uguali

Esercizio

- Supponiamo di voler "risparmiare" bit casuali modificando l'algoritmo nel modo seguente:
  1. Poni $i = i + 1$ e ritorna al passo 2
- È una buona modifica? Argomentare la risposta.

Teoria e algoritmi per la primalità

La domanda di numeri primi

- I numeri primi (o semplicemente i primi) giocano un ruolo fondamentale in tutti gli algoritmi di crittografia asimmetrica.
- Di primi ne servono tanti, da utilizzare sia in diversi protocolli sia nell'ambito dello stesso protocollo con partecipanti diversi.
- È dunque naturale chiedersi quanto siano abbondanti i primi delle dimensioni che interessano in crittografia e come si possa determinarli.
- Sappiamo che i primi sono infiniti (si provi a dimostrarlo o a ricordare la dimostrazione); riguardo la loro abbondanza ci viene in aiuto il c.d. Prime number theorem, che descrive la distribuzione asintotica dei primi.
- Se $\pi(n)$ indica il numero di primi non maggiori di n, vale quanto segue

$$\pi(n) \sim \frac{n}{\ln n}$$

- Questo implica che, mediamente (e al crescere di n) fra i primi n numeri circa una frazione $1/\ln n$ sono primi.
- Da un punto di vista pratico, anche se con passaggio matematicamente assai discutibile, possiamo affermare che, se scegliamo a caso un numero n di sufficiente grandezza, abbiamo una probabilità circa $1/\ln n$ che esso sia primo.

Un esperimento con numeri non troppo grandi

In [ ]:
```
# pip install pycryptodome
```

In [ ]:
```
from Crypto.Util import number
from math import log
```

In [ ]:
```
numtrials = 10000                        # Numero di esperimenti
e = 2048                                 # Lunghezza in bit dei numeri generati
prob_estimate = 1.0/(e*log(2))           # Stima della probabilità
expected_num = numtrials*prob_estimate   # Valore atteso di primi
```

In [ ]:
```
# Generazione e test
numprimes = 0
for _ in range(numtrials):
    if number.isPrime(number.getRandomInteger(e)):
        numprimes += 1
```

In [ ]:
```
# Confronto con il numero atteso
print(f"Numero di primi trovati (stima, potrebbero esserci duplicati):\t{numprimes}")
print(f"Numero atteso di primi con {numtrials} esperimenti:\t\t\t{expected_num:.2f}")
print(f"Rapporto fra le due quantità:\t\t\t\t\t{numprimes/expected_num:.3f}")
```

Ricerca di un primo

- Una lettura "speculare" del risultato precedente ci consente di affermare che, per trovare un numero primo, scegliendo a caso fra i numeri di n bit, dobbiamo mediamente effettuare circa $n \ln 2 \approx 0.69n$ tentativi.
- Per i numeri comunemente utilizzati in crittografia, n può essere dell'ordine di qualche migliaia (2048 o 3072 per RSA, molto meno per protocolli su curve ellittiche).
- Disponendo di algoritmi efficienti per verificare se un numero è primo, il metodo comunemente utilizzato con buona efficienza consiste proprio nel generare e verificare, finché il numero generato non è primo
- Non tutti i numeri primi possono però andar bene, anche in dipendenza del protocollo utilizzato.
- Il seguente esperimento mostra come il numero medio di tentativi per trovare un primo sia in accordo con la prescrizione teorica.

In [ ]:
```
experiments = 1000
e = 512
avg = 0
for _ in range(experiments):
    attempts = 1
    while not number.isPrime(number.getRandomInteger(e)):
        attempts+=1
    avg+=attempts
print(f"Numero medio di tentativi effettuati:\t{float(avg)/experiments:.2f}")
print(f"Valore teorico:\t\t\t\t{0.69*e:.2f}")
```

Sul numero di tentativi per trovare primo sicuro

In [ ]:
```
experiments = 5
e = 512
avg = 0
for _ in range(experiments):
    attempts = 1
    q = number.getRandomInteger(e)
    while True:
        while not number.isPrime(q):
            attempts+=1
            q = number.getRandomInteger(e)
        p = 2*q+1
        if number.isPrime(p):
            break
        else:
            attempts+=1
            q = number.getRandomInteger(e)
    avg+=attempts
print(f"Numero medio di tentativi effettuati:\t{float(avg)/experiments:.2f}")
```

Test basato sul Piccolo Teorema di Fermat

- Il piccolo teorema di Fermat afferma che, se p è un numero primo, allora vale:

$$x^{p-1} = 1 \bmod p$$

per ogni x tale che $\mathrm{MCD}(x, p) = 1$. In particolare, quindi, il teorema vale per ogni x positivo e minore di p.

- Se n non è un numero primo, è possibile che risulti tanto $x^{n-1} \bmod n = 1$ quanto $x^{n-1} \bmod n \neq 1$.
- In particolare, se x è composto e risulta $x^{n-1} \bmod n = 1$, n è detto pseudo-primo base-x.
- L'esistenza di pseudo-primi impedisce di poter utilizzare il Piccolo Teorema di Fermat come criterio di primalità.
- Ad esempio, se non esistessero pseudo-primi base-2 basterebbe calcolare $2^{p-1} \bmod p$ e verificare se il risultato è 1 o meno.
- Il fatto importante è che gli pseudo-primi base-2 sono comunque pochi e tendono ad essere "straordinariamente pochi" quanto più la grandezza aumenta.
- Stime abbastanze precise ne limitano la frequenza, fra i numeri di 512 bit, a circa 1 su 10^{20}.

Un semplice test di primalità per numeri scelti a caso

- L'ultima osservazione suggerisce che, se n è scelto a caso e se si usa 2 come base, è quanto meno "improbabile" che si vada a cadere su un numero pseudo-primo base 2.
- L'idea può quindi facilemente tradotta nel sequente "algoritmo".
- Si noti (qui e nel seguito) l'importazione di funzioni dal package *ACLIB*, che contiene un po' di programmi Python usati (se non proprio spiegati) a lezione.

In [ ]:
```
from ACLIB.utils import modexp
def FLT_primality(n):
    "Fermat's Little Theorem primality test"
    return modexp(2,n-1,n) == 1
```

- Si presti bene attenzione e si rifletta sulle caratteristiche di questo test.
- Si tratta certamente di un algoritmo one-sided error, ma non di un algoritmo Monte Carlo, perché seimplicemente non effettua alcuna scelta casuale.
- Se dunque non sappiamo come è scelto l'input, non possiamo fare alcuna affermazione probabilistica! E pure se gli pseudo-primi base-2 sono rari, non possiamo escludere che il meccanismo di scelta vada a considerare proprio questi più frequentemente.
- Per questa ragione, se il risultato è True, non ha alcun senso ripetere il test per cercare di ridurre la probabilità di errore.

- Facciamo ora una piccola verifica sull'affidabilità del metodo, con numeri casuali di grandezza relativamente piccola.
- Al riguardo, riporto qui il codice per la funzione esponenziale modulare (e, conseguentemente, anche il codice per il prodotto modulare) vista nel precedente notebook.

In [ ]:
```
from Crypto.Util import number
numerrors = 0
e = 50    # Numeri di e bit
numexp = 50000
for _ in range(numexp):
    n = number.getRandomInteger(e)|1    # n deve essere dispari
    if FLT_primality(n) != number.isPrime(n):
        numerrors += 1
print(numerrors)
```

Numeri di Carmichael

- Si potrebbe pensare di migliorare il test basato sul Piccolo Teorema di Fermat provando più di una base e non solo 2.
- Ad esempio potremmo provare, oltre al valore 2, anche un'altra base $a \in \mathbf{Z}_n^*$ scelta a caso.
- Questa è certamente una buona idea, come suggerisce il seguente esempio

In [ ]:
```
FLT_primality(561)  # 341 = 11*31
```

In [ ]:
```
modexp(3,340,341) == 1
```

- 341 è il più piccolo pseudo-primo base 2 ma evidentemente non è uno pseudo-primo base 3.
- Possiamo quindi incorporare l'idea in un test più "sofisticato".

In [ ]:
```
from random import randint
from ACLIB.utils import Euclid
def improved_FLT_primality(n):
    "Improved Fermat's Little Theorem primality test."
    a = randint(3,n-1)
    if Euclid(a,n)>1:
        return False
    return (modexp(2,n-1,n) == 1) and (modexp(a,n-1,n) == 1)
```

In [ ]:
```
improved_FLT_primality(341)
```

- Il nuovo test è certamente più efficace per almeno due motivi.

  1. Perché, se n è composto, con probabilità non nulla il calcolo del MCD fra n e a lo rivela;
  2. Perché n potrebbe essere psedo-primo base 2 ma non pseudo-primo base a
- Se il test restituisce True (suggerendo che il numero sia primo=) "può", in linea di principio può avere senso ripeterlo perché esso effettua una scelta casuale e questo può portare a risultati differenti proprio per le ragioni 1. e 2. esposte sopra.
- Esistono però numeri n composti per i quali, qualsiasi sia la base a t.c. $\mathrm{MCD}(a,n) = 1$, risulta $a^{n-1} \bmod n = 1$.
- In altri termini, esistono pseudo-primi base a per ogni a t.c. $\mathrm{MCD}(a,n) = 1$.
- Questi numeri sono chiamati (numeri) di Carmichael.
- Se n è un numero di Carmichael, qualora la base a generata passi il test di Euclide, il test effettuato nell'ultima riga è totalmene inutile.
- I numeri di Carmichael sono estrememente rari (certamente più rari degli pseudo-primi base 2).
- Ce n'è però una relativa abbondanza fra i numeri "piccoli".
- I primi 5 sono: 561, 1105, 1729, 2465 e 2821.

In [ ]:
```
from ACLIB.utils import factorize
```

In [ ]:
```
factorize(2821)
```

In [ ]:
```
# Proviamo il test per ogni i t.c. MCD(i,n)=1
n = 1105
for i in range(2,n):
    if Euclid(i,n)==1:
        if modexp(i,n-1,n) != 1:
            print(f"{n} non è un numero di Carmichael")
print(f"{n} è primo oppure è un numero di Carmichael")
```

- Gli algoritmi utilizzati in pratica sono "veri" algoritmi probabilistici di tipo Monte Carlo.
- Questo vuol dire che possono essere ripetuti per "abbattere" la probabilità di errore.
- Si tratta di algoritmi la cui implementazione è quasi sempre semplice.
- La difficoltà risiede, anche in questo caso "quasi sempre", nella dimostrazione che la probabilità di errore (one-sided) è effettivamente limitata da una costante.
- Per gli studenti interessati, rimando ad una semplice ricerca con le parole chiave Rabin-Miller e Solovay-Strassen (dal nome dei ricercatori che li hanno ideati).

Residui quadratici e test di primalità di Solovay-Strassen

- Un elemento $a \in \mathbf{Z}_n$ è detto residuo quadratico (modulo n) se esiste $x \in \mathbf{Z}_n$ tale che $a = x^2 \bmod n$.
- Elenchiamo, senza dimostrazione, un po' di proprietà dei residui quadratici in $\mathbf{Z}_n^*$.

1. Se a è un r.q., allora anche $a^{-1} \bmod n$ è un r.q.
2. L'insieme dei r.q. forma un sottogruppo di $\mathbf{Z}_n^*$, che indicheremo con Q_n.
3. Se n è primo, la cardinalità di Q_n è esattamente $|\mathbf{Z}_n^*|/2 = \frac{n-1}{2}$ ed esso è formato da tutte le potenze pari di un generatore g.

```
In [ ]: # Esempio, i residui quadratici in Z*_23
        Q = {(i**2)%23 for i in range(0,23)}
        Q
```

Simbolo di Legendre

- Se p è un numero primo e $a \in \mathbf{Z}_p$, il simbolo di Legendre $\left(\frac{a}{p}\right)$ è la funzione così definita

$$\left(\frac{a}{p}\right) = \begin{cases} 0 & \text{se } p \text{ divide } a \\ 1 & \text{se } a \text{ è un r. q. } \bmod p \\ -1 & \text{se } a \text{ non è un r. q. } \bmod p \end{cases}$$

- C'è un modo semplice per calcolare il simbolo di Legendre, che è dato dal seguente criterio di Eulero

$$\left(\frac{a}{p}\right) = a^{\left(\frac{p-1}{2}\right)} \bmod p$$

che quindi riduce il problema al calcolo di un'esponenziale modulare.

- Il criterio di Eulero consente facilmente di dimostrare che:

$$\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right)$$

- È inoltre immediato verificare che, se $a \equiv b \pmod p$, allora

$$\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right)$$

- Più difficili da dimostrare sono le seguenti proprietà:

1. Legge di reciprocità quadratica. Se p e q sono primi dispari, vale:

$$\left(\frac{q}{p}\right)\left(\frac{p}{q}\right) = (-1)^{\frac{(p-1)(q-1)}{4}}$$

2. Criterio affinché 2 sia un r.q. modulo p

$$\left(\frac{2}{p}\right) = (-1)^{\frac{p^2-1}{8}}$$

da cui risulta che 2 è un r.q. se e solo se $p \equiv 1, 7 \pmod 8$, cioè se $p = 8k+1$ oppure $p = 8k+7$, per un qualche intero k.

3. Criterio affinché 5 sia un r.q. modulo p è che $p \equiv 1, 4 \pmod 5$.

- 2 e 5 ci interessano da vicino in quanto sono utilizzati come generatori nell'implementazione OPENSSL del protocollo di Diffie e Hellman.

Simbolo di Jacobi

- Il *simbolo di Jacobi* generalizza il simbolo di Legendre al caso dei numeri composti e per esso si usa la *stessa notazione*
- Sia dunque n un intero dispari e sia $n = p_1 \cdot p_2 \cdot \ldots \cdot p_k$ *la sua scomposizione in fattori primi*, non necessariamente distinti. Per ogni intero non negativo a definiamo

$$\left(\frac{a}{n}\right) = \prod_{i=1}^{k} \left(\frac{a}{p_i}\right)$$

- Si noti dunque che il simbolo di Jacobi coincide con il simbolo di Legendre se n è primo.
- Il seguente semplice esempio mostra tuttavia che se, se n è composto, la condizione $\left(\frac{a}{n}\right) = 1$ non assicura che a sia un residuo quadratico modulo n:

$$\left(\frac{2}{15}\right) = \left(\frac{2}{5}\right)\left(\frac{2}{3}\right) = (-1)(-1) = 1$$

- È vero invece che se $\left(\frac{a}{n}\right) = -1$ allora a non è un residuo quadratico.
- Se viceversa sappiamo che a è un residuo quadratico modulo n, allora può succedere che $\left(\frac{a}{n}\right) = 0$, ma solo nel caso in cui $\mathrm{MCD}(a, n) > 1$, altrimenti $\left(\frac{a}{n}\right) = 1$

- Il simbolo di Jacobi è dunque un valore nell'insieme $\{0, 1, -1\}$ e gode di *molte delle proprietà "di calcolo"* del simbolo di Legendre
  1. Proprietà *moltiplicativa*:
  $$\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right)\left(\frac{b}{n}\right)$$
  2. Se $a \equiv b \pmod{n}$, allora
  $$\left(\frac{a}{n}\right) = \left(\frac{b}{n}\right)$$
  3. Se n ed m sono *interi dispari*:
  $$\left(\frac{m}{n}\right) = \begin{cases} -\left(\frac{n}{m}\right) & \text{se } n, m \equiv 3 \pmod{4} \\ \left(\frac{n}{m}\right) & \text{altrimenti} \end{cases}$$
  4. Vale inoltre:
  $$\left(\frac{2}{n}\right) = 1 \quad \text{sse } n \equiv 1, 7 \pmod{8}$$
- Il simbolo di Jacobi, tuttavia, *non può essere preso come criterio* per stabilire se un numero a è un r.q. modulo n.

Calcolo del simbolo di Jacobi

- Le proprietà elencate in precedenza consentono di calcolare il simbolo di Jacobi $\left(\frac{a}{n}\right)$ *senza dover fattorizzare n*, requisito questo di importanza decisiva.

```python
from ACLIB.utils import Euclid,modexp
def jacobiS(a,n):
    '''Computes the Jacobi simbol (a/n). It is assumed (in the top leve call)
       the n is an odd positive integer'''
    if Euclid(a,n)>1:
        return 0
    if a==1:        # By definition, as the product of Legendre symbols
        return 1
    if a==2:        # Apply property 4 above
        return 1 if n%8==1 or n%8==7 else -1
    if a%2==0:      # Apply property 1 above
        return jacobiS(2,n)*jacobiS(a//2,n)
    if a>n:         # Apply property 2 above
        return jacobiS(a%n,n)
    if a%4==3 and n%4==3:     # Otherwise, apply property 3 above, checking for the case
        return -jacobiS(n,a)  # a=n=3 (mod 4)
    return jacobiS(n,a)
```

- Provare a dimostrare che la precedente funzione termina sempre (e dunque è corretta perché se viene restituito un valore questo riflette le proprietà note
- La dimostrazione è solo un fatto di logica, assumendo vere le proprietà sopra elencate

```python
jacobiS(756479,1298351)
```

Test di primalità di Solovay-Strassen

- Se n è un numero primo sappiamo che vale l'uguaglianza

$$\left(\frac{a}{n}\right) = a^{\frac{n-1}{2}} \mod n$$

- Nel caso l'uguaglianza non valga possiamo quindi certamente dire che il numero è composto. che invece sappiamo *valere sempre se n è primo*.
- Cosideriamo ancora, come esempio, il caso di $n = 15$

- Ricordiamo che, se n è primo, vale l'uguaglianza

$$\left(\frac{a}{n}\right) = a^{\frac{n-1}{2}} \mod n$$

- Nel caso l'uguaglianza non valga possiamo quindi certamente dire che il numero è composto.
- È chiaro che, se non sappiamo che n è primo (e ovviamente non lo sappiamo, perché è ciò che vogliamo determinare) dobbiamo calcolare il simbolo di Jacobi, che vale per n qualsiasi (primo o composto)

- Vediamo un esperimento che suggerisce perché il test di Solovay-Strassen, essenzialmente basato su questa osservazione, si può progettare come algoritmo Monte Carlo

```python
# totient è la funzione di Eulero: totient(n) calcola il numero di numeri più piccoli di n
# e coprimi con n (1 incluso). E' l'ordina del gruppo Z*_n
from sympy.ntheory.factor_ import totient
```

In [ ]:
```python
while ((n := number.getRandomInteger(10)) and (number.isPrime(n) or n%2==0)): pass
print(f"Abbiamo generato {n}")
# Abbiamo generato un numero composto dispari
# Andiamo ora a verificare quanti numeri a in Z*_n "ingannano" il test di Eulero
count=0
order = totient(n)
for a in range(1,n):
    if Euclid(a,n)>1: continue
    J = jacobiS(a,n)%n
    E = modexp(a,(n-1)>>1,n)
    if (J==E):
        count+=1
        #print(a)
print(order,count,order%count)
```

- Come si può vedere, l'uguaglianza non vale quasi per tutti in numeri in $\mathbf{Z}_n$
- Un numero composto n tale che l'uguaglianza vale per un certo intero a è detto *pseudo-primo di Eulero rispetto alla base a*.
- L'algoritmo di Solovay e Strassen si basa sul fatto che un numero n composto non è pseudo-primo di Eulero per tutte le possibili basi a.
- Una valore a per cui $\mathrm{MCD}(a,n) > 1$ o tale per cui risulta $\left(\frac{a}{n}\right) \neq a^{\frac{n-1}{2}} \mod n$ viene detto *testimone della non primalità di n*

Pseudo codice dell'algoritmo

1. Dato $n > 2$ dispari, scegli $a \in \{1, 2, 3, \ldots, n-1\}$ uniformemente a caso
2. Calcola $m = \mathrm{MCD}(a,n)$ e verifica se $m > 1$; in caso affermativo restituisci `composto`
3. Altrimenti calcola $J = \left(\frac{a}{n}\right)$ e $P = a^{\frac{n-1}{2}} \mod n$
4. Se $P \neq J$ restituisci `composto`
5. Altrimenti restituisci `probabilmente primo`

Proprietà dell'algoritmo (non dimostrate)

- Se il numero è primo, l'algoritmo restituisce sempre la *risposta corretta* perché i test di riga 2 e 4 sono sempre falsi e dunque l'output è `probabilmente primo`
- Tuttavia, se n è composto, l'algoritmo *può restituire la risposta errata* `probabilmente primo` nel caso in cui n sia pseudo-primo di Eulero rispetto alla base a
- È però possibile dimostrare che, per ogni intero composto dispari n, ci sono al più $\frac{n-1}{2}$ interi a tale che n è uno pseudo-primo di Eulero rispetto ad a
- Ne consegue che almeno $\frac{n-1}{2}$ numeri sono testimoni della non primalità e dunque che l'algoritmo erra (su input un numero composto) con *probabilità non superiore a $\frac{1}{2}$*

Una funzione candidato trapdoor

Fattorizzazione di interi

- Sul problema di fattorizzare efficientemente numeri composti è basata, in particolare, la "forza" del protocollo RSA per cifratura e firma digitale
- Dato un numero composto n, l'algoritmo brute-force prova tutti i potenziali divisori $d \leq \sqrt{n}$ e impiega chiaramente tempo esponenziale nella lunghezza di n
- Per i numeri in gioco nella crittografia, si tratta di un approccio destinato al fallimento
- Esistono però algoritmi molto più efficienti, non di costo polinomiale ma neppure esponenziale
- Il migliore algoritmo noto è il General Number Field Sieve, di cui si "stima" in modo euristico che la complessità sia una funzione del tipo $G(n) = e^{c\sqrt[3]{(\ln n)(\ln \ln n)^2}}$, con $c < 3$ costante
- Una funzione del tipo $(1+c)^{d \cdot n}$, per ogni $c, d > 0$ costanti, cresce più velocemente rispetto a $G(n)$ e questo fornisce una spiegazione del perché RSA richieda numeri molto elevati (fattori di almeno 2048 bit)
- In generale, gli algoritmi di fattorizzazione sub-exponenziali sono piuttosto complessi e non li discuteremo ulteriormente
- Presenteremo invece l'algoritmo di fattorizzazione ρ di Pollard, che (non a caso) ha lo stesso nome dell'algoritmo per trovare collisioni di funzioni hash che vedremo.
- Il pregio dell'algoritmo è la semplicità, il difetto è che richiede un numero di passi che è ancora esponenziale nella lunghezza di n, sebbene l'esponente possa essere sostanzialmente minore di n

Algoritmo di fattorizzazione ρ di Pollard

- L'algoritmo produce, a partire da un valore $x_1 \in \mathbf{Z}_n$, una successione di valori $x_{j+1} = f(x_j) \bmod n$, $j = 1, 2, \ldots$ che, almeno idealmente, dovrebbe esibire le proprietà di una sequenza casuale che tuttavia, da un certo punto in poi, inizierà a ripetersi
- Ogni tanto, uno dei valori calcolati viene salvato ed utilizzato nelle successive iterazioni (fino al salvataggio successivo) per verificare se è stato determinato un fattore di n
- Le iterate salvate sono quelle di indice $i = 2^t$, $t = 0, 1, 2, \ldots$
- Ogni iterata salvata viene dunque utlizzata nei test per un numero di passi che raddoppia ad ogni successivo salvataggio
- Presentiamo prima lo pseudocodice e poi un'implementazione in Python
- Risultati in linea con le aspettative teoriche sono tipicamente ottenuti utilizzando la seguente funzione

$$f(x) = (x^2 - 1) \bmod n$$

di cui anche qui facciamo uso

Algoritmo ρ di Pollard per la fattorizzazione di interi

1. $j = 1, i = 2$ */ j indica l'iterata corrente, i è l'indice del successivo salvataggio /*
2. $y = x_1 = r \in \mathbf{Z}_n$ */ y è il valore salvato /*
3. while True
4. $\quad x_{j+1} = (x_j^2 - 1) \bmod n$
5. $\quad j = j + 1$
6. $\quad m = \mathrm{MCD}(y - x_j, n)$
7. $\quad$ if $m > 1$ and $m \neq n$
8. $\qquad$ print(m); break
9. $\quad$ if $j == i$ */ Controlliamo se siamo arrivati al prossimo salvataggio /*
10. $\qquad i = 2 \cdot i$
11. $\qquad y = x_j$

Osservazioni "semplici"

- È di facile constatazione il fatto che l'algoritmo o non termina mai oppure, se termina, restituisce effettivamente un fattore non banale di n
- Come secondo step (per comprendere l'idea elegante che sta alla base dell'algoritmo), analizziamo il test che porta eventualmente alla scoperta di un tale fattore, indicato con m nello pseudo-codice
- Evidentemente, in quanto $m = \mathrm{MCD}(y - x_j, n)$, esso divide anche la differenza $y - x_j$
- Detto in altri termini, questo vuol dire y e x_j sono congrui modulo m
- Scopo dell'algoritmo è dunque di andare a cercare coppie di iterate che siano diverse ma congrue modulo un divisore di n
- La parte "geniale" dell'algoritmo è la strategia con cui viene effettuata la ricerca, ovvero tenendo fissa una delle due iterate nel confronto (la y dello pseudo-codice) per periodi via via più lunghi

Implementazione Python e qualche esperimento

- L'implementazione è data come funzione che (eventualmente) restituisce un fattore non banale di n

```python
from ACLIB.utils import Euclid, rand
def pollard_rho(n):
    '''Implements the Pollard's rho heuristics and (possibly)
       returns a non-trivial divisor of n.
       Strictly follows the description in
       Cormen et al. Introduction to Algorithms, Third ed.
    '''
    i,j = 2,1
    xprec = rand(n)  # xprec = x_1
    y = xprec        # Salvataggio del primo valore
    while True:
        j += 1
        x = (xprec*xprec - 1)%n            # calcolo iterata successiva
        if (p:=Euclid(y-x,n))!=1 and p!=n:
            return p
        if j==i:
            y = x
            i *= 2
        xprec = x
```

```python
n = (1097**5)*12983*(397**2); n
```

```python
m = pollard_rho(n); m
```

```python
n //= m; n
```

```python
pollard_rho(n)
```

```python
n = 13**2*7**6*17**3; n
```

```python
pollard_rho(n)
```

```python
from ACLIB.utils import getprime
```

```python
p = getprime(10**12);p
```

```python
q = getprime(10**12);q
```

```python
n = p*q; n
```

```python
pollard_rho(n)
```

La strategia spiegata

- Per ogni possibile fattore q di n esiste una sorta di "successione ombra" $\{z_i\}_{i\geq 1}$, ottenuta riducendo modulo q ogni termine x_i della successione effettivamente costruita
- Naturalmente, non conoscendo i fattori (che sono proprio ciò che andiamo a cercare) noi tali successioni non le conosciamo, ma questo non importa
- Ciò che importa sono le proprietà che di esse possiamo postulare e che, come vedremo, dicono qualcosa della successione $\{x_i\}_{i\geq 1}$
- Concentriamoci su una tale successione, per un qualche fattore q di n
- Ciò che possiamo subito dimostrare è che la successione ombra ha la stessa struttura della $\{x_i\}_{i\geq 1}$
- Al riguardo, utilizziamo la seguente proprietà, di facile verifica. Per ogni intero x, vale

$$(x \bmod n) \bmod q = x \bmod q$$

qualora q sia un divisore di n (basta usare la definizione di resto)

- Possiamo in tal caso scrivere:

$$\begin{aligned} z_{i+1} &= x_{i+1} \bmod q \\ &= ((x_i^2 - 1) \bmod n) \bmod q \\ &= (x_i^2 - 1) \bmod q \\ &= ((x_i \bmod q)^2 - 1) \bmod q \\ &= (z_i^2 - 1) \bmod q \end{aligned}$$

- Ogni passo dell'algoritmo ha quindi una identica controparte nei valori che abbiamo definito "ombra". In particolare possiamo affermare che:
  1. Nell'ipotesi di comportamento realmente casuale, anche la successione ombra è caratterizzata da una sorta di "antiperiodo" (il gambo del ρ) e un periodo (la testa) le cui lunghezze (indicate con λ e σ) soddisfano $\lambda, \sigma = O(\sqrt{q})$
  2. Con la strategia di salvare i valori ad intervalli sempre più grandi abbiamo la certezza che, per un certo valore dell'indice i, il valore ombra z_{2^i} salvato è dentro la testa del ρ; e ci aspettiamo che risulti $2^i = O(\sqrt{q})$
  3. Sempre per la stessa ragione, per un qualche indice i maggiore o uguale al precedente, la quantità 2^i è maggiore σ (e dunque, ancora, $2^i = O(\sqrt{q})$)
  4. Questo però implica che esiste $j > 2^i, j \le 2^{i+1}$ tale che $z_j = z_{2^i}$
- A questo punto, però, ricordiamo che $z_i = x_i \bmod q$, per ogni i, e dunque, nella successione originale

$$x_j \bmod q = x_{2^i} \bmod q$$

  e quindi che $x_j - x_{2^i}$ è un divisore di q e dunque di n
- La seguente figura (tratta dal libro Cormen et al. Introduction to Algorithms, Third ed.) illustra la situazione per il particolare valore $n = 1387 = 19 \cdot 73$

```
In [ ]: (84-814)%73
```

```
In [ ]: # Con riferimento alla successione mod 73
        (31**2-1)%73 == 11 and 814%73 == 11  # I due modi di ottenere z_8=11
```

```
In [ ]: # Poiché z_8 = z_12, risulta che x_8-x_12 è divisibile per 73
        Euclid(814-84,1387)
```

Cosa può andare storto?

- Potrebbe succedere che i valori di λ e σ per più di una sequenza ombra siano tali che la "scoperta" di due iterate identiche avvenga nello stesso istante
- Per esemplificare il caso "meno improbabile", se $n = p \cdot q$, con p e q primi, la concomitanza della scoperta fa sì che le corrispondenti iterate x_j e x_{2^i} siano congruenti modulo p e modulo q
- Poiché p e q sono primi, questo implica che x_j e x_{2^i} sono anche congrui modulo n e dunque che $\mathrm{MCD}(x_{2^i}, x_j) = n$
- Questo ovviamente non aiuta
- Nel caso di scomposizione $n = p^k$, può "semplicemente" succedere che quando $z_{2^i} = z_j$ risulti anche $x_{2^i} = x_j$
- Anche in questo caso non viene scoperto nulla

```
In [ ]: pollard_rho(7**2)
```

```
In [ ]: pollard_rho(23**3)
```

Efficienza

- Avendo ben presente il carattere euristico dell'analisi relativa alla lunghezza di λ e σ per le varie sequenze ombra, è ragionevole attendersi che il numero di operazioni necessarie per trovare il più piccolo fattore primo p di n sia $O(\sqrt{p})$
- Poiché $p < \sqrt{n}$, la stima complessiva diviene $O(\sqrt[4]{n})$ operazioni su numeri di $O(\log n)$ bit

Fattorizzazione completa

- Possiamo ora utilizzare la funzione pollard_rho per tentare la fattorizzazione completa di n
- Dobbiamo però prima di tutto decidere come rappresentare le fattorizzazioni

In [ ]:
```
class primefact(dict):
    '''Simple class to represent factorizations'''
    def __new__(cls,*args):
        return super().__new__(cls,{})
    def __init__(self,*args):
        assert not len(args)&1, "Wrong arguments"
        for i in range(0,len(args),2):
            self[args[i]]=args[i+1]
    def primes(self):
        return set(self.keys())
    def __repr__(self):
        return " * ".join([str(p) if e==1 else str(p)+"**"+str(e) for p,e in self.items()])
    def merge(self,other):
        for k in self.primes().union(other.primes()):
            self[k]=self.get(k,0)+other.get(k,0)
```

In [ ]:
```
a = primefact(2,2,3,4);a
```

In [ ]:
```
a.primes()
```

In [ ]:
```
b = primefact()
```

```python
from ACLIB.utils import isPrime, BSGS
from math import sqrt
def BruteForceFact(n):
    '''Computes the prime facorization using brute force.
       Use carefully for small (and odd) values of n
    '''
    assert n&1, "n must be odd"
    F = primefact()
    while n>1 and not isPrime(n):
        for i in range(3,int(sqrt(n))+1,2):
            if n%i==0:
                F.merge(primefact(i,1))
                n=n//i
                break
    if n>1:
        F.merge(primefact(n,1))
    return F

def factorize(n,nmax=10**6):
    '''Return the prime factorization of n'''
    def fact(n):
        '''Inner function that actually does the heavy job.
           Uses brute force for small n and Pollard's rho heuristics otherwise.
           Not guaranteed to terminate, since pollard_rho
           does not halt occasionally
        '''
        if n<=nmax:
            return BruteForceFact(n)
        elif isPrime(n):
            return primefact(n,1)
        f = pollard_rho(n)
        F = fact(f)
        F.merge(fact(n//f))
        return F
    # We first deal with the case n=p 2^q and compute q (p is odd)
    i = 0
    while not n&1:
        i += 1
        n = (n>>1)
    if i==0:
        d = primefact()
    else:
        d = primefact(2,i)
    # What remains is p
    if n != 1:
        d.merge(fact(n))
    return d
```

In [ ]:
```python
n = 17**4*31**3*59; n
```

In [ ]:
```python
factorize(n)
```

In [ ]:
```python
n=1387; factorize(n)
```

In [ ]:
```python
n = 8965*10001**2; n
```

In [ ]:
```python
f=factorize(n); f
```

In [ ]:
```python
eval(str(f))
```

In [ ]:
```python

```