Tecniche di Progettazione Automatica **Anno Acc. 1995-96**

PROF. ALFREDO DE SANTIS

ALGORITMI IN TEORIA DEI NUMERI

a cura di: Antonio Iannicelli Laura Lamaina e Antonio Licciardi *5 giugno 1996*

Indice

1 Introduzione

Tra i più importanti problemi nella teoria dei numeri vi sono il problema di testare la primalità di un numero, il problema di fattorizzare interi di grossa taglia e quello del calcolo del logaritmo discreto.
Prima di analizzare alcuni tra i migliori algoritmi attualmente conosciuti che risolvono questi problemi, introduciamo i concetti base della teoria dei numeri, [1].

2 Nozioni elementari della Teoria dei numeri

Questo capitolo presenta le nozioni elementari della teoria dei numeri concernenti l'insieme $\mathbf{Z} = \{..., -2, -1, 0, 1, 2, ...\}$ di interi e l'insieme $\mathbf{N} = \{0, 1, 2, ...\}$ dei numeri naturali.

2.1 Divisori

La notazione $d|a$ (si legge d **divide** a) significa che $a = kd$ per qualche intero k. Se $d|a$, a è detto **multiplo** di d. $d \not| a$ significa che d non divide a. Se $d|a$ e $d \geq 0$ d è detto **divisore** di a. Un divisore di un intero a è compreso tra 1 e $|a|$. Per esempio, i divisori di 24 sono $1, 2, 3, 4, 6, 8, 12$, e 24. Ogni intero a è divisibile dai **divisori banali** 1 e a. I divisori non banali di a sono detti **fattori** di a. Per esempio, i fattori di 20 sono $2, 4, 5$, e 10.

2.2 Numeri primi e composti

Un intero $a > 1$ che ha come divisori 1 e a (cioè solo i divisori banali) è detto numero **primo**. Un intero $a > 1$ non primo è detto numero **composto**. Per esempio, 39 è composto perchè $3|39$.

2.3 Teorema della divisione, resti, e classi di equivalenza

Teorema 1 (Teorema della divisione) *Per ogni intero a e per ogni intero positivo n, $\exists!$ intero q ed r tale che $0 \leq r < n$ e $a = qn + r$.*

Il valore $q = \lfloor a/n \rfloor$ è il **quoziente** della divisione. Il valore $r = a \bmod n$ è il **resto** della divisione.
Noi abbiamo che $n|a \iff a \bmod n = 0$; questo segue da : $a \bmod n = a - \lfloor a/n \rfloor n$.
Se $(a \bmod n) = (b \bmod n)$, noi scriviamo $a \equiv b \pmod{n}$ e diciamo che a è **equivalente** a b modulo n. In altre parole, $a \equiv b \pmod{n}$ se a e b hanno lo stesso resto quando dividono n. Inoltre, $a \equiv b \pmod{n} \iff n|\,(b-a)$. Se a non è equivalente a b modulo n, equivale a scrivere : $a \not\equiv b \pmod{n}$.
Per esempio, $61 \equiv 6 \pmod{11}$, come pure $-13 \equiv 22 \equiv 2 \pmod{5}$.
Gli interi possono essere divisi in n classi di equivalenza a seconda del resto modulo n. Le **classi di equivalenza modulo n** contengono degli interi che sono del tipo:

$$[a]_n = \{a + kn : k \in \mathbf{Z}\}$$

Per esempio, $[3]_7 = \{..., -11, -4, 3, 10, 17, ...\}$; Questo insieme è uguale a $[-4]_7$ e $[10]_7$. Dire che $a \in [b]_n$ è lo stesso che scrivere $a \equiv b(\bmod n)$. L'insieme di tutte le classi di equivalenza è:

$$\mathbf{Z}_n = \{[a]_n : 0 \le a \le n-1\}.$$

Oppure con facilità di notazione $\mathbf{Z}_n = \{0, 1, ..., n-1\}$.

Proprietà 1 *Se a e b sono due qualsiasi interi tale che $a|b$ e $b > 0$ allora :*
$(x \bmod b) \bmod a = x \bmod a \;\; \forall x$.
Inoltre se $x \equiv y \pmod{b} \Longrightarrow x \equiv y \pmod{a} \;\; \forall x$ e y.

2.4 Comuni divisori e M.C.D.

Se d è un divisore di a e di b, d è un **comune divisore** di a e b. Per esempio, i divisori di 30 sono 1,2,3,5,6,10,15, e 30, cosi i comuni divisori di 24 e 30 sono 1,2,3 e 6. Nota che 1 è un comune divisore per qualsiasi coppia di interi.
Un' importante propietà dei comuni divisori è la seguente :
$d|a$ e $d|b \Longrightarrow d|(a+b)$ e $d|(a-b)$.
In generale abbiamo che:
$d|a$ e $d|b \Longrightarrow d|(ax+by) \; \forall x, y \in \mathbf{Z}$.
Inoltre, se $a|b$ allora o $|a| \le |b|$ oppure b=0, ciò significa che se $a|b$ e $b|a \Longrightarrow a = \pm b$.
Il **Massimo Comune Divisore** tra due interi a e b, non entrambi zero, è il più grande tra i divisori comuni di a e di b; Esso si denota con mcd(a,b).
Per esempio, mcd(24,30)=6, mcd(5,7)=1, e mcd(0,9)=9.
Se a e b non sono entrambi zero, allora mcd(a,b) è un intero compreso tra 1 e min($|a|$,$|b|$). Si assume che il mcd(0,0)=0. Questa definizione serve per le seguenti proprietà elementari della funzione mcd:
$\mathrm{mcd}(a,b) = \mathrm{mcd}(b,a)$
$\mathrm{mcd}(a,b) = \mathrm{mcd}(-a,b)$
$\mathrm{mcd}(a,b) = \mathrm{mcd}(|a|,|b|)$
$\mathrm{mcd}(a,0) = |a|$
$\mathrm{mcd}(a,ka) = |a| \; \forall k \in Z$.

Teorema 2 *Se a e b sono interi, non entrambi zero, il mcd(a,b) è il più piccolo elemento positivo dell'insieme $\{ax + by : x, y \in Z\}$ della combinazione lineare di a e b.*

Corollario 1 *Per ogni intero* a *e* b, *se* $d|a$ *e* $d|b$ *allora* $d|mcd(a,b)$.

Corollario 2 *Per ogni intero* a *e* b *e per ogni positivo* n :
$mcd(an, bn) = n \cdot mcd(a,b)$.

Corollario 3 *Per ogni intero positivo* n, a, *e* b, *se* $n|ab$ *e* $mcd(a,n)=1$, *allora* $n|b$.

Proprietà 2 *Se* p *è un numero primo e* $0 \leq k \leq p$, *allora* $mcd(k, p) = 1$.

2.5 Primi interi relativi

Due interi a e b sono **primi relativi** se il loro solo comune divisore è 1, cioè se $\mathrm{mcd}(a, b) = 1$. Per esempio, 8 e 15 sono primi relativi perchè, i divisori di 8 sono 1,2,4 e 8 mentre quelli di 15 sono 1,3,5 e 15.

Teorema 3 *Per ogni intero* a, b *e* p, *se* $mcd(a, p) = 1$ *e* $mcd(b, p) = 1$ *allora* $mcd(ab, p)=1$.

2.6 Fattorizzazione unica

Un importante fatto elementare riguardo la divisibilità dei numeri primi è il seguente:

Teorema 4 *Per ogni primo* p *e per ogni intero* a, b, *se* $p|ab$ *allora o* $p|a$ *oppure* $p|b$.

Una conseguenza di tale teorema è che un intero ha un'unica fattorizzazione in numeri primi.

Teorema 5 (Fattorizzazione unica) *Un numero composto* a *può essere scritto esattamente in un modo e cioè come prodotto di:*
$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}$
dove i p_i *sono primi,* $p_1 < p_2 < \cdots < p_r$, *e gli* e_i *sono interi positivi.*

Per esempio, il numero 6000 può essere unicamente fattorizzato come: $2^4 \cdot 3 \cdot 5^3$.

2.7 Algoritmi per il Massimo Comune Divisore

In questo paragrafo per computare efficientemente il M.C.D. di due interi usiamo l'algoritmo di Euclide, usando solo numeri interi positivi. Questo è giustificato da una proprietà del M.C.D. $\mathrm{mcd}(a, b)=\mathrm{mcd}(|a|, |b|)$.
Eseguiamo il M.C.D. partendo dalla fattorizzazione di a e b in numeri primi, cioè :

$$a = p_1^{e_1} p_2^{e_2} \cdots p_r^{e_r}, \quad b = p_1^{f_1} p_2^{f_2} \cdots p_r^{f_r}$$

così il M.C.D. è semplicemente :

$$mcd(a, b) = p_1^{min(e_1, f_1)} p_2^{min(e_2, f_2)} \cdots p_r^{min(e_r, f_r)}$$

L'algoritmo di Euclide, per computare il M.C.D., si basa sul seguente teorema:

Teorema 6 (Ricorsione del M.C.D.) *Per ogni intero nonnegativo* a *e per ogni intero positivo* b, $mcd(a, b) = mcd(b, a \bmod b)$.

2.7.1 Algoritmo di Euclide

Gli input a e b sono due interi non negativi qualsiasi.

```
EUCLIDE(a, b)

1. if b = 0
2.     then return a
3.     else return EUCLIDE(b, a mod b)
```

Esempio 1 *Consideriamo la computazione del mcd(30, 21):*

$$\begin{aligned}\mathrm{EUCLIDE}(30,\ 21) &= \mathrm{EUCLIDE}(21,\ 9)\\ &= \mathrm{EUCLIDE}(9,\ 3)\\ &= \mathrm{EUCLIDE}(3,\ 0)\\ &= 3\end{aligned}$$

In questa computazione ci sono solo tre chiamate ricorsive.

L'algoritmo non sarà ricorsivo all'infinito poichè il secondo decresce strettamente in ogni chiamata ricorsiva. Così l'algoritmo di Euclide termina sempre con una risposta corretta.

2.7.2 La complessità dell'algoritmo di Euclide

Analizziamo il peggior caso di EUCLIDE come funzione di grandezza di a e di b. Assumiamo con una piccola perdita di generalità il caso che $a > b \geq 0$. Questa assunzione viene giustificata dal fatto che, se $b > a \geq 0$, allora EUCLIDE(a, b) genera la chiamata ricorsiva EUCLIDE(b, a). Cioè, in questo caso, EUCLIDE fa una sola chiamata in più rispetto al caso $a > b$. Nel caso in cui $b = a > 0$ la procedure termina dopo una chiamata ricorsiva poichè $a \bmod b = 0$. Il tempo di esecuzione complessivo di EUCLIDE é proporzionale al numero di chiamate ricorsive che fa. Si osserva che questo numero è $\mathbf{O}(\ln b)$.

2.7.3 Forma estesa dell'algoritmo di Euclide

L'estensione dell'algoritmo di Euclide è dovuta al fatto di computare due coefficienti x e y tali che :
$d = mcd(\mathrm{a,b}) = ax + by$.
Nota che x e y devono essere ≥ 0. La procedura EUCLIDE-ESTESO prende in input due interi arbitrari e restituisce una tripla del tipo (d, x, y) che soddisfa l'equazione precedente.

EUCLIDE-ESTESO(a, b)

1. **if** $b = 0$
2. **then return** $(a, 1, 0)$
3. $(d', x', y') \longleftarrow$ EUCLIDE-ESTESO(b, $a \bmod b$)
4. $(d, x, y) \longleftarrow (d', y', x' - \lfloor a/b \rfloor y')$
5. **return** (d, x, y)

Esempio 2 (EUCLIDE-ESTESO(99,78))

a	b	$\lfloor a/b \rfloor$	d	x	y
99	78	1	3	-11	14
78	21	3	3	3	-11
21	15	1	3	-2	3
15	6	2	3	1	-2
6	3	2	3	0	1
3	0	-	3	1	0

Tabella 1: EUCLIDE-ESTESO(99,78)

2.8 Aritmetica Modulare

L'aritmetica modulare è come quella normale, eccetto per il fatto che si lavora in modulo n, così ogni risultato è un valore compreso tra 0 e $n-1$. Introduciamo un modello formale per l'aritmetica modulare rifacendoci alla teoria dei gruppi.

2.8.1 Gruppi finiti

Un **Gruppo** $(S, \oplus)$ è un insieme S su cui è definita l'operazione binaria $\oplus$ che ha le seguenti proprietà:

1. **CHIUSURA** : $\forall a, b \in S \Longrightarrow a \oplus b \in S$
2. **IDENTITÀ** : $\exists e \in S : e \oplus a = a \oplus e = a \ \forall a \in S$
3. **ASSOCIATIVITÀ** : $\forall a, b, c \in S$ abbiamo $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
4. **INVERSI** : $\forall a \in S \ \exists! \ b \in S : a \oplus b = b \oplus a = e$

Per esempio, consideriamo il gruppo $(\mathbf{Z}, +)$ degli interi: 0 è l'identità, e l'inverso di a è $-a$. Se un gruppo $(S, \oplus)$ soddisfa la proprietà commutativa ($a \oplus b = b \oplus a \ \forall a, b \in S$) è detto **gruppo abeliano**. Inoltre se $|S| < \infty$ allora esso è detto **gruppo finito**.

$+_6$	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

$\cdot_{15}$	1	2	4	7	8	11	13	14
1	1	2	4	7	8	11	13	14
2	2	4	8	14	1	7	11	13
4	4	8	1	13	2	14	7	11
7	7	14	13	4	11	2	1	8
8	8	1	2	11	4	13	14	7
11	11	7	14	2	13	1	8	4
13	13	11	7	1	14	8	4	2
14	14	13	11	8	7	4	2	1

Tabella 2: $(\mathrm{Z}_6, +_6)$ e $(\mathrm{Z}_{15}^*, \cdot_{15})$

2.8.2 Addizione e moltiplicazione modulare

Possiamo formare due gruppi abeliani finiti usando l'addizione e la moltiplicazione modulo n, dove n è un intero positivo. Questi gruppi sono basati sulle classi di equivalenza degli interi modulo n.
Per definire un gruppo su $\mathbf{Z}_n$ abbiamo bisogno di un'operazione binaria che può essre ottenuta ridefinendo le ordinarie operazioni di addizione e di moltiplicazione. È facile definire le operazioni di addizione e di moltiplicazione per $\mathbf{Z}_n$, perchè le classi di equivalenza di due interi determinano univocamente le classi di equivalenza della loro somma o prodotto. Cioè se
$a \equiv a' \pmod{n}$ e $b \equiv b' \pmod{n}$ allora
$a + b \equiv a' + b' \pmod{n}$,
$ab \equiv a'b' \pmod{n}$.
Così noi definiamo l'addizione e moltiplicazione modulo n, denotate con $+_n$ e $\cdot_n$:

$$[a]_n +_n [b]_n = [a+b]_n$$

$$[a]_n \cdot_n [b]_n = [ab]_n$$

Usando questa definizione di somma modulo n possiamo definire il **gruppo additivo modulo n** come $(\mathbf{Z}_n, +_n)$. La cardinalità del gruppo additivo modulo n è $|\mathbf{Z}_n| = n$. La tavola 2 mostra la tabella completa dell'operazione somma per il gruppo $(\mathbf{Z}_6, +_6)$.

Teorema 7 *Il gruppo $(\mathbf{Z}_n, +_n)$ è un gruppo finito abeliano.*

Usando la definizione di prodotto modulo n, possiamo definire il **gruppo moltiplicativo modulo n** come $(\mathbf{Z}_n^*, \cdot_n)$. Gli elementi di questo gruppo sono gli elementi di Z_n che sono primi con n :

$$\mathbf{Z}_n^* = \{[a]_n \in \mathbf{Z}_n : mcd(a, n) = 1\}.$$

Un esempio di tale gruppo è

$$\mathbf{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$$

dove l'operazione è la moltiplicazione modulo 15. La tavola 2 mostra il gruppo $(\mathbf{Z}_{15}^*, \cdot_{15})$.
Per esempio $8 \cdot 11 \equiv 13 \pmod{15}$ lavorando in $\mathbf{Z}_{15}^*$. L'identità per questo gruppo è 1.

Teorema 8 *Il gruppo $(\mathbf{Z}_n^*, \cdot_n)$ è un gruppo finito abeliano.*

L'inverso moltiplicativo di un elemento a è denotato $(a^{-1} \bmod n)$. La divisione in $\mathbf{Z}_n^*$ è definita dall'equazione $a/b \equiv ab^{-1} (\bmod n)$.
Per esempio in $\mathbf{Z}_{15}^*$ abbiamo che $7^{-1} \equiv 13 \pmod{15}$, poichè $7 \cdot 13 \equiv 91 \equiv 1 \pmod{15}$ cosichè $4/7 \equiv 4 \cdot 13 \equiv 7 \pmod{15}$.
La cardinalità di $\mathbf{Z}_n^*$ è denotata con $\phi(n)$. Questa funzione è detta **Funzione di Eulero** e soddisfa la seguente equazione:

$$\phi(n) = n \prod_{p|n} \left(1 - \frac{1}{p}\right)$$

dove p sono tutti i divisori primi di n (includendo n stesso se n è primo).
Per esempio, poichè i divisori primi di 45 sono 3 e 5,

$$\begin{aligned} \phi(45) &= 45\left(1 - \tfrac{1}{3}\right)\left(1 - \tfrac{1}{5}\right) \\ &= 45\left(\tfrac{2}{3}\right)\left(\tfrac{4}{5}\right) \\ &= 24. \end{aligned}$$

Se p è primo, allora $\mathbf{Z}_p^* = \{1, 2, \cdots, p-1\}$, e $\phi(p) = p - 1$.
Se n è composto allora $\phi(n) < n - 1$.
Se p e q sono primi tali che $m = pq$, $\phi(m) = (p-1)(q-1)$.

Teorema 9 (Teorema di Eulero) *Per ogni intero $n > 1$,$a^{\phi(n)} \equiv 1 (\bmod n)$ $\forall a \in \mathbf{Z}_n^*$*

Teorema 10 (Teorema di Fermat) *Se p è un primo allora $a^{p-1} \equiv 1 (\bmod p)$ $\forall a \in \mathbf{Z}_p^*$*

2.8.3 Sottogruppi

Se $(S, \oplus)$ è un gruppo, $S' \subseteq S$, e $(S', \oplus)$ è un gruppo allora $(S', \oplus)$ è detto **sottogruppo** di $(S, \oplus)$.

Esempio 3 *Consideriamo il gruppo (Z_8, $+_8$) e l'insieme $S' = \{0, 2, 4, 6\}$ allora (S', $+_8$) è un sottogruppo; infatti guardando la tavola 3 vediamo che l'operazione è interna a S'.*

$+_8$	*0*	*2*	*4*	*6*
0	*0*	*2*	*4*	*6*
2	*2*	*4*	*6*	*0*
4	*4*	*6*	*0*	*2*
6	*6*	*0*	*2*	*4*

Tabella 3: il sottogruppo (S', $+_8$)

Teorema 11 (Teorema di Lagrange) *Se $(S, \oplus)$ è un gruppo finito e $(S', \oplus)$ è un suo sottogruppo, allora $|S'|$ è un divisore $|S|$.*

2.8.4 Sottogruppi generati da un elemento

Dato un gruppo $(S, \oplus)$ scegliamo un elemento $a \in S$ e troviamo tutti gli elementi che possono essere generati da a usando l'operazione del gruppo. Definiamo $a^{(k)}$ per $k \geq 1$ come

$$a^{(k)} = \bigoplus_{i=1}^{k} a = a \oplus a \oplus \cdots \oplus a \quad k\, volte$$

Per esempio, se prendiamo $a = 2$ nel gruppo $\mathbf{Z}_6$, la sequenza $a^{(1)}, a^{(2)}, \cdots$ è
2, 4, 0, 2, 4, 0, 2, 4, 0,.... .
Nel gruppo $\mathbf{Z}_n$ abbiamo che $a^{(k)} = ka \bmod n$ e nel gruppo $\mathbf{Z}_n^*$ abbiamo che $a^{(k)} = a^{(k)} \bmod n$.
Il **sottogruppo generato da a**, denotato con $\langle a \rangle$ oppure $(\langle a \rangle, \oplus)$ è definito così:

$$\langle a \rangle = \{a^{(k)} : k \geq 1\}.$$

Noi diciamo che a **genera** il sottogruppo di $\langle a \rangle$ oppure che a è il **generatore** di $\langle a \rangle$. Poichè S è finito, $\langle a \rangle$ è un sottoinsieme finito di S, possibilmente includendo tutto S.

Teorema 12 *Siano p primo, $g \in \mathbf{Z}_p^*$ e $p - 1 = q_1^{e_1} q_2^{e_2} \cdots q_m^{e_m}$, con q_i primo per $i = 1, 2, ..., m$. Se esiste un q_i tale che :*

$$g^{\frac{p-1}{q_i}} \equiv 1 \bmod p$$

allora g non è un generatore di $\mathbf{Z}_p^*$.

Esempio 4 *In $\mathbf{Z}_6$ abbiamo:*

$$\begin{aligned}
&\langle 0 \rangle = \{0\}, \\
&\langle 1 \rangle = \{0, 1, 2, 3, 4, 5\} = \mathbf{Z_6}, \\
&\langle 2 \rangle = \{0, 2, 4\}.
\end{aligned}$$

In questo caso vediamo che $\langle 1 \rangle$ genera tutto $\mathbf{Z}_6$ e si dice che $\mathbf{Z}_6$ è un **gruppo ciclico**.
L'**ordine** di a (nel gruppo S), denotato con ord(a), è definito come il più piccolo $t > 0$ tale che $a^{(t)} = e$, dove e è l'elemento neutro di S.

Teorema 13 *Per ogni gruppo finito $(S, \oplus)$ e per ogni $a \in S$, l'ordine di un elemento è uguale alla grandezza del sottogruppo da esso generato, ovvero ord$(a) = |\langle a \rangle|$.*

Corollario 4 *Se $(S, \oplus)$ è un gruppo finito con identità e, allora per tutti gli $a \in S$,*

$$a^{(|S|)} = e$$

Il teorema di Lagrange implica che ord(a) divide $|S|$ e così $|S| \equiv 0 \pmod t$, dove $t = ord(a)$.

2.9 Risoluzione di equazioni lineari modulari

Consideriamo il problema di trovare la soluzione per l'equazione

$$ax \equiv b(\text{mod} n), \quad \textit{dove } n > 0.$$

Assumiamo che a, b, e n sono noti, e vogliamo trovare quella x modulo n che soddisfa l'equazione. Devono esserci zero, uno, o più soluzioni.

Corollario 5 *L'equazione $ax \equiv b(\text{mod} n)$ è risolvibile per un sconosciuto x se e solo se $mcd(a,n)|b$.*

Corollario 6 *L'equazione $ax \equiv b(\bmod n)$ ha d distinte soluzioni modulo n, dove $d = mcd(a,n)$, oppure non ha soluzioni.*

Teorema 14 *Sia $d = mcd(a,n)$ e supponiamo che $d = ax' + ny'$ per qualche intero x' e y'. Se $d|b$ allora l'equazione $ax \equiv b(\text{mod} n)$ ha come una delle sue soluzioni il valore*

$$x_0 = x'(b/d) \bmod n.$$

Teorema 15 *Supponiamo che l'equazione $ax \equiv b(\bmod n)$ è risolvibile (cioè $d|b$, dove $d = mcd(a,n)$) e che x_0 è una soluzione. Allora questa equazione ha esattamente d distinte soluzioni, modulo n, date da*

$$x_i = x_0 + i(n/d) \quad per \;\; i = 1, 2, ..., d-1.$$

Abbiamo adesso tutte le formulazioni matematiche per risolvere l'equazione $ax \equiv b$ (mod$n$); Il prossimo algoritmo stampa tutte le soluzioni di quest'equazione. Gli input a e b sono degli interi arbitrari, e n è un intero arbitrario positivo.

RISOLUZIONE-EQUAZIONI-LINEARI-MODULARI (a, b, n)

1. $(d, x', y') \longleftarrow$ EUCLIDE-ESTESO (a, n)
2. **if** $d|b$
3. **then** $x_0 \longleftarrow x'(b/d)$ mod$n$
4. **for** $i \longleftarrow 0$ **to** $d-1$
5. **do** print $(x_0 + i(n/d))$ mod$n$
6. **else** print "nessuna soluzione"

2.10 Il teorema cinese del resto

Il teorema cinese del resto fornisce una corrispondenza tra un sistema di equazioni modulari, un insieme di moduli primi tra loro a coppie (per esempio 3, 5 e 7) e un'equazione modulare del loro prodotto. Ad esempio vogliamo trovare tutti gli interi x che danno come resto 2, 3 e 2 quando vengono divisi rispettivamente per 3, 5 e 7. Una delle soluzioni è $x = 23$ e la soluzione generale è $23 + 105k$ per qualche arbitrario intero k.

Teorema 16 (Teorema cinese del resto) *Dato un sistema di equazioni modulari:*

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \vdots \quad \vdots \\ x \equiv a_r \pmod{m_r} \end{cases}$$

Con gli m_i primi fra loro a due a due (cioè mcd$(m_i, m_j) = 1$ e $i \neq j$).
Sia $M = \prod_{i=1}^{r} m_i$, $M_i = \frac{M}{m_i}$ *e* $Y_i = M_i^{-1}$
La soluzione generale del sistema è :

$$x = \sum_{i=1}^{r} a_i M_i Y_i \pmod{M}$$

2.11 Quadrati modulo p

Definizione 1 *Sia p primo. Un elemento $y \in \mathbf{Z}_p^*$ si dice* **quadrato modulo p** *se esiste $x \in \mathbf{Z}_p^*$ tale che:*

$$y = x^2 \bmod p$$

in tal caso x si dice **radice** *di y modulo p.*

Teorema 17 *Sia p primo, y un quadrato modulo p e x una radice di y allora:*

(a) *$p - x$ è un'altra radice di y*

(b) *x e $p - x$ sono tutte e solo le radici di y.*

Lemma 1 *Se p è un numero primo, risulta:*

1. *Il numero di quadrati modulo p è $\frac{p-1}{2}$*
2. *Dato un generatore g di $\mathbf{Z}_p^*$, tutti i quadrati modulo p sono i seguenti:*

$$y = g^{2s} \quad s = 1, 2, ..., \frac{p-1}{2}$$

3. *Se g è un generatore di $\mathbf{Z}_p^*$, risulta:*

$$g^x = g^y \bmod p \Longleftrightarrow x = y \bmod \phi(p)$$

Per determinare se un numero è un quadrato modulo p, esiste il seguente:

Teorema 18 (Criterio di Eulero) *x è un quadrato modulo $p \Longleftrightarrow x^{\frac{p-1}{2}} = 1 \bmod p$.*

2.12 Residui Quadratici

Se p è un primo e a è minore di p allora a è un residuo quadratico modulo p se la congruenza:

$$x^2 \equiv a \bmod p$$

ha un'unica soluzione $x \in Z_p$.
Osserviamo che non tutti i valori di a soddisfano questa proprietà.

Ad esempio se $p = 7$ calcolo $x^2 \equiv a \bmod p \;\; \forall x \in Z_p$:

$$1^2 \equiv 1 \bmod 7$$
$$2^2 \equiv 4 \bmod 7$$
$$3^2 \equiv 2 \bmod 7$$
$$4^2 \equiv 2 \bmod 7$$
$$5^2 \equiv 4 \bmod 7$$
$$6^2 \equiv 1 \bmod 7$$

Non c'è nessun valore in Z_p che invece soddisfa le equazioni:

$$x^2 \equiv 3 \bmod 7$$
$$x^2 \equiv 5 \bmod 7$$
$$x^2 \equiv 6 \bmod 7$$

Segue che 1, 2 e 4 sono residui quadratici modulo 7 mentre 3, 5 e 6 non lo sono. Se a è un residuo quadratico modulo p allora a ha esattamente due radici quadrate: x_1 compresa tra 0 e $(p-1)/2$, e l'altra compresa tra $(p-1)/2$ e $(p-1)$. Una di queste radici è anche un residuo quadratico modulo p ed è detta radice quadrata principale.

Se n è il prodotto tra due primi p e q ci sono esattamente $(p-1)(q-1)/4$ residui quadratici modulo n.

Si dimostra che ci sono esattamente $(p-1)/2$ residui quadratici modulo p e lo stesso numero di residui non quadratici. Una definizione alternativa di residuo quadratico è fornita dal Criterio di Eulero ed è la seguente:

Definizione 2 (Criterio di Eulero) *Sia p un primo, allora x è un residuo quadratico modulo p se e solo se*

$$x^{(p-1)/2} \equiv 1 \bmod p$$

Infatti ad esempio $2^{(7-1)/2} \equiv 1 \bmod 7$

2.12.1 Simbolo di Legendre

Il simbolo di Legendre è indicato con $L(a,p)$, dove a è un intero e p è un primo maggiore di 2. Esso assume i seguenti valori:

$$L(a,p) = \begin{cases} 0 & \text{se } p \text{ divide } a \ (a \equiv 0 \bmod p) \\ 1 & \text{se } a \text{ è un residuo quadratico modulo } p \\ -1 & \text{se } a \text{ non è un residuo quadratico modulo } p \end{cases}$$

In modo alternativo (per il Criterio di Eulero) possiamo scrivere:

$$L(a,p) = \begin{cases} 0 & \text{se } a^{(p-1)/2} \equiv 0 \bmod p \\ 1 & \text{se } a^{(p-1)/2} \equiv 1 \bmod p \\ -1 & \text{se } a^{(p-1)/2} \not\equiv 1 \bmod p \end{cases}$$

2.12.2 Simbolo di Jacobi

Il simbolo di Jacobi viene indicato con $J(a,n)$ ed è una generalizzazione del simbolo di Legendre, infatti esso è definito per una qualsiasi coppia di interi a ed n. $J(a,n)$ è una funzione definita su un insieme ridotto di residui dei divisori di n e viene calcolato nel seguente modo:

1. *Se n è primo ed a è un residuo quadratico modulo n allora*

$$J(a,n) = 1$$

2. *Se n è primo ed a non è un residuo quadratico modulo n allora*

$$J(a,n) = -1$$

3. *Se n è composto allora*

$$J(a,n) = J(h,p_1) \times \cdots J(h,p_m)$$

   dove $p_1, \ldots, p_m$ sono i fattori primi di n.

Un modo equivalente per esprimere il simbolo di Jacobi, è il seguente:

$$J(a,n) = \prod_{i=1}^{k} J(a,p_i)^{e_i}$$

dove n è un intero positivo dispari mentre $p_1^{e_1} \ldots p_k^{e_k}$ rappresentano la fattorizzazione di n.
Il simbolo di Jacobi può essere valutato anche senza fattorizzare n, utilizzando alcuni risultati di teoria dei numeri.

2.12.3 Calcolo alternativo del simbolo di Jacobi

Proprietà 1 (Simbolo di Jacobi) *Sia n un intero dispari e $m_1 \equiv m_2 \bmod n$, allora*

$$J(m_1,n) = J(m_2,n)$$

Proprietà 2 *Se n è un intero dispari allora*

$$J(m_1 m_2, n) = J(m_1, n) \times J(m_2, n)$$

In particolare se $m = 2^k t$, *dove* t *è dispari, allora*

$$J(m, n) = J(2, n)^k \times J(t, n)$$

Proprietà 3 *Se n è un intero dispari allora*

$$J(-1, n) = -1^{(n-1)/2}$$

Proprietà 4 *Se n è un intero dispari allora*

$$J(2, n) = -1^{(n^2-1)/8}$$

Proprietà 5 *Siano m ed n interi dispari, allora*

$$J(m, n) = \begin{cases} -J(n, m) & \textit{se } m \equiv n \equiv 3 \bmod 4 \\ J(n, m) & \textit{altrimenti} \end{cases}$$

Esempio 5 (Calcolo del simbolo di Jacobi con le proprietà precedenti)

$$\begin{array}{lrl}
\textit{per la Prop2} & J(9,13) & = J(3,13) \times J(3,13) \\
\textit{per la Prop1} & J(3,13) & = J(16,13) \\
\textit{per la Prop2} & & = J(2,13)^4 \times J(1,13) \\
\textit{per la Prop4} & & = (-1)^{(13^2-1)/2} \times J(1,13) \\
\textit{per la Prop2} & & = (-1)^{84} \times J(-1,13) \times J(-1,13) \\
\textit{per la Prop3} & & = (-1)^6 \times (-1)^6
\end{array}$$

In definitiva

$$J(9, 13) = 1 \times 1 = 1$$

cioè 9 é un residuo quadratico mod 13

3 Test di primalità

I numeri primi sono divenuti molto importanti per la costruzione di critto-sistemi a chiave pubblica. In genere dopo aver generato casualmente un numero grande (ad esempio per RSA n è di 512 bit) la sua primalità viene testata attraverso opportuni algoritmi.
Per questo tipo di problema non si conoscono algoritmi deterministici efficienti che testano la primalità.
Gli algoritmi conosciuti che risolvono questo problema in maniera efficiente sono probabilistici. La determinazione dei numeri primi viene realizzata attraverso diversi test che possono determinare se un numero casuale è primo con un coefficiente di attendibilità superiore al 50%. Se un numero presumibilmente primo non supera il test di primalità allora esso è sicuramente non primo. Se invece supera il primo test allora la probabilità che sia non primo è al più 1/2. Se poi supera anche il secondo test allora la probabilità non supera il valore di 1/4. Allo stesso modo se supera 10 di questi tests la probabilità che non è primo è al più $1/2^{10}$, cioè 1 su 1024 (cioè è quasi primo). Un calcolo più preciso di questa probabilità viene realizzato nella sezione 3.1.2.
Gli algoritimi che vedremo appartengono alla classe degli algoritmi MONTE CARLO.

Definizione 3 (Algoritmo Monte Carlo) *Un algoritmo* MONTE CARLO yes-biased *è un algoritmo probabilistico per un problema di decisione in cui la risposta* "si" *è sempre corretta, mentre quella* "no" *può essere inesatta.*
In maniera analoga si può definire un algoritmo MONTE CARLO no-biased *in cui la risposta* "no" *è sempre corretta, mentre quella* "si" *può essere inesatta.*
Un algoritmo MONTE CARLO yes-biased *ha un errore probabilistico uguale ad ϵ se, per qualche test in cui la risposta (esatta) è* "si"*, l'algoritmo darà la risposta (incorretta)* "no" *con probabilità al più ϵ.*

Lo schema generale per la generazione di un numero primo di k bit è il seguente:

Genera primo(k)

1. **while true do**
2. genera a caso un intero x di k bit
3. **if** test-primalità(x,ϵ)=vero **then** stop

3.1 Algoritmo di Solovay-Strassen

L'algoritmo di Solovay-Strassen [2] è di tipo probabilistico e usa il simbolo di Jacobi.

Solovay-Strassen(p, 1/2)

1. Sceglie a caso $a \in \mathbf{Z}_p$
2. **If** $\gcd(a,p) \neq 1$ **then** *p non supera il test perchè è composto*
3. **Set** $j = a^{(p-1)/2} \bmod p$
4. **If** $j \neq J(a,p)$ **then** *p è definitivamente non primo*
5. **If** $j = J(a,p)$ **then** (p è primo)

Dopo un primo round se l'algoritmo dà la risposta "p è primo" allora p lo è al 50%. Se ripetiamo più volte l'algoritmo ottenendo sempre la risposta "p è primo" allora la probabilità che p è primo è quasi 1.
In un'implementazione reale l'algoritmo viene eseguito molto velocemente, e procede nel seguente modo: genera un numero a caso p e prima di applicare il test di primalità controlla che non è divisibile per i primi più piccoli. Poi genera un altro numero a caso r e calcola $\gcd(r,p)$ e $r^{(p-1)/2} \bmod p$; esegue poi i passi 2 e 4. Se p non li supera ripete il test così come è stato detto in precedenza.

Esempio 6 (Con numero primo) *Supponiamo che $p = 13$ e $a = 9$.*
Nel passo 2 dell'algoritmo viene calcolato il $\gcd(9,13)$*, siccome esso è* 1 *allora viene calcolato* $j = 9^6 \bmod 13 = 1$ *ed il simbolo di Jacobi* $J(9,13) = 1$ *(vedi esempio 4). Nel passo 5 vengono confrontati i due valori calcolati, essi sono uguali e quindi p è dichiarato primo.*

3.1.1 Running-time dell'algoritmo

L'algoritmo viene realizzato attraverso operazioni elementari, fatta eccezione per il calcolo di $a^{(p-1)/2} \bmod p$ che viene eseguito attraverso l'algoritmo per l'esponenziazione modulare il cui running-time è $O((\log n)^3)$. Per quanto riguarda il calcolo del simbolo di Jacobi osserviamo che esso non viene computato in termini di fattorizzazione, in quanto se possiamo fattorizzare n, allora sappiamo già se esso è primo o meno.

Attraverso le proprietà viste calcoliamo il simbolo di Jacobi con operazioni modulari ed elementari, per cui la complessità totale rimane $O((\log n)^3)$.

3.1.2 Probabilità di errore dell'algoritmo

Pensiamo di avere generato un numero random n e testiamo la sua primalità utilizzando l'algoritmo di Solovay-Strassen. Se eseguiamo l'algoritmo m volte, in prima analisi potremmo dire che la probabilità che tale intero n sia primo è $1 - 2^{-m}$. Approfondendo l'analisi probabilistica otteniamo un risultato diverso.
Definiamo le seguenti variabili random a e b che indicano gli eventi:

a = "un intero dispari random n di una specifica taglia è composto"

b = "l'algoritmo risponde che n è primo dopo m esecuzioni".

Sicuramente la $prob(b|a) \leq 2^{-m}$. Ad ogni modo la probabilità che a noi interessa è $prob(a|b)$ che di solito è diversa da $prob(b|a)$. Ma avendo a disposizione $prob(b|a)$, con il teorema di Bayes possiamo determinare la probabilità richiesta. A tale scopo calcoliamo $prob(a)$. Supponiamo che $N \leq n \leq 2N$, applicando il teorema sui numeri primi (Il numero dei primi su N numeri è asintoticamente : $\frac{N}{\ln(N)}$) abbiamo che il numero dei primi compreso tra N e 2N è approssimativamente:

$$\frac{2N}{\ln(2N)} - \frac{N}{\ln(N)} \approx \frac{N}{\ln(N)} \approx \frac{n}{\ln(n)}$$

Poichè ci sono $N/2 \approx n/2$ interi dispari tra N e $2N$ useremo la stima:

$$prob(a) \approx 1 - \frac{\frac{n}{\ln(n)}}{\frac{n}{2}} = 1 - \frac{2}{\ln(n)}$$

Calcoliamo allora la seguente probabilità:

$$\begin{aligned}
prob(a|b) &= \frac{prob(a)prob(b|a)}{prob(b)} \\
&= \frac{prob(a)prob(b|a)}{prob(b|a)prob(a) + prob(b|\bar{a})prob(\bar{a})} \\
&\approx \frac{prob(b|a)(1 - \frac{2}{ln(n)})}{prob(b|a)(1 - \frac{2}{\ln(n)}) + \frac{2}{\ln(n)}} \\
&= \frac{prob(b|a)(\ln(n) - 2)}{prob(b|a)(\ln(n) - 2) + 2} \\
&\leq \frac{2^{-m}(\ln(n) - 2)}{2^{-m}(\ln(n) - 2) + 2} \\
&= \frac{\ln(n) - 2}{\ln(n) - 2 + 2^{m+1}}
\end{aligned}$$

In definitiva la

$$prob(a|b) \leq \frac{\ln(n) - 2}{\ln(n) - 2 + 2^{m+1}}$$

Si può dimostrare che 2^{-m} tende più velocemente a zero rispetto a $\frac{\ln(n)-2}{\ln(n)-2+2^{m+1}}$. Questo significa che la probabilità che p sia primo dopo m esecuzioni è minore di quanto credevamo. In pratica prendendo un $m \approx 50$ o 100 si può ridurre la probabilità d'errore ad un numero molto piccolo.

Esempio 7 *Supponiamo che $n \approx 2^{256} \approx e^{177}$, scegliamo questi numeri perchè in genere queste sono le taglie dei primi usati in RSA. Nella tabella ci sono alcuni valori delle due quantità in funzione di m. Osserviamo che sebbene $\frac{175}{175+2^{m+1}}$ tenda a zero molto velocemente, essa non raggiungerà mai la velocità di 2^{-m}.*

m	2^{-m}	$\frac{175}{175+2^{m+1}}$
1	.500	.978
2	.250	.956
5	$.312 \times 10^{-1}$	.732
30	$.931 \times 10^{-9}$	$.815 \times 10^{-7}$
50	$.888 \times 10^{-15}$	$.777 \times 10^{-13}$
100	$.789 \times 10^{-30}$	$.690 \times 10^{-28}$

3.2 Algoritmo di Miller-Rabin

Vediamo ora un altro algoritmo Monte-Carlo per la composizione dei numeri, esso è anche considerato il più forte test di primalità probabilistico.
Ovviamente è di tipo polinomiale, la sua complessità è $O(\log n)^3$ che è la stessa per il test di Solovay-Strassen. L'algoritmo è stato sviluppato da Rabin ed è basato su una teoria sviluppata da Miller [3, 4].

Miller-Rabin(p,1/4)

1. **Set** $p - 1 = 2^b m$, m è dispari
2. Sceglie casualmente a $a < p$
3. **Set** $z = a^m \bmod p$
4. **if** $z \equiv 1 \bmod p$ **then** p *è primo* **quit**
5. **for** $i = 0$ **to** $b - 1$ **do**
6.   **if** $z \equiv -1 \bmod p$ **then** p *è primo* **quit**
7.   **else** $z = z^2 \bmod p$
8. *risponde p è composto*

Esempio 8 (Con un numero composto) *Prima di esaminare l'esempio, osserviamo che b è la più grande potenza di 2 che divide $p-1$. Supponiamo che $p = 21$ allora $b = 2$ ed $m = 5$. Nel passo 2 dell'algoritmo viene scelto a casualmente, poniamo $a = 19$; in 3 viene calcolato*

$z = 10$, poi si testa il valore di z dato che $z \not\equiv 1 \bmod p$ allora si dà inizio al loop for: per $i = 0$ $z \not\equiv -1 \bmod p$ e quindi il valore di z viene aggiornato alla riga 7, così da avere $z = 16$, per $i = 1$ ancora una volta $z \not\equiv -1 \bmod p$ e di nuovo z viene aggiornato diventando $z = 4$. A questo punto si esce dal loop e al passo 8 l'algoritmo risponde che p è composto.

Esempio 9 (Con un numero primo) *Questa volta supponiamo che $p = 17$, $b = 4$, $m = 1$ ed $a = 15$. Nel passo 3 viene calcolato $z = 15$, poi siccome $z \not\equiv 1 \bmod p$ allora si dà inizio al loop for: per $i = 0$ $z \not\equiv -1 \bmod p$, allora il valore di z viene aggiornato e $z = 4$, anche per $i = 1$ $z \not\equiv -1 \bmod p$ così $z = 16$, nell'ultima iterazione per $i = 2$, $16 \equiv -1 \bmod 17$ e quindi nel passo 6 l'algoritmo risponde che p è primo e si ferma.*

Mostriamo ora che questo algoritmo non può rispondere che n *è composto* se n è primo, cioè l'algoritmo è yes-biased.

Teorema 19 (Miller-Rabin) *L'algoritmo di Miller-Rabin per i numeri composti è di tipo Monte-Carlo yes-biased.*

Dimostrazione Supponiamo per assurdo che l'algoritmo risponde n *è composto* per qualche n primo.
Poichè l'algoritmo risponde n *è composto*, esso deve trovarsi nel caso in cui $a^m \not\equiv 1 \bmod n$.
Consideriamo la sequenza di valori testati nel passo 4 dell'algoritmo. Poichè b è un quadrato, in ogni iterazionedel ciclo for, testiamo i valori a^m, a^{2m}, $\ldots$, $a^{2^{k-1}m}$.
Dato che l'algoritmo risponde n *è composto* concludiamo che $a^{2^i m} \not\equiv -1 \bmod n$ per $0 \leq i \leq k-1$ Ora usando l'assunzione che n é primo, il teorema di Fermat ci dice che $a^{2^k m} \equiv 1 \bmod n$ poichè $n - 1 = 2^k m$, allora $a^{2^{k-1}m}$ è una radice quadrata di $1 \bmod n$. Siccome n è primo ci sono solamente due radici quadrate di $1 \bmod n$ e sono ± 1.
Abbiamo che $a^{2^{k-1}m} \not\equiv -1 \bmod n$ segue allora che $a^{2^{k-1}m} \equiv 1 \bmod n$; cosi $a^{2^{k-2}m}$ deve essere una radice quadrata di 1. Ripetendo più volte l'argomento $a^{2^{k-2}m} \equiv 1 \bmod n$, si ottiene $a^m \equiv 1 \bmod n$, ma questa è una contraddizione, poichè in questo caso l'algoritmo avrebbe dovuto rispondere n *è primo*.
Errore probabilistico: L'errore probabilistico di questo algoritmo è al più 1/4.

3.3 Algoritmo di Lehmann

Un altro semplice algoritmo che testa la primalità di un numero è stato realizzato da Lehmann [5]. Un'applicazione di tale algoritmo di tipo probabilistico viene eseguita k volte ed è basata sulla seguente idea:
Sia p un intero dispari, a un intero qualsiasi, $a^{(p-1)/2} \in \{1, -1\}$ se e solo se p è primo.

Lehmann(p)

1. Sceglie un numero p da testare
2. Per essere sicuro che p non è divisibile per i primi più piccoli, va a testare 2, 3, 5, 7 e 11 così da aumentare anche la velocità dell'algoritmo
3. Sceglie k numeri casuali $a_1, \ldots, a_k$
4. Calcola $a_i^{(n-1)/2} \bmod p \;\; \forall i = 1...k$
5. Se almeno uno degli a_i calcolati è diverso da ± 1, allora *p è composto*
6. Se tutti i valori calcolati sono uguali a ± 1 e il valore -1 è stato trovato almeno una volta, allora *p è primo*.
7. Se tutti i valori determinati sono uguali ad 1, allora *allora p è composto*

Nel passo 6 p è composto con probabilità maggiore di $1 - 2^{-k}$. Nel passo 7 p è primo con probabilità maggiore di $1 - 2^{-k}$.

3.3.1 Confronto tra l'algoritmo di Lehmann e quello di Solovoy-Strassen

Questo algoritmo se confrontato con quello di Solovay-Strassen, risulta essere più efficiente, perchè non prevede il calcolo del simbolo di Jacobi, la probabilità di errore di entrambi, nel caso peggiore è la stessa. Lo svantaggio dell'algoritmo di Lehmann è che entrambe le conclusioni "composto" e "primo" possono essere inesatte, mentre in quello di Solovay-Strassen il risultato "composto" è totalmente attendibile.

3.4 Metodo dei primi forti

Spesso la letteratura dell'RSA suggerisce che i primi forti potrebbero essere usati per p e q. Questi sono numeri primi con particolari proprietà che rendono il prodotto n difficile da fattorizzare attraverso specifici metodi di fattorizzazione. Tra le proprietà suggerite (vedere [6, 7]) abbiamo:

1. Il gcd tra $(p-1)$ e $(q-1)$ sarà piccolo
2. Entrambi $p-1$ e $q-1$ hanno fattori primi molto grandi che sono rispettivamente p' e q'
3. Entrambi $p'-1$ e $q'-1$ hanno fattori primi molto grandi che sono rispettivamente p'' e q''
4. Entrambi $(p-1)/2$ e $(q-1)/2$ saranno primi [8].

4 Fattorizzazione

Fattorizzare un numero n significa trovare i suoi fattori primi. Il problema della fattorizzazione è uno dei più vecchi nella teoria dei numeri.
Abbiamo già visto alcuni tra i migliori algoritmi per testare la primalità di un numero.
In questa sezione faremo una discussione informale sui migliori algoritmi di fattorizzazione

conosciuti.
Attualmente due sono gli algoritmi capaci di fattorizzare efficientemente numeri molto grandi:

- **Quadratic Sieve** [9, 12]. Questo è l'algoritmo più veloce conosciuto per numeri con meno di 150 cifre decimali ed è stato molto usato. Una versione più veloce di questo algoritmo è chiamata **Multiple Polynomial Quadratic Sieve.** La versione più veloce di questo algoritmo è **Double Large Prime Variation of the Multiple Polynomial Quadratic Sieve**.

- **Number Field Sieve (NFS)** [13]. Questo è l'algoritmo di fattorizzazione conosciuto più veloce. Quando fu proposto non era molto pratico, ma c'è stato un cambiamento dovuto alle implementazioni degli ultimi anni.
L' algoritmo Multiple Polynomial Quadratic Sieve è più veloce per numeri più piccoli (da 110 a 135 cifre decimali a secondo delle implementazioni).

Esistono altri algoritmi di fattorizzazione che però sono stati sostituiti da questi due menzionati sopra:

- **Elliptic Curve Method (ECM)** [11]. Questo metodo è stato molto usato per trovare fattori di numeri a 38 cifre, ma non più grandi. Per numeri più grandi, gli altri algoritmi sono più veloci.

- **Trial Division.** Questo è il più vecchio algoritmo di fattorizzazione, ed implica il controllo di ogni numero primo minore o uguale della radice quadrata del numero candidato. Se $n < 10^{12}$, questo è un metodo di fattorizzazione ragionevole, ma per n più grande abbiamo tecniche più sofisticate.

- **p-1 Method** [10]. Questo algoritmo è utile quando il numero candidato ha un fattore p tale che $p-1$ ha solo fattori piccoli.

Vedremo in particolare due metodi: **p-1 Method** e **Quadratic Sieve.**

4.1 Il Metodo p-1

Questo algoritmo ha due input: l'intero (dispari) n che deve essere fattorizzato, e un limite B. Supponiamo che p sia un primo che divide n, e $q \leq B$ tale che $q|(p-1)$.
Allora si deve avere che $(p-1)|B!$.
Alla fine del ciclo for (passo 2.), $a \equiv 2^{B!} \bmod n$,e poichè $p|n$ avremo che $a \equiv 2^{B!} \bmod p$.
E per il teorema di Fermat si verifica che $2^{p-1} \equiv 1 \bmod p$.
Poichè $(p-1)|B!$ abbiamo che $a \equiv 1 \bmod p$ (nel passo 3.).
Quindi $(a-1) \equiv 0 \bmod p$.
Così, nel passo 4., $p|(a-1)$ ma $p|n$, quindi $p|d = \gcd(a-1, n)$.
L' intero d sarà un divisore non banale di n (tranne se $a = 1$ nel passo 3.).
Avendo trovato un fattore non banale d, procederemo per cercare di fattorizzare d e $\frac{n}{d}$ se sono composti.

p-1 Method(n,B)

1. **Set** $a = 2$
2. **for** $j = 2$ **to** B **do**
   $a = a^j \bmod n$
3. $d = \gcd(a-1, n)$
4. **if** $1 < d < n$ **then** "d è un fattore di n"
   else
   "non è stato trovato nessun fattore di n (fallimento)"

Esempio 10 *Vogliamo fattorizzare* `n`= 15770708441. *Se applichiamo l'algoritmo* **p-1** *con* `B`= 180, *troviamo che* `a`= 11620221425 *nel passo 3., e* `d` *risulta essere* 135979.
Infatti, la completa fattorizzazione di n *in primi è* $15770708441 = 135979 \times 115979$.
In questo caso la fattorizzazione ha successo poichè 135978 *ha solo fattori primi "piccoli":* $135978 = 2 \times 3 \times 131 \times 173$.
Quindi, prendendo `B`≥ 173, *sarà il caso che* 135978|`B!`, *come desiderato.*

Adesso analizziamo la complessità del metodo.

Nell' algoritmo ci sono $B-1$ esponenzazioni modulari, ognuna delle quali richiede al più $\log_2 B$ moltiplicazioni modulari usando square-and-multiply.
Il calcolo del gcd richiede $O(\log n)^3)$ operazioni su bit. Quindi la complessità dell' algoritmo è $O(B \log B (\log n)^2 + (\log n)^3)$, considerando le operazioni su bit.
Se B è $O((\log n)^i)$ per un intero fissato i, allora l' algoritmo è polinomiale: comunque, per tale scelta di B la probabilità di successo sarà molto piccola.
D' altra parte se aumentiamo la grandezza di B drasticamente, diciamo $\sqrt{n}$, l' algoritmo avrà successo, ma non sarà più veloce della Trial Division.
Così lo svantaggio di questo metodo è che richiede che n abbia un fattore primo p tale che $p-1$ abbia solo fattori primi piccoli.
Sarebbe molto facile costruire un RSA modulo $n = pq$ che resista alla fatto rizzazione con questo metodo. Si potrebbe partire trovando un primo grande p_1 tale che $p = 2p_1 + 1$ sia ancora primo, e un primo grande q_1 tale che $q = 2q_1 + 1$ sia ancora primo (usando gli algoritmi Montecarlo per controllare la primalità).
Allora l' RSA modulo $n = pq$ sarà resistente alla fattorizzazione usando il $(p-1)$ method.

Osservazione: più efficiente è l' algoritmo delle **Elliptic Curve**, sviluppato da Lenstra nella metà degli anni '80, ed è una generalizzazione del $(p-1)$ Method. Qui non ne discuteremo la teoria, ma diciamo che il successo delle curve ellittiche dipende dalle situazioni più probabili che un intero vicino a p abbia solo fattori piccoli.
Mentre il $p-1$ Method dipende da una relazione che si ottiene nel gruppo $\mathbf{Z_p}$, il metodo delle curve ellittiche coinvolge gruppi definiti sulle curve ellittiche modulo p.

4.2 Quadratic Sieve

Questo algoritmo fu inventato da Pomerance agli inizi dgli anni '80.
È basato su un' idea molto semplice: se riusciamo a trovare x e y tali che

$$x^2 \equiv y^2 \bmod n$$

e

$$x \neq \pm y \bmod n$$

allora $\gcd(x-y, n)$ è un fattore non banale di n.
Il metodo usa una *factor base*, che è un insieme B di primi piccoli.
Ricaviamo diversi interi x tali che tutti i fattori primi di $x^2 \bmod n$ stiano nella *factor base* B. (Come questo sia fatto lo vedremo dopo).
L' idea è di prendere il prodotto di diversi x in modo che ogni primo nella *factor base* venga usato un numero pari di volte. Così avremo una congruenza del tipo desiderato $x^2 \equiv y^2 \bmod n$, che può portarci ad una fattorizzazione di n.

Esempio 11 *Consideriamo* `n`$= 15770708441$.
Sia `B`$= \{2, 3, 5, 7, 11, 13\}$ *la factor base. Consideriamo le tre congruenze:*

$$8340934156^2 \equiv 3 \times 7 \bmod n$$

$$12044942944^2 \equiv 2 \times 7 \times 13 \bmod n$$

$$2773700011^2 \equiv 2 \times 3 \times 13 \bmod n$$

Se prendiamo il prodotto di queste tre congruenze, abbiamo:

$$(8340934156 \times 12044942944 \times 2773700011)^2 \equiv (2 \times 3 \times 7 \times 13)^2 \bmod n$$

Riducendo l'espressione nella parentesi modulo n, abbiamo

$$9503435785^2 \equiv 546^2 \bmod n$$

Quindi calcolando $\gcd(9503435785 - 546, 15770708441)$, *troviamo il fattore 115759 di n.*

Supponiamo che $B = \{p_1, \ldots, p_B\}$ sia la *factor base*. Sia C leggermente più grande di B (diciamo $C = B + 10$) e supponiamo di aver ottenuto C congruenze:

$$x_j^2 \equiv p_1^{\alpha_{1_j}} \times p_2^{\alpha_{2_j}} \times \cdots \times p_B^{\alpha_{B_j}} \bmod n \quad \textit{per } 1 \leq j \leq C.$$

Per ogni j, consideriamo il vettore:

$$a_j = (\alpha_{1_j} \bmod 2, \ldots, \alpha_{B_j} \bmod 2) \in (\mathbf{Z_2})^B$$

Se riusciamo trovare un sottoinsieme degli a_j che sommati modulo 2 danno il vettore (0,0, ...), allora il prodotto dei corrispondenti x_j userà ogni fattore di B un numero pari di volte. Vediamo con un esempio che esiste una dipendenza anche se $C < B$.

Esempio 12 *I tre vettori a_1, a_2, a_3 sono:*

$$a_1 = (0, 1, 0, 1, 0, 0)$$

$$a_2 = (1, 0, 0, 1, 0, 1)$$

$$a_3 = (1, 1, 0, 0, 0, 1)$$

È facile vedere che

$$a_1 + a_2 + a_3 = (0, 0, 0, 0, 0, 0) \bmod 2$$

Questo dà origine alla congruenza che abbiamo visto nell'esempio 1 che fattorizza n.

Osserviamo che per trovare un sottoinsieme di C vettori $a_1, \ldots, a_C$, la cui somma modulo 2 è il vettore di tutti zeri, non è niente di più che trovare una combinazione lineare (su $\mathbf{Z_2}$) di questi vettori.
Dato $C > B$, tale dipendenza lineare deve esistere e può essere trovata facilmente usando il metodo standard dell' eliminazione gaussiana.
La ragione del perchè $C > B + 1$ è che non c' è nessuna garanzia che una data congruenza $x^2 \equiv y^2 \bmod n$ produrrà la fattorizzazione di n.
Approssimativamente solo per il 50% delle volte accade che $x \equiv \pm y \bmod n$.
Ma se $C > B + 1$, allora possiamo ottenere varie di queste congruenze (che sorgono da dipendenze lineari diverse tra gli a_j).
Resta da discutere come otteniamo gli interi x_j tali che $x_j^2 \bmod n$ fattorizzino completamente sulla *factor base* B.
Un metodo usato comunemente è usare gli interi della forma:

$$x_i = i + \lfloor \sqrt{n} \rfloor \qquad i = 1, 2, \ldots$$

Allora una procedura al setaccio è usata per determinare quegli x_i che sono fattori su B. C'è naturalmente un trade-off: se $B = |B|$ è grande, allora abbiamo bisogno di più congruenze prima di poter trovare una relazione di dipendenza.
Una scelta ottimale per B è approssimativamente:

$$\sqrt{e^{\sqrt{\ln n \ln \ln n}}}$$

e questo porta ad un running time di

$$O(e^{(1+o(1))\sqrt{\ln n \ln \ln n}}).$$

4.3 Algoritmi di fattorizzazione nella pratica

I running time asintotici sono:

Quadratic sieve	$O(e^{(1+o(1))\sqrt{\ln n \ln \ln n}})$
Elliptic curve	$O(e^{(1+o(1))\sqrt{2 \ln p \ln \ln p}})$
Numer field sieve	$O(e^{(1.92+o(1))(\ln n)^{\frac{1}{3}}(\ln \ln n)^{\frac{2}{3}}})$

La notazione $o(1)$ denota una funzione di n che tende a 0 per n che tende a ∞, p denota il fattore primo più piccolo di n.
Nel caso peggiore, $p \approx \sqrt{n}$, il running time asintotico degli algoritmi quadratic sieve e elliptic curve è essenzialmente lo stesso.
Ma in tale situazione, quadratic sieve supera elliptic curve poichè le costanti nascoste sono più piccole.
Il metodo elliptic curve è più utile se i fattori primi di n sono di taglia diversa.
Un numero molto grande che fu fattorizzato usando il metodo dell'elliptic curve fu il numero di Fermat $2^{2^{11}} - 1$ nel 1988 da Brent.
Per fattorizzare i moduli dell'RSA (dove $n = pq$, p e q sono primi e sono approssimativamente della stessa grandezza) il quadratic sieve è attualmente l'algoritmo che ha più successo.
Nel 1983, il quadratic sieve fattorizzò con successo un numero con 69 cifre che era un fattore (composto) di $2^{251} - 1$ (questo calcolo fu fatto da Davis, Holdredge e Simmons).
Con il progresso i numeri sono saliti fino a 106 cifre e furono fattorizzati con questo metodo da Lenstra e Manasse, che distribuirono i calcoli a centinaia di postazioni di lavoro separate (chiamarono questo approccio "fattorizzazione tramite posta elettronica").
La fattorizzazione di un RSA-120 (che è una lista di numeri composti che sono dell'ordine di 120 cifre) richiese 825 MIPS-anni (MIPS= un milione di istruzioni per secondo) di tempo di computazione e si usò una factor base di cardinalità 245810.
Il Number Field Sieve è il più recente dei tre algoritmi, e sembra avere una grande potenzialità poichè il suo runnig time asintotico è più veloce sia del Quadratic sieve sia dell'elliptic curve.
È ancora in fase di sviluppo, ma si pensa che il Number field sieve potrebbe provare di essere più veloce per numeri che hanno più di 125-130 cifre.

Carl Pomerance [12] ha progettato una macchina di fattorizzazione modulare.
La grandezza del numero che si può fattorizzare dipende da quanto è grande la macchina che si può fornire.
Il modello di dimostrazione di Pomerance è un' implementazione di \$ 25.000 che può fattorizzare numeri di 100 cifre in due settimane.
Una macchina di \$ 10 milioni potrebbe fattorizzare un numero di 150 cifre in un anno.
Teoricamente non ci sono limiti per questo modello di fattorizzazione. Per fattorizzare un numero di 200 cifre in un anno, la macchina costerebbe \$ 100 bilioni.

5 Logaritmi Discreti

Uno dei più importanti problemi nella teoria della complessità è quello di trovare un algoritmo efficiente che calcoli il logaritmo discreto (modulo un primo p, avente come base un generatore $\alpha \in \mathbf{Z}_p^*$), poiché la sicurezza di molti sistemi di crittografia a chiave pubblica è basata su questo problema.

L'utilità del problema del logaritmo discreto nell'ambiente crittografico è dovuta al fatto che esso è l'operazione inversa dell'esponenziazione modulare, la quale è una funzione one-way. Valutare l'espressione $\alpha^a \bmod p$ è facile, mentre trovare l'elemento a tale che $\alpha^a \equiv \beta \bmod p$ è difficile, scegliendo opportuni primi p. In particolare non si conoscono algoritmi polinomiali per il problema del logaritmo discreto.

Prima di andare ad analizzare alcuni degli algoritmi che calcolano il logaritmo discreto, analizziamo in dettaglio l'operazione di esponenziazione modulare.
L'operazione $\alpha^a \bmod p$ può essere effettuata efficientemente, con il metodo di elevamento al

quadrato ripetuto, utilizzando la rappresentazione binaria di a.

Esponenziazione_modulare (α, a, p)

1. Sia $< a_n, a_{n-1}, \ldots, a_1, a_0 >$ la rappresentazione binaria di a
2. $y = 1$
3. **for** $i = n$ **to** 0 **do**
4. $\quad y = y^2 \bmod p$
5. $\quad y = y \cdot \alpha^{a_i} \bmod p$
6. **return** y

Questo algoritmo effettua $n+1 = 1+\lfloor \log_2 a \rfloor$ quadrati e $1+\lfloor \log_2 a \rfloor$ prodotti modulari. Inoltre per ottenere la rappresentazione binaria di a bisogna effettuare altre $1 + \lfloor \log_2 a \rfloor$ divisioni. Quindi, in totale vengono effettuate $O(\log a)$ operazioni aritmetiche per ogni esponenziazione modulare.

Descriveremo il problema del logaritmo discreto nel campo finito $\mathbf{Z}_p$, dove p è primo.

- **Istanza del problema**: $I = (p, \alpha, \beta)$
  dove p è primo, $\alpha \in \mathbf{Z}_p^*$ è un elemento primitivo
  (cioè un generatore di $\mathbf{Z}_p^*$) e $\beta \in \mathbf{Z}_p^*$.
- **Obiettivo**: Trovare quell'unico intero $a (0 \leq a \leq p-2)$ tale che

$$\alpha^a \equiv \beta \mod p$$

(cioè calcolare $a = \log_\alpha \beta$).

Notiamo che a è un intero modulo $p-1$ poiché, per il teorema di Fermat

$$\alpha^{p-1} \equiv 1 \bmod p \equiv \alpha^0$$

Ciò implica che a è da ricercare tra zero e $p-2$.

NOTA: Non tutti i logaritmi discreti hanno soluzione. Ad esempio, non esiste alcun x tale che

$$3^x \equiv 7 \bmod 13$$

La complessità dei migliori algoritmi conosciuti che risolvono tale problema in $\mathbf{Z}_p$ è essenzialmente la stessa dei migliori algoritmi conosciuti che risolvono il problema della fattorizzazione di un intero n della stessa taglia di p, quando n è prodotto di due primi che hanno approssimativamente la stessa lunghezza.

Tale complessità è:

$$e^{(\ln p)^{1/2} (\ln \ln p)^{1/2}} = e^{\sqrt{(\ln p)(\ln \ln p)}}$$

Vedremo in seguito un algoritmo che presenta questa complessità.

Per contrastare gli attacchi conosciuti, p dovrebbe essere di almeno 150 digit (equivalentemente 498 bit) e $p-1$ dovrebbe avere almeno un fattore primo "grande".

Infatti, Pohlig e Hellman hanno trovato un modo veloce per computare il logaritmo discreto in $\mathbf{Z}_p$, se $p-1$ ha solo fattori primi "piccoli". Per questa ragione, alcuni campi non sono usati nella crittografia.

Sono tre i gruppi principali in cui i logaritmi discreti sono di interesse per la crittografia:

- Il gruppo moltiplicativo dei campi primi: GF(p)
- Il gruppo moltiplicativo dei campi finiti di caratteristica 2: GF(2^n)
- I gruppi di curve ellittiche sui campi finiti: EC(F)

5.1 Algoritmi per il calcolo del logaritmo discreto

Esaminiamo dapprima due soluzioni di "forza bruta" per il problema del logaritmo discreto.

- Possiamo risolvere il problema tramite una ricerca esaustiva su tutti i possibili valori a. In questo caso dobbiamo effettuare, nel caso pessimo, $p-1$ esponenziazioni modulari, con una complessità in spazio $O(1)$.
- Un'alternativa al metodo descritto sopra è quella di precomputare tutti i possibili valori $\alpha^a \bmod p$ e poi ordinare le coppie (a, α^a) rispetto alla seconda coordinata, creando una tabella.
  Dopo la precomputazione, che richiede $p-1$ esponenziazioni modulari, tempo $O(p \log p)$ per l'ordinamento e spazio $O(p)$, il problema può essere risolto in tempo costante.

5.1.1 Algoritmo di Shanks

Rispetto ai due algoritmi di "forza bruta" esaminati precedentemente, lo sforzo computazionale dell'algoritmo di Shanks [15] non è tutto concentrato sul tempo o sullo spazio, ma è distribuito più uniformemente. Per questo motivo viene definito time-memory trade-off.

Sia $m = \lceil \sqrt{p-1} \rceil$

Algoritmo di Shanks

1. Computa $\alpha^{mj} \bmod p$, per $0 \le j \le m-1$
2. Ordina le m coppie ordinate (j, α^{mj}) rispetto alla seconda coordinata, ottenendo una lista $\mathcal{L}_1$
3. Computa $\beta \cdot \alpha^{-i} \bmod p$, per $0 \le i \le m-1$
4. Ordina le m coppie ordinate $(i, \beta \cdot \alpha^{-i})$ rispetto alla seconda coordinata, ottenendo una lista $\mathcal{L}_2$
5. Trova una coppia $(j, y) \in \mathcal{L}_1$ e una coppia $(i, y) \in \mathcal{L}_2$ (aventi la seconda coordinata identica)
6. $\log_\alpha \beta = mj + i \bmod (p-1)$

Per far vedere che l'algoritmo funziona, analizziamo i passi 5. e 6.

Sia

$$(j, y) \in \mathcal{L}_1 \tag{1}$$

$$(i, y) \in \mathcal{L}_2 \tag{2}$$

allora dalla (1) otteniamo	$y = \alpha^{mj} \bmod p$
e dalla (2) si ha che	$y = \beta \cdot \alpha^{-i} \bmod p$
quindi	$\alpha^{mj} \equiv \beta \cdot \alpha^{-i} \bmod p$
da cui	$\alpha^{mj+i} \equiv \beta \bmod p$
in definitiva	$mj + i \equiv \log_\alpha \beta \bmod (p-1)$

come si voleva. Viceversa, per ogni β di cui esiste il logaritmo discreto, possiamo scrivere $\log_\alpha \beta = mj + i$, dove $0 \leq j, i \leq m-1$. Quindi, la ricerca al passo 5. avrà successo.
Questo prova la correttezza dell'algoritmo.
Ovviamente, se il passo 5. fallisce, significa che il logaritmo discreto non esiste.

Adesso analizziamo la complessità dell'algoritmo di Shanks.
Una volta fissati p e α, i passi 1. e 2. possono essere eseguiti con una precomputazione, in quanto ogni volta ci interessa calcolare il $\log_\alpha \beta$ con β che varia in $\mathbf{Z}_p^*$.
Ciò comunque non fa variare la complessità asintotica dell'algoritmo.

Analizziamo la complessità passo per passo:

1. Si effettuano m esponenziazioni modulari.
2. Si effettuano $O(m \log m)$ confronti.
3. per calcolare $\alpha^{-i} \bmod p$ bisogna prima calcolare $\alpha^i \bmod p$ e poi trovare l'inverso $(\alpha^i)^{-1} \bmod p$.
   (L'inverso si può calcolare efficientemente con l'applicazione dell'algoritmo Extended_Euclid(α^i, p), il quale effettua $O(\log p)$ operazioni aritmetiche)
   Infine bisogna moltiplicare $(\alpha^i)^{-1}$ per β.
   In definitiva, in questo passo si effettuano:
   m esponenziazioni modulari
   m inversi (che equivalgono a $O(m \log p)$ operazioni aritmetiche)
   m prodotti modulari.
4. Si effettuano $O(m \log m)$ confronti.
5. questo passo può essere effettuato con uno scorrimento simultaneo delle liste $\mathcal{L}_1$ e $\mathcal{L}_2$ con $O(m)$ confronti.
6. questo passo viene effettuato in tempo $O(1)$.

Per quanto visto prima, ogni esponenziazione modulare equivale a $O(\log p)$ operazioni aritmetiche (poichè l'esponente è un numero modulo p) e quindi l'algoritmo effettua $O(m \log p)$ operazioni aritmetiche, $O(m \log m)$ confronti ed ha una complessità in spazio $O(m)$.

Esempio 13 *Sia* $p = 29$*,* $\alpha = 2$ *e* $\beta = 18$*.*
Risulta $m = \lceil\sqrt{28}\rceil = 6$*.*
Al passo 1. computiamo le coppie $(j, \alpha^{mj}) \bmod p = (j, 64^j) \bmod 29$*, per* $j = 0, 1, \cdots 5$*.*
Al passo 2. ordiniamo queste coppie ed otteniamo la seguente lista $\mathcal{L}_1$*:*

$$(0,1)\ \ (5,4)\ \ (1,6)\ \ (2,7)\ \ (3,13)\ \ (4,20)$$

Al passo 3. computiamo le coppie $(i, \beta \cdot \alpha^{-i}) \bmod p = (i, 18 \cdot 2^{-i}) \bmod 29$*, per* $i = 0, 1, \cdots 5$*.*
Al passo 4. ordiniamo queste coppie ed otteniamo la seguente lista $\mathcal{L}_2$*:*

$$(5,6)\ \ (1,9)\ \ (4,12)\ \ (0,18)\ \ (2,19)\ \ (3,24)$$

Adesso, procedendo simultaneamente attraverso le due liste ordinate $\mathcal{L}_1$ *e* $\mathcal{L}_2$*, troviamo (1,6) in* $\mathcal{L}_1$ *e (5,6) in* $\mathcal{L}_2$*.*
Per il passo 6. si ha:

$$\log_2 18 = 6 \cdot 1 + 5 = 11 \mod 28$$

5.1.2 Algoritmo di Pohlig-Hellman

Adesso esaminiamo l'algoritmo di Pohlig-Hellman [14] già citato precedentemente.

Sia

$$p - 1 = \prod_{i=1}^{k} p_i^{c_i}$$

dove i p_i sono tutti primi distinti.

Per ogni $1 \leq i \leq k$ risulta:

$$\begin{cases} p - 1 \equiv 0 \mod p_i^{c_i} \\ p - 1 \not\equiv 0 \mod p_i^{c_i+1} \end{cases}$$

L'idea dell'algoritmo è quella di calcolare i valori $a \equiv \log_\alpha \beta \bmod p_i^{c_i}$, per $i = 1, 2, \ldots, k$, ottenendo un sistema di k equazioni modulari.
Per il teorema cinese del resto, esiste un unico elemento a modulo $(p-1)$ che risolve il sistema.
Quindi, risolvendo il sistema otteniamo $a \equiv \log_\alpha \beta \bmod (p-1)$.

Facciamo vedere quali sono i passi per ottenere una singola equazione modulare del sistema.
Fissato i, $(1 \leq i \leq k)$, sia $q = p_i$ e $c = c_i$.
L'algoritmo che segue calcola il valore $a \equiv \log_\alpha \beta \bmod q^c$.
In particolare, esprimiamo a nella rappresentazione q-aria:

$$a = \sum_{j=0}^{c-1} a_j q^j \mod q^c$$

dove $(0 \leq a_j \leq q-1)$, per $j = 0, 1, \ldots, c-1$.
Poichè q e c sono dati in input all'algoritmo, lo scopo di quest'ultimo sarà quello di calcolare i valori a_j, per $j = 0, 1, \ldots, c-1$.

Algoritmo per computare $\log_\alpha \beta \bmod q^c$

1. Computa $\gamma^i = \alpha^{(p-1)i/q} \mod p$, per $(0 \le i \le q-1)$
2. Poni $j = 0$ e $\beta_j = \beta$
3. **while** $j \le c-1$ **do**
4. computa $\delta = \beta_j^{(p-1)/q^{j+1}} \mod p$
5. trova i tale che $\delta = \gamma^i$
6. $a_j = i$
7. $\beta_{j+1} = \beta_j \cdot \alpha^{-a_j q^j} \mod p$
8. $j = j + 1$

Dimostriamo la correttezza dell'algoritmo.
Posto

$$x_0 = \sum_{j=0}^{c-1} a_j q^j \mod q^c$$

osserviamo che possiamo esprimere a come

$$a = x_0 + q^c s, \text{ per qualche intero } s.$$

Per prima cosa l'algoritmo computa a_0.
Innanzitutto facciamo vedere che

$$\beta^{(p-1)/q} \equiv \alpha^{(p-1)a_0/q} \mod p \tag{3}$$

Poichè $\beta = \alpha^a \mod p$, si ha:

$$\beta^{(p-1)/q} \equiv \alpha^{(p-1)(x_0+q^c s)/q} \mod p \tag{4}$$

perciò basta mostrare che

$$\alpha^{(p-1)(x_0+q^c s)/q} \equiv \alpha^{(p-1)a_0/q} \mod p \tag{5}$$

per il teorema del logaritmo discreto, la (5) è vera se e solo se

$$\frac{(p-1)(x_0+q^c s)}{q} \equiv \frac{(p-1)a_0}{q} \mod \phi(p)$$

poichè p è primo, $\phi(p) = p-1$
e quindi la (5) è vera se e solo se

$$\frac{(p-1)(x_0+q^c s)}{q} \equiv \frac{(p-1)a_0}{q} \mod (p-1)$$

Abbiamo che:

$$\begin{aligned}
\frac{(p-1)(x_0+q^c s)}{q} - \frac{(p-1)a_0}{q} &= \frac{p-1}{q}\cdot(x_0+q^c s-a_0)\\
&= \frac{p-1}{q}\cdot(\sum_{j=0}^{c-1} a_j q^j + q^c s - a_0)\\
&= \frac{p-1}{q}\cdot(\sum_{j=1}^{c-1} a_j q^j + q^c s)\\
&= (p-1)\cdot(\sum_{j=1}^{c-1} a_j q^{j-1} + q^{c-1}s)\\
&\equiv 0 \mod (p-1)
\end{aligned}$$

quindi la (5) è vera e da questo deriva che è vera la (3).

Per $k=0$ si ha:

$$\beta_0 = \beta, \text{ per il passo 2.}$$

e

$$\delta = \beta^{(p-1)/q^{0+1}} = \beta^{(p-1)/q}, \text{ per il passo 4.}$$

inoltre, per la (3)

$$\delta = \alpha^{(p-1)a_0/q} \mod p$$

Quindi, se i è tale che $\delta = \gamma^i$, calcolato al passo 5., allora risulta:

$$\alpha^{(p-1)a_0/q} \equiv \gamma^i \equiv \alpha^{(p-1)i/q}$$

da cui

$$\frac{p-1}{q}\cdot a_0 \equiv \frac{p-1}{q}\cdot i$$

e perciò

$$a_0 = i$$

così come calcolato al passo 6.

Quindi il coefficiente a_0 è calcolato correttamente.
Se $c=1$ abbiamo terminato e risulta:

$$a = a_0 \mod q^c \equiv \log_\alpha \beta \mod q^c$$

Se $c>1$, dobbiamo determinare i coefficienti $a_1, a_2, \ldots, a_{c-1}$.

In generale, per $(1 \le k \le c-1)$ si ha:

$$\begin{aligned}
\beta_k &= \beta_{k-1}\alpha^{-a_{k-1}q^{k-1}} \mod p\\
&= \alpha^{a-a_0-a_1q-\cdots-a_{k-1}q^{k-1}} \mod p
\end{aligned}$$

inoltre $\delta = \beta_k^{(p-1)/qk+1}$

Innanzitutto facciamo vedere che

$$\beta_k^{(p-1)/qk+1} \equiv \alpha^{(p-1)a_k/q} \mod p \tag{6}$$

Poniamo

$$x_k = \log_\alpha \beta_k \mod q^c = a - a_0 - a_1 q - \cdots - a_{k-1} q^{k-1} \mod q^c$$

Allora

$$x_k = \sum_{j=k}^{c-1} a_j q^j \mod q^c$$

inoltre possiamo scrivere

$$a - a_0 - a_1 q - \cdots - a_{k-1} q^{k-1} = x_k + q^c s, \text{ per qualche intero } s.$$

Si ha:

$$\beta_k^{(p-1)/qk+1} = \alpha^{(p-1)(a-a_0-a_1q-\cdots-a_{k-1}q^{k-1})/q^{k+1}} = \alpha^{(p-1)(x_k+q^c s)/q^{k+1}} \mod p \tag{7}$$

quindi basta provare che

$$\alpha^{(p-1)(x_k+q^c s)/q^{k+1}} \equiv \alpha^{(p-1)a_k/q} \mod p \tag{8}$$

la (8) è vera se e solo se

$$\tfrac{(p-1)(x_k+q^c s)}{q^{k+1}} \equiv \tfrac{(p-1)a_k}{q} \mod (p-1)$$

Abbiamo che:

$$\begin{aligned}
\frac{(p-1)(x_k+q^c s)}{q^{k+1}} - \frac{(p-1)a_k}{q} &= \frac{(p-1)(x_k+q^c s) - (p-1)a_k q^k}{q^{k+1}} \\
&= \frac{p-1}{q^{k+1}} \cdot (x_k + q^c s - a_k q^k) \\
&= \frac{p-1}{q^{k+1}} \cdot (\sum_{j=k}^{c-1} a_j q^j + q^c s - a_k q^k) \\
&= \frac{p-1}{q^{k+1}} \cdot (\sum_{j=k+1}^{c-1} a_j q^j + a_k q^k + q^c s - a_k q^k) \\
&= (p-1) \cdot (\sum_{j=k+1}^{c-1} a_j q^{j-(k+1)} + q^{c-(k+1)} s) \\
&\equiv 0 \mod (p-1)
\end{aligned}$$

quindi la (8) è vera e da questo deriva che è vera la (6).

Per la (6)

$$\delta = \alpha^{(p-1)a_k/q} \mod p$$

Quindi, se i è tale che $\delta = \gamma^i$, allora risulta:

$$\alpha^{(p-1)a_k/q} \equiv \gamma^i \equiv \alpha^{(p-1)i/q}$$

da cui

$$\frac{p-1}{q} \cdot a_k \equiv \frac{p-1}{q} \cdot i$$

e perciò

$$a_k = i$$

Questo mostra che ogni coefficiente a_k, $(1 \leq k \leq c-1)$, è calcolato correttamente.

Una volta determinati tutti i coefficienti $a_0, a_1, \ldots, a_{c-1}$, possiamo determinare l'equazione modulare

$$a = \sum_{j=0}^{c-1} a_j q^j \mod q^c \equiv \log_\alpha \beta \mod q^c$$

Supposto che

$$p - 1 = \prod_{i=1}^{k} p_i^{c_i}$$

una volta applicato l'algoritmo illustrato per ognuno dei fattori $p_i^{c_i}$ di $p-1$, otteniamo il seguente sistema di k equazioni

$$\left\{ \; a \equiv \log_\alpha \beta \mod p_i^{c_i} \qquad i = 1, 2, \ldots, k \right.$$

Per comodità riscriviamo il sistema nella forma

$$\left\{ \; a \equiv a_i \mod p_i^{c_i} \qquad i = 1, 2, \ldots, k \right. \tag{9}$$

A questo punto possiamo applicare il teorema cinese del resto, il quale afferma che il sistema (9) ha un'unica soluzione modulo $(p-1)$.

In particolare, la soluzione è data da

$$a = \sum_{i=1}^{k} a_i \cdot m_i \cdot y_i \mod (p-1)$$

dove

$$m_i = \tfrac{p-1}{p_i^{c_i}} \quad \text{e} \quad y_i = m_i^{-1} \mod p_i^{c_i}, \text{ per } 1 \leq i \leq k.$$

Adesso analizziamo la complessità dell'algoritmo di Pohlig-Hellman.

Fissati q e c, analizziamo dapprima la complessità dell'algoritmo per computare $\log_\alpha \beta \mod q^c$. Senza perdere di generalità, possiamo considerare c costante, in quanto esso è l'esponente di uno dei fattori primi di $p-1$.

Il passo 1. effettua q esponenziazioni modulari e i passi da 3. a 8. sono eseguiti $O(c)$ volte. In ogni ciclo viene effettuata una esponenziazione modulare al passo 4. e una al passo 7., inoltre,

al passo 7. viene eseguito anche il calcolo di un inverso e di un prodotto modulare.
Quindi il lavoro dominante è quello effettuato dal passo 1. e perciò la complessità totale è di $O(q)$ esponenziazioni modulari, ovvero di $O(q \log p)$ operazioni aritmetiche.
Se k sono i distinti fattori di $p-1$, il lavoro per ottenere il sistema di equazioni modulari finale è

$$\sum_{i=1}^{k} O(p_i) \text{ esponenziazioni modulari.}$$

Se indichiamo con q il fattore primo più grande fra tutti, possiamo dire che la complessità è $O(kq) = O(q)$ esponenziazioni modulari (poichè k è costante).
Per risolvere il sistema bisogna effettuare: k divisioni per ottenere i coefficienti m_i, k inversi modulari per ottenere i coefficienti y_i ed inoltre la sommatoria finale. Tutto questo lavoro si può riassumere in $O(\log p)$ operazioni aritmetiche a causa degli inversi modulari e quindi la complessità totale dell'algoritmo di Pohlig-Hellman è di $O(q)$ esponenziazioni modulari e cioè di $O(q \log p)$ operazioni aritmetiche, dove $q = p_i^{c_i}$, per un certo i tale che: $p_i^{c_i} \geq p_j^{c_j}$, per $1 \leq j \leq k$.
Da questa analisi si comprende che se $p-1$ ha solo fattori primi piccoli, l'algoritmo di Pohlig-Hellman riesce a computare in tempo ragionevole il logaritmo discreto.

NOTA: Ovviamente questo algoritmo può essere applicato solo quando si conosce la fattorizzazione di $p-1$.

Esempio 14 *Sia $p = 29$, $\alpha = 2$ e $\beta = 18$. Allora $p - 1 = 28 = 2^2 \cdot 7^1$ e quindi $p_1 = 2$, $c_1 = 2$, $p_2 = 7$ e $c_2 = 1$.*
Dobbiamo calcolare $a = \log_2 18$. Procediamo prima computando $a \bmod 4$ e poi $a \bmod 7$, utilizzando l'algoritmo per computare $\log_\alpha \beta \bmod q^c$.
Partiamo ponendo $q = p_1 = 2$ e $c = c_1 = 2$.
Al passo 1. vengono computati i valori:

$$\gamma^0 = 2^{28 \cdot 0/2} = 2^0 = 1$$

e

$$\gamma^1 = 2^{28 \cdot 1/2} = 2^{14} \equiv 28 \mod 29$$

Al passo 4. della prima iterazione del ciclo while viene computato

$$\delta = \beta^{28/2^{0+1}} = 18^{14} \equiv 28 \mod 29$$

Quindi, al passo 6.

$$a_0 = 1$$

poichè $\delta = \gamma^1$.
Al passo 7. viene computato

$$\beta_1 = \beta_0 \cdot \alpha^{-a_0 q^0} = \beta \cdot \alpha^{-1} \equiv 9 \mod 29$$

Al passo 4. della seconda e ultima iterazione del ciclo while, viene computato

$$\delta = \beta_1^{28/4} = 9^7 \equiv 28 \mod 29$$

Quindi, al passo 6.

$$a_1 = 1$$

poichè $\delta = \gamma^1$.
A questo punto bisogna computare

$$a = (a_0 q^0 + a_1 q) \mod q^c \equiv 3 \mod 4$$

Adesso poniamo $q = p_2 = 7$ *e* $c = c_2 = 1$.
Poichè $c = 1$*, il ciclo while viene eseguito una sola volta e al passo 4. viene computato*

$$\delta = \beta^{28/7} = 18^4 \equiv 25 \mod 29$$

Adesso vediamo i valori computati al passo 1. dell'algoritmo:

$$\begin{aligned}
&\gamma^0 = 1\\
&\gamma^1 = \alpha^{28/7} = 2^4 \equiv 16 \mod 29\\
&\gamma^2 = \alpha^{56/7} = 2^8 \equiv 24 \mod 29\\
&\gamma^3 = \alpha^{84/7} = 2^{12} \equiv 7 \mod 29\\
&\gamma^4 = \alpha^{112/7} = 2^{16} \equiv 25 \mod 29
\end{aligned}$$

quindi, per il passo 6., risulta

$$a_0 = 4$$

e perciò

$$a \equiv 4 \mod 7$$

Una volta terminata l'applicazione dell'algoritmo per computare $\log_\alpha \beta \mod q^c$*, bisogna risolvere il sistema (9), che nel nostro caso è:*

$$\begin{cases} a \equiv 3 \mod 4 \\ a \equiv 4 \mod 7 \end{cases}$$

A questo punto, per il teorema cinese del resto otteniamo la soluzione

$$a \equiv 11 \mod 28$$

e cioè

$$\log_2 18 \equiv 11 \ \textit{in}\ \mathbf{Z}_{29}$$

5.1.3 Index Calculus Method

In questa sezione presentiamo il metodo più veloce, tra quelli conosciuti, per il calcolo del logaritmo discreto. In particolare, esso ha una complessità molto simile alla complessità dei migliori algoritmi per la fattorizzazione.
Le idee alla base del metodo sono dovute a Western e Miller [20] (vedere anche [18]).
L'algoritmo fu inventato indipendentemente da Adleman [16], Merkle [17] e Pollard [19] e la sua complessità computazionale è stata parzialmente analizzata da Adleman [16].
A differenza delle precedenti tecniche, l'index calculus method è probabilistico piuttosto che deterministico.
Il metodo utilizza un insieme $\mathcal{B}$ di numeri primi piccoli, detto *factor base*.
Supponiamo di prendere $\mathcal{B} = \{p_1, p_2, \ldots, p_B\}$.
Il metodo si divide in due fasi e la prima viene effettuata attraverso una precomputazione.

FASE 1. Trovare $\log_\alpha p_i \mod (p-1)$, per $1 \le i \le B$.
Cioè calcolare il logaritmo discreto di tutti gli elementi nell'insieme $\mathcal{B}$ (*factor base*).

FASE 2. Dato $\beta \in \mathbf{Z}_p^*$ qualsiasi, calcolare $\log_\alpha \beta \mod p$ utilizzando i logaritmi calcolati nella Fase 1.

La Fase 1. viene realizzata attraverso la costruzione di $C = B + 10$ congruenze modulo p della forma:

$$\alpha^{x_j} \equiv p_1^{a_{1j}} p_2^{a_{2j}} \cdots p_B^{a_{Bj}} \mod p, \ 1 \le j \le C \tag{10}$$

con i coefficienti $a_{ij} \ge 0$, per $1 \le j \le C$.
Queste congruenze possono essere scritte nella forma:

$$x_j \equiv a_{1j} \log_\alpha p_1 + \cdots + a_{Bj} \log_\alpha p_B \mod (p-1), \ 1 \le j \le C \tag{11}$$

infatti, dalla (10) deriva che

$$\begin{aligned} x_j = \log_\alpha \alpha^{x_j} &\equiv \log_\alpha(p_1^{a_{1j}} \cdots p_B^{a_{Bj}}) \\ &\equiv \log_\alpha p_1^{a_{1j}} + \cdots + \log_\alpha p_B^{a_{Bj}} \\ &\equiv a_{1j} \log_\alpha p_1 + \cdots + a_{Bj} \log_\alpha p_B \mod \phi(p) \end{aligned}$$

Una volta conosciuto x_j e i coefficienti a_{ij}, $(1 \le i \le B\ ,\ 1 \le j \le C)$, le (11) costituiscono un sistema di C congruenze nelle B incognite $\log_\alpha p_i$, $(1 \le i \le B)$.
Se questo sistema ha un'unica soluzione $\mod (p-1)$, allora risolvendolo abbiamo a disposizione i valori $\log_\alpha p_i$, per $i = 1, 2, \ldots, B$. Bisogna perciò affrontare il problema di generare le congruenze (10).
Un modo elementare per fare questo è scegliere un valore x a caso, computare $\alpha^x \mod p$ e poi controllare se $\alpha^x \mod p$ ha tutti i suoi fattori in $\mathcal{B}$ (*factor base*).

Nota: l'ultima operazione può essere fatta verificando se è possibile ottenere 1 dividendo $(\alpha^x \mod p)$ per gli elementi in $\mathcal{B}$ presi un numero arbitrario di volte.

Osservazione: In realtà, per trovare i logaritmi discreti dei numeri primi nel *factor base*, basterebbe generare B congruenze indipendenti tra loro, ma poichè non è possibile effettuare a priori un controllo sull'indipendenza delle congruenze generate, si preferisce costruire un numero di congruenze C maggiore di B. Se non fosse adottato questo accorgimento si potrebbe facilmente generare un sistema con più di una soluzione. Quindi, le congruenze aggiuntive servono per restringere lo spazio delle soluzioni.

Una volta completata la Fase 1., possiamo realizzare la Fase 2. attraverso un algoritmo probabilistico di tipo Las Vegas.

Algoritmo

1. Scegli un intero casuale s, $(1 \leq s \leq p-1)$
2. Computa $\gamma = \beta\alpha^s \mod p$
3. Tenta di fattorizzare γ con i primi in $\mathcal{B}$
   (ad esempio attraverso divisioni successive)
4. Se γ può essere fattorizzato su $\mathcal{B}$, allora otteniamo una congruenza della forma:
   $$\beta\alpha^s \equiv p_1^{c_1} p_2^{c_2} \cdots p_B^{c_B} \mod p$$
   da cui
   $$\log_\alpha(\beta\alpha^s) \equiv \log_\alpha(p_1^{c_1} p_2^{c_2} \cdots p_B^{c_B}) \mod (p-1)$$
   che equivale a
   $$\log_\alpha \beta \equiv c_1 \log_\alpha p_1 + \cdots + c_B \log_\alpha p_B - s \mod (p-1)$$
5. Poichè l'unica incognita nell'ultima congruenza del passo 4. è $\log_\alpha \beta$, possiamo facilmente calcolarla risolvendo la congruenza.

Adesso analizziamo la complessità dell'index calculus method.

Sono state fatte analisi euristiche di varie versioni dell'algoritmo. Il risultato è che sotto assunzioni ragionevoli, il running time asintotico della precomputazione è

$$O(e^{(1+O(1))\sqrt{(\ln p)(\ln \ln p)}})$$

e il tempo per calcolare un singolo logaritmo discreto è

$$O(e^{(1/2+O(1))\sqrt{(\ln p)(\ln \ln p)}})$$

Esempio 15 *Sia* $p = 10007$, $\alpha = 5$ *e prendiamo* $\mathcal{B} = \{2, 3, 5, 7\}$.
Poichè $\log_5 5 = 1$, *dobbiamo solo calcolare il logaritmo discreto dei primi* $2, 3$ *e* 7.
Alcuni esempi di esponenti x_j *di* α *adatti sono:* $x_1 = 4063$, $x_2 = 5136$, $x_3 = 9865$.
Per $j = 1$ *abbiamo:*

$$5^{4063} \mod 10007 = 42 = 2 \cdot 3 \cdot 7$$

da cui otteniamo la congruenza

$$\log_5 2 + \log_5 3 + \log_5 7 = 4063 \mod 10006$$

Per $j = 2$ *abbiamo:*

$$5^{5136} \mod 10007 = 54 = 2 \cdot 3^3$$

da cui otteniamo la congruenza

$$\log_5 2 + 3\log_5 3 = 5136 \mod 10006$$

Per $j = 3$ *abbiamo:*

$$5^{9865} \mod 10007 = 189 = 3^3 \cdot 7$$

da cui otteniamo la congruenza

$$3\log_5 3 + \log_5 7 = 9865 \mod 10006$$

A questo punto abbiamo ottenuto tre congruenze in tre incognite e il sistema ammette un'unica soluzione modulo 10006*, che risulta essere:*

$$\begin{cases} \log_5 2 = 6578 \\ \log_5 3 = 6190 \\ \log_5 7 = 1301 \end{cases}$$

Terminata la Fase 1., passiamo alla Fase 2.
Sia $\beta = 9451$.
Supponiamo che l'algoritmo scelga $s = 7736$ *e computiamo*

$$\gamma = \beta\alpha^s = 9451 \cdot 5^{7736} \mod 10007 = 8400$$

Poichè $8400 = 2^4 \cdot 3^1 \cdot 5^2 \cdot 7^1$, *otteniamo:*

$$\begin{aligned} \log_\alpha \beta &= \log_5 9451 \\ &= 4\log_5 2 + \log_5 3 + 2\log_5 5 + \log_5 7 - s \mod 10006 \\ &= 4 \cdot 6578 + 6190 + 2 \cdot 1 + 1301 - 7736 \mod 10006 \\ &= 6057 \end{aligned}$$

Riferimenti

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest *Introduzione agli algoritmi*, v. 3, 1995, pp. 759-797.

[2] R. Solovay e V. Strassen, *A Fast Monte Carlo Test for Primality*, SIAM Journal on Computing,v.6, Mar 1977, pp.84-85; erratum in ibid, v.7, 1978, p.118.

[3] G. L. Miller, *Riemann's Hypothesis and Tests for Primality*, Journal of Computer Sistems Science, v.13, n. 3, Dec 1976, pp.300-317.

[4] M. O. Rabin, *Probabilistic Algorithm for Testing Primality*, Journal of Number Theory, v.12, n.1, Feb 1980, pp.128-138.

[5] D. J. Lehmann, *On Primality Tests*, SIAM Journal on Computing, v.11, n.2, May 1982, pp.374-375.

[6] J. A. Gordon, *Strong Primes Are Easy to Find*, Advances in Criptology-EUROCRYPT '84 Proceedings, Berlin:Springer-Verlag, 1985, pp.216-223.

[7] R. Rivest, A. Shamir, e Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the AMC, v.21, n.2, Feb 1978, pp.120-126.

[8] G. R. Blakley, *Safeguarding Cryptographic Keys*, Proceedings of the National Computer Conference, 1979, American Federation of Information Processing Societies, v.48, 1979, pp.242-268.

[9] C.Pomerance, "The Quadratic Sieve Factoring Algorithm", *Advance in Cryptology-EUROCRYPT '84 Proceedings*, Berlin: Springer-Verlg, 1985.

[10] J.M.Pollard, "Theorems on factorization and primality testing", Proc. Cambridge Philos Soc. 76 (1974).

[11] H.W.Lenstra, *Factoring integers with elliptic curves.*

[12] C.Pomerance, J.W.Smith, R.Tuler "A Pipe-Line Architecture for Factoring Large Integers with the Quadratic Sieve Algorithm", *SIAM Journal on Computing*, v.17, Apr 1988.

[13] A.K.Lenstra, H.W.Lenstra, Jr, M.S.Manasse, and J.M.Pollard, "The Number Field Sieve", *Proceedings of the 22nd ACM Symposium on the Theory of the Computing*, 1990.

[14] S. C. Pohlig and M. E. Hellman, "An improved Algorithm for Computing Logarithms in GF(p) and Its Cryptografic Significance", *IEEE Transactions on information Theory*, v.24, n.1, Jan 1978, pp.106-111.

[15] D. E. Knuth, *The art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[16] L. M. Adleman, "A subexponential algorithm for the discrete logarithm problem with applications to cryptography, *Proc. 20th IEEE Found. Comp. Sci. Symp.* (1979), 55-60.

[17] R. Merkle, "Secrecy, authentication, and public key system", Ph.D. dissertation, Dept. of Electrical Engineering, Stanford Univ., 1979.

[18] J. C. P. Miller, "On factorization, with a suggested new approach", *Math. Comp. 29* (1975), 155-172.

[19] J. Pollard, "Monte Carlo methods for index computations (mod p), *Math. Comp. 32* (1978), 918-924.

[20] A. E. Western and J. C. P. Miller, *Tables of Indices and Primitive Roots*, Royal Society Mathematical Tables, vol. 9, Cambridge Univ. Press, 1968.