**Aritmetica e crittografia: l’algoritmo RSA**

[Enrico Sailis](mailto:enricosailis@gramsciamaldi.edu.it)

I.I.S. “Gramsci-Amaldi”

Carbonia (SU)

email: enricosailis@gramsciamaldi.edu.it

**Introduzione**

Questo lavoro fa parte di un progetto per la realizzazione di un software in python per la criptazione di messaggi digitali, finalizzato ad accrescere l’interesse dei giovani per la matematica e l’informatica. Il progetto prevede una serie di incontri pomeridiani rivolti agli alunni interessati e si baserà su materiale reperibile gratuitamente in rete. Il tema quì trattato è la **crittografia a chiave pubblica**, e in particolare l’algoritmo **RSA**. Per progettare un percorso sulla crittografia RSA realizzabile e soprattutto sperimentabile dagli studenti, ho sfruttato un lavoro del 2006 del **prof. Michele Impedovo** che ho copiato per la quasi totalità e di cui ho voluto riscrivere solo alcune parti e la parte relativa al software nel linguaggio python da me scelto.

Procederemo a ritroso, dall’algoritmo RSA al Teorema di Eulero e al Piccolo Teorema di Fermat, fino alle operazioni di addizione e moltiplicazione in Zn e al codice ASCII. Gli algoritmi saranno implementati in python utilizzando l’ambiente di sviluppo software libero [Visual Studio Code](https://code.visualstudio.com/Download) della Microsoft.

Tutti i file utilizzati descritti nell’articolo sono contenuti nel repository:

<https://github.com/EnricoSailis/Criptografia_RSA>

Come nell’esperienza precedente del prof. Impedovo, spero che questo lavoro dia ai ragazzi e al sottoscritto tanta soddisfazione.

**La crittografia a chiave segreta**

Il primo codice crittografico di cui si ha testimonianza storica è il semplicissimo codice di Cesare (Svetonio, Vitae Caesarum, I, 56); consiste nel sostituire ad ogni carattere C di un messaggio un altro carattere C’ che si trova, nell’ordine alfabetico, n posti più avanti rispetto a C; se si supera la “z” si ricomincia a contare dalla “a”.

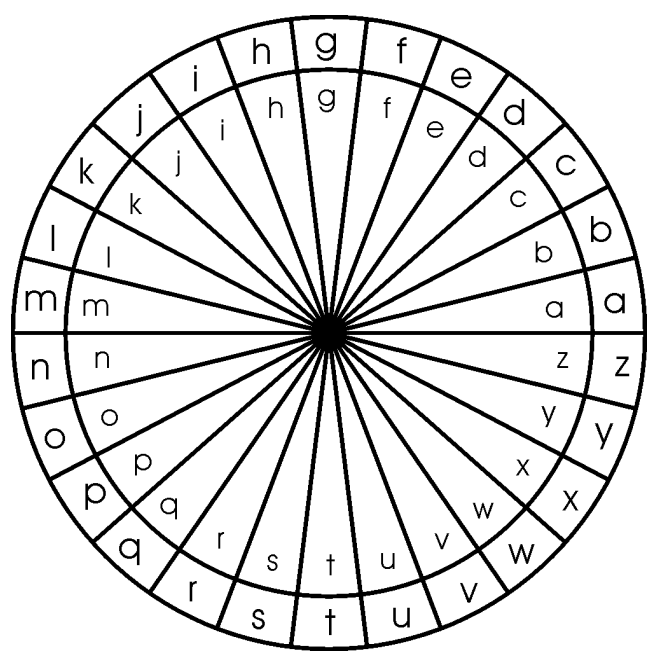
Per esempio, se n = 7, allora il messaggio

“matematica”

viene codificato in

“thalthapjh”.

Chi riceve il messaggio associa ad ogni carattere la lettera che si trova 7 posti più indietro (se si supera la “a” si ricomincia a contare dalla “z”) e lo decodifica.



L’immagine precedente suggerisce uno strumento (due dischi sovrapposti che ruotano uno sull’altro) che si potrebbe costruire alla scuola elementare per realizzare un gioco-attività con lettere e numeri sul codice di Cesare.

Il numero n è la **chiave** dell’algoritmo, la cosiddetta **chiave segreta**: non deve essere nota ad altri e deve essere preliminarmente concordata dai due utenti.

Il codice di Cesare è facilmente scardinabile: sia perché le possibili diverse chiavi sono solo 26 (basta provarle tutte), sia perché in ogni lingua le lettere compaiono con una certa frequenza statistica (vedi ad esempio www.riksoft.com/criafreq.htm) e dunque, soprattutto se il messaggio è abbastanza lungo, non è difficile formulare congetture sensate e verificarle con rapidità. Nel nostro esempio è naturale osservare che nella parola thalthapjh ci sono tre “h”, e che con tutta probabilità rappresentano una vocale; con pochi tentativi si ottiene la decodifica.

Inoltre la chiave n, per essere affidabile, dovrebbe essere modificata spesso, con ovvie difficoltà di comunicazione tra i due utenti.

Il **codice di Vigenère** (Blaise de Vigenère, 1523-1596) introduce una variante che lo rende decisamente più sicuro del codice di Cesare: le lettere non sono traslate tutte dello stesso valore n. Più precisamente: i due utenti si scambiano preliminarmente un vettore numerico di lunghezza arbitraria, tipicamente la sequenza numerica corrispondente ad una parola; per esempio [1,13,15,18,5] corrisponde alla parola “amore” ed è la chiave segreta dell’algoritmo. La codifica del messaggio avviene nel seguente modo:

− il 1° carattere viene traslato di 1 posto;

− il 2° carattere viene traslato di 13 posti;

− il 3° carattere viene traslato di 15 posti;

− il 4° carattere viene traslato di 18 posti

− il 5° carattere viene traslato di 5 posti;

− il 6° carattere viene traslato di 1 posto;

− il 7° carattere viene traslato di 13 posti

− ...

e così via. La parola “matematica” viene così trasformata in

“nniwrbgxuf”.

Lettere uguali nel messaggio originale non corrispondono in generale a lettere uguali nel messaggio cifrato (e viceversa), e questo lo mette al riparo da una decodifica immediata; ovviamente, tanto più lunga è la chiave segreta, tanto più sicuro è il codice. Tuttavia non è difficile immaginare che, per esempio utilizzando un calcolatore, si possa in breve tempo scoprire la chiave segreta e decodificare il messaggio. Un metodo efficiente per scardinare il codice di Vigenère è stato proposto, ben prima dell’avvento dei calcolatori, da Friedrich Kasiski (1805-1881).

**La rivoluzione della chiave pubblica**

Gli algoritmi di Cesare e di Vigenère hanno la stessa struttura: ad una stringa S il mittente applica una certa funzione ƒ e invia al destinatario il messaggio criptato ƒ(S). Il destinatario applica a ƒ(S) la funzione inversa ƒ−1 e legge il messaggio in chiaro ƒ−1 (ƒ(S)) = S. In entrambi gli algoritmi la segretezza del messaggio è affidata alla segretezza della chiave; nota questa, la funzione inversa è immediatamente disponibile: se la chiave di codifica è il numero n, o il vettore v, quella di decodifica è il numero −n, o il vettore −v.

La crittografia a chiave pubblica ha rivoluzionato i meccanismi di codifica e decodifica dei messaggi: non occorre più che mittente e destinatario si mettano preliminarmente d’accordo su quale chiave segreta utilizzare.

Nell’epoca di Internet e delle transazioni elettroniche, è indispensabile che B possa mandare un messaggio segreto ad A (per esempio il numero della pro-

pria carta di credito), senza dover preliminarmente comunicare ad A quale chiave segreta utilizzare. Infatti tale comunicazione viaggerebbe anch’essa su canali esposti ad eventuali malintenzionati. Se C intercetta la chiave segreta, può facilmente decodificare qualunque messaggio.

Gli algoritmi a chiave pubblica eliminano questa falla: ad ogni utente vengono assegnate, da una Certification Authority (spesso abbreviata in CA, vedi per esempio [www.thawte.com](http://www.thawte.com)), una **chiave pubblica**, che compare in un elenco consultabile da chiunque, e una **chiave privata**, che l’utente deve tenere gelosamente segreta.

Se B vuole mandare un messaggio ad A, non deve prima concordare con A quale chiave usare; deve semplicemente cercare sull’elenco la chiave pubblica di A, e cifrare il messaggio, mediante un opportuno algoritmo di pubblico dominio, con questa chiave. Una volta cifrato, il messaggio può essere decodificato solo da chi conosce la chiave privata, e dunque solo da A. Perché questo sia possibile, è necessario che la funzione ƒ (che codifica il messaggio) e la funzione ƒ−1 (che lo decodifica) non siano simmetriche: cioè non deve essere possibile ricavare ƒ−1 direttamente da ƒ.

Si obietterà: ma se la funzione ƒ è pubblica, non dovrebbero esserci problemi particolari a ricavare ƒ−1, soprattutto nell’era della rivoluzione elettronica. Ci si chiede: è possibile progettare un algoritmo per il quale sia praticamente impossibile ricavare ƒ−1 da ƒ? La metafora dell’elenco telefonico può servire a capire che un tale algoritmo, almeno in linea di principio, è possibile. Trovare il numero di telefono di un utente a partire dal nome è immediato (questa è la funzione ƒ); invece trovare il nome di un utente partendo dal suo numero di telefono (la funzione inversa ƒ−1) è praticamente impossibile, perché richiederebbe una quantità di tempo tale da risultare nei fatti impraticabile. Nel seguito utilizzeremo spesso il termine “impossibile” in questo senso: non calcolabile in tempi ragionevoli.

Il primo algoritmo che soddisfa i requisiti della chiave pubblica è stato messo a punto solo nel 1977 da Ron **R**ivest, Adli **S**hamir e Leonard **A**dlemann, ed è noto, dalle iniziali degli autori, come **algoritmo RSA**. Ancor oggi è uno dei più utilizzati. Come vedremo in dettaglio, per l’algoritmo RSA la funzione “facile da calcolare” è la potenza di due numeri in Zn con n dato dal prodotto di due numeri primi molto grandi; oggi l’ordine di grandezza di assoluta sicurezza utilizzato per p e q è circa 10300: noti p e q, il prodotto n = pq si calcola immediatamente. La funzione inversa, quella “impossibile da calcolare”, è la fattorizzazione: se un numero n è il prodotto di due numeri primi “grandi”, la sua fattorizzazione richiede di norma tempi superiori all’età dell’universo. Infatti non si conoscono a tutt’oggi (e c’è ragione di ritenere che non si conosceranno mai), algoritmi efficienti di fattorizzazione.

È interessante notare, a questo proposito, l’enorme sproporzione degli attuali risultati della ricerca matematica avanzata sui due problemi apparentemente analoghi:

1) stabilire se un numero naturale è primo;

2) fattorizzare un numero naturale.

Il primo problema è “facile”, il secondo è “impossibile”.

Il riconoscimento (e quindi la costruzione) di numeri primi grandi non offre oggi particolari difficoltà computazionali. Esistono algoritmi molto efficienti in grado di riconoscere numeri primi anche di migliaia di cifre in pochi decimi di secondo.

I più efficienti di essi (vedi per esempio l’algoritmo di Miller-Rabin, oppure l’algoritmo di Solovay-Strassen, che viene illustrato nell’Appendice C) sono di tipo probabilistico; viene effettuato un certo numero di test sul numero e:

− se tutti i test vengono superati, il numero viene dichiarato primo con una probabilità che può essere resa vicina a 1 quanto si vuole in tempi rapidi;

− se uno solo dei test non viene superato allora il numero è certamente composto.

Con Visual Studio Code il programma **GeneraNumeriPrimi** in python determina il più piccolo numero primo maggiore di un numero casuale di n cifre.

# Generazione di numeri primi casuali con un numero qualsiasi di cifre con la funzione nextprime() di sympy

import sympy as sp

import secrets as sc

n=int(input('Digita il numero delle cifre del numero primo che vuoi ottenere: '))

# Si calcola il numero nbit di bit che occorrono per rappresentare il numero di n cifre in notazione binaria

nbit = int(3\*sp.log(10\*\*n,8))

# viene generato un numero binario casuale sn con un numero nbit di bit

sn = sc.randbits(nbit)

# viene cercato il numero primo p successivo al numero intero sn

p=sp.nextprime(sn)

print("Numero primo random con ",n," cifre: ")

print(p)

Oggi si conoscono numeri primi con milioni di cifre. Per i record sui numeri primi fino ad oggi conosciuti visita il sito: <https://t5k.org/>.

Per quanto riguarda invece la fattorizzazione del prodotto di due numeri primi vedi per esempio <https://mathworld.wolfram.com/RSANumber.html>, vi sono state sfide aperte con in palio migliaia di dollari per chi sarebbe riuscito a fattorizzare dei numeri dati.

Si può utilizzare un qualunque software di matematica per rendersi conto, almeno in modo empirico e del tutto qualitativo, dei tempi necessari per la fattorizzazione di un numero naturale n in funzione della lunghezza di n, cioè del numero di cifre di n.

Con i più potenti algoritmi e con i più potenti calcolatori disponibili oggi, i tempi di fattorizzazione restano esponenziali rispetto alla lunghezza di n. La fattorizzazione del prodotto di due numeri primi di 300 cifre richiede tempi superiori all’età dell’universo. L’algoritmo RSA funziona perché a tutt’oggi non si conoscono algoritmi efficienti di fattorizzazione (nonostante la ricerca matematica sia attivissima

in questo settore), ma non è stato dimostrato da alcuno che sia inattaccabile. RSA perderebbe istantaneamente la sua inattaccabilità il giorno in cui si scoprisse un algoritmo di fattorizzazione capace di lavorare in tempi polinomiali (e non esponenziali) rispetto alla lunghezza dell’input.

Dato che RSA si basa sulla non esistenza di algoritmi rapidi di fattorizzazione, nella trattazione che segue dovremo sempre porre attenzione non solo agli aspetti matematici, ma anche a quelli computazionali. Non avrebbe senso illustrare un procedimento senza preoccuparci, passo-passo, della sua effettiva calcolabilità.

Noi lavoreremo con Visual Studio Code e utilizzeremo fattori p e q di 300 cifre, perciò la lunghezza di n sarà di 600 cifre. Quindi lavoreremo con numeri il cui ordine di grandezza è quello effettivamente utilizzato nella pratica, mostreremo che Visual Studio Code con la libreria python di calcolo numerico sympy può essere adatto allo scopo.

**Lo schema generale**

Vediamo lo schema generale adottato da un codice crittografico; se l’utente B deve mandare ad A un testo T, cioè una stringa di caratteri, allora:

1) B trasforma T in un numero naturale N, mediante una funzione g nota a tutti gli utenti;

2) B applica al numero N la funzione di codifica ƒ, nota a tutti, ottenendo un numero M, e manda M ad A:

N = g(T) ; M = f(N)

3) A riceve il messaggo M e lo trasforma in N mediante la funzione di decodifica ƒ−1, nota solo ad A, e ottiene il numero N;

4) A applica a N la funzione g−1 e ottiene finalmente il messaggio originale:

N=f-1(M) ; T = g-1(N)

Ci occuperemo inizialmente delle funzioni g e g−1 che trasformano un testo in un numero naturale e viceversa, e poi delle due funzioni aritmetiche ƒ e ƒ−1 che costituiscono il cuore dell’algoritmo crittografico.

**Dal testo al numero e viceversa**

Il punto di partenza per trasformare caratteri in numeri è il codice ASCII (American Standard Code for International Interchange), che definisce una corrispondenza biunivoca tra i 128 (o 256 per l’ASCII esteso) caratteri dello standard ISO e i numeri naturali 0, 1, ..., 127 (o 0, 1, ..., 255 per l’ASCII esteso).

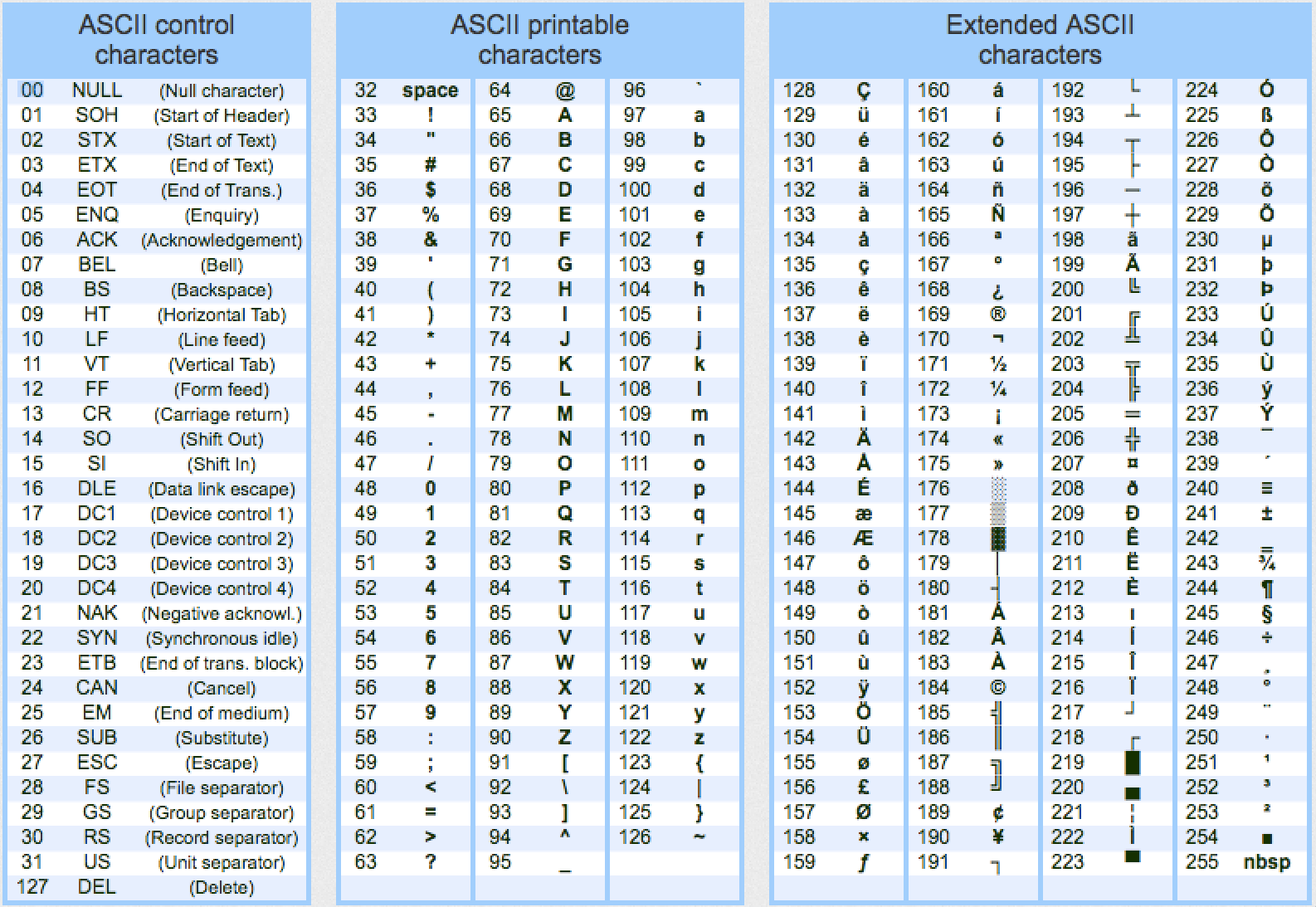


Tabella ASCII

Il seguente programma python “DaStringaANumero” , la funzione g, trasforma una stringa s in un numero intero n sfruttando la funzione python ord(c) che trasforma un carattere c nel codice ASCII corrispondente:

# Trasforma una stringa di caratteri ASCII in un numero intero

n=0

s = str(input("Inserisci una stringa da convertire in numero: "))

for c in s:

n=n\*128+ord(c)

print(n)

Ora spieghiamo il funzionamento del programma.

Per trasformare una stringa in numero costruiamo la funzione g (l’abbiamo chiamata “DaStringaANumero”). A questo scopo è sufficiente esprimere il numero in base 128, con le “cifre” uguali ai codici ASCII (da 0 a 127) delle lettere del testo da trasformare. Per esempio, alla stringa “abc” associamo il numero (97 98 99) in base 128, cioè il numero:

97·1282 + 98·128 + 99 = 1601891.

L’algoritmo iterativo per trasformare una stringa in un numero naturale consiste nel partire da n = 0, e leggere una alla volta le lettere del testo da convertire e ad ogni passo moltiplicare n per 128 e aggiungere ad n il codice ASCII, la “cifra” , dell’ultima lettera letta della stringa in lettura. Nel nostro esempio i passi per la conversione di “abc” sono:

n = 0

n = 0·128 + 97

n = 97·128 + 98

n = (97·128 + 98)·128 + 99

n = 97·1282+98·128+99

La funzione seguente (la chiameremo “DaNumeroAStringa” ), la funzione g-1, trasforma un numero in un testo sfruttando la funzione chr(r) che trasforma un numero naturale r < 128 nel carattere ASCII corrispondente:

# Trasforma un numero intero in una stringa di caratteri ASCII

s = ''

n = int(input("Inserisci un numero intero da convertire in stringa: "))

while n > 0 :

r = n % 128

c = chr(r)

s = c + s

n = n // 128

print('testo:',s)

Per tornare indietro, dal numero alla stringa s, si parte dal numero 1601891 e ad ogni passo si calcola il resto della divisione del numero n con 128 (n % 128), si trasforma questo resto nel carattere ASCII corrispondente e lo si giustappone alla stringa che si sta creando. Si divide poi lo stesso numero n per 128 e si considera la parte intera della divisione (n = n // 128) . Indicando con [n]128 il resto della divisione di n per 128 questo algoritmo espresso in altro modo diviene:

[1601891]128 = 99 → c

s = c

1601891//128 =12514

[12514]128 = 98 → b

s = bc

12514//128 = 97

[97]128 = 97 → a

s = abc

97//128 = 0

Stop!

Abbiamo così ottenuto le lettere dal numero, a partire dall’ultima lettera a destra del messaggio originale.

**Il Teorema di Fermat**

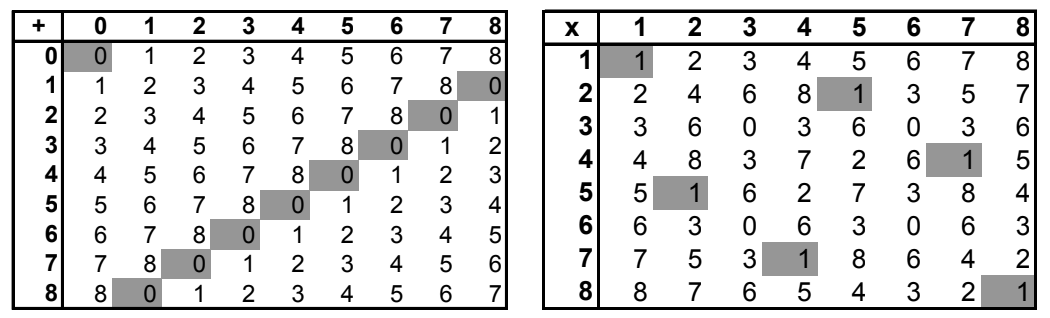
L’aritmetica che serve per implementare l’algoritmo RSA è quella della struttura algebrica **Zn**: ovvero l’insieme delle **classi di resto modulo n**, cioè l’insieme {0, 1, ..., n−1} dei resti delle divisioni di un qualunque numero intero per n. In questo insieme sono definite l’addizione e la moltiplicazione nel seguente modo:

[a+b]n = mod(a+b, n)

[a⋅b]n = mod(a⋅b, n)

dove mod(x,y) è la funzione (implementata in qualsiasi sistema di calcolo) che restituisce il resto del quoziente x/y tra i numeri interi x e y.

La figura seguente illustra le tavole pitagoriche dell’addizione e della moltiplicazione in Z9 (nella tabella del prodotto è stato ignorato lo 0); sono evidenziate le celle che contengono i rispettivi elementi neutri 0 (per l’addizione) e 1 (per la moltiplicazione).



Nella tabella dell’addizione si osserva che la distribuzione degli opposti è estremamente regolare ed è facile formulare una legge generale:

l’opposto di a in Zn è n−a.

Nella tabella della moltiplicazione si osserva innanzitutto che non tutti gli elementi hanno inverso (solo 2, 4, 5, 7 e 8 sono invertibili); come è noto, in Zn ammettono inverso solo i numeri a primi con n, cioè tali che MCD(a, n) = 1; il che significa che se (e solo se) n è primo allora tutti gli elementi (non nulli) di Zn ammettono inverso. In altri termini Zn è un anello per ogni n, ed è un campo se e solo se n è primo.

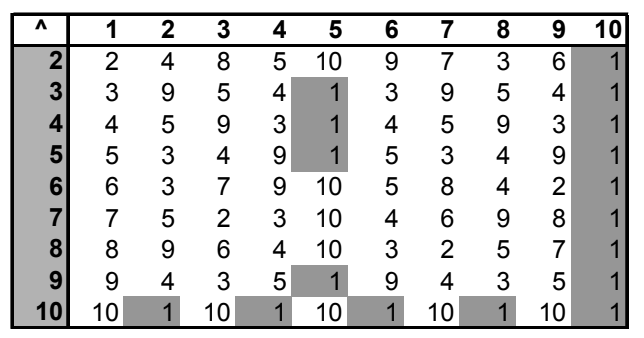
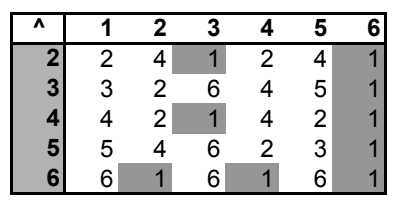
Inoltre la distribuzione degli inversi non mostra una regolarità evidente, a parte la simmetria rispetto alla diagonale principale (quella che và dall’estremo superiore sinistro all’estremo inferiore destro della tabella) e alla diagonale secondaria (quella che và dall’estremo inferiore sinistro all’estremo superiore destro della tabella).

**Domanda**: esiste una legge generale anche per l’inverso di a in Zn? Per esempio, qual è l’inverso di 1234 in Z1789? È possibile calcolarlo direttamente oppure dobbiamo calcolare

1234⋅1, 1234⋅2, 1234⋅3, ...

in Z1789 fino a che il risultato è 1?

La tabella della moltiplicazione non fornisce indizi; proviamo allora con la tabella delle potenze, cioè eleviamo ogni elemento di Zn (escluso 1, le cui potenze sono tutte uguali a 1) alle successive potenze 1, 2, ..., n−1. Ecco la tabella delle potenze in Z7 e in Z11.



Si osservi l’ultima colonna: è composta di soli 1, cioè per ogni a non nullo risulta:

− in Z7: a6 = 1;

− in Z11: a10 = 1.

Formuliamo la seguente congettura:

Per ogni a∈Zn, a≠0, risulta an−1 = 1.

Se così fosse, avremmo trovato una legge generale per l’inverso di a in Zn: l’inverso di a sarebbe an−2, perché

a⋅an−2 = an−1 = 1.

In realtà la congettura è corretta se e solo se n è primo. Questo è infatti quanto afferma il seguente, celebre teorema.

**Teorema di Fermat**

Se e solo se n è primo, per ogni a∈Zn non nullo risulta an−1 = 1.

**Corollario**

Se n è primo, allora l’inverso di a (a≠0) in Zn è an−2.

Dunque l’inverso di 1234 in Z1789 è 12341787 = 303.

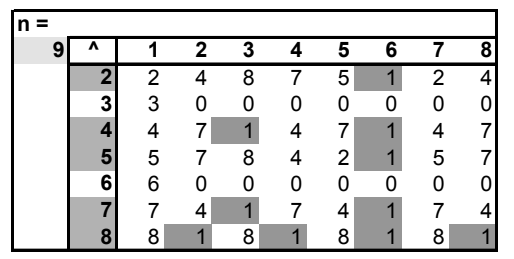
A questo punto è doverosa una domanda: se l’inverso di a in Zn è an−2, calcolare l’inverso di a comporta necessariamente il prodotto di n−2 fattori uguali ad a? Se per calcolare l’inverso di a in Zn dovessimo effettuare un numero di operazioni elementari circa uguale a n, i tempi di calcolo sarebbero proibitivi. Per esempio, se anche ogni prodotto venisse eseguito in 10−12 s, per elevare ad un esponente di 20 cifre (quindi “piccolo” per i calcoli che saranno richiesti da RSA), occorrerebbero 108 s ≈ 3 anni; con un esponente di 30 cifre occorrerebbero 30 miliardi di anni.

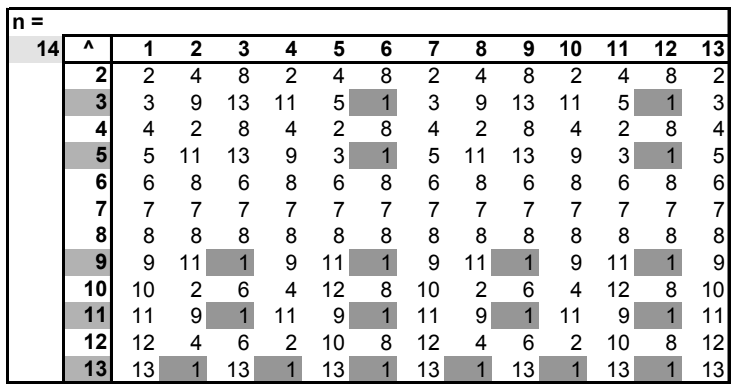
Fortunatamente esistono algoritmi molto efficienti per calcolare la potenza di un numero, il cui tempo di calcolo è lineare rispetto alla lunghezza dell’esponente (vedi in Appendice A l’**algoritmo di Legendre**).

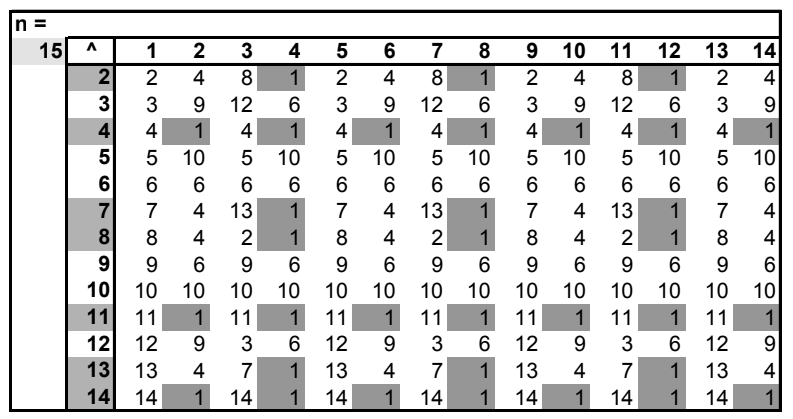
**Il Teorema di Eulero**

Se il teorema di Fermat risolve il problema del calcolo dell’inverso di a in Zn quando n è primo, rimane aperto il problema nel caso n non sia primo. Occorrerà aspettare più di un secolo perché questo problema venga risolto da Eulero.

Diamo ancora un’occhiata alla tabella delle potenze degli elementi di Zn, con n composto, per scoprire qualche eventuale regolarità. Le figure seguenti mostrano le potenze in Z9, Z14, Z15.







Nella prima colonna sono evidenziati gli elementi di Zn che ammettono inverso: si osservi che, per tutti gli elementi invertibili, risulta:

− in Z9: a6 = 1

− in Z14: a6 = 1 (e quindi a12 = (a6)2 = 1)

− in Z15: a4 = 1 (e quindi a8 = (a4)2 = 1 e a12 = (a4)3 = 1)

Sembra che anche in Zn, con n composto, esista un esponente (6 in Z9 e in Z14, 4 in Z15), tale che ogni elemento invertibile elevato a quell’esponente sia uguale a 1. Si tratta di capire qual è la relazione tra questo esponente e n. La risposta è nel seguente, altrettanto celebre, teorema, che generalizza il teorema di Fermat.

**Teorema di Eulero**

Per ogni n ≥ 2 e per ogni a∈Zn invertibile cioè MCD(a, n) = 1, risulta

aφ(n)= 1,

dove φ(n) è il numero di naturali compresi tra 1 e n che sono primi con n.

**Corollario**

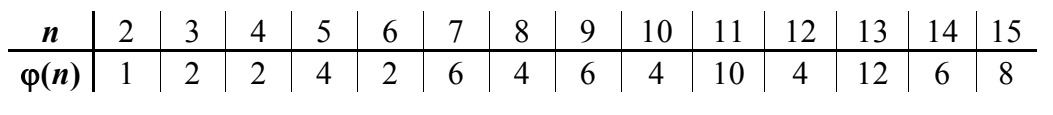
Per ogni n ≥ 2 e per ogni a∈Zn invertibile, cioè MCD(a, n) = 1, l’inverso di a è

aφ(n)−1.

La funzione φ(n) è chiamata **funzione di Eulero** ed associa ad ogni numero naturale n il numero di numeri a∈{1, 2, ..., n} tali che MCD(a, n) = 1.

Per esempio, φ(10) = 4, poiché i numeri di Z10 primi con 10 sono 1, 3, 7, 9.

La tabella elenca i valori della funzione di Eulero per n = 2, ..., 15.

Se n è primo allora φ(n) = n−1, e dunque il teorema di Fermat è un caso particolare del teorema di Eulero.

Per la funzione φ è possibile dimostrare le seguenti proprietà:

− Se MCD(a, b) = 1 allora φ(ab) = φ(a)φ(b).

Per esempio: φ(72) = φ(8)⋅φ(9) = 4⋅6 = 24.

− Se p è primo allora φ(pm) = pm−1(p−1).

Per esempio: φ(2401) = φ(74) = 73⋅6 = 343⋅6 = 2058.

In particolare, se p e q sono due numeri primi distinti, allora φ(pq) = (p−1) · (q−1). Per esempio φ(15) = φ(3)φ(5) = 2⋅4 = 8. Questa sarà una proprietà fondamentale che utilizzeremo nell’algoritmo RSA.

Noto il valore di φ(n), possiamo calcolare l’inverso di qualsiasi elemento invertibile di Zn. Per esempio, poiché 1789 è primo, risulta MCD(1789, 2401) = 1, e dunque 1789 è invertibile in Z2401. Poiché φ(2401) = 2058, l’inverso di 1789 è 17892057 = 51 in Z2401.

Mettiamo subito in evidenza che il calcolo di φ(n) per numeri grandi non è realizzabile in tempi ragionevoli. Le proprietà della funzione φ mostrano che il calcolo di φ(n) è immediato se si conosce la fattorizzazione di n; ma la fattorizzazione di n è esattamente ciò che verrà tenuto nascosto e che costituisce il motivo della inattaccabilità di RSA; le difficoltà di calcolare φ(n) e di fattorizzare n sono equivalenti. Infatti:

− dalla fattorizzazione di n, n = pq si ricava immediatamente φ(n):

φ(n) = (p−1)(q−1);

− dalla conoscenza di φ(n) e n si ricava la fattorizzazione di n:

φ(n) = (p−1)(q−1) = pq − (p+q) + 1 = n − (p+q) + 1

da cui p+q= n − φ(n) + 1;

quindi di p e q si conosce il prodotto pq = n e la somma p+q = n−φ(n)+1 ed è immediato calcolare p e q, che sono le soluzioni dell’equazione di secondo grado

x2 − (p+q)x + pq = 0.

In effetti i tempi di calcolo di φ(n) sono dello stesso ordine di grandezza della fattorizzazione di n.

**L’algoritmo di Euclide per l’inverso di a in Zn**

Fortunatamente non è necessario conoscere φ(n) per calcolare l’inverso di a in Zn: un’opportuna applicazione dell’algoritmo di Euclide ci consente di effettuare il calcolo dell’inverso di a con strumenti del tutto differenti e in tempi rapidi.

Come è noto, l’algoritmo di Euclide è un algoritmo molto efficiente per il calcolo del MCD tra due numeri naturali n e a. Consiste nel calcolare la sequenza strettamente decrescente dei resti delle divisioni a partire dal numero n ed a, nel calcolare poi la divisione tra a e il resto r1 della divisione precedente per ottenere il resto r2, nel dividere poi il resto r1 per il resto r2 così da ottenere un resto r3, nel dividere r2 con r3 e così di seguito, dividendo i resti successivi tra loro nell’ordine decrescente, si ottiene alla fine come ultimo resto lo 0. Possiamo esprimere l’algoritmo di Euclide come segue:

x0 = n

x1 = a

x2 = [x0]x1 ([x0]x1 è il resto della divisione di x0 per x1)

x3 = [x1]x2 ([x1]x2 è il resto della divisione di x1 per x2)

x4 = [x2]x3 ([x2]x3 è il resto della divisione di x2 per x3) **(I)**

.

.

.

xi = [xi−2] xi−1

fino a che, per un certo i, risulta xi = 0. Allora xi−1 è l’MCD(n, a).

L’algoritmo di Euclide, letto “al contrario”, consente altresì di esprimere il numero MCD(n,a) come combinazione lineare di n e a, consente cioè di trovare due numeri interi s e t tali che

sn+ta = MCD(n, a).

Di seguito vediamo un esempio. Calcoliamo MCD(1789, 1234) utilizzando l’algoritmo di Euclide. Indichiamo in grassetto i quozienti delle divisioni

x0 = 1789

x1 = 1234

1789 = **1**⋅1234 + 555 → x2 = [1789]1234 = 555

1234 = **2**⋅555 + 124 → x3 = [1234]555 = 124

555 = **4**⋅124 + 59 → x4 = [555]124 = 59

124 = **2**⋅59 + 6 → x5 = [124]59 = 6

59 = **9**⋅6 + 5 → x6 = [59]6 = 5

6 = **1**⋅5 + 1 → x7 = [6]5 = **1 = MCD(1789, 1234)**

5 = **5**⋅1 + 0 → x8 = [5]1 = 0

Stop!

Rappresentiamo in una tabella quozienti qi e resti xi delle divisioni successive indicate sopra. L’indice i conformemente a quanto avviene in python parte da 0.

| *i* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *xi* | 1789 | 1234 | 555 | 124 | 59 | 6 | 5 | 1 | 0 |
| ***qi*** | **1** | **2** | **4** | **2** | **9** | **1** | **5** |  |  |

Risulta: x7 = MCD(1789, 1234) = 1 in quanto l’MCD(x0, x1) è l’ultimo resto non nullo della catena di divisioni (1 nel nostro caso). A partire dal resto x7 = MCD = 1 possiamo esprimere ciascun resto xj come combinazione lineare dei due resti precedenti xj−2 e xj−1 e grazie a ciò in ultimo possiamo esprimere l’MCD = 1 come una combinazione lineare dei numeri dati: x0 = 1789 e x1 = 1234. Vediamo i passaggi.

A partire dai calcoli precedenti esprimiamo ogni resto come combinazione lineare dei due resti precedenti:

1789 = **1**⋅1234 + 555 → 555 = 1789−**1**⋅1234

1234 = **2**⋅555 + 124 → 124 = 1234−**2**⋅555

555 = **4**⋅124 + 59 → 59 = 555−**4**⋅124

124 = **2**⋅59 + 6 → 6 = 124−**2**⋅59

59 = **9**⋅6 + 5 → 5 = 59−**9**⋅6

6 = **1**⋅5 + 1 → 1 = 6−**1**⋅5

Ora, procedendo a ritroso, sostituiamo nell’ultima uguaglianza al posto del resto 5 il risultato della uguaglianza precedente, così da ottenere l’MCD = 1 come combinazione lineare dei due resti precedenti a 5 ovvero 6 e 59.

1 = 6−**1**⋅5 =

= 6−**1**⋅(59−**9**⋅6) = −**1**⋅59 + [1 -(-**1)**(**9**)]⋅6 = −1⋅59+10⋅6

Procedendo a ritroso, sostituendo al posto di 6 la sua combinazione lineare attraverso i due resti precedenti 59 e 124 si ha:

1 = −1⋅59+10⋅6 =

1 = −1⋅59+10⋅(124−**2**⋅59) = 10⋅124 + [-1 - 10(**2**)]⋅59 = 10⋅124−21⋅59

così da ottenere l’MCD = 1 come combinazione lineare dei due resti precedenti a 6 ovvero 59 e 124. Ripetendo il procedimento di seguito con sostituzioni analoghe, alla fine si ottiene l’MCD = 1 come combinazione lineare dei numeri dati 1789 e 1234. Ecco il riepilogo dei passaggi:

1 = 6−**1**⋅5 =

= 6−**1**⋅(59−**9**⋅6) = −**1**⋅59 + [1 -(-**1)**(**9**)]⋅6 = −1⋅59+10⋅6

= −1⋅59+10⋅(124−**2**⋅59) = 10⋅124 + [-1 - 10(**2**)]⋅59 = 10⋅124−21⋅59

= 10⋅124−21⋅(555−**4**⋅124) = −21⋅555 + [10 - (-21)(**4)**]⋅124 = −21⋅555+94⋅124 =

= −21⋅555+94⋅(1234−**2**⋅555) = 94⋅1234 + [-21 - 94(**2**)] ⋅555 = 94⋅1234−209⋅555 =

= 94⋅1234−209⋅(1789−**1**⋅1234)=−209⋅1789+[94-(-209)(**1**)]⋅1234=

= −209⋅1789+303⋅1234

Nota: Scopriamo che la legge di formazione dei coefficienti è la seguente: se [s, t] sono i coefficienti della combinazione lineare dell’MCD in funzione dei resti xi e xi+1 allora [t, s−t⋅qi-1] sono i coefficienti della combinazione lineare dell’MCD in funzione dei resti xi-1 e xi.

Ricapitolando abbiamo ottenuto:

1 = −209⋅1789+303⋅1234

da cui si ottiene:

303⋅1234 = 1+209⋅1789

Ora ‘leggendo’ questa combinazione lineare in Z1789 risulta:

[303⋅1234]1789 = 1

cioè il resto della divisione di 303⋅1234 per 1789 è uguale a 1, il che vuol dire che 303 è l’inverso di 1234 in Z1789.

Riassumendo: in Zn (sia che n sia primo o composto), a è invertibile se e solo se MCD(n, a) = 1. Utilizzando l’algoritmo di Euclide possiamo esprimere 1 come combinazione lineare di n e a:

sn+ta = 1

da cui

[ta]n = 1

ossia il resto della divisione di ta per n è 1, cioè t è l’inverso di a in Zn.

**L’algoritmo RSA: dalla parte della Certification Authority**

Abbiamo tutto quello che ci serve per descrivere l’algoritmo di crittazione RSA. Articoliamo la descrizione in due fasi: la scelta delle chiavi (svolta dalla CA) e l’algoritmo di codifica-decodifica, svolto dagli utenti.

La scelta delle chiavi (pubblica e privata) da assegnare all’utente A da parte della Certification Authority si fonda sulla possibilità di procurarsi due numeri primi p e q “grandi”; oggi l’ordine di grandezza standard di p e q è circa 10300. Come abbiamo visto, questo non è un problema dal punto di vista computazionale.

Dunque la CA:

− si procura due numeri primi p e q;

− calcola il loro prodotto n = pq;

− calcola φ(n) = (p−1)(q−1); il numero n viene reso pubblico, ma non la sua fattorizzazione pq, né φ(n).

Per ciascun utente si costruiscono ora due chiavi, quella pubblica H e quella privata K. Una chiave, per esempio quella pubblica, è un qualunque numero H minore di φ(n) e primo con φ(n) (e quindi invertibile in Zφ(n)). L’altra chiave K, quella privata, è l’inverso di H in Zφ(n).

Svolgiamo un esempio con numeri “piccoli”: partiamo dai numeri primi:

p = 1997739811 e q = 3532090361; risulta

n = p⋅q = 7056197530219061771 e φ(n) = (p-1)⋅(q-1) = 7056197524689231600

Prendiamo come H un qualsiasi numero primo con φ(n), per esempio il numero primo

H = 1093531

Ora calcoliamo K = H−1 in Zφ(n). Applicando direttamente il teorema di Eulero per calcolare K dobbiamo elevare H a φ(φ(n))−1 in Zφ(n) e risulta:

K =[ Hφ(φ(n))−1 ] φ(n) = 6502603384024117171

Noi abbiamo utilizzato invece il programma **Inverso** (descritto in appendice B )che si basa sull’algoritmo di Euclide.

La prima fase è terminata. La CA consegna all’utente A:

− la chiave H, che viene resa pubblica insieme al valore n ;

− la chiave privata K, che A tiene nascosta.

Un estraneo conosce dunque H e n, ma non conosce K, né φ(n). Se vuole conoscere K (e questo, come vedremo, è l’unico modo di decodificare il messaggio) deve fattorizzare n, impresa impossibile in tempi ragionevoli se p e q sono numeri primi “grandi”.

**L’algoritmo RSA: dalla parte dell’utente**

La proprietà fondamentale che lega H, K, n (e che costituisce il cuore dell’algoritmo RSA) è la seguente: per ogni x∈Zn risulta

[(xH ) K ] n= x

In altri termini, se MCD(H, φ(n)) = 1, cioè se H è invertibile in Zφ(n) allora la funzione

ƒ(x) = xH

è invertibile in Zn, e la sua funzione inversa è

ƒ−1(x) = xK,

dove [HK]φ(n) = 1.

La proprietà è facile da dimostrare se x è primo con n; infatti se HK = 1 in Zφ(n) allora in Zn risulta

( xH) K = xHK =(x )t φ(n) + 1 = (x) t φ(n) x =(xφ(n))t x

ma per il teorema di Eulero xφ(n) = 1, dunque

(xφ(n))t x = 1t x = x

Se x non è primo con n la dimostrazione è solo un po’ più laboriosa. Bene, abbiamo trovato le due funzioni ƒ (facile da calcolare) e ƒ−1 (praticamente impossibile da calcolare) che ci servono per l’algoritmo. La funzione ƒ è la potenza con esponente H in Zn, la funzione inversa ƒ−1 è la potenza con esponente K (l’inverso di H in Zφ(n)) in Zn. H è noto a tutti e quindi chiunque può mandare un messaggio segreto ad A, ma solo A può decodificarlo perché solo lui conosce la chiave segreta K.

Riassumendo, le quattro funzioni g, g−1, ƒ, ƒ−1 per l’algoritmo RSA sono le seguenti:

− g(T) è la funzione numero;

− g−1(N) è la funzione testo;

− ƒ(N) è la funzione xH in Zn, dove H è la chiave pubblica dell’utente;

− ƒ−1(M) è la funzione xK in Zn, dove K è l’inverso di H in Z(p−1)(q−1) ed è la chiave privata dell’utente.

Vediamo l’algoritmo al lavoro utilizzando le chiavi H e K precedenti. Sup-

poniamo che B voglia mandare il messaggio segreto

T := “tvtb”

ad A. B legge sull’elenco pubblico la chiave pubblica H di A e il valore n. Poi calcola

ƒ(g(T)).

Risulta

N = g(T) = DaStringaANumero(“tvtb”) = 245217890

occorre assicurarsi che N risulti minore di n,

B codifica il numero come un numero cifrato M di Zn

M = ƒ(N) = [NH]n = 61066390659944073

Il programma che svolge questo compito si chiama **Potenza** descritto in appendice A

B manda dunque ad A il messaggio cifrato M. A riceve M e calcola

N = ƒ−1(M) = [MK]n = 245217890

Noi useremo sempre il programma Potenza

e infine trasforma il messaggio in lettere con la funzione DaNumeroAStringa

T = DaNumeroAStringa(N) = tvtb

finalmente A legge il messaggio decodificato “tvtb”.

Ovviamente, con p e q così piccoli, la segretezza è facilmente scardinabile: si fattorizza n e si ricavano p e q, e di conseguenza φ(n) = (p-1)(q-1). Poiché H è noto, è immediato calcolare K = H−1 in Zφ(n).

Per garantire la segretezza la CA genera numeri n grandi come prodotto di due numeri primi di 300 cifre. Per generare numeri primi di tale grandezza utilizzeremo il programma GeneraNumeriPrimi descritto in precedenza.

Un aspetto importante che non abbiamo ancora detto è che occorre assicurarsi che risulti

DaStringaANumero(T) < n.

Infatti, poiché tutte le operazioni vengono svolte in Zn, per garantire la biunivocità della corrispondenza testo-numero occorre che il numero corrispondente ad un messaggio sia minore di n; altrimenti, poco male: è sufficiente suddividere il messaggio in più sottomessaggi. A questo scopo, con n ≈ 10600 e con un alfabeto di 128 caratteri, è sufficiente che il messaggio abbia un numero di caratteri (spazi compresi) minore di log128(10600)≈ 285 caratteri.

**Segretezza e firma digitale**

Il meccanismo che abbiamo appena illustrato permette di ottenere la segretezza di un messaggio: nessun malintenzionato può decifrare il messaggio mandato da B ad A. Ma questo non è l’unico scopo della crittografia. Per esempio, quando A riceve il messaggio è sicuro che nessun altro tranne lui può averlo letto, ma non può essere sicuro che sia stato proprio B a mandarlo. Di norma il messaggio codificato è accompagnato da un messaggio in chiaro che ne illustra lo scopo e dichiara il mittente. Un malintenzionato C potrebbe scrivere alla banca A, dichiarando di essere B, e di voler bonificare una somma di denaro a C. In questo caso ciò che importa non è tanto la segretezza del messaggio (l’intenzione di B bonificare una certa somma a C) quanto l’autenticità del mittente.

I sistemi a chiave pubblica consentono di risolvere questo problema: un utente può apporre, al messaggio codificato che manda, la cosiddetta firma digitale. Se B vuole mandare ad A un messaggio di cui non gli importa tanto la segretezza (chiunque lo può leggere), quanto il fatto che A sia certo che solo B può averglielo mandato, basta modificare l’ordine di codifica; vediamo come.

Anche B chiede alla Certification Authority una chiave pubblica HB e una chiave privata KB e il corrispondente numero nB. Codifica il testo T con la propria chiave privata KB e lo manda ad A. A legge nel messaggio in chiaro chi è il mittente. Si procura la chiave pubblica HB di B e decodifica il testo.

A è sicuro che solo B può avergli mandato quel testo, poiché è l’unico che possiede la chiave privata KB. Naturalmente il testo mandato da B ad A può essere decodificato da chiunque (poiché serve solo la chiave pubblica di B), ma lo scopo non era in questo caso la segretezza ma la firma digitale da parte di B.

È possibile ottenere sia la segretezza sia la firma digitale? Sì, la simme-

tria delle funzioni ƒ e ƒ−1 consente a B di mandare un messaggio ad A con le seguenti caratteristiche:

− solo A può leggerlo: il messaggio è segreto;

− solo B può averlo mandato: il messaggio possiede la firma digitale.

Funziona in questo modo: B codifica il messaggio prima con la propria chiave privata, e poi codifica il risultato con la chiave pubblica di A. A riceve il messaggio, lo decodifica con la propria chiave privata (solo A può farlo, il messaggio è segreto) e poi decodifica il risultato con la chiave pubblica di B (solo B può averlo mandato, il messaggio ha la firma digitale).

In questo modo la comunicazione tra A e B è perfettamente tutelata da occhi indiscreti ed è sempre garantita l’identità sia del mittente sia del destinatario.

**Conclusioni**

Il funzionamento dell’algoritmo RSA è sancito, come abbiamo visto, da una rete di proprietà aritmetiche e costituisce dunque un notevole successo della matematica, proprio in un settore, l’aritmetica, tradizionalmente lontano dalle applicazioni. A questo proposito, solo nel 1940 Godfrey Hardy, nel suo Apologia di un matematico, scriveva:

*“È giusto che Gauss e anche gli altri matematici minori si rallegrino per almeno una scienza, e proprio la loro [la Teoria dei Numeri], che il distacco stesso dalle contingenze umane conserva benigna e pulita”. Anche per questo motivo l’algoritmo RSA rappresenta un grande successo dell’aritmetica: è paradossale, tuttavia, che la segretezza che l’algoritmo RSA promette (e mantiene) sia garantita proprio da una sconfitta della matematica: il non saper fattorizzare in tempi ragionevoli il prodotto di due numeri primi grandi*.

**APPENDICE A**

**L’algoritmo di Legendre**

**Premessa**

Prima di mostrare in dettaglio l’algoritmo di Legendre per il calcolo delle potenze descriviamo l’algoritmo per la trasformazione di un numero in notazione binaria, ovvero per la riscrittura dello stesso numero nella forma:

2ncn + 2n-1cn-1 + 2n-2cn-2 + …+23c3 +22c2 + 21c1 + 20c0

dove ci rappresenta una cifra binaria che può valere 0 oppure 1, mentre 2i diciamo che è il suo peso. Vediamo il tutto con un esempio, assumiamo di voler scrivere in notazione binaria il numero 100.

100 = 26∙1+ 25∙1+ 24∙0+ 23∙0+ 22∙1+ 21∙0 + 20∙0

come è facile verificare. In notazione binaria il numero 100 lo indicheremo in questo modo:

(1 10010 0)2

Con la notazione che usiamo normalmente, quella decimale, senza mettere le parentesi e la base numerica 10 il numero **100** vuol dire:

102∙**1** + 101∙**0** + 100∙**0**

Ma ritorniamo al numero 100 che in notazione binaria è: 1100100. Questo numero si ricava facilmente mediante la seguente procedura: dividiamo il numero 100 per 2 , quindi dividiamo il quoziente così ottenuto ancora per due e così di seguito dividiamo i quozienti per 2 fino ad ottenere come ultimo quoziente 0, in tale processo si considerano i resti ottenuti delle divisioni che sono proprio le cifre di 100 in notazione binaria a partire da quella destra (quella di peso minore) e via via quelle di peso maggiore. Di seguito sviluppiamo lo svolgimento della suddetta procedura.

100 = 2∙**50** + 0

**50** = 2∙**25** + 0

**25** = 2∙**12** + 1

**12** = 2∙**6** + 0

**6** = 2∙**3** + 0

**3**  = 2∙**1** + 1

**1** = 2∙ **0** + 1

Giustifichiamo ora la procedura appena esposta per il calcolo delle cifre binarie di 100. Ora

100 = 2ncn + 2n-1cn-1 + 2n-2cn-2 + … +23c3 +22c2 + 21c1 + 20c0

mettendo in evidenza 2 si ottiene:

100 = 2∙**(... + 25∙c6 +24∙c5 + 23∙c4 + 22∙c3 +2∙c2 + c1 )**+ c0

ma 100 diviso 2 ha quoziente **50** e resto 0 cioè: 100 = 2∙**50** + 0

da cui si deduce:

**50 = … + 25∙c6 +24∙c5 + 23∙c4 + 22∙c3 +2∙c2 + c**1 e c0 = 0.

Da

50 = … + 25∙c6 +24∙c5 + 23∙c4 + 22∙c3 +2∙c2 + c1

mettendo in evidenza 2 si ottiene:

50 = 2∙**(... + 24∙c6 +23∙c5 + 22∙c4 + 2∙c3 +c2)**+ c1

ma 50 diviso 2 ha quoziente **25** e resto 0 , cioè: 50 = 2∙**25** + 0

da cui si deduce:

**25 = … + 24∙c6 +23∙c5 + 22∙c4 + 2∙c3 +c2** e c1 = 0

e così di seguito,

**25** = 2∙**12** + 1 → c2 = 1

**12** = 2∙**6** + 0 → c3 = 0

**6**  = 2∙**3** + 0 → c4 = 0

**3** = 2∙**1** + 1 → c5 = 1

**1** = 2∙**0** + 1 → c6 = 1

**0** = 2∙**0** + 0 → c7 = 0

Stop!

cn = 0 per n

Quindi in conclusione

100 = (c6 c5 c4 c3 c2 c1 c0 )2 = (1 1 0 0 1 0 0)2

Terminata la premessa sulla trasformazione binaria di un numero riprendiamo l’argomento di questa appendice, ovvero l’algoritmo di Legendre.

Come abbiamo visto, l’algoritmo RSA prevede che si sappia calcolare rapidamente una potenza ab in Zn, con b ed n numeri grandi. Esiste un algoritmo, noto come algoritmo di Legendre, che consente di effettuare un numero di operazioni elementari che è lineare rispetto al numero di cifre di b, cioè è lineare rispetto alla lunghezza dell’input.

Per capirne il funzionamento, supponiamo per semplicità che b sia una potenza di 2, b = 2c. Per calcolare ab sarebbero necessari allora non b prodotti, ma c elevamenti al quadrato, cioè c prodotti:

ab = = =

e quindi il numero di operazioni elementari sarebbe c = log2(b). Con

b ≈ 10600 ≈ 22000,

dovremmo calcolare soltanto 2000 operazioni elementari; se anche ciascuna richiedesse ben 10−6 s, il tempo di calcolo sarebbe due millesimi di secondo.

Se b non è una potenza di 2, come accade in generale, è sufficiente un piccolo aggiustamento; vediamo ad esempio come calcolare a100. Il numero 100 in base 2 si scrive:

1100100

si pone inizialmente la variabile z (che conterrà il risultato) uguale a 1 e la variabile x uguale ad a. Si scorrono le cifre di 1100100 a partire dall’ultima a destra: se la cifra è 0 si eleva x al quadrato e si mette il risultato in x; se la cifra è 1 si moltiplica z per x e si mette il risultato in z poi si eleva x al quadrato e si mette il risultato in x . Se poi il calcolo avviene in Zn, avremo cura di ridurre il risultato modulo n ad ogni moltiplicazione, ovvero di calcolare il resto della divisione per n dei numeri x e z calcolati. Mostriamo la validità dell’algoritmo di Legendre prendendo come esempio il calcolo di a100 .

a100 = (a2)50 = (a4)25 = (a4)2∙12+1 = (a4)∙(a8)12 = (a4)∙(a16)6 =

= (a4)∙(a32)3 = (a4)∙(a32)2+1 = (a4)∙(a32)∙(a64) = (a36)∙(a64) = a100

di seguito i valori di z, x , c dove le c sono le cifre binarie c di 100 in ordine inverso

| z | 1 | 1 | 1 | 1∙a4 | a4 | a4 | a4∙a32=a36 | a36∙a64 = a100 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x | a | a2 | a4 | a8 | a16 | a32 | a64 | a128 |
| c |  | 0 | 0 | 1 | 0 | 0 | 1 | 1 |

Nel linguaggio python l’algoritmo di Legendre è così implementato nel programma **Potenza**:

# Calcola la potenza a^h in Zn

a = int(input("Inserisci la base a: "))

h = int(input("Inserisci l'esponente h: "))

n = int(input("Inserisci n (di Zn): "))

x=a

y=h

z=1

while y>0:

if y%2==1:

z=z\*x

z=z%n

y=y-1

x=x\*x

x=x%n

y=y//2

print(a,'^',h,'=',z,'(modulo',n,')')

Riportiamo di seguito i valori dei parametri x, y, z nel caso esaminato in precedenza del calcolo di a100 per ogni ciclo while.

| z | 1 | 1 | 1 | a4 | a4 | a4 | a4∙a32=a36 | a36∙a64 = a100 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| x | a | a2 | a4 | a8 | a16 | a32 | a64 | a128 |
| y | 100 | 50 | 25 | 12 | 6 | 3 | 1 | 0 |

**APPENDICE B**

**L’inverso di a in Zn con l’algoritmo euclideo**

Riprendiamo l’esempio del calcolo dell’inverso di 1234 in Z1789. Ci procuriamo innanzitutto la sequenza dei resti x0, ..., x8 e dei quozienti q0, ..., q6:

| *i* | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *xi* | 1789 | 1234 | 555 | 124 | 59 | 6 | 5 | 1 | 0 |
| *qi* | 1 | 2 | 4 | 2 | 9 | 1 | 5 |  |  |

Ora analizziamo il modo in cui l’MCD(1789, 1234) = x7 = 1 si esprime come combinazione lineare di x6 e x5, di x5 e x4, e così via fino ad esprimere x7 = 1 come combinazione lineare di x0 = 1789 e x2 = 1234 (nella colonna a destra abbiamo messo in evidenza le coppie dei coefficienti della combinazione cosi come ottenuti precedentemente nel paragrafo sull’algoritmo di Euclide, fatta eccezione per la prima coppia [0, 1]).

1 = **0**⋅5+**1**⋅1 [0, 1]

= **1**⋅6−**1**⋅5 [1, −1]

= **−1**⋅59+**10**⋅6 [−1, 10]

= **10**⋅124**−21**⋅59 [10, − 21]

= **− 21**⋅555+**94**⋅124 [−21, 94]

= **94**⋅1234**−209**⋅555 [94, −209]

= **−209**⋅1789+**303**⋅1234 [−209, 303]

Precedentemente abbiamo scoperto che la legge di formazione dei coefficienti è la seguente: se [s, t] sono i coefficienti della combinazione lineare dell’MCD = 1 in funzione di xi e xi+1 allora [t, s−t⋅qi-1] sono i coefficienti della combinazione lineare dell’MCD = 1 in funzione di xi-1 e xi. Se si parte dall' ultimo resto non nullo dell’algoritmo di Euclide, x7 = 1, si inizia con la coppia di coefficienti [0,1].

Di seguito il programma **Inverso** in python che implementa l’algoritmo su descritto per la ricerca dell’inverso di un numero a in Zn. Nella funzione viene eseguito un controllo sul fatto che risulti MCD(a, n) = 1, in modo che a sia effettivamente invertibile in Zn.

# Determina l'inverso di a in Zn se MCD(n,a)=1

lx=[]

lq=[]

lt=[]

n = int(input("Inserisci il numero n: "))

a = int(input("Inserisci il numero da invertire a < n in Zn, a: "))

lx.append(n)

lx.append(a)

r=1

i=1

q=1

while (r!=0) :

q=lx[i-1]//lx[i]

lq.append(q)

r=lx[i-1]%lx[i]

lx.append(r)

i=i+1

# viene stampato l'MCD(a,n)

if lx[i-1] != 1:

print("MCD(",n, ",",a,")=",lx[i-1])

print('Inverso non calcolabile con questa routine')

quit

for m in range(i-1):

lt.append(m)

lt[m]=1

lt[m-1]=-lq[m-1]

lt[m-1]=lt[m-1]%n

m=m-2

while (m>=0):

lt[m]=-lq[m]\*lt[m+1]+lt[m+2]

lt[m]=lt[m]%n

m=m-1

# lt[0] è l'inverso di a in Zn se MCD(n,a)=1

if lx[i-1] == 1:

print("l'inverso di",a, 'in Z',n,'è:',lt[0])

quit

Con questo algoritmo i tempi di calcolo sono molto ridotti. In effetti si può dimostrare che il numero di operazioni elementari di questo algoritmo è lineare rispetto alla lunghezza di n.

Descriviamo ora i passaggi del programma **Inverso**.

Il simbolo cancelletto # precede un commento utile per chiarire il fine di un insieme di istruzioni, con

# Determina l'inverso di a in Zn se MCD(n,a)=1

viene indicato lo scopo del programma. Con

lx=[]

lq=[]

lt=[]

vengono creati tre vettori vuoti: lx, lq, lt, l è l’iniziale di “lista” sinonimo di “vettore” in python. Con:

n = int(input("Inserisci il numero n: "))

a = int(input("Inserisci il numero da invertire a < n in Zn, a: "))

lx.append(n)

lx.append(a)

vengono richiesti e caricati n ed a di seguito nel vettore lx che ora è divenuto [n, a], nel nostro esempio lx = [1789, 1234]. con:

r=1

i=1

q=1

Vengono inizializzate a 1 tre variabili r, i, q che stanno per resto, indice, quoziente. Con:

while (r!=0) :

q=lx[i-1]//lx[i]

lq.append(q)

r=lx[i-1]%lx[i]

lx.append(r)

i=i+1

nel ciclo while (finché) viene ripetuto il codice indentato finché il resto r della divisione tra lx[i-1] e lx[i] è diverso da zero, quì vengono calcolati il quoziente q della divisione lx[i-1] // lx[i] e il resto r della divisione lx[i-1] % lx[i] e questi valori vengono inseriti di seguito rispettivamente nei vettori lx e lq; di seguito viene incrementato l’indice i dell’elemento dei vettori lx e lq. In python si assume che l’indice del primo elemento di un vettore è 0. In notazione matematica convenzionale il ciclo while svolge l’algoritmo di Euclide indicato precedentemente in (I).

Si riporta qui di seguito per comodità nella notazione matematica l’algoritmo di Euclide da cui ricaviamo poi procedendo nel senso contrario l’algoritmo per il calcolo dell’inverso di a in Zn.

Posto:

x0 = n

x1 = a

seguono le definizioni di q e x successive come quoziente e resto della divisione dei due precedenti valori dei resti x, finché l’ultimo resto x diviene 0.

x0 = q0x1 + x2

x1 = q1x2 + x3

x2 = q2x3 + x4

.

. **(I)**

.

xi-4 = qi-4xi-3 + xi-2

xi-3 = qi-3xi-2 + xi-1

xi-2 = qi-2xi-1 + xi

Ci si ferma quando xi = 0, da cui si ricava: MCD(n, a) = xi-1.

Nel nostro esempio dopo l’esecuzione del ciclo while il vettore lx è: [1789, 1234, 555, 124, 59, 6, 5, 1, 0], il vettore lq è: [1,2,4,2,9,1,5], ed i=8. Detti valori sono quelli che abbiamo visto in precedenza nella tabella con le x e le q, riportata per comodità qui sotto.

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| xi | 1789 | 1234 | 555 | 124 | 59 | 6 | 5 | 1 | 0 |
| **qi** | **1** | **2** | **4** | **2** | **9** | **1** | **5** |  |  |

Terminato il ciclo while, se l’MCD(a, n) che è l’ultimo resto non nullo della catena di divisioni tra i resti successivi è un numero diverso da 1 allora questo viene stampato e si segnala che l’inverso di a in Zn non è calcolabile e il programma termina. Questo è ciò che fa la seguente parte del programma:

# viene stampato l'MCD(a,n)

if lx[i-1] != 1:

print("MCD(",n, ",",a,")=",lx[i-1])

print('Inverso non calcolabile con questa routine')

quit

Se l’MCD(n, a) = 1 il programma prosegue con il ciclo for (per):

for m in range(i-1):

lt.append(m)

viene creato il vettore lt con un numero i-1 di componenti (gli indici vanno da 0 a i-2, il range nel nostro esempio è da 0 a 6 ) e alla fine m = 6. Con:

lt[m]=1

lt[m-1]=-lq[m-1]

lt[m-1]=lt[m-1]%n

m=m-2

si inizializza l’ultimo e il penultimo elemento di lt, nel nostro esempio risulta:

lt = [ , , , , ,1788 ,1] ed m = 5 alla fine.

Alla fine del ciclo:

while (m>=0):

lt[m]=-lq[m]\*lt[m+1]+lt[m+2]

lt[m]=lt[m]%n

m=m-1

il vettore lt conterrà i coefficienti dell’algoritmo inverso di Euclide modulo n ovvero i valori lt =[ 303 , 1580 , 94 , 1768 , 10 , 1788 ,1] che consentiranno di esprimere l’MCD(n, a) = 1 come una combinazione lineare di n ed a da cui poi ricavare l’inverso di a = 1234 in Zn = Z1789 che è proprio lt[0] = 303. Per capire la validità del codice nel ciclo while rivediamo ora l’algoritmo inverso di Euclide in dettaglio in notazione matematica consueta, ottenuto con riferimento alla **(I)** dove abbiamo usato xj+1= - qj-1xj + xj-1 e dove abbiamo messo come coefficienti della combinazione lineare dei resti tj+1xj + tjxj+1 gli elementi tj+1 e tj di un vettore t ottenuto con le sostituzioni che sono indicate con colori uguali in passaggi contigui.

1 = MCD(n, a) = xi-1

1 = xi-1

= 1xi-3 - qi-3xi-2

= ti-2 xi-3 + ti-3xi-2

= ti-2 xi-3 + ti-3( - qi-4xi-3 + xi-4 )

= ti-3xi-4 + (- qi-4ti-3+ti-2) xi-3

= ti-3xi-4 + ti-4xi-3

= ti-3xi-4 + ti-4(-qi-5xi-4+xi-5)

= ti-4xi-5 + ( -qi-5ti-4+ti-3)xi-4

= ti-4xi-5 + ti-5xi-4

.

.

.

= t3x2 + t2x3

= t3x2 + t2(-q1x2+x1 )

= t2x1 + (-q1t2+t3)x2

= t2x1 + t1x2

= t2x1 + t1(-q0x1+x0 )

= t1x0 + (-q0t1+t2)x1

= t1x0 + t0x1

= t1n + t0a

Cioè:

t1n + t0a = 1

ovvero t0a ha resto 1 se viene diviso per n, pertanto t0 è l’inverso di a in Zn. In questi passaggi abbiamo assunto m compreso tra i-2 e 0 e che:

ti-2 = 1

ti-3 = -qi-3

ti-4 = -qi-4ti-3+ti-2

e per ogni valore di m da i-4 a 0

tm = -qmtm+1+tm+2

inoltre dobbiamo considerare il resto della divisione di tm con n in quanto dobbiamo restare nell’ambito di Zn. Nel nostro caso specifico si ottengono i valori:

| m | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **tm** | **303** | **1580 (-209)** | **94** | **1768 (-21)** | **10** | **1788 (-1)** | **1** |

che corrispondono a quelli ottenuti nei passaggi riportati di seguito dove non si sono considerati i resti della divisione di di tm per n (n= 1789).

1 = **1**⋅6−**1**⋅5

= 6−1⋅(59−9⋅6) = −**1**⋅59+**10**⋅6

= −1⋅59+10⋅(124−2⋅59) = **10**⋅124−**21**⋅59

= 10⋅124−21⋅(555−4⋅124) = −**21**⋅555+**94**⋅124

= −21⋅555+94⋅(1234−2⋅555) = **94**⋅1234−**209**⋅555

= 94⋅1234−209⋅(1789−1⋅1234) = −**209**⋅1789+**303**⋅1234

L’ultima parte del codice effettua la stampa a video dell’inverso di a in Zn

# lt[0] è l'inverso di a in Zn se MCD(n,a)=1

if lx[i-1] == 1:

print("l'inverso di",a, 'in Z',n,'è:',lt[0])

**APPENDICE C**

**L’algoritmo di Solovay-Strassen per la primalità**

Il teorema di Fermat afferma che se n è primo (e solo in questo caso) allora per ogni a = 1, 2, ..., n−1 risulta

an−1= 1 in Zn.

Dunque, se per un certo a∈Zn non nullo risulta an−1 ≠ 1 allora n è certamente composto: il numero a è così un testimone contro la primalità di n.

Per esempio:

713 è primo?

Prendiamo il testimone più piccolo, cioè a = 2; risulta in Z713:

2712 = 624 ≠ 1.

Quindi 713 è composto (= 23·31). Si osservi che in questo modo possiamo dichiarare 713 composto senza conoscerne alcun fattore. E se invece, per un certo a, risulta an−1 = 1, possiamo affermare che n è primo? La risposta è no, e il più piccolo controesempio è n = 341. Infatti

2340 = 1 in Z341

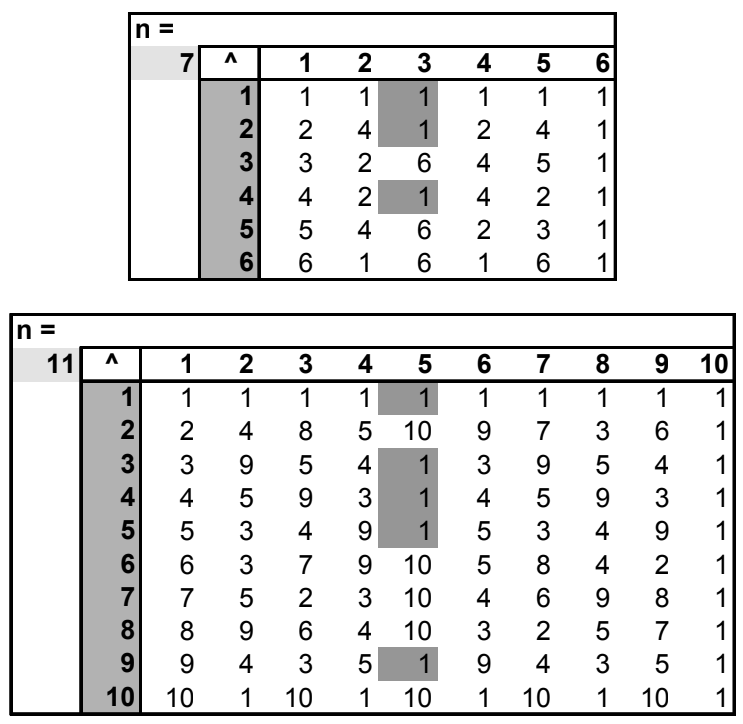
ma 341 è composto (= 11·31). Insomma, il numero 2 è un testimone a favore della primalità di 341, ma è un testimone che si rivela inattendibile.

E se aumentiamo il numero di testimoni a favore? Può essere che, pur di aumentare il numero di testimoni, si possa dichiarare n primo, almeno con una certa probabilità?

Anche in questo caso la risposta è no. Infatti esistono infiniti numeri composti, chiamati **numeri di Carmichael**, per i quali tutti gli a primi con n sono testimoni a favore della primalità e sono tutti inattendibili. Il più piccolo numero di Carmichael è 561 = 3·11·17 (i successivi sono 1105, 1729, 2465, 2821, 6601, 8911, ...). È ovvio che se si prende un testimone a che non sia primo con 561 non può risultare a560 = 1 in Z561 (altrimenti a sarebbe invertibile). Sappiamo che i numeri primi con 561 sono φ(561) = 320: bene, per tutti questi risulta a560 = 1.

Quindi il teorema di Fermat può essere utilizzato per affermare che un certo n è composto, ma non per riconoscere se n è primo.

Cerchiamo qualche altra proprietà. Si osservino le seguenti tabelle delle potenze in Z7 e Z11.

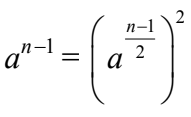


Si nota che:

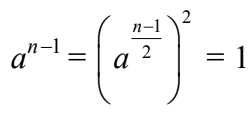
− in Z7 le potenze terze sono tutte uguali a 1 oppure a 6 (che è −1 in Z7);

− in Z11 le potenze quinte sono tutte uguali a 1 oppure a 10 (cioè -1 in Z11).

Si tratta di una proprietà generale: se n è primo allora per ogni a∈Zn non nullo risulta



se n è primo allora per il teorema di Fermat



e l’equazione x2 = 1 in Zn ammette le sole soluzioni x = 1 e x = −1 (questo non è più vero se n è composto; per esempio in Z8 l’equazione

x2 = 1 ammette 4 soluzioni: 1, 3, 5, 7 = −1).

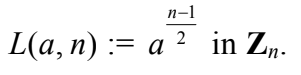
Quindi se n è primo possiamo suddividere i numeri non nulli di Zn in due insiemi: quelli che elevati a (n−1)/2 danno 1 (si dimostra che sono esattamente la metà di essi), e quelli che elevati a (n−1)/2 danno −1 (l’altra metà).

Possiamo definire così la funzione di **Legendre**.

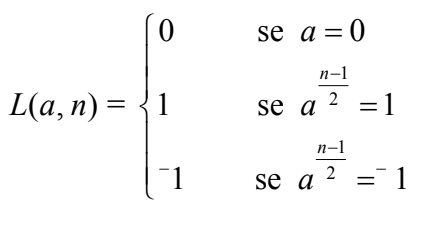
Definizione

Sia n un numero primo maggiore di 2 e a∈Zn. la funzione di Legendre L(a, n)

è così definita:

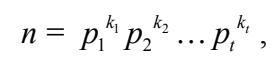


Da quanto abbiamo visto, la funzione di Legendre assume solo tre valori:



Per costruire l’algoritmo di Solovay-Strassen ci serve estendere la funzione di Legendre (che si applica solo a numeri primi) a qualsiasi numero dispari (esattamente come abbiamo fatto per estendere il teorema di Fermat con il teorema di Eulero). Si definisce così la **funzione di Jacobi**.

**Definizione**

Sia n un numero dispari maggiore di 2, la cui scomposizione in fattori primi è la seguente

e sia a∈Zn. La funzione di Jacobi è così definita:



Per esempio, in Z21 risulta

*J*(2, 21) = *L*(2, 3)*L*(2, 7) = −1⋅1 = −1

*J*(3, 21) = *L*(3, 3)*L*(3, 7) = 0⋅1 = 0

*J*(4, 21) = *L*(4, 3)*L*(4, 7) = 1⋅1 = 1

È ovvio che se n è primo allora la funzione di Legendre coincide con la funzione di Jacobi. Anche la funzione di Jacobi, come la funzione di Legendre, non può che as-

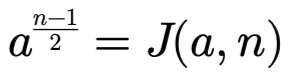
sumere uno dei tre valori 0, −1, 1: in particolare risulta

J(a, n) = 0

se e solo se MCD(n, a) ≠ 1, cioè se a non è primo con n.

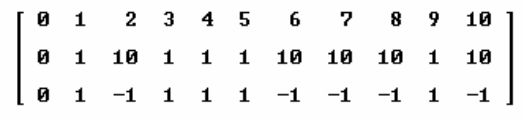
Il fatto interessante è che per calcolare la funzione di Jacobi non occorre conoscere la fattorizzazione di n e per esso esistono algoritmi di calcolo efficienti, sui quali siamo costretti a sorvolare.

Quindi per un n dispari, ci sono φ(n) numeri a∈Zn tali che J(a, n) = −1 oppure J(a, n) = 1.

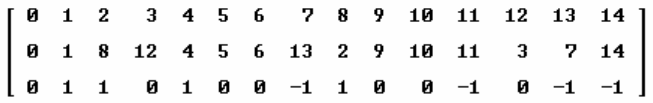
Sappiamo che se n è primo, allora tutti gli a∈Zn soddisfano l’uguaglianza



la figura seguente esemplifica questa proprietà dei numeri primi con n = 11 e mette a confronto, per tutti gli a∈Z11 (elencati sulla prima riga), il valore della potenza

(seconda riga) con il valore di J(a, n) (terza riga). Ricordiamo che in Z11 risulta 10 = −1.

E se n non è primo? Costruiamo la stessa tabella per un numero dispari composto, per esempio n = 15.



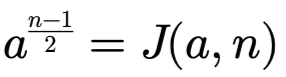
Ignoriamo i numeri non primi con 15 (sono 0, 3, 5, 6, 9, 10, 12, per i quali *J*(a, n) = 0); per gli altri 8 = φ(15) può risultare:

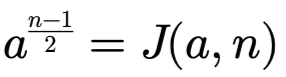


- = *J*(a, n); ce ne sono solo 2: 1 e 14; oppure



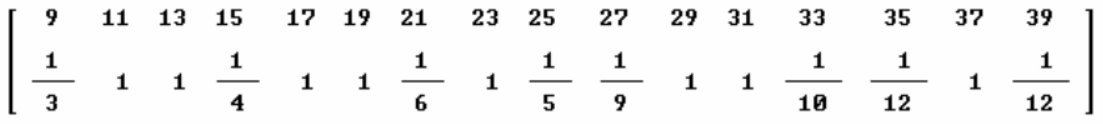
- ≠ *J*(a, n); ce ne sono 6: 2, 4, 7, 8, 11, 13.



Se allora a è un testimone a favore della primalità di n (si comporta come se n fosse primo). Quindi per n = 15 ci sono 2 testimoni a favore, e 6 contro; invece per n = 11 tutti sono testimoni a favore. In generale, consideriamo un numero dispari composto n e sia φ(n) il numero di numeri primi con esso. Di questi, siano T(n) i testimoni a favore, cioè quelli che soddisfano l’uguaglianza , 

e quindi φ(n)−T(n) i testimoni contro, cioè quelli che non la soddisfano.

Se si svolge qualche esperimento con diversi numeri composti dispari, si scopre che T(n) è solitamente piccolo; cioè: se n è composto, sono in pochi a testimoniare per la sua primalità. La funzione test(n,a), restituisce 1 se a è un testimone a favore, 0 altrimenti. Possiamo così facilmente svolgere qualche esperimento per contare quanti sono, per i numeri composti dispari, i testimoni a favore. Grazie alla funzione utilizzata otteniamo il rapporto tra T(n) e φ(n). Sfruttando la funzione per valutare quanti sono i testimoni a favore per i numeri dispari da 9 a 39 otteniamo la tabella



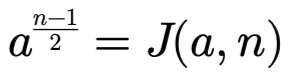
Come si vede, il rapporto T(n)/φ(n) riportato nella seconda riga vale 1 se n, riportato nella prima riga è primo, vale in generale molto meno di 1 se n è composto. Per esempio tale rapporto vale solo 1/12 per n = 39.

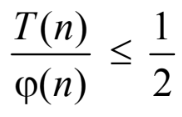
In altri termini: se n è primo tutti sono testimoni a favore della primalità di n; se n è composto, i testimoni a favore della primalità di n sono “pochi” (in un senso che tra poco preciseremo).

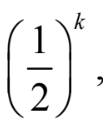
Gli n che si comportano peggio, da questo punto di vista, sono ancora una volta i numeri di Carmichael: per esempio, per n = 1729 risulta T(n)/φ(n) = 1/2, cioè ben la metà dei numeri primi con n sono pronti a testimoniare a favore della primalità di n. Bene, questo è il caso peggiore che possa capitare. Poniamoci la stessa domanda che ci siamo posti per il teorema di Fermat: è possibile, pur di aumentare il numero di testimoni a favore, che si possa dichiarare n primo, almeno con una certa probabilità? Questa volta la risposta è SI.

Il teorema seguente costituisce la chiave dell’algoritmo di **Solovay-Strassen**.

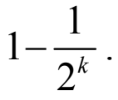
**Teorema**

Sia n un numero composto dispari n, φ(n) il numero di numeri a primi con n, T(n) il numero di testimoni a per i quali

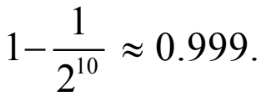
Allora qualunque sia n risulta

Questo significa che se prendiamo a caso un a primo con n (n composto dispari), la probabilità che a sia un testimone a favore della primalità di n è minore del 50%. Oppure, in altri termini: se per un certo n dispari troviamo un testimone a favore a, scegliendolo a caso tra i φ(n) primi con n, allora la probabilità che n sia composto è minore di 1/2. Se di testimoni ne troviamo k, allora la probabilità che n sia composto è minore di

il che significa che la probabilità che n sia primo è maggiore di

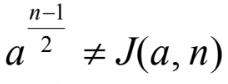


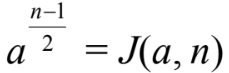
Se riusciamo a procurarci 10 testimoni per n, la probabilità che n sia primo è



Con 20 testimoni la probabilità sale a circa 0.999999. La probabilità che n abbia 50 testimoni (solitamente questo è lo standard di sicurezza) e sia composto è circa di 10−15.

Illustriamo finalmente l’algoritmo di Solovay-Strassen. Sia dato un numero dispari n di cui vogliamo stabilire la primalità; si scelga a caso un numero a compreso tra 1 e n−1. Allora:

− se MCD(n, a) > 1, allora n è certamente composto;

− altrimenti, se allora n è certamente composto;

− altrimenti, se allora n è probabilmente primo, con probabilità

maggiore di 1/2.

− si itera il procedimento fino a che: o si conclude che n è composto, oppure si raggiungono 50 testimoni a favore, nel qual caso il numero viene dichiarato primo, con probabilità maggiore di 1−10−15.

Lo scopo di questa appendice è stato, in definitiva, quello di mostrare come sia possibile fabbricarsi numeri primi grandi, di dimensioni utili per definire le chiavi dell’algoritmo RSA.

Rivisto nel mese di agosto del 2024