NOTE DI CRITTOGRAFIA

A CHIAVE PUBBLICA

Fascicolo 1. Prerequisiti di Matematica

Versione 1.0

Francesco Pappalardi

10 maggio 2003

Indice

Introduzione

Queste note sono estratte dai corsi di MA2 e CR1 tenuti all'Università Roma Tre negli anni dal 1998 al 2002.

L'obiettivo è di produrre un testo in italiano per un corso di crittografia a chiave pubblica per studenti del secondo o terzo anno di studio.

È previsto che gli studenti abbiano già seguito un corso di algebra.

Questo è il primo di cinque fascicoli che riguarderanno i seguenti argomenti:

1. Prerequisiti di Matematica.
2. Il crittosistema RSA.
3. Introduzione ai campi finiti.
4. Logaritmi discreti e crittosistemi derivati.
5. Le curve ellittiche in crittografia.

La crittografia a chiave pubblica

La crittografia è una scienza antichissima che avuto importanti sviluppi recenti. Si dice che anche Giulio Cesare per comunicare con le sue truppe in Gallia usasse un sistema crittografico. La sua tecnica consisteva nell'associare a ciascuna lettera dell'alfabeto latino la lettera ottenuta traslando di tre, cioè applicando la sostituzione

$$\begin{array}{cccccccccccc} \mathtt{A} & \mathtt{B} & \mathtt{C} & \mathtt{D} & \mathtt{E} & \mathtt{F} & \cdots & \mathtt{U} & \mathtt{V} & \mathtt{Z} \\ \mathtt{D} & \mathtt{E} & \mathtt{F} & \mathtt{G} & \mathtt{H} & \mathtt{I} & \cdots & \mathtt{A} & \mathtt{B} & \mathtt{C} \end{array}$$

Secondo il metodo di Cesare, la cifratura del messaggio "*morte ai Galli*" è "*ptuzh dn Ldoon*".

Oggi ci riferiamo ad un "*crittosistema*" come al dato di

$$(\mathcal{P}, \mathcal{C}, \mathcal{K}, E, D)$$

dove

- $\mathcal{P}$ è l'insieme (o spazio) dei *messaggi in chiaro*;
- $\mathcal{C}$ è l'insieme (o spazio) dei *messaggi cifrati*;
- $\mathcal{K}$ è l'insieme (o spazio) delle *chiavi*;
- E è la *funzione di cifratura*

$$E : \mathcal{P} \times \mathcal{K} \to \mathcal{C}, (p, k) \mapsto c = E_k(p);$$

- D è la *funzione di decifratura*

$$E : \mathcal{C} \times \mathcal{K} \to \mathcal{P}, (c, k) \mapsto D_k(c).$$

Per il momento l'unica ipotesi che facciamo è per ogni $p \in \mathcal{P}$ risulti

$$D_k(E_k(p)) = p.$$

Come vedremo lo spazio delle chiavi potrebbe essere complesso e gestito in modi molto diversi in diverse circostanze. Per questa ragione è naturale dividere la crittografia in due branche: **crittografia a chiave privata** e **crittografia a chiave pubblica**.

La differenza tra questi due volti della crittografia sta nel fatto che con un sistema a chiave privata è previsto che due soggetti che vogliono comunicare utilizzando il sistema debbano essersi scambiati la chiave in un momento precedente alla comunicazione. Invece, un crittosistema a chiave pubblica non richiede necessariamente che i due soggetti si siano mai incontrati.

La crittografia a chiave privata prevede due tipi di algoritmi:

- Algoritmi a blocchi (*Block Cipher*);
- Algoritmi a flusso (*Stream Cipher*).

Esempi di algoritmi a blocchi sono il celebre **DES** (Data encryption standard) e il recente **AES** (Advanced encryption standard). Consistono nel cifrare contemporaneamente blocchi di bit di testo in chiaro, tipicamente 64 bits.

Veniamo ora alla crittografia a chiave pubblica a cui sono dedicate queste note.

> *Ogni crittosistema a chiave pubblica basa la propria sicurezza su un problema matematico MOLTO difficile da risolvere.*

Gli algoritmi di crittografia a chiave pubblica si possono dividere in famiglie a seconda del problema matematico su cui basano la propria sicurezza. In queste note considereremo le seguenti famiglie:

- **RSA** introdotti da R. Rivest, Shamir e L. Adleman nel 1977;
- **DL** (logaritmi discreti in $\mathbb{F}_p$) introdotti da Diffie e Hellman nel 1977;
- **ECC** (Crittosistemi con curve ellittiche) introdotti da N. Koblitz all'inizio degli anni 80;
- **KNAPSACK** (zainetti) introdotti da Merkle e Hellman nel 1979;
- **NTRU** introdotti da Hoffstein, Silverman e Phfipher all'inizio degli anni 90.

La classificazione precedente è arbitraria e sicuramente non è condivisa da alcuni esperti del settore. Tuttavia queste note seguono questa classificazione.

Per questo primo fascicolo ci siamo ispirati ai testi di N. Koblitz [3] e D. Stinson [8]. Altri testi in cui trovare lo stesso materiale sono [4], [10] e [7].

Capitolo 1

Tempo di esecuzione di un algoritmo

1.1 Notazioni

Sia $n \in \mathbb{N}$. Con l'espressione

$$n = (d_{k-1}, d_{k-2}, \ldots, d_1, d_0)_b$$

indichiamo *l'espansione in base* b *di* n ($b \geq 2$ è un intero). Questo significa che

1. $0 \leq d_i < b$ (per ogni $i = 0, \ldots, k-1$);
2. $d_{k-1} \neq 0$;
3. $n = \sum_{i=0}^{k-1} d_i b^i$.

Poniamo inoltre $0 = (0)_b$ per ogni $b \geq 2$. È ben noto che ogni intero si può espandere in modo unico in qualsiasi base. L'intero k è il *numero di cifre di* k *in base* b e spesso si indica con k_n quando non è necessario specificare la base b. Si ha che

$$k_n = \begin{cases} 1 & \text{se } n = 0 \\ \left[\frac{\log n}{\log b}\right] + 1 & \text{altrimenti} \end{cases} \tag{1.1}$$

dove $[x] = \max\{m \in \mathbb{N} |\ m \leq x\}$ denota la *parte intera di* $x \in \mathbb{R}$ e $\log x = \int_1^x dt/t$ denota il *logaritmo naturale di* $x \in \mathbb{R}$. Per vedere questo è sufficiente notare che da $n = \sum_{i=0}^{k-1} d_i b^i$ e $d_{k-1} \neq 0$ segue $n \geq b^{k-1}$. Quindi $k - 1 \leq \frac{\log n}{\log b}$

ed essendo $k \in \mathbb{N}$, si ha che $k-1 \leq [\frac{\log n}{\log b}]$. Inoltre dal fatto che $n = \sum_{i=0}^{k-1} d_i b^i$ e $d_i < b$ per ogni $i = 0, \ldots, k-1$ otteniamo

$$n \leq (b-1) \sum_{i=0}^{k-1} b^i = b^k - 1 < b^k,$$

da cui $k > \frac{\log n}{\log b}$ ed essendo $k \in \mathbb{N}$, $k \geq \left[\frac{\log n}{\log b}\right] + 1$ e questo implica (1.1). Si noti che con queste disuguaglianze abbiamo anche mostrato che n ha k cifre in base b se e solo se $b^{k-1} \leq n < b^k$.

Se $b = 2$ allora useremo il termine espansione *binaria*, se $b = 10$ espansione *decimale* e se $b = 2^m$ espansione in m*–bit* .

Se n e m hanno rispettivamente k_n e k_m cifre in base b, allora

$$k_{n+m} = \begin{cases} \max\{k_n, k_m\} & \text{oppure} \\ \max\{k_n, k_m\} + 1. \end{cases} \tag{1.2}$$

Infatti $n + m \geq b^{k_n - 1} + b^{k_m - 1} \geq b^{\max\{k_n, k_m\} - 1}$ e $n + m < b^{k_n} + b^{k_m} \leq 2b^{\max\{k_n, k_m\}} \leq b^{\max\{k_n, k_m\} + 1}$ in quanto $b \geq 2$ e da ciò segue

$$\max\{k_n, k_m\} \leq k_{n+m} \leq \max\{k_n, k_m\} + 1.$$

Inoltre

$$k_{nm} = \begin{cases} k_n + k_m & \text{oppure} \\ k_n + k_m - 1. \end{cases} \tag{1.3}$$

Infatti $nm < b^{k_n + k_m}$ e $nm > b^{(k_n + k_m - 1) - 1}$, dunque

$$k_n + k_m - 1 \leq k_{nm} \leq k_n + k_m.$$

Esempio.

i. $(110010010)_2 = 2 + 2^5 + 2^8 + 2^9 = (402)_{10}$;

ii. $(\texttt{BAD})_{26} = 4 \cdot 26^0 + 1 \cdot 26^1 + 2 \cdot 26^2 = (1382)_{10}$;

iii. Calcoliamo (a mano) l'espansione binaria di $(10^6)_{10}$ che ha $[6 \cdot \log 10 / \log 2] + 1 = 20$ cifre decimali. Cominciamo scrivendo:

$$10^6 = 2^6(1 + (5^6 - 1))$$

Adesso

$$5^6 - 1 = (5-1)(5^2+5+1)(5+1)(5^2-5+1) = 2^3 \cdot 3 \cdot 21 \cdot 31.$$

Scrivendo $3 = 1+2$, $31 = (2^5-1)/(2-1) = 1+2+2^2+2^3+2^4$ e $21 = 1+2^2 5 = 1+2^2(1+2^2) = 1+2^2+2^4$ e moltiplicando, otteniamo:

$$\begin{aligned}(10^6)_{10} &= 2^6(1+2^3(1+2)(1+2^2+2^4)(1+2+2^2+2^3+2^4)\\ &= 2^6+2^9+2^{14}+2^{16}+2^{17}+2^{18}+2^{19}\\ &= (11110100001001000000)_2\end{aligned}$$

1.2 La nozione di operazione bit

Vogliamo definire un'unità di misura per il tempo necessario per effettuare i calcoli. Introduciamo questa unità con un esempio.

SOMMA DI NUMERI BINARI: Siano $n = (1111000)_2$ e $m = (11110)_2$, vogliamo calcolare $n+m$. Mettendo i numeri in colonna (scrivendo il numero più grande per primo) e procedendo come ai tempi delle elementari procediamo come segue:

	1	1	1	0	0	0	0		$\longleftarrow$ riporti
$n \rightarrow$	1	1	1	1	0	0	0	+	
$m \rightarrow$	0	0	1	1	1	1	0	=	
$n+m \rightarrow$	1	0	0	1	0	1	1	0	$\longleftarrow$ risultato

Analizziamo questo esempio. Poniamo $\mathbb{F}_2 = \{0,1\}$ e osserviamo che per calcolare la j–esima cifra d_j di $m+n$ si considera il vettore a 3 bit

$$\left|\begin{array}{c} r_{j-1} \\ a_j \\ b_j \end{array}\right| \in \mathbb{F}_2^3$$

dove a_j e b_j sono rispettivamente le j–esime cifre di m e di n e r_{j-1} è il riporto ottenuto quando si è calcolato d_{j-1} e $r_0 = 0$. A questo vettore si associa il vettore

$$\left|\begin{array}{c} r_j \\ d_j \end{array}\right| \in \mathbb{F}_2^2$$

dove d_j è la cifra cercata. Si ha che

$$\left|\begin{array}{c} r_j \\ d_j \end{array}\right| = \boxplus\left(\left|\begin{array}{c} r_{j-1} \\ a_j \\ b_j \end{array}\right|\right)$$

dove la funzione $\boxplus$ è definita come segue

$$\boxplus : \mathbb{F}_2^3 \longrightarrow \mathbb{F}_2^2$$

$$\left|\begin{array}{c}0\\0\\0\end{array}\right| \mapsto \left|\begin{array}{c}0\\0\end{array}\right| \qquad \left|\begin{array}{c}1\\0\\0\end{array}\right|, \left|\begin{array}{c}0\\1\\0\end{array}\right|, \left|\begin{array}{c}0\\0\\1\end{array}\right| \mapsto \left|\begin{array}{c}0\\1\end{array}\right|$$

$$\left|\begin{array}{c}1\\1\\1\end{array}\right| \mapsto \left|\begin{array}{c}1\\1\end{array}\right| \qquad \left|\begin{array}{c}0\\1\\1\end{array}\right|, \left|\begin{array}{c}1\\0\\1\end{array}\right|, \left|\begin{array}{c}1\\1\\0\end{array}\right| \mapsto \left|\begin{array}{c}1\\0\end{array}\right|$$

La funzione $\boxplus$ deve essere applicata tante volte quante sono le cifre di n.

Definizione. La funzione $\boxplus$ si dice *operazione bit (tipo somma)*.

Per calcolare $n + m$, dove $n, m \in \mathbb{N}$ espresso in base binaria è necessario applicare la funzione $\boxplus$ $\max\{k_n, k_m\}$ volte. Quindi diremo che sono necessarie $\max\{k_n, k_m\}$ operazioni bit. A causa dell'architettura di un computer la nozione di operazione bit è adatta a misurare la quantità di tempo-macchina necessario ad effettuare un dato calcolo.

Se $f : \mathbb{Z}^s \to \mathbb{Z}^r$ che può essere calcolata mediante un certo numero di operazioni bit (cioè applicando $\boxplus$ un certo numero di volte), definiamo il tempo di calcolo di $f(\underline{n})$ (e lo indichiamo $\mathfrak{T}(f(\underline{n}))$) come il **minimo** numero di operazioni bit necessarie per calcolare $f(\underline{n})$.

Quindi

$$\boxed{\mathfrak{T}(m+n) = \max\{k_m, k_n\}.}$$

Nella pratica potrebbe essere molto difficile e poco interessante calcolare il preciso valore di $\mathfrak{T}(f(\underline{n}))$. Infatti ci sono molti modi per calcolare la stessa funzione. Una stima superiore è quasi sempre sufficiente. Per quanto riguarda una stima inferiore, osserviamo che il numero di operazioni bit necessarie a calcolare il risultato di una certa operazione è senz'altro superiore al numero di cifre binarie del risultato. Infatti tutte le cifre del risultato devono essere calcolate e ognuna di questa richiede almeno un'operazione bit. Quindi

$$\boxed{\mathfrak{T}(f(\underline{n})) \geq k_{f(\underline{n})}}$$

dove con $k_{f(\underline{n})}$ indichiamo la somma delle cifre di tutte le componenti di $f(\underline{n})$. Quella sopra è l'unica sottostima generale che siamo in grado di produrre.

Vedremo che questa nozione è utile per valutare i tempi necessari a calcolare tutte le funzioni che si incontrano in crittografia. Ciò ci permetterà di distinguere tra operazioni che in pratica si possono fare ed altre che, richiedendo troppo tempo, non possono essere effettuare. Mostreremo subito che

$$\boxed{\mathfrak{T}(mn) \leq k_m k_n.}$$

PRODOTTI DI NUMERI BINARI: Siano $m = (11101)_2$ e $n = (1101)_2$. Per calcolare $m \cdot n$, mettiamo i numeri i colonna come facevamo al tempo delle scuole elementari:

$$\begin{array}{lccccccccc}
m & & & & & 1 & 1 & 0 & 1 & \times \\
n & & & & 1 & 1 & 1 & 0 & 1 & = \\
\cline{6-9}
 & & & & & 1 & 1 & 0 & 1 & \\
 & & & 1 & 1 & 0 & 1 & - & - & \\
 & & 1 & 1 & 0 & 1 & - & & & \\
 & 1 & 1 & 0 & 1 & - & & & & \\
\cline{2-6}
m \cdot n & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 &
\end{array}$$

Per ottenere $m \cdot n$ scandiamo le cifre di n (da destra); ogni volta che troviamo (un) 1 riscriviamo sotto una copia di m e sotto la prima cifra di questa copia scriviamo un trattino. Se la cifra di n successiva è uno zero allora scriviamo un altro trattino accanto a quello che abbiamo appena scritto. In questo modo otteniamo tante righe (tutte uguali a m) quante sono le cifre di n uguali a 1. Questa prima fase è pura trascrizione delle informazioni e non richiede alcuna operazione.

La fase successiva consiste nel sommare tutte le righe. In totale vanno fatte tante somme quante sono le righe trascritte meno una. Si noti che dopo ogni somma, le cifre che sono sopra i trattini possono essere trascritte automaticamente, quindi in ogni passo viene sempre fatta una somma tra m e un numero che ha un numero di cifre inferiore.

In totale il numero di operazioni bit che verranno fatte è pari a k_m moltiplicato per il numero di righe meno 1. Possiamo scrivere questo nel seguente modo

$$\mathfrak{T}(m \cdot n) = k_m \times (\#\{1 \text{ nell'espansione decimale di } n\} - 1).$$

Da questo segue subito la stima

$$\mathfrak{T}(mn) \leq k_m k_n.$$

Siccome in media n conterrà $k_n/2$ uno e $k_n/2$ zero, la stima precedente in media diventa $\mathfrak{T}(mn) \leq 0.5 k_m k_n$. Ma come vedremo "l'ordine di grandezza" rimane lo stesso.

Il passo successivo è quello di decomporre le sottrazioni in un certo numero di operazioni bit. Per far ciò avremo il bisogno di introdurre un nuovo tipo di operazioni bit (tipo sottrazione).

SOTTRAZIONE DI NUMERI BINARI: Siano $n = (1011010)_2$ e $m = (110111)_2$, vogliamo calcolare $n - m$. Mettiamo i numeri in colonna come abbiamo fatto per la somma (scrivendo il numero più grande per primo) e procediamo, come ai tempi delle elementari:

$$\begin{array}{rcccccccc} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 0} & {\scriptstyle 0} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 0} \\ n \rightarrow & & 1 & 0 & 1 & 1 & 0 & 1 & 0 \quad - \\ m \rightarrow & & 0 & 1 & 1 & 0 & 1 & 1 & 1 \quad = \\ \hline n - m \rightarrow & & & 1 & 0 & 0 & 0 & 1 & 1 \end{array} \quad \begin{array}{l} \longleftarrow \text{ prestiti} \\ \\ \\ \longleftarrow \text{ risultato} \end{array}$$

Vediamo che anche per la sottrazione, come per la somma, per calcolare la cifra j–esima c_j di $n - m$ si tratta di consultare la seguente tavola.

$$\boxminus : \mathbb{F}_2^3 \longrightarrow \mathbb{F}_2^2$$

$$\left|\begin{array}{c} 0 \\ 1 \\ 0 \end{array}\right| \mapsto \left|\begin{array}{c} 0 \\ 1 \end{array}\right| \qquad \left|\begin{array}{c} 0 \\ 1 \\ 1 \end{array}\right|, \left|\begin{array}{c} 1 \\ 1 \\ 0 \end{array}\right|, \left|\begin{array}{c} 0 \\ 0 \\ 0 \end{array}\right| \mapsto \left|\begin{array}{c} 0 \\ 0 \end{array}\right|$$

$$\left|\begin{array}{c} 1 \\ 0 \\ 1 \end{array}\right| \mapsto \left|\begin{array}{c} 1 \\ 0 \end{array}\right| \qquad \left|\begin{array}{c} 1 \\ 0 \\ 0 \end{array}\right|, \left|\begin{array}{c} 0 \\ 0 \\ 1 \end{array}\right|, \left|\begin{array}{c} 1 \\ 1 \\ 1 \end{array}\right| \mapsto \left|\begin{array}{c} 1 \\ 1 \end{array}\right|$$

Si ha che

$$\left|\begin{array}{c} p_j \\ c_j \end{array}\right| = \boxminus\left(\left|\begin{array}{c} p_{j-1} \\ a_j \\ b_j \end{array}\right|\right)$$

dove p_j è il prestito dell'operazione precedente e $p_0 = 0$ mentre a_j e b_j sono rispettivamente la j–esima cifra di n e di m. La funzione $\boxminus$ deve essere applicata tante volte quante sono le cifre di n.

Definizione. La funzione $\boxminus$ è detta *operazione bit (tipo sottrazione)*.

Anche in questo caso sono necessarie $\max\{k_n, k_m\}$ operazioni bit per calcolare $n - m$.

Un'operazione bit tipo sottrazione è simile ad una tipo somma e per questo motivo è ragionevole assumere che applicarle richieda lo stesso tempo. Per questo motivo le confonderemo e parleremo semplicemente di operazioni bit. Quindi

Quindi

$$\boxed{\mathfrak{T}(n-m) = \max\{k_m, k_n\}.}$$

Concludiamo la nostra discussione sulle operazioni fondamentali con la divisione. Dati $n, m \in \mathbb{N}$ sappiamo che esiste un unica coppia di numeri $(q, r) \in \mathbb{N}^2$ che verifica:

1. $n = q \cdot m + r$;
2. $0 \leq r < n$.

Con il simbolo n/m indichiamo la coppia (q, r) over q e r si chiamano quoziente e resto. Anche nel caso di n/m, possiamo decomporre il calcolo in un certo numero di operazioni bit.

DIVISIONE DI NUMERI BINARI: Al solito cominciamo con un esempio. Siano $n = (11001001)_2$ e $m = (100111)_2$, vogliamo calcolare n/m.

$n \rightarrow$	1	1	0	0	1	0	0	1		1	0	0	1	1	1	$\leftarrow m$
	1	0	0	1	1	1				1	0	1				$\leftarrow q$
			1	0	1	1	0	1								
			1	0	0	1	1	1								
$r \rightarrow$						1	0	0								

La divisione procede nel seguente modo: si abbassano tante cifre di n in modo tale che il numero abbassato sia maggiore o uguale a m. Ciò vuol dire che bisogna abbassare k_m oppure $k_m + 1$ cifre. Poi si scrive 1 nello spazio riservato per le cifre di q, si trascrive m sotto le cifre abbassate e si sottrae. Accanto al risultato di questa sottrazione si abbassano altre cifre di n. Ogni volta che si abbassa la cifra si aggiunge uno zero accanto a q fino ad ottenere un numero che sia maggiore o uguale a m. Adesso si aggiunge 1 su q e via così. Alla fine, quando sono finite le cifre di n, il numero rimasto è necessariamente minore di m ed è il resto r.

Questo metodo permette di decomporre la divisione un certo numero di sottrazioni. Ogni sottrazione coinvolge numeri che hanno k_m oppure $k_m + 1$ cifre. E il numero di sottrazioni è pari al numero di cifre del quoziente k_q. In realtà bastano tante sottrazioni quanti sono gli 1 nell'espansione binaria di q.

Quindi

$$\boxed{\mathfrak{T}(n/m) \leq (k_m + 1)k_q.}$$

Sfruttando il fatto che $n \geq mq$ e quindi $k_q \leq k_n - k_m + 1$, possiamo anche dedurre le stime

$$\boxed{\mathfrak{T}(n/m) \leq (k_m + 1)(k_n - k_m + 1.) \leq (k_m + 1)(k_n - 1) \leq k_n^2}$$

che a volte saranno sufficienti.

Negli algoritmi che analizziamo spesso accade di dover sommare o moltiplicare un numero crescente di numeri interi. Per questa ragione è utile avere delle stime più generali. Si ha

Proposizione 1 *Siano $m_1, \ldots, m_s \in \mathbb{N}$ e sia $M = \max\{m_1, \ldots, m_s\}$. Allora*

1. $\mathfrak{T}(m_1 + \cdots + m_s) \leq (s-1) \cdot k_{(s-1)M}$
2. $\mathfrak{T}(m_1 \cdots m_s) \leq (s-1)^2 \cdot k_M^2$

Dimostrazione. Per quanto riguarda la prima stima, procediamo per induzione osservando che $\mathfrak{T}(m_1 + m_2) = k_M$ ed è quindi verificata la base dell'induzione. Se $s > 2$, allora, per calcolare $m_1 + \cdots + m_s$, bisogna prima aver calcolato $m_1 + \cdots + m_{s-1}$, quindi

$$\mathfrak{T}(m_1 + \cdots + m_s) = \mathfrak{T}(m_1 + \cdots + m_{s-1}) + \mathfrak{T}((m_1 + \cdots + m_{s-1}) + m_s).$$

Per induzione, il primo tempo è minore o uguale a $(s-2)k_{(s-2)M}$ mentre $m_1 + \cdots + m_{s-1} \leq (s-1)M$. Infine

$$\mathfrak{T}(m_1 + \cdots + m_s) \leq (s-2)k_{(s-2)M} + k_{\max\{m_1+\cdots+m_{s-1}, m_s\}} \leq$$

$$(s-2)k_{(s-1)M} + k_{(s-1)M} = (s-1) \cdot k_{(s-1)M}$$

come volevasi dimostrare. La seconda stima è del tutto analoga osservando che $m_1 \cdots m_{s-1} \leq M^{s-1}$ è un numero con al più $(s-1)k_M$ cifre, si ha

$$\begin{aligned} \mathfrak{T}(m_1 \cdots m_s) &= \mathfrak{T}(m_1 \cdots m_{s-1}) + \mathfrak{T}((m_1 \cdots m_{s-1}) \cdot m_s) \leq \\ &\leq (s-2)^2 k_M^2 + k_{M^{s-1}} \cdot k_{m_s} \leq (s-1)^2 k_M^2 . \square \end{aligned} \tag{1.4}$$

1.3 Il simbolo "O"-grande

Vogliamo alleggerire le nostre notazioni e scegliamo di non distinguere tra tempi che hanno lo stesso ordine di grandezza. Per far ciò analizziamo i tempi usando un simbolo che viene dall'analisi.

Definizione. Siano f e g due funzioni di s variabili reali a valori in $\mathbb{R}$ (o anche in $\mathbb{C}$). Diremo che

$$f = O(g)$$

se esistono costanti $c > 0$ e $N_c > 0$ tali che $|f(n_1, \dots, n_s)| \leq c|g(n_1, \dots, n_s)|$ per ogni $(n_1, \dots, n_s)$ su cui entrambe f e g sono definite e $n_i > N_c$ per $i = 1, \dots, s$. Se $f = O(g)$ e $g = O(f)$, allora scriveremo $f \asymp g$.

Un esempio tipico è $(\log n)^\beta = O(n^\alpha)$ per ogni $\alpha, \beta > 0$. Per noi un altro esempio essenziale è

$$k_n = O\left(\log n / \log b\right).$$

Inoltre quando b è fissato e si può assumere costante ai fini della discussione, possiamo anche scrivere

$$k_n = O\left(\log n\right).$$

È anche vero che $k_n \asymp \log n$.

Tutti i risultati del paragrafo precedente possono essere riscritti usando il simbolo "O"–grande.

$\mathfrak{T}(n+m) = O(\log n)$	$\mathfrak{T}(n \cdot m) = O((\log n)^2)$
$\mathfrak{T}(n-m) = O(\log n)$	$\mathfrak{T}(n/m) = O((\log n)^2)$
$\mathfrak{T}(m_1 + \cdots + m_s) = O(s(\log M + \log s)$	$\mathfrak{T}(m_1 \cdots m_s) = O(s^2(\log M)^2)$

dove $n \geq m$ e $M = \max\{m_1, \dots m_s\}$. Osserviamo che se s è fisso e nell'analisi di un dato problema non è da considerarsi come una variabile, allora

$$\mathfrak{T}(m_1 + \cdots + m_s) = O(\log M) \quad \text{e} \quad \mathfrak{T}(m_1 \cdots m_s) = O((\log M)^2).$$

Esempio 1. Consideriamo il fattoriale $n!$ di un numero naturale n. Si ha che $k_{n!} \asymp n \log n$. Infatti $n! < n^n$ dunque $k_{n!} \leq nk_n = O(n \log n)$; inoltre $n! > ([n/2])^{[n/2]}$ e quindi $k_{n!} \geq [n/2](k_{[n/2]} - 1) \asymp n \log n$.

Questo discende subito anche dalla formula di Stirling:

$$n! \sim \left(\frac{n}{e}\right)^n \sqrt{2\pi n}.$$

Per calcolare $n!$ bisogna calcolare $n-1$ prodotti di numeri inferiori a n. Quindi da una delle stime nel paragrafo precedente, otteniamo

$$\mathfrak{T}(n!) = O(n^2(\log n)^2).$$

Esempio 2. Siano $\underline{a} = (a_1, \ldots, a_s)$ e $\underline{b} = (b_1, \ldots, b_s)$ vettori in $\mathbb{N}^s$. Allora se $\langle \underline{a}, \underline{b} \rangle$ è il prodotto scalare, si ha

$$\mathfrak{T}(\langle \underline{a}, \underline{b} \rangle) = O(s(\log^2 M + \log s))$$

dove $M = \max\{a_1, \ldots, a_s, b_1, \ldots, b_s\}$. Infatti per calcolare

$$\langle \underline{a}, \underline{b} \rangle = a_1 b_1 + \cdots + a_s b_s$$

è necessario precalcolare gli s prodotti $a_1 b_1, \ldots a_s b_s$ e in seguito sommarli. La prima operazione richiede $O(s \log^2 M)$ operazioni bit mentre la seconda (osservando che $a_i b_i \leq M^2$ richiede $O(s(\log M + \log s))$ operazioni bit. In totale trascurando il termine $s \log M = O(s \log^2 M)$ sono necessarie $O(s(\log^2 M + \log s))$ operazioni bit.

Esempio 3. Siano $f, g \in \mathbb{Z}[x]$ e indichiamo con D il massimo tra i gradi di f e g e con M il massimo di tutti i coefficienti sia di f che g, allora

$$\mathfrak{T}(f \cdot g) = O(D^2(\log M^2 + \log D)).$$

Infatti i coefficienti del polinomio prodotto $h = fg$ sono dati dalle formule

$$c_k = \sum_{i+j} a_i b_j$$

Questa somma contiene meno di $D+1$ addendi e per l'esempio precedente

$$\mathfrak{T}(c_k) = O(D(\log M^2 + \log D)).$$

In tutto bisogna calcolare $O(D)$ coefficienti e quindi $\mathfrak{T}(f \cdot g) = O(D^2(\log M^2 + \log D))$.

Esempio 4. Se m, n sono interi e $\binom{n}{m}$ è il coefficiente binomiale, si ha che

$$\mathfrak{T}(\binom{n}{m}) = O(m^2 \log^2 m).$$

Per far vedere questo sfruttiamo l'identità

$$\binom{n}{m} = \frac{n(n-1)\cdots(n-m+1)}{1\cdot 2\cdots m}.$$

Per prima cosa bisogna moltiplicare m numeri minori di n e questo richiede $O(m^2 \log^2 n)$ operazioni bit. Poi bisogna dividere il risultato m volte per numeri minori di m. ciascuna di queste divisioni richiede $O(m \log n \log m)$. Sfruttando il fatto che si può assumere $m \leq n/2$ in quanto $\binom{n}{m} = \binom{n}{n-m}$ e $\binom{n}{m} = 0$ se $m > n$.

Osservazioni. Indubbiamente tramite la notazione O–grande si impone una grossa approssimazione nell'analisi del tempo necessario e fare i calcoli. Ad esempio non si distingue tra il tempo necessario a sommare due interi e quello necessario a sommarne centomila. La costante implicità nel simbolo O–grande potrebbe rendere un calcolo teoricamente rapido in uno completamente inefficiente. Dovremo vivere con questa limitazione.
Il concetto di numero di operazioni bit dipende in modo cruciale dal metodo che si usa per effettuare il calcolo. Un semplice esempio è il seguente: Supponiamo di voler stimare il numero di operazioni bit necessarie a sommare i primi n numeri. Trattandosi di n minori di n, si ottiene

$$\mathfrak{T}(\sum_{j=0}^{n} j) = O(n \log n).$$

Se però si usa il fatto che $\sum_{j=0}^{n} j = \frac{n(n+1)}{2}$ e si calcola sfruttando questa identità si ha che

$$\mathfrak{T}(\sum_{j=0}^{n} j) = O(\log^2 n).$$

È quindi chiaro che il numero di operazioni bit necessarie ad effettuare un dato calcolo dipende dall'algoritmo usato. Modifichiamo quindi la nostra definizione di tempo nel seguente modo:

Definizione. Sia f una funzione in k variabili e $\mathcal{A}$ un algoritmo per calcolare il valore di $f(n_1, \ldots, n_k)$. Allora

$$\mathfrak{T}_{\mathcal{A}}(f(n_1, \ldots, n_k) = \# \left\{ \begin{array}{l} \text{operazioni bit necessarie} \\ \text{a calcolare } f(n_1, \ldots, n_k) \text{ usando } \mathcal{A}. \end{array} \right\}$$

Malgrado questa precisazione, continueremo ad usare la notazione $\mathfrak{T}$ quando l'algoritmo è implicito.

Persino per moltiplicare due numeri interi n, m ($n \geq m$) esiste un algoritmo trasformata di Fourier veloce (FFT) che permette di risparmiare il numero di operazioni bit. Si ha infatti che

$$\mathfrak{T}_{\text{FFT}}(nm) = O(\log n \ \log\log n \ \log\log\log n).$$

Per quanto ci riguarda assumeremo che i numeri vengono sempre moltiplicati con l'algoritmo elementare spiegato nel paragrafo precedente e quindi che $\mathfrak{T}(nm) = O(\log^2 n)$.

Un altro semplice esempio di come il numero di operazioni dipende dall'algoritmo è il seguente: Supponiamo di voler moltiplicare due polinomi di primo grado a coefficienti interi $ax + b$ e $cx + d$ con $a, b, c, d \in \mathbb{N}$. Tutti sanno che

$$(ax + b)(cx + d) = acx^2 + (ad + bc)x + bd$$

Quindi per moltiplicare i polinomi bisogna calcolare i 4 prodotti ac, ab, bc, db e poi la somma $ad + bc$. Se invece di fare ciò osserviamo che $(ad + bc) = (a + b)(c + d) - ac - bd$ e sfruttiamo questa identità per fare i calcoli, allora dobbiamo fare solo 3 prodotti ($(a + b)(c + d)$, ac e bd) e 4 somme. In numero delle somme sale ma il numero dei prodotti scende. Perciò questo metodo è più veloce.

Siamo adesso in grado di definire in modo preciso quando consideriamo difficile calcolare una funzione

Definizione. Data una funzione f di k variabili e un'algoritmo $\mathcal{A}$ per calcolarla, diremo che f si calcola in tempo *polinomiale* con $\mathcal{A}$ se

$$\mathfrak{T}_{\mathcal{A}}(f(n_1, \ldots, n_k)) = O(\log n_1^{\alpha_1} \cdots \log n_k^{\alpha_k})$$

per opportuni valori positivi $\alpha_1, \cdots, \alpha_k$ indipendenti da $n_1, \ldots, n_k$. Inoltre se $\alpha_1 + \cdots + \alpha_k = 1$ diremo che f si calcola in tempo *lineare* e se $\alpha_1 + \cdots + \alpha_k = 2$ diremo che f si calcola in tempo *quadratico*. Infine se

$$\mathfrak{T}_{\mathcal{A}}(f(n_1, \ldots, n_k)) = O(\exp(\log n_1^{\beta_1}) \cdots \exp(\log n_k^{\beta_k}))$$

per opportuni valori positivi $\beta_1, \cdots, \beta_k$ indipendenti da $n_1, \ldots, n_k$ e tali che $\beta_1 + \cdots + \beta_k < 1$ allora diremo che f si calcola in tempo *sub–esponenziale*. Altrimenti diremo che f si calcola in tempo esponenziale.

Ad esempio Le quattro operazioni fondamentali si possono effettuare in tempo polinomiale. Anzi la somma e la sottrazione si possono effettuare in tempo lineare ed il prodotto e la divisione in tempo quadratico.

Concludiamo il paragrafo con altri due esempi.

Esempio 5. (Cambiamento di base). Supponiamo n sia un intero espresso in base binaria, vogliamo stimare il tempo necessario per calcolare le cifra decimali di n.

Fissiamo la corrispondenza:

$$\begin{array}{cccccccccc} 0 & 1 & 10 & 11 & 100 & 101 & 110 & 111 & 1000 & 1001 \\ \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ (0)_{10} & (1)_{10} & (2)_{10} & (3)_{10} & (4)_{10} & (5)_{10} & (6)_{10} & (7)_{10} & (8)_{10} & (9)_{10} \end{array}$$

Definiamo ricorsivamente:

$$\begin{aligned} n/(1010)_2 &= (q_1, r_1) \qquad 0 \le r_1 < (1010)_2 \\ q_1/(1010)_2 &= (q_2, r_2) \qquad 0 \le r_2 < (1010)_2 \\ q_2/(1010)_2 &= (q_3, r_3) \qquad 0 \le r_3 < (1010)_2 \\ &\vdots \quad \vdots \quad \vdots \qquad\qquad \vdots \\ q_{k-1}/(1010)_2 &= (q_k, r_k) \qquad 0 \le r_k < (1010)_2. \end{aligned}$$

Ci si ferma non appena $q_k < (1010)_2 = 10$. Questo avviene sempre in quanto $n > q_1 > q_2 > \cdots$ è una successione di numeri decrescenti e prima o poi supererà 10. Si verifica facilmente che

$$n = (q_k r_k r_{k-1} \cdots r_2 r_1)_{10}.$$

Infatti i numeri $r_1, \ldots, r_k, q_k$ sono cifre nel sistema decimale e scrivendo 10 al posto di $(1010)_{10}$, si ha

$$\begin{aligned} n = 10q_1 + r_1 &= 10(10q_2 + r_2) + r_1 \\ &= \vdots \\ &= 10^k q_k + 10^{k-1} r_k + 10^{k-2} r_{k-2} + \cdots + 10 r_1 + r_1. \end{aligned}$$

Rimane da analizzare il numero di operazioni bit necessarie a compiere questo calcolo. Si tratta di compiere k divisioni per $(1010)_2$. Notare che k è il numero di cifre decimali di n. Pertanto $k = O(\log n)$. Ogni divisione richiede $O(\log n)$ operazioni bit in quanto $q_i < n$. Infine

$$\mathfrak{T}((n)_{10}) = O(\log^2 n).$$

Se invece di voler calcolare l'espansione in base 10 avessimo voluto calcolare l'espansione in base b (variabile), allora avremmo fissato b simboli per in numeri

$\{(0)_2, (1)_2, \ldots, (b-1)_2\}$ e poi diviso ripetutamente per b. I resti di tali divisioni sono le cifre in base b di n. In questo caso ogni divisione richiede $O(\log n \log b)$ operazioni bit e il numero di divisioni è pari al numero di cifre in base b di n cioè $O(\log n / \log b)$. Quindi si ha il tempo quadratico

$$\mathfrak{T}((n)_b) = O(\log^2 n).$$

Deduciamo che il numero di operazioni bit per calcolare le cifre in una qualsiasi base non dipende dalla base.

È naturale chiedersi come stimare il numero di operazioni bit necessarie a calcolare le cifre binarie di un dato numero n espresso in base decimale. Se scriviamo

$$n = (r_k r_{k-1} \cdots r_1)_{10}$$

con

$$r_i \in \{(0)_2, (1)_2, (10)_2, (11)_2, (100)_2, (101)_2, (110)_2, (111)_2, (1000)_2, (1001)_2\}$$

sfruttando la corrispondenza precedente, allora

$$n = \sum_{i=1}^{k} r_i (1010)_2^i.$$

Quindi possiamo calcolare le cifre binarie di n sfruttando le stime della Proposizione 1. Lasciamo come esercizio di verificare che questo calcolo richiede $O(\log^2 n)$ operazioni bit.

Proposizione 2 (La parte intera della radice quadrata) *Dato un numero binario n, esiste un algoritmo per calcolare $[\sqrt{n}]$ tale che*

$$\mathfrak{T}([\sqrt{n}]) = O(\log^3 n).$$

Dimostrazione. Supponiamo che n abbia k cifre binarie. Dal fatto che $2^{k-1} \leq n < 2^k$ segue che $2^{(k-1)/2} \leq \sqrt{n} < 2^{k/2}$. Dunque se k è pari $2^{k/2-1} \leq [\sqrt{n}] < 2^{k/2}$ e il numero di cifre binarie di $[\sqrt{n}]$ è $k/2$ e se k è dispari, allora $2^{(k+1)/2-1} \leq [\sqrt{n}] < 2^{(k+1)/2}$ e dunque $[\sqrt{n}]$ ha $(k+1)/2$ cifre binarie. Quindi

$$k_{[\sqrt{n}]} = [(k_n + 1)/2].$$

Assumiamo che $k = 2t$ sia pari (il caso k dispari è analogo). Dalla precedente analisi segue che

$$[\sqrt{n}] = (1\epsilon_2\epsilon_3 \cdots \epsilon_t)_2$$

con $\epsilon_i \in \{0,1\}$. Vogliamo determinare $\epsilon_2, \epsilon_3, \ldots, \epsilon_t$ per approssimazioni successive. Consideriamo la successione definita per ricorrenza:

$$x_0 = 2^{t-1} = (1 \overbrace{0 \cdots 0}^{t-1 \text{ volte}})_2$$

$$x_i = \begin{cases} x_{i-1} & \text{se } (x_{i-1} + 2^{t-i-1})^2 > n \\ x_{i-1} + 2^{t-i-1} & \text{altrimenti.} \end{cases}$$

Affermiamo che $x_{t-1} = [\sqrt{n}]$. Per costruzione è chiaro che $x_{t-1} \leq [\sqrt{n}]$. Vogliamo anche mostrare che $[\sqrt{n}] - x_i < 2^{t-1-i}$ e quindi $[\sqrt{n}] - x_{t-1} < 1$.

Da $2^{t-1} \leq [\sqrt{n}] < 2^t$ segue che $[\sqrt{n}] - x_0 < 2^t - 2^{t-1} = 2^{t-1}$. Per induzione su i, si ha che se $x_i = x_{i-1}$, allora $(x_{i-1} + 2^{t-i-1})^2 > n$ da cui

$$[\sqrt{n}] - x_i = [\sqrt{n}] - x_{i-1} < 2^{t-i-1}.$$

Se invece $x_i = x_{i-1} + 2^{t-i-1}$, allora

$$[\sqrt{n}] - x_i = [\sqrt{n}] - x_{i-1} - 2^{t-i-1} < 2^{t-i} - 2^{t-i-1} = 2^{t-i-1}.$$

Rimane da calcolare il numero di operazioni bit necessarie per calcolare x_{t-1}. Per calcolare i $t-1$ valori di x_i è necessario calcolare il quadrato di un numero con t cifre decimali e poi (eventualmente) sommare 2^{t-i-2}. Ciascuna di queste operazioni richiede $O(t^2)$ operazioni bit. Quindi

$$\mathfrak{T}([\sqrt{n}]) = O(t^3) = O(\log^3 n).$$

□

Concludiamo il paragrafo osservando che malgrado abbiamo fatto tutte le nostre considerazioni per numeri naturali in $\mathbb{N}$, non c'è un grosso problema ad estenderle a numeri interi in $\mathbb{Z}$. Per lavorare con numero binario intero è sufficiente aggiungere un bit contenente il segno ed ogni volta che si effettua un calcolo precalcolare il segno del risultato secondo le regole dei segno. Questa può essere considerata come una aggiuntiva operazione bit che non cambia l'essenza di tutte le nostre stime.

Esercizi.

1. Sia $f(x) = \sin(x)$.
   (a) Si stimi il tempo necessario per calcolare il valore del polinomio di Taylor di grado 3 intorno a 0 in un valore intero $x = n$.

   (b) Si stimi il tempo necessario per calcolare il valore del polinomio di Taylor di grado k intorno a 0 in un valore intero $x = n$.

2. Si dia una stima per il numero di operazioni bit necessarie al calcolo del determinante di una matrice 3×3 a coefficienti interi in cui gli elementi della prima colonna sono in valore assoluto minori di M, quelli della seconda colonna sono in valore assoluto minori di N e quelli della terza sono in valore assoluto minori di L.

3. Calcolare la parte intera di $\sqrt{(1101010001101)_2}$ utilizzando l'algoritmo delle approssimazioni successive. (si tratta di un numero binario.)

4. Si dia una stima per il numero di operazioni bit necessarie per calcolare l'inversa di una matrice 4×4 in cui tutte le componenti sono $\leq N$.

5. Calcolare la parte intera di $\sqrt{10110101011}$ utilizzando l'algoritmo delle approssimazioni successive. (si tratta di un numero binario.)

6. Dare una stima per il numero di operazioni bit necessarie per verificare se un numero intero m ha fattori primi minori di Y.

7. Si calcoli il numero di operazioni bit necessarie per calcolare la seguente somma:

$$\sum_{k=1}^{n} ka^k.$$

8. Data una matrice A triangolare superiore n per n a coefficienti interi che non superano n. Si dia una stima per il numero di operazioni bit necessarie a calcolare il quadrato di A.

9. Dati due polinomi f e g di grado k a coefficienti in un campo finito con 64 elementi. Si dia una stima per il numero di operazioni bit necessarie a calcolare il prodotto fg e per il numero di operazioni bit necessarie a calcolare il polinomio derivato f'.

10. Si dia una stima per il numero di cifre binarie del numero $(n!)^{n!}$.

11. Si calcoli la parte itera del numero binario

$$\sqrt{101011101}$$

    utilizzando l'algoritmo delle approssimazioni successive.

12. Dati $a = (a_1, \ldots, a_n)$, $b = (b_1, \ldots, b_n) \in \mathbb{N}^n$ tali che $a_i, b_i \leq e^{n^3}$ per ogni $i = 1, \ldots, n$, si dia una stima che dipende solo da n per il numero di operazioni bit necessarie a calcolare il prodotto scalare

$$a_1 b_1 + \ldots + a_n b_n.$$

13. Dato il numero binario $n = (1001101010)_2$, calcolare $[\sqrt{n}]$ usando l'algoritmo delle approssimazioni successive.

14. Si scrivano tutte le cifre esadecimali del numero binario $(100101011)_2$.

15. Si descriva un algoritmo per calcolare la parte intera della radice r–esima di un numero intero rappresentato in base due. Si stimi il numero di operazioni bit necessarie per eseguirlo.

16. Se $n \in \mathbb{N}$, sia $\sigma(n)$ la somma dei divisori di n. Supponiamo che sia nota la fattorizzazione (unica) di $n = p_1^{\alpha_1} \cdots p_s^{\alpha_s}$. Calcolare il numero di operazioni bit necessarie per calcolare $\sigma(n)$. *Suggerimento: Usare il fatto che σ è una funzione moltiplicativa e calcolare una formula per $\sigma(p^\alpha)$)*

    Soluzione. Dalla moltiplicatività di σ segue che

    $$\sigma(n) = \sigma(p_1^{\alpha_1}) \cdots \sigma(p_1^{\alpha_1}).$$

    Inoltre $\sigma(p^a) = \sum_{i \le a} p^i = \frac{p^{a+1}-1}{p-1}$. Notare che $\mathfrak{T}(\sigma(p^a)) = O(a^2 \log^2 p)$ infatti l'operazione che richiede il maggior numero di operazioni bit è il calcolo di p^{a+1}. Questo implica che per calcolare $\sigma(p_1^{\alpha_1}), \ldots, \sigma(p_s^{\alpha_s})$ sono necessarie $O(\sum_{i \le s} \alpha_i^2 \log^2 p_i) = O(\log^2 n)$ operazioni bit. Quindi, calcolando $\sigma(n)$ moltiplicando gli s numeri $\sigma(p_1^{\alpha_1}), \ldots, \sigma(p_s^{\alpha_s})$ che sono tutti minori di n^2, otteniamo (usando la formula standard) $\mathfrak{T}(\sigma(n)) = O(s^2 \log^2 n)$. Infine da $s = O(\log s)$ si ha $\mathfrak{T}(\sigma(n)) = O(\log^4 n)$. Con un pò di lavoro in più si può anche mostrare che $\mathfrak{T}(\sigma(n)) = O(\log^3 n)$.

17. Dato il numero binario $n = (10011100101)_2$, calcolare $[\sqrt{n}]$ usando l'algoritmo delle approssimazioni successive (Non passare a base 10 e non usare la calcolatrice!)

    Soluzione. Applicando l'algoritmo delle approssimazioni successive otteniamo:

    $$\begin{array}{ll} x_0 = (100000) = 2^5 & \\ x_1 = (100000) & x_0 + 2^4 > n; \\ x_2 = (100000) & x_1 + 2^3 > n; \\ x_3 = (100000) & x_2 + 2^2 > n; \\ x_4 = (100010) & x_3 + 2 < n; \\ x_5 = (100011) & x_4 + 1 < n; \end{array}$$

    Quindi $[\sqrt{n}] = 100011$.

18. Si stimi il numero di operazioni bit necessarie a calcolare la derivata di un polinomio di grado n^2 in cui tutti i coefficienti sono minori di n.

19. Si stimi il numero di operazioni bit necessarie a calcolare l'integrale a di un polinomio di grado n in cui tutti i coefficienti sono minori di e^n.

20. Si dia una stima (in funzione del parametro t) per il numero di operazioni bit necessarie al calcolo del determinante di una matrice 3×3 a coefficienti interi in cui gli elementi della prima colonna sono in valore assoluto minori di t, quelli della seconda colonna sono in valore assoluto minori di e^t e quelli della terza sono in valore assoluto minori di t^t.

Capitolo 2

Richiami di Teoria dei numeri elementare

La teoria dei numeri è basata sulla *divisibilità*. Dati due numeri $a, b \in \mathbb{Z}$ diremo che a *divide* b e scriveremo $a \mid b$ se esiste un numero $k \in \mathbb{Z}$ tale che $b = ka$. In caso contrario diremo che a *non divide* b e scriveremo $a \nmid b$. Le seguenti proprietà sono fondamentali.

Per ogni $a, b, c \in \mathbb{Z}$:

1. Se $a \mid b$ allora $a \mid bc$;
2. Se $a \mid b$ e $b \mid c$ allora $a \mid c$;
3. Se $a \mid b$ e $b \mid a$ allora $a = \pm b$;
4. Se $a \mid b$ e $a \mid c$ allora $a \mid (b \pm c)$.

Un intero $p \neq \pm 1$ si dice *numero primo* se la condizione $b \mid p$ è verificata solo per i quattro interi $\{\pm 1, \pm p\}$. In caso contrario il numero si dice *composto* . La lettera p nel seguito sarà utilizzata esclusivamente per indicare numeri primi.

Tutti conoscono il seguente Teorema Fondamentale dell'aritmetica (TFA).

Teorema 1 *Dato un intero $n \in \mathbb{N}$, $n > 1$ esistono numeri primi $p_1, \ldots, p_s$ e numeri interi $\alpha_1, \ldots, \alpha_s \in \mathbb{N}$ tali che*

1. $n = p_1^{\alpha_1} \cdots p_s^{\alpha_s}$;
2. $p_1 < p_2 < \cdots < p_s$.

Inoltre i primi $p_1, \ldots, p_s$ e gli interi $\alpha_1, \ldots, \alpha_s$ sono gli unici con questa proprietà.

Il Teorema fondamentale dell'aritmetica purtroppo (o per fortuna) è soltanto un risultato di esistenza e nessuna dimostrazione conosciuta fornisce un algoritmo efficiente per determinare la fattorizzazione di n. In un certo senso il TFA è paragonabile al Teorema di esistenza e unicità per equazioni differenziali ordinarie. Il fatto di sapere che la soluzione esiste non vuol dire in generale che siamo in grado di costruirla o approssimarla. La famiglia degli algoritmi crittografici che va sotto il nome di RSA si basa su questo principio ma ci occuperemo di questo nel fascicolo successivo.

Data la fattorizzazione unica $n = p_1^{\alpha_1} \cdots p_s^{\alpha_s}$ definiamo le funzioni

$$\begin{aligned} \omega(n) &= \alpha_1 + \cdots + \alpha_s \\ \nu(n) &= s. \end{aligned}$$

$\nu(n)$ e $\omega(n)$ contano rispettivamente il numero dei fattori primi di n ed il numero dei fattori primi contati con molteplicità. Ovviamente $\nu(n) \leq \omega(n)$ e dato che $n = p_1^{\alpha_1} \cdots p_s^{\alpha_s} \geq 2^{\alpha_1} \cdots 2^{\alpha_s} = 2^{\omega(n)}$, si ha la stima

$$\nu(n) \leq \omega(n) = O(\log n). \tag{2.1}$$

Questa ci sarà utile in seguito. Osserviamo che con del lavoro in più è possibile mostrare che

$$\nu(n) = O(\log n / \log\log n).$$

Non daremo la dimostrazione di questo risultato. Chi è interessato può trovare la dimostrazione nel libro di Hardy e Wright [2]. Un'altro risultato che enunciamo e non dimostriamo è il *Teorema dei numeri primi* (TNP). Dimostrato nel 1897 (indipendentemente) da Hadamard e de la Valle Puissen a culmine di un sforzo partito da età antichissima e con il contributo essenziale di Riemann ed altri.

Teorema 2 *Sia x un numero reale positivo. Definiamo la funzione $\pi(x)$ come il numero complessivo di numeri primi positivi p con $p \leq x$. Allora*

$$\pi(x) \sim \frac{x}{\log x}.$$

Una dimostrazione (quella classica) del TNP può essere trovata nel libro di Davenport [1]. Esiste una versione più esplicita del TNP dovuta a Rosser e Schoenfeld [6]

$$\frac{x}{\log x}\left(1 + \frac{1}{2\log x}\right) < \pi(x) < \frac{x}{\log x}\left(1 + \frac{3}{2\log x}\right) \qquad (x \geq 52).$$

Per molte applicazioni non è necessario scomodare il TNP ed è spesso sufficiente la stima più debole

$$\pi(x) \asymp \frac{x}{\log x}.$$

Quest'ultima è nota come il Teorema di Chebicev ed è molto più facile da dimostrare del TNP. Noi dimostreremo la seguente versione che appare nel libro di Tenenbaum [9].

Teorema 3 *Per ogni $x > 3$, si ha*

$$\log 2\frac{x}{\log x} \leq \pi(x) \leq \left(\log 4 + \frac{8\log\log x}{\log x}\right)\frac{x}{\log x}.$$

Dimostrazione. Consideriamo la seguente bella disuguaglianza valida per $x \geq 1$:

$$\prod_{n\leq x} p \leq 4^x$$

che dimostriamo più tardi. Da questa segue che

$$t^{\pi(x)-\pi(t)} \leq \prod_{t<p\leq x} p \leq 4^n.$$

Prendendo i logaritmi in entrambi i lati otteniamo

$$(\pi(x) - \pi(t))\log t \leq x\log 4$$

da cui

$$\pi(x) \leq \frac{x\log 4}{\log t} + \pi(t) \leq \frac{x\log 4}{\log t} + t.$$

Adesso scegliamo $t = x/\log^2 x$ (per cui $\log t = \log x - 2\log\log x$) e otteniamo

$$\pi(x) < \frac{x}{\log x - 2\log\log x} + \frac{x}{\log^2 x}$$

Usando il fatto che per $u \in (0, 1/2)$ si ha $1/(1-u) < 1 + 2u$, sappiamo che

$$\frac{1}{\log x - 2\log\log x} < \frac{1}{\log x}\left(1 + 4\frac{\log\log x}{\log x}\right)$$

in quanto $2\log\log x/\log x < 0,18$ per $x \geq 3$. Quindi

$$\pi(x) \leq \frac{x}{\log}\left(\log 4 + \frac{4\log 4\log\log x + 1}{\log x}\right)$$

e infine $4\log 4\log\log x + 1 \leq 8\log\log x$ per $x \geq 5$ mentre per gli altri due valori la disuguaglianza può essere verificata direttamente.

Rimane da far vedere che il prodotto dei primi x numeri primi è minore di 4^x. Procedendo per induzione su $x \geq 3$, osserviamo che se x è pari, allora x non è primo e quindi

$$\prod_{p \leq x} p = \prod_{p \leq x-1} \leq 4^{x-1} < 4^x.$$

Se x è dispari, allora scriviamo $x = 2m+1$. Consideriamo l'intero $\binom{2m+1}{m} = (2m+1)!/(m!(m+1)!)$ che è divisibile per tutti i primi p con $m+1 < p \leq 2m+1$. Pertanto

$$\left(\prod_{m+1<p\leq 2m+1} p \right) \Bigg| \binom{2m+1}{m}$$

Notare inoltre che

$$\begin{aligned} 2^{2m+1} = & \ (1+1)^{2m+1} = \\ = & \ \sum_{i=0}^{2m+1} \binom{2m+1}{j} \geq \binom{2m+1}{m} + \binom{2m+1}{m+1} = 2\binom{2m+1}{m}. \end{aligned}$$

Quindi $\binom{2m+1}{m} \leq 4^m$ per induzione su $x = 2m+1$,

$$\prod_{p \leq x} p = \prod_{p \leq m+1} p \prod_{m+1<p\leq 2m+1} p \leq 4^{m+1} 4^m = 4^x$$

il che completa la dimostrazione della maggiorazione.

Per quanto riguarda l'altra disuguaglianza, assumiamo $x \geq 7$ (per gli altri valori di x si può verificare a mano). Sia $[1, 2, 3, \ldots, x]$ il minimo comune multiplo di tutti gli interi minori o uguali a x. Dalla bella disuguaglianza

$$[1, 2, 3, \ldots, x] \geq 2^x \qquad (x \geq 7) \tag{2.2}$$

che dimostriamo tra poco, e dal fatto che

$$[1, 2, 3, \ldots x] = \prod_{p \leq x} p^{\nu_p}$$

dove ν_p è il più grande intero tale che $p^{\nu_p} \mid [1, 2, 3, \ldots x]$, otteniamo:

$$2^x \leq [1, 2, 3, \ldots, x] \leq \prod_{p \leq x} p^{\nu_p} \leq x^{\pi(x)}.$$

Dove l'ultima disuguaglianza segue dal fatto che $p^{\nu_p} \leq x$ in quanto p^{ν_p} deve dividere un numero minore o uguale a x. Quindi prendendo i logaritmi otteniamo

$$\pi(x) \geq \log 2 \frac{x}{\log x}.$$

Rimane solo da dimostrare (2.2). L'idea è dovuta a Nair e consiste nel considerare

$$I(m,x) = \int_0^1 t^{m-1}(1-t)^{x-m}dt \qquad (1 \le m \le x).$$

Calcolando l'integrale sostituendo

$$(1-t)^{x-m} = \sum_{i=0}^{x-m} \binom{x-m}{j}(-t)^j,$$

si ottiene facendo i calcoli

$$I(m,x) = \sum_{i=0}^{x-m} \binom{x-m}{j}(-1)^j \frac{1}{m+j}$$

e quindi $I(m,x)$ è un numero razionale il cui denominatore divide $[1,2,\ldots,x]$.

Calcolando invece $I(m,x)$ osservando che:

$$\sum_{m=1}^{x} \binom{x-1}{m-1} s^{m-1} I(m,x) = \int_0^1 (1-t-st)^{x-1}ds = \frac{1}{x}\sum_{m=1}^{x} s^{m-1}$$

da cui, confrontando i coefficienti di s^{m-1},

$$I(m,x) = \frac{1}{x\binom{x-1}{m-1}} = \frac{1}{m\binom{x}{m}}, \qquad (1 \le m \le x),$$

otteniamo che

$$m\binom{x}{m} \mid [1,2,\ldots,x], \qquad (1 \le m \le x).$$

Questo implica

$$x\binom{2x}{x} \mid [1,2,\ldots,x] \mid [1,2,\ldots,2x+1]$$

e

$$(x+1)\binom{2x+1}{x} = (2x+1)\binom{2x}{x} \mid [1,2,\ldots,2x+1].$$

Siccome x e $2x+1$ sono coprimi, ne segue che

$$x(2x+1)\binom{2x}{x} \mid [1,2,\ldots,2x+1]$$

e infine, essendo $\binom{2x}{x}$ il più grande dei $2x+1$ coefficienti binomiali che appaiono nell'espansione di $(1+1)^{2x}$, otteniamo per $x \ge 1$

$$[1,2,\ldots,2x+1] \ge x4^x$$

Da cui per $x \geq 2$

$$[1, 2, \ldots, 2x+1] \geq 24^x = 2^{2x+1}$$

e per $n \geq 4$

$$[1, 2, \ldots, 2x+2] \geq [1, 2, \ldots, 2x+1] \geq 4^{x+1}$$

il che dimostra la disuguaglianza $d_x \geq 2^x$ per $x \geq 9$ mentre per gli altri valori basta controllarla osservando che $d_7 = 420 > 2^7$ e $d_8 = 840 > 2^8$. □

Analizziamo anche una semplice conseguenza del teorema di Chebicev: Se p_n denota l'n–esimo numero primo, allora $p_n \asymp n \log n$. Infatti $\pi(p_n) = n$ e dal fatto che $\frac{p_n}{\log p_n} \asymp n$ è facile dedurre che $\log p_n \asymp \log n$ e dunque

$$p_n \asymp n \log n.$$

Analogamente dal TNP segue che $p_n \sim n \log n$.

Adesso torniamo alla divisibilità. Diremo che una potenza di un primo p^α *divide esattamente* un intero n (e scriveremo $p^\alpha \| n$) se $p^\alpha \mid n$ e $p^{\alpha+1} \nmid n$. Se $p^\alpha \| n$, allora $v_p(n) = \alpha$ è la *valutazione p–adica* dell'intero n. Chiaramente $v_p(n) = 0$ se e solo se $p \nmid n$. Tra le proprietà della valutazione p–adica vi sono le seguenti.

1. $n = \prod_p p^{v_p(n)}$ dove con il simbolo $\prod_p$ intendiamo il prodotto esteso a tutti i numeri primi p. Chiaramente il prodotto sopra è in realtà un prodotto finito esteso ai divisori primi di n;
2. $n|m$ se e solo se per ogni p, $v_p(n) \leq v_p(m)$;
3. $\omega(n) = \sum_p v_p(n)$;
4. $v_p(n) = O(\log n / \log p)$.

La seguente nozione è tra le più importanti dell'argomento:

Definizione. Dati $a, b \in \mathbb{Z}$, diremo che un intero positivo d è un *massimo comun divisore* di a e b se

1. $d|a$, $d|b$;
2. per ogni c tale che $c|a$ e $c|b$, si ha $c|d$.

dalla seconda proprietà e dal fatto di insistere che il massimo comun divisore è positivo segue subito la sua unicità. Si ha inoltre la seguente

Proposizione 3 *Comunque scelti due interi a e b esiste il massimo comun divisore di a e b.*

Alla luce della precedente proposizione, indicheremo il massimo comun divisore di a e b con il simbolo (a, b). Esistono varie dimostrazioni dell'esistenza del massimo comun divisore. Noi ne daremo due. La prima è elegante ma totalmente inutile da un punto di vista algoritmico in quanto usa il TFA.

Dimostrazione. Per le proprietà delle valutazioni p–adiche abbiamo

$$n = \prod_p p^{v_p(n)}, \quad m = \prod_p p^{v_p(m)}.$$

Basta verificare che il numero

$$\prod_p p^{\min\{v_p(n), v_p(m)\}}$$

verifica i due assiomi di massimo comun divisore usando la proprietà 2. delle valutazioni p–adiche. □

Esempio. Se $n = 4200 = 2^3 \cdot 3 \cdot 5^2 \cdot 7$ e $m = 10780 = 2^7 \cdot 5 \cdot 7 \cdot 11$, allora $(4200, 10790) = 2^7 \cdot 5 \cdot 7 = 140$.

Se la fattorizzazione di n e m è data, è facile vedere che

$$\mathfrak{T}((m, n)) = O(\log^4 nm).$$

Il problema è che nella pratica incontreremo numeri per i quali la fattorizzazione non è nota. La precedente stima e quindi di scarsa utilità. Pre–fattorizzare il numero per calcolare il massimo comun divisore non è il modo di procedere. Dedicheremo il prossimo paragrafo ad una dimostrazione alternativa dell'esistenza del massimo comun divisore.

2.1 L'algoritmo di Euclide

Alcuni definiscono l'algoritmo di Euclide come il padre di tutti gli algoritmi. È sorprendente il fatto che dopo migliaia di anni il modo più efficiente per calcolare il massimo comun divisore di due interi sia ancora l'algoritmo di Euclide.

Teorema 4 *Esiste un algoritmo che permette di calcolare il massimo comun divisore (a, b) di a e b. Utilizzando questo algoritmo si ha questo algoritmo si ha*

$$\mathfrak{T}_{EUCL}((a, b)) = O(\log a \log b).$$

Dimostrazione. A meno di scambiare a e b, possiamo supporre che $a \geq b$. Se fosse $b = 0$, allora si avrebbe (con poco sforzo) $(a, 0) = a$. Dunque assumiamo $a \geq b > 0$. Consideriamo la sequenza di divisioni euclidee:

$$\begin{array}{rcll} a & = & b \cdot q_0 & + \quad r_1 \\ b & = & r_1 \cdot q_1 & + \quad r_2 \\ r_1 & = & r_2 \cdot q_2 & + \quad r_3 \\ r_2 & = & r_3 \cdot q_3 & + \quad r_4 \\ r_3 & = & r_4 \cdot q_4 & + \quad r_5 \\ & \vdots & & \vdots \\ r_{k-2} & = & r_{k-1} \cdot q_{k-1} & + \quad r_k \\ r_{k-1} & = & r_k \cdot q_k & \end{array}$$

Dove $r_k \neq 0$ è l'ultimo resto non nullo. Esiste un resto non nullo in quanto $b > r_1 > r_2 > \cdots > r_k \geq 0$ è una successione strettamente decrescente di interi positivi che prima o poi deve annullarsi. Si ha che $r_k = (a, b)$ infatti si ha la catena di identità

$$(a, b) = (b, r_1) = (r_1, r_2) = (r_2, r_3) = \cdots = (r_{k-1}, r_k) = (r_k, 0) = r_k.$$

Di queste identità è sufficiente dimostrare la prima e tutte le altre seguono per induzione. In $(a, b) = (b, r_1)$ in quanto si ha $(a, b)|b$ e $(a, b)|r_1 = a - q_0 b$ e perció $(a, b)|(b, r_1)$. Analogamente $(b, r_1)|b$ e $(b, r_1)|a = bq_0 + r_1$ implica $(b, r_1)|(a, b)$ e quindi $(a, b) = (b, r_1)$.

Rimane da stimare il numero di operazioni bit necessarie a fare tutte queste divisioni: La prima divisione richiede $O(\log b \log q_0)$ operazioni bit, la seconda ne richiede $O(\log r_1 \log q_1)$ e in generale la $j + 1$–esima divisione richiede $O(\log r_j \log q_j)$. Sommando tutti questi tempi e usando il fatto che $\log r_i \leq \log b$, otteniamo che

$$\mathfrak{T}_{\text{EUCL}}((a, b)) = O(\log b \sum_{j=0}^{k} \log q_k) = O(\log b \log(q_0 \cdots q_k)).$$

L'ultima osservazione è che $a \geq q_0 \cdots q_k$ in quanto

$$\begin{array}{rl} a = bq_0 + r_1 \geq bq_0 \geq (r_1 q_1 + r_2)q_0 & \geq r_1 q_1 q_0 \geq \cdots \\ \cdots & \geq r_k q_k q_{k-1} \cdots q_0 \geq q_k q_{k-1} \cdots q_0, \end{array}$$

e la stima segue. □

Il tempo necessario a calcolare il massimo comun divisore di due interi è pertanto dello stesso ordine di grandezza a quello necessario a moltiplicarli. Sebbene tra i più usati, l'algoritmo euclideo non è l'unico algoritmo efficiente. Un'altro esempio di un algoritmo per calcolare il massimo comun divisore in modo efficiente è l'*algortimo MCD–binario*.

ALGORITMO MCD-BINARIO

(a,b)	$=$	`if`	$a < b$	`then`	(b,a)
		`if`	$b = 0$	`then`	a
		`if`	$2 \mid a, 2 \mid b$	`then`	$2(a/2, b/2)$
		`if`	$2 \mid a, 2 \nmid b$	`then`	$(a/2, b)$
		`if`	$2 \nmid a, 2 \mid b$	`then`	$(a, b/2)$
				`else`	$((a-b)/2, b)$

L'algoritmo MCD–binario è stato scoperto da J. Stein nel 1967 e ha il vantaggio che come operazioni prevede soltanto differenze e divisioni per due che richiedono tempo lineare. Per i computer la divisione per due corrisponde allo "spegnimento" dell'ultimo bit che è un operazione estremamente veloce. Per questo motivo l'algoritmo di Stein è più veloce di quello euclideo.

Ad ogni iterazione almeno uno dei due argomenti a o b viene sostituito da un argomento che ha almeno un bit in meno. Dunque l'algoritmo termina in meno di k iterazioni se $2^k > ab$. In totale quindi

$$\mathfrak{T}_{\text{BIN–MCD}}((a,b)) = O(\log^2 ab)$$

che è lo stesso ordine di grandezza del numero di operazioni bit necessarie nell'algoritmo di Euclide.

Diamo un esempio del calcolo del massimo comun divisore di usando i due algoritmi:

Esempio. $(1547, 560) = 7$
Algoritmo Euclideo:

$$\begin{aligned} 1547 &= 2 \cdot 560 + 427 \\ 560 &= 1 \cdot 427 + 133 \\ 427 &= 3 \cdot 133 + 28 \\ 133 &= 4 \cdot 28 + 21 \\ 28 &= 1 \cdot 21 + 7 \qquad \leftarrow \text{MCD} \\ 21 &= 3 \cdot 7 \end{aligned}$$

Algoritmo MCD–binario:

$$\begin{aligned} 1.\quad (1547, 560) &= (1547, 280) \\ 2.\quad (1547, 280) &= (1547, 140) \\ 3.\quad (1547, 140) &= (1547, 70) \\ 4.\quad (1547, 70) &= 1547, 35) \\ 5.\quad (1547, 35) &= (756, 35) \\ 6.\quad (756, 35) &= (378, 35) \\ 7.\quad (378, 35) &= (189, 35) \\ 8.\quad (189, 35) &= (77, 35) \\ 9.\quad (77, 35) &= (35, 21) \\ 10.\quad (35, 21) &= (7, 21) \\ 11.\quad (21, 7) &= (7, 7) \\ 12.\quad (7, 7) &= (7, 0) \\ 13.\quad (7, 0) &= 7 \end{aligned}$$

L'algoritmo di Euclide ha anche un altro vantaggio, cioè quello di produrre anche un'identità di Bezout.

Definizione. Se a, b sono interi, dei *coefficienti di Bezout* di a e b è una soluzione (α, β) dell'equazione

$$(a, b) = \alpha \cdot a + \beta \cdot b.$$

È immediato verificare che se (α, β) sono coefficienti di Bezout di a e b, allora anche $(\alpha + kb, \beta - ka)$ per ogni scelta di $k \in \mathbb{Z}$. Quindi i coefficienti di Bezout non sono unici. Però esistono sempre e siamo in grado di calcolarli in modo efficiente.

Si ha infatti

Teorema 5 *Dati a e b interi esistono coefficienti di Bezout (α, β) che possono essere calcolati in $O(\log a \log b)$ operazioni bit.*

Dimostrazione. Assumiamo (come è lecito) che $a \geq b > 0$ e consideriamo l'algoritmo euclideo per calcolare (a, b). Questo produce i quozienti $q_0, \cdots, q_k$ e i resti

$r_1, \ldots, r_k$. Affermiamo che ogni resto si può scrivere come combinazione lineare a coefficienti interi. Cioè che esistono $\alpha_1, \ldots, \alpha_k$ e $\beta_1, \ldots, \beta_k$ tali che per ogni $i = 1, \ldots, k$

$$r_i = \alpha_i \cdot a + \beta_i \cdot b.$$

Essendo $(a, b) = r_k$, questo dimostra l'esistenza dei coefficienti di Bezout (in numeri (α_i, β_i), $i = 0, \ldots, k$ si chiamano *coefficienti di Bezout parziali*). Inoltre si hanno le formule ricorsive:

$$\begin{cases} \alpha_0 = 0 \\ \alpha_1 = 1 \\ \alpha_i = \alpha_{i-2} - q_{i-1} \cdot \alpha_{i-1} \end{cases} \qquad \begin{cases} \beta_0 = 1 \\ \beta_1 = -q_0 \\ \beta_i = \beta_{i-2} - q_{i-1} \cdot \beta_{i-1} \end{cases} \tag{2.3}$$

che in forma matriciale possono essere riscritte nel seguente modo:

$$\begin{pmatrix} \alpha_0 & \alpha_1 \\ \beta_0 & \beta_1 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & -q_0 \end{pmatrix}, \qquad \begin{pmatrix} \alpha_i \\ \beta_i \end{pmatrix} = \begin{pmatrix} \alpha_{i-2} & \alpha_{i-1} \\ \beta_{i-2} & \beta_{i-1} \end{pmatrix} \begin{pmatrix} 1 \\ -q_{i-1} \end{pmatrix}.$$

Dimostriamo per induzione che $r_i = \alpha_i a + \beta_i b$: $r_1 = a - q_0 b$ quindi è vero che $(\alpha_1, \beta_1) = (1, -q_0)$, inoltre $r_2 = b - q_1 r_1 = -q_1 a + (1 + q_0 q_1) b$. Dunque $(\alpha_2, \beta_2) = (-q_1, 1 + q_0 q_1) = (\alpha_0 - q_1 \alpha_1, \beta_0 - q_1 \beta_1)$. In generale

$$r_i = r_{i-2} - q_{i-1} r_{i-1} = \alpha_{i-2} \cdot a + \beta_{i-2} \cdot b - q_{i-1}(\alpha_{i-1} \cdot a + \beta_{i-1} \cdot b) = \alpha_i \cdot a + \beta_i \cdot b$$

dove (α_i, β_i) sono quelli definiti sopra.

Rimane da mostrare che possiamo svolgere questi calcoli in $O(\log a \log b)$ operazioni bit il che è la parte più delicata della dimostrazione. Si ha che per $i \geq 2$

$$\begin{aligned} |\alpha_i| &\leq (1 + q_1)(1 + q_2) \cdots (1 + q_{i-1}) \\ |\beta_i| &\leq (1 + q_0)(1 + q_1) \cdots (1 + q_{i-1}). \end{aligned}$$

Infatti

$$|\alpha_2| = q_1 \leq (1 + q_1), \quad |\alpha_3| = 1 + q_1 q_2 \leq (1 + q_1)(1 + q_2)$$

e induttivamente

$$\begin{aligned} |\alpha_i| &\leq |\alpha_{i-2}| + q_{i-1} |\alpha_{i-1}| \\ &\leq (1 + q_1) \cdots (1 + q_{i-3})[1 + q_{i-1}(1 + q_{i-2})] \\ &\leq (1 + q_1) \cdots (1 + q_{i-1}). \end{aligned}$$

L'argomento per β_i è lo stesso e lo omettiamo.

Da queste stime deduciamo che il numero di cifre binarie di α_i è $O(\log a)$. Infatti

$$\log |\alpha_i| \leq \sum_{j \leq k} \log(1 + q_j) = \log(q_1 \cdots q_k) + \sum_{j \leq k} \log(1 + q_j^{-1})$$

e osservando che $(1 + q_j^{-1}) \leq 2$, otteniamo

$$\log |\alpha_i| \leq \log a + k \log 2$$

dove k è il numero di divisioni nell'algoritmo di euclide per calcolare (a, b).

Infatti affermiamo che dati $a, b \in \mathbb{N}$ con $a \geq b > 0$, il numero k di divisioni nell'algoritmo di euclide per calcolare (a, b) verifica

$$k = O(\log b).$$

Da cui segue subito che $\log |\alpha_i| = O(\log a)$ e pertanto che α_i ha $O(\log a)$ cifre.

Se $r_0 = b, r_1, \ldots, r_k$ sono i resti della divisione euclidea, allora per ogni $i = 0, \ldots, k-2$ si ha che $r_{i+2} \leq \frac{1}{2} r_i$. Infatti se $r_{i+1} \leq \frac{1}{2} r_i$ allora a maggior ragione $r_{i+2} < r_{r+1} \leq \frac{1}{2} r_2$. Se invece $r_{i+1} > \frac{1}{2} r_i$, allora da $r_i = q_{i+1} r_{i+1} + r_{r+2}$ otteniamo che $q_{i+1} = 1$ e $r_{i+2} = r_i - r_{i+1} < \frac{1}{2} r_{i+1}$.

Adesso dalle disuguaglianze $1 \leq r_k < \frac{1}{2} r_{k-2} < \ldots < \frac{1}{2^{(k+1)/2}} r_0$ e dal fatto che $b = r_0$, otteniamo $k = O(\log b)$.

Adesso notiamo che per ogni $j > 1$ se $\alpha_2, \ldots, \alpha_{j-1}$ sono stati calcolati, allora

$$\mathfrak{T}(\alpha_j) = O(\log a \log q_1 \cdots q_{j-1}).$$

Infatti $\alpha_j = \alpha_{j-2} - q_{j-1} \alpha_{j-1}$, il prodotto $q_{j-1} \alpha_{j-1}$ richiede

$$O(\log q_{j-1} \log |\alpha_{j-1}|) = O(\log q_{j-1} \log a)$$

operazioni bit e la sottrazione ne richiede $O(\max\{\log |\alpha_{i-2}|, \log q_{i-1} |\alpha_{i-1}|\}) = O(\log a)$. Infine

$$\mathfrak{T}(\alpha_2, \ldots, \alpha_k) = \sum_{2 \leq j \leq k} O(\log a \log q_{j-1}) = O(\log a \log q_1 \cdots q_{k-1}).$$

Il che è $O(\log a \log b)$ in quanto $b \geq q_1 \cdots q_{k-1}$. □

Esempio. Tornando al massimo comun divisore $(1547, 560) = 7$, notiamo che in questo caso i quozienti sono: $2, 1, 3, 4, 1$ pertanto

i	q_i	α_i	β_i
0	2	0	1
1	1	1	-2
2	3	-1	3
3	4	4	-11
4	1	-17	47
5		21	-58

infatti $7 = 21 \cdot 1547 - 58 \cdot 560$.

Due numeri interi a, b si dicono *coprimi* se $(a, b) = 1$. Questo implica che esistono α, β tali che $\alpha \cdot a + \beta \cdot b = 1$. Quest'ultima identità caratterizzà la nozione di coprimalità infatti se esistono α, β tali che $\alpha \cdot a + \beta \cdot b = 1$ allora necessariamente $(a, b) = 1$. In altre parole

Proposizione 4 *Dati due numeri interi $a, b \in \mathbb{Z}$, l'equazione*

$$aX + bY = 1$$

ammette soluzione in interi X, Y se e solo se $(a, b) = 1$. □

2.2 La funzione di Eulero

La funzione di eulero ha un ruolo importante in crittografia. Conta il numero dei numeri positivi coprimi e minori del suo argomento. Cioè se $n \in \mathbb{N}$

$$\varphi(m) = \#\{n \in \mathbb{N}, 1 \leq n \leq m, (n, m) = 1\}.$$

Dalla definizione segue subito che

1. $\varphi(1) = 1$;
2. $\varphi(p) = p - 1$ (se p è un numero primo);
3. $\varphi(p^\alpha) = p^\alpha - p^{\alpha-1}$.

Inoltre dimostreremo più avanti che

$$\varphi(nm) = \varphi(n)\varphi(m) \qquad \text{se } (n, m) = 1.$$

Da quest'ultima proprietà otteniamo immediatamente che se $m = p_1^{\alpha_1} \cdots p_s^{\alpha_s}$ è la fattorizzazione di m come prodotto di primi, allora

$$\varphi(m) = m \prod_{i=1}^{s} \left(1 - \frac{1}{p_i}\right). \tag{2.4}$$

Da questa identità otteniamo il fatto che se la fattorizzazione di m è nota, allora è facile calcolare $\varphi(m)$. Infatti

Proposizione 5 *Se sono noti i fattori primi $p_1, \ldots, p_s$ di m, allora*

$$\mathfrak{T}(\varphi(m)) = O(\log^3 m).$$

Dimostrazione. Dalla formula (2.4) deduciamo che è possibile calcolare $\varphi(m)$ dividendo m per $p_1, p_2, \ldots, p_s$ e poi moltiplicando il risultato per $p_1 - 1, p_2 - 1, \ldots, p_s - 1$.

Dunque sono necessarie s divisioni e s moltiplicazioni. La i–esima divisione è tra il numero $m/(p_1 \cdots p_{i-1})$ e p_i. Quindi tra un numero minore di m^2 e un numero minore di m. Pertanto questa divisione richiede $O(\log^2 m)$ operazioni bit. La i–esima moltiplicazione che segue poi sarà tra il numero $m \cdot (p_1 - 1) \cdots (p_{i-1} - 1)/(p_1 \cdots p_s)$ e $(p_i - 1)$. Anche questa operazione richiede $O(\log^2 m)$ operazioni bit. Infine

$$\mathfrak{T}(\varphi(m)) = O(s \log^2 m)$$

e l'enunciato segue dal fatto che $s = \nu(m) = O(\log m)$ come mostrato in (2.1). □

Nel caso in cui i primi della fattorizzazione di m non sono noti, può essere molto difficile calcolare $\varphi(m)$. Infatti non si conosce alcun algoritmo generale che in tempo polinomiale permetta di calcolare φ. Se un tale algoritmo venisse scoperto, renderebbe tutti i protocolli crittografici della famiglia RSA totalmente insicuri. Gli esperti tendono a pensare che un tale algoritmo non esiste. Tuttavia non è possibile dimostrarlo.

Nel caso speciale in cui $m = p_1 \cdot p_2$, conoscere il valore di $\varphi(m)$ permettere di calcolare rapidamente i fattori p_1 e p_2. Si ha infatti

Proposizione 6 *Se $m = p_1 p_2$ è il prodotto di due numeri primi e il valore di $\varphi(m)$ è dato allora*

$$\mathfrak{T}(\{p_1, p_2\}) = O(\log^3 m).$$

Viceversa se p_1 e p_2 sono noti allora

$$\mathfrak{T}(\varphi(m)) = O(\log m).$$

Dimostrazione. Scrivendo $m = xy$, e $\varphi(m) = (x-1)(y-1)$ ed eliminando y da queste due equazioni otteniamo $x\varphi(m) = (x-1)(m-x)$ e dunque

$$x^2 + (\varphi(m) - m - 1)x + m = 0.$$

Risolvendo l'equazione di secondo grado si trova

$$x_{1,2} = \frac{(m + 1 - \varphi(m)) \pm \sqrt{(\varphi(m) - m - 1)^2 - 4m}}{2}.$$

Chiaramente le soluzioni sono i primi cercati. Per fare il calcolo sono necessarie $O(\log^3 m)$ operazioni bit in quanto la radice quadrata richiede $O(\log^3 m)$ operazioni bit con l'algoritmo della Proposizione 2 e le altre operazioni richiedono $O(\log^2 m)$ operazioni bit.

Il viceversa si può dedurre facilmente dal fatto che $\varphi(m) = m - p_1 - p_2 + 1$ e quindi è possibile calcolarla con due sottrazioni e un'addizione. □

Vedremo nel prossimo paragrafo che $\varphi(m)$ è l'ordine di un gruppo che useremo estesamente. È naturale domandarsi quanto sia grande $\varphi(m)$ rispetto ad m. Chiaramente $\varphi(m) \leq m$. Mostreremo che

Proposizione 7 *Ogni intero m verifica.*

$$\varphi(m) \geq \frac{m}{\frac{6}{\pi^2}(\log m + 1)}.$$

Dimostrazione. Dalla formula (2.4) segue che è sufficiente mostrare che se $\prod_{p|m}$ indica il prodotto esteso a tutti i fattori primi di m, allora

$$\prod_{p|m}\left(1-\frac{1}{p}\right)^{-1} \leq \frac{\pi^2}{6}(\log m + 1).$$

Adesso usiamo il fatto che

$$\begin{aligned}\prod_{p|m}\left(1-\frac{1}{p}\right)^{-1} = \prod_{p|m}\left(1-\frac{1}{p^2}\right)^{-1}\cdot\prod_{p|m}\left(1+\frac{1}{p}\right) \leq \\ \prod_{p \text{ primo}}\left(1-\frac{1}{p^2}\right)^{-1}\cdot\prod_{p|m}\left(1+\frac{1}{p}\right) = \frac{\pi^2}{6}\prod_{p|m}\left(1+\frac{1}{p}\right).\end{aligned}$$

L'ultima uguaglianza segue dalla formula classica

$$\prod_{p \text{ primo}}\left(1-\frac{1}{p^2}\right)^{-1} = \sum_{m=1}^{\infty}\frac{1}{m^2} = \frac{\pi^2}{6}.$$

Adesso notiamo che per il teorema fondamentale dell'aritmetica,

$$\prod_{p|m}\left(1+\frac{1}{p}\right) \leq \sum_{n|m}\frac{1}{n} \leq \sum_{n\leq m}\frac{1}{n}$$

Utilizzando le disuguaglianze

$$\sum_{n<m}\frac{1}{n} \leq 1 + \int_1^m dt/t = 1 + \log m.$$

Mettendo tutto insieme otteniamo l'enunciato. □

Il precedente risultato non è il migliore. È possibile mostrare che se $m \geq 5$, allora

$$\varphi(m) \geq 6\frac{m}{\log\log m}.$$

Per la dimostrazione si veda [2]. Infine è possibile dimostrare che in media il valore di $\varphi(m)$ è $0.61m$. Ciò è espresso dal risultato classico:

$$\sum_{m\leq x} \varphi(m) \sim \frac{6}{\pi^2}m.$$

Anche se utili e affascinanti, questi risultati esulano dagli obbiettivi di questo corso e rimandiamo a testi classici come [5] per la dimostrazione.

Esercizi.

1. Calcolare il massimo comun divisore $(30, 125)$ usando sia l'algoritmo di Euclide che quello MCD-binario. Poi calcolare l'identità di Bezout usando i quozienti ottenuti nell'algoritmo di Euclide.

2. Calcolare il massimo comun divisore tra 240 e 180 utilizzando sia l'algoritmo euclideo che quello binario. Calcolare anche l'identità di Bezout.

   Soluzione. L'esecuzione dell'algoritmo di Euclide per calcolare $(240, 180)$ è

   $$240 = 180 + 60; \;\; 180 = 3 \cdot 60 + 0; \;\; (240, 180) = 60.$$

   mentre quella di quello binario è

   $$(240, 180) = 4(60, 45) = 4(15, 45) = 4((45 - 15)/2, 15) =$$

   $$4(15, 15) = 4(0, 15) = 4 \cdot 15 = 60.$$

   L'identità di Bezout è $60 = 240 - 180$.

Capitolo 3

L'aritmetica in $\mathbb{Z}/m\mathbb{Z}$

In molti conoscono l'anello $\mathbb{Z}/m\mathbb{Z}$ e sono abituati a lavorare con esempi di tali anelli in cui m è piccolo. Nelle applicazioni crittografiche m è solitamente dell'ordine di 10^{300} che è un numero molto grande. È quindi opportuno sapere cosa possiamo e cosa non possiamo fare con gli elementi di questo anello. Nel primo caso è anche utile sapere come fare ciò che vogliamo fare.

Premettiamo un breve richiamo sulle congruenze che ci è utile per fissare le notazioni. Diremo che due interi a, b sono *congruenti modulo* m e scriveremo $a \equiv b \bmod m$ se $m|a-b$. Alle volte scriveremo anche $a \equiv_m b$ oppure $a = b$ se la natura dell'uguaglianza è chiara nel contesto.

La congruenza è una relazione di equivalenza l'insieme delle classi di equivalenza rispetto alla congruenza è $\mathbb{Z}/m\mathbb{Z}$. Ogni intero a è congruente al resto della sua divisione per m. Indicheremo con $a \bmod m$ tale resto. Questo fatto permette di scegliere gli interi tra 0 e $m-1$ come rappresentanti delle classi di questa relazione. Così facendo si identifica

$$\mathbb{Z}/m\mathbb{Z} \simeq \{0, 1, \ldots, m-1\}.$$

mediante l'applicazione che manda la classe dell'intero a in $a \bmod m$. Noi confonderemo i due insiemi usando il secondo al posto del primo.

La relazione di congruenza è compatibile con le operazioni di somma e moltiplicazione. Ciò implica che $\mathbb{Z}/m\mathbb{Z}$ ha la struttura di anello commutativo dove se $a, b \in \mathbb{Z}/m\mathbb{Z}$ (cioè a e b sono interi con $0 \leq a, b < m$), allora

$$a +_m b = (a+b) \bmod m \begin{cases} a+b & \text{se } a+b < m \\ a+b-m & \text{se } a+b \geq m \end{cases}$$

Quindi l'opposto di a è $m-a$. Inoltre da questo si vede che per sommare due elementi in $\mathbb{Z}/m\mathbb{Z}$ sono necessarie $O(\log m)$ operazione bit.

Per quanto riguarda il prodotto definiamo

$$a \cdot_m b = (ab) \bmod m$$

quindi per moltiplicare due numeri in $\mathbb{Z}/m\mathbb{Z}$ basta moltiplicare due numeri minori di m e poi dividere il risultato per m. Ciò richiede $O(\log^2 m)$ operazioni bit.

Indichiamo con $U(\mathbb{Z}/m\mathbb{Z})$ il gruppo delle unità di $\mathbb{Z}/m\mathbb{Z}$. Sappiamo che tale gruppo ha $\varphi(m)$ elementi in quanto $x \in U(\mathbb{Z}/m\mathbb{Z})$ se e solo se $(x, m) = 1$. l'intero y, $1 \leq y \leq m-1$ che è l'inverso moltiplicativo di x in $U(\mathbb{Z}/m\mathbb{Z})$ si chiama inverso aritmetico di x modulo m e si indica con x^*.

Per calcolare x^*, un volta appurato che $(x, m) = 1$ si può ricorrere all'algoritmo di Euclide e l'identità di Bezout che in $O(\log^2 m)$ operazioni bit produce i coefficienti (α, β) tali che

$$\alpha x + \beta m = 1$$

Calcolando $\alpha \bmod m$ (il che richiede altre $O(\log^2 m)$ operazioni bit) otteniamo x^*. Infatti $x\alpha \equiv xx^* \equiv 1 \bmod m$. Infine

$$\mathfrak{T}(x^*) = O(\log^2 m).$$

Per dare un idea di come avviene il calcolo consideriamo un esempio non numeri di 30 cifre:

Esempio. Sia $m = 738873402423833494183027176791$. Nell'anello $\mathbb{Z}/m\mathbb{Z}$ ci sono molti elementi (circa 10^{30}). Sia $x = 379672818919642755759402217519$. I quozienti $q_0, q_1, \ldots$ con resto non nullo dell'algoritmo di Euclide per $(x, m) = 1$ sono [1]

$$0, 1, 1, 17, 1, 1, 4, 1, 27, 1, 4, 1, 1, 1, 1, 5, 3, 1, 1, 1, 1, 1, 1, 2, 1, 5, 1, 51, 1, 1, 1, 6, 1,$$

$$3, 1, 1, 3, 1, 84, 1, 1, 1, 6, 1, 2, 3, 15, 3, 1, 30, 1, 2, 5, 2, 8, 2, 1, 28, 12, 1, 15, 5, 1, 5$$

Consideriamo i primi 63 valori e per le formule (2.3) per i coefficienti di Bezout parziali [2] otteniamo

$$\alpha_{61} = 1266265360577070807260792549 99,$$
$$\beta_{61} = -650675118868085112223586862 80.$$

[1] In Pari i quozienti si calcolano così:

```
Q=vector(100,x,0);q=m;r=x;j=1;
while(r<>0,S=divrem(q,r);q=r;r=S[2];Q[j]=S[1];j=j+1);
print(MCD=q quozientiQ)
```

[2] In Pari i coefficienti di parziali di Bezout si calcolano così:

```
a=vector(64,x,0);a[1]=0;a[2]=1;
for(j=3,63,a[j]=a[j-2]-a[j-1]*Q[j-1]);a
b=vector(64,x,0);b[1]=1;b[2]=-Q[1];
for(j=3,64,b[j]=b[j-2]-b[j-1]*Q[j-1]);b
```

Pertanto

$$x^* = \alpha_{61} \bmod m = 126626536057707080726079254999.$$

Riassumiamo tutta la discussione nel seguente

Teorema 6 *$\mathbb{Z}/m\mathbb{Z}$ è un anello commutativo unitario con m elementi in cui le somme richiedono tempo lineare, i prodotti tempo quadratico. Il gruppo delle unità di questo anello $U(\mathbb{Z}/m\mathbb{Z})$ contiene $\varphi(m)$ elementi e il calcolo degli inversi $U(\mathbb{Z}/m\mathbb{Z})$ richiede tempo quadratico.* □

3.1 L'algoritmo dei quadrati successivi

Sia $b \in \mathbb{Z}/m\mathbb{Z}$ e $n \in \mathbb{N}$. Per calcolare $b^n \bmod m$ (cioé il resto della divisione di b^n per m) possiamo procedere in vari modi. Il primo (e il peggiore) consiste nel calcolare b^n e poi dividere il risultato per m. La prima parte richiede $O(n^2 \log m)$ operazioni bit mentre la seconda richiede $O(n \log^2 m)$ operazioni bit. Quindi il tempo è necessario è in tutto $O(n^2 \log^2 m)$ e ciò vuol dire che in pratica il calcolo non è fattibile.

Un secondo modo di procedere consiste nel calcolare

$$b,\ b^2 \bmod m,\ b^3 \bmod m,\ \ldots\ ,\ b^n \bmod m.$$

Questo metodo richiede in ogni passo $O(\log^2 m)$ operazioni bit e in tutto sono necessari n passi. Il tempo in questo caso è $O(n \log^2 m)$ che rimane esponenziale in n.

Esiste per fortuna un algoritmo che permette di migliorare sensibilmente la stima in n. Si tratta dell'*algoritmo dei quadrati successivi*.

Teorema 7 *Dati $n \in \mathbb{N}$ e $b \in \mathbb{Z}/m\mathbb{Z}$, esiste un algoritmo $\mathcal{A}$ per calcolare $b^n \bmod m$ tale che*

$$\mathfrak{T}_{\mathcal{A}}(b^n \bmod m) = O(\log n \log^2 m).$$

Dimostrazione. Supponiamo che n abbia k cifre binarie e scriviamo

$$n = (\epsilon_{k-1} \cdots \epsilon_1 \epsilon_0)_2.$$

Calcoliamo nell'ordine le $k-1$ potenze in $\mathbb{Z}/m\mathbb{Z}$

$$b^2 \bmod m,\ (b^2)^2 \bmod m,\ (b^{2^2})^2 \bmod m,\ \ldots\ ,\ (b^{2^{k-2}})^2 \bmod m.$$

Ciascun termine è il quadrato di un elemento in $\mathbb{Z}/m\mathbb{Z}$. Pertanto ogni passo richiede $O(\log^2 m)$ operazioni bit ed in tutto sono necessarie

$$O(k \log^2 m) = O(\log n \log^2 m)$$

operazioni bit. Consideriamo anche la successione di elementi di $\mathbb{Z}/m\mathbb{Z}$ ottenuti come segue:

$$c_0 = \begin{cases} b \bmod m & \text{se } \epsilon_0 = 1 \\ 1 \bmod m & \text{se } \epsilon_0 = 0 \end{cases} \quad c_1 = \begin{cases} c_0 b^2 \bmod m & \text{se } \epsilon_1 = 1 \\ c_0 \bmod m & \text{se } \epsilon_1 = 0 \end{cases} \quad \dots$$

$$\dots \; c_i = \begin{cases} c_{i-1} b^{2^i} \bmod m & \text{se } \epsilon_i = 1 \\ c_{i-1} \bmod m & \text{se } \epsilon_i = 0 \end{cases} \quad \dots$$

che nella pratica si calcolano contemporaneamente ai precedenti. Per calcolare ogni c_i sono necessarie $O(\log^2 m)$ operazioni bit in quanto $b^{2^i} \bmod m$ è stato già calcolato nella prima parte dell'esecuzione. Inoltre $c_i = c_{i-1} b^{\epsilon_i 2^i}$, quindi

$$c_{k-1} \equiv b^{\epsilon_0} b^{\epsilon_1 2} \cdots b^{\epsilon_{k-1} 2^{k-1}} \bmod m \equiv b^{\epsilon_0 + \epsilon_1 2 + \cdots + \epsilon_{k-1} 2^{k-1}} \bmod m \equiv b^n \bmod .$$

Infine il numero di operazioni bit necessarie a calcolare c_{k-1} è $O(k \log^2 m)$ e questo conclude la dimostrazione. □

L'algoritmo dei quadrati successivi può essere descritto sotto forma di programma (in un linguaggio inventato al momento) nel seguente modo:

ALGORITMO DEI QUADRATI SUCCESSIVI

```
1.  B = b, C = b^{ε_0}
2.  FOR j from 1 to k − 1
3.  B = B^2 (mod m) ; C = C ∗ B^{ε_j} (mod m)
4.  ROF
5.  PRINT C.
```

Osservazione. L'algoritmo dei quadrati successivi è molto generale e non si applica solo alle potenze di $\mathbb{Z}/m\mathbb{Z}$. Se G è un qualsiasi gruppo algebrico finito e t_G è il massimo numero di operazioni bit necessarie a moltiplicare (secondo l'operazione del gruppo) due elementi. Allora preso comunque $b \in G$ e $n \in \mathbb{N}$,

$$\mathfrak{T}(b^n) = O(t_G \log n).$$

La descrizione dell'algoritmo (e la sua analisi) del caso della potenza di un elemento in un gruppo algebrico è completamente analoga a quella per le potenze degli elementi di $\mathbb{Z}/m\mathbb{Z}$. Nei prossimi capitoli considereremo i caso in cui il gruppo è i gruppo moltiplicativo di un campo finito o il caso in cui il gruppo è il gruppo $E(\mathbb{F}_q)$ dei punti razionali di una curva ellittica definita su un campo finito $\mathbb{F}_q$.

Esempio. Vogliamo calcolare[3] il valore di $40^{1999} \bmod 1492$. Partiamo dall'espansione binaria $1999 = (1,1,1,1,1,0,0,1,1,1,1)_2$. Se $b_i = 40^{2^i} \bmod 1492$, e $c_i = c_{i-1} b_i^{\epsilon_i}$ allora

i	0	1	2	3	4	5	6	7	8	9	10
b_i	40	108	1220	876	488	916	552	336	996	1328	40
c_i	40	1336	656	236	236	236	468	569	784	1228	1376

Pertanto $40^{1999} \bmod 1492 = 1376$. Notare che un qualsiasi pacchetto in cui l'aritmetica modulare è implementata (come MAGMA, MAPLE, PARI, MATHEMATICA) le potenze vengono sempre calcolate usando l'algoritmo dei quadrati successivi.

Esercizi.

1. Si calcoli
$$62400^{5461} \pmod{61345}$$
utilizzando il metodo dei quadrati successivi. (RISPOSTA 39205)

2. Si dia una stima per il numero di operazioni bit necessarie al calcolo della parte intera della norma di un vettore di $\mathbf{R^n}$ assumendo che tutte le coordinate sono minori di 1000000.

3. Dimostrare che $n^7 - n$ è sempre divisibile per 42.

4. Si calcoli 2^{300} modulo 23 e si calcoli anche l'espansione binaria di $(2^{150}-1)/(2^{30}-1)$.

5. Quante operazioni bit sono necessarie per calcolare $(n!)^{n!}$ modulo m?

6. Si fattorizzi completamente $5^{24} - 1$.

7. Mostrare che se a, b e c sono interi positivi con $a, b, c \leq p$ (p primo), allora è possibile calcolare $a^{(b^c)} \bmod p$ in $O(\log^3 p)$ operazioni bit. *(Sugg. Usare sia l'algoritmo delle potenze successive che il Piccolo Teorema di Fermat.)*

[3] In Pari si fa così:
```
b=40;m=1492;n=1999;N=binary(n);L=matsize(N)[2];B=b;C=bˆN[1];
for(j=1,L-1,B=(Bˆ2)%m;C=(C*BˆN[L-j])%m);print(C)
```

8. Si descriva un'algoritmo per calcolare in tempo polinomiale $2^m \pmod{m+1}$. Si stimi anche il numero di operazioni bit necessarie.

9. Mostrare che le moltiplicazioni nell'anello quoziente $\mathbf{Z}/n\mathbf{Z}[x]/(x^d)$ si possono calcolare in $O(\log^2 n^d)$ operazioni bit mentre le addizioni in $O(\log n^d)$ operazioni bit.

   Soluzione. Scriviamo

   $$f_1 = \sum_{j=0}^{d-1} a_j x^j,\ f_2 = \sum_{j=0}^{d-1} b_j x^j \in \mathbf{Z}/n\mathbf{Z}[x]/(x^d).$$

   Allora $f_1+f_2 = \sum_{j=0}^{d-1}(a_j+b_j)x^j$. Pertanto $\mathfrak{T}(f_1+f_2)$=$\mathfrak{T}(a_0+b_0,\ldots,a_{d-1}+b_{d-1}) = O(d\log n)$. Invece se scriviamo $f_1 f_2 = \sum_{k=0}^{d-1} c_k x^k$, allora $c_k = \sum_{i+j=k} a_i b_j$. Inoltre per ogni $k = 0,\ldots,d-1$, $\mathfrak{T}(c_k)) = O(d\log^2 n)$ (infatti si tratta di k moltiplicazioni e $k-1$ somme in $\mathbf{Z}/n\mathbf{Z}$. Infine

   $$\mathfrak{T}(f_1 f_2) = \mathfrak{T}(c_0,\ldots,c_{d-1})) = O(d^2\log^2 n).$$

Capitolo 4

Altri risultati elementari di Teoria dei numeri

In questo paragrafo richiameremo alcuni risultati di Teoria dei numeri che verranno utilizzati spesso in seguito. Tutte le dimostrazioni sono incluse anche se immaginiamo che il lettore le abbia già viste prima in qualche altro corso.

Proposizione 8 (SOLUZIONI DELLE CONGRUENZE LINEARI) *Siano $a, b \in \mathbb{Z}$ e sia $m \in \mathbb{N}$. Detto $d = (m, a)$, la congruenza*

$$aX \equiv b \bmod m$$

ammette soluzione se e solo se $d|b$. In tal caso la congruenza ammette d soluzioni in $\mathbb{Z}/m\mathbb{Z}$. Se x_0 è la più piccola di tali soluzioni, allora tutte le soluzioni sono $x_k = x_0 + km/d$ ($k = 0, \ldots, d-1$). Inoltre, se $a, b \in \mathbb{Z}/m\mathbb{Z}$, allora tutte le soluzioni possono essere calcolate in $O(d \log^2 m)$ operazioni bit.

Dimostrazione. Se $d = 1$, allora la congruenza ammette come soluzione $ba^* \bmod m$ dove a^* è l'inverso aritmetico di a modulo m. È immediato verificare che tale soluzione è l'unica modulo m. Infine, il numero di operazioni bit necessarie per calcolare $ba^* \bmod m$ sono quelle necessarie per un inversione e una moltiplicazione cioè $O(\log^2 m)$.

Se $d > 1$ e $d|b$, allora la congruenza lineare è equivalente a

$$\frac{a}{d}X \equiv \frac{b}{d} \bmod \frac{m}{d}$$

alla quale è possibile applicare il caso precedente in quanto $a/d \in U(\mathbb{Z}/(m/d)\mathbb{Z})$. La soluzione di tale congruenza da luogo alle d soluzioni nell'enunciato ciascuna delle quali può essere calcolata in $O(\log^2 m)$ operazioni bit.

Infine se la congruenza è risolubile e x è una soluzione, allora $b = xa - km$ quindi $d = (a, m)|b$. □

Proposizione 9 (IL PICCOLO TEOREMA DI FERMAT) *Sia p un numero primo e sia $a \in \mathbb{Z}$ allora*

$$a^p \equiv a \bmod p.$$

Inoltre se $n_1, n_2 \in \mathbb{Z}$ con $n_1 \equiv n_2 \bmod p-1$ e $(a, p) = 1$, allora

$$a^{n_1} \equiv a^{n_2} \bmod p.$$

Dimostrazione. Si tratta di un argomento standard. Se $p|a$ allora l'enunciato è chiaramente verificato. Se $p \nmid a$, allora l'enunciato è equivalente a

$$a^{p-1} \equiv 1 \bmod p.$$

Consideriamo l'insieme $S = \{a, 2a \bmod p, \ldots, (p-1)a \bmod p\}$. Si ha che $S = \mathbb{F}_p^*$ infatti $S \subseteq F_q^*$ e se due degli elementi $aj \bmod p$ e $ai \bmod p$ coincidessero, si avrebbe $j \equiv i \bmod p$. Quindi prodotto degli elementi di S è pari al prodotto degli elementi di $\mathbb{F}_p^*$. Infine

$$\prod_{i=1}^{p-1} ai = a^{p-1} \prod_{i=1}^{p-1} i = \prod_{i=1}^{p-1} i \bmod p$$

e semplificando $(p-1)! \in \mathbb{F}_p^*$ si ottiene il risultato.

Per quando riguarda la seconda parte dell'enunciato, osserviamo che $n_1 \equiv n_2 \bmod p-1$ vuol dire che $n_1 = n_2 + t(p-1)$ per un opportuno intero t. Quindi

$$a^{n_1} = a^{n_2 + t(p-1)} = a^{n_2}(a^{p-1})^t \equiv a^{n_2} 1^t \bmod p.$$

□

Osservazione. Il piccolo Teorema di Fermat può aiutare in alcuni casi a rendere più veloce il calcolo di $b^n \bmod p$. Infatti se per esempio n è molto grande rispetto a p (diciamo $n \asymp e^p$ per fissare le idee) allora l'algoritmo dei quadrati successivi richiede $O(p \log^2 p)$ operazioni bit. Se calcoliamo $n \bmod p-1$ (il che può essere fatto in $O(p \log p)$ operazioni bit) e sfruttiamo il fatto che per il Piccolo Teorema di Fermat

$$a^n \equiv a^{n \bmod p-1} \bmod p,$$

possiamo applicare l'algoritmo dei quadrati successivi a $a^{n \bmod p-1}$. Quindi

$$\mathfrak{T}(a^n \bmod p) = O(p \log p).$$

Così abbiamo risparmiato un fattore $O(\log p)$ di operazioni bit.

Esempi. Calcoliamo $3^{9^{17}} \bmod 11$. Siccome $9^{17} \equiv (-1)^{17} \equiv 9 \bmod 10$, per il piccolo Teorema di Fermat si ha

$$3^{9^{17}} \equiv_{11} 3^9 \equiv_{11} 27^3 \equiv_{11} 5^3 \equiv_{11} 5 \cdot 25 \equiv_{11} 15 \equiv_{11} 4.$$

Se invece vogliamo calcolare l'ultima cifra in base 13 di $2^{10^{50}}$, allora sapendo che

$$10^{50} \equiv_{12} 2^{50} \equiv_{12} 32^{10} \equiv_{12} (-4)^{10} \equiv_{12} 16^5 \equiv_{12} \equiv_{12} 4 \cdot 16^2 \equiv_{12} 4$$

Quindi

$$2^{10^{50}} \equiv_{13} 2^4 \equiv_{13} 3.$$

Proposizione 10 (TEOREMA CINESE DEI RESTI) *Siano $m_1, \ldots, m_s \in \mathbb{N}$ coprimi a due a due e siano $a_1, \ldots, a_s \in \mathbb{Z}$. Posto $M = m_1 \cdots m_s$, esiste un unico elemento $x \in \mathbb{Z}/M\mathbb{Z}$ tale che*

$$\begin{cases} x \equiv a_1 \bmod m_1 \\ x \equiv a_2 \bmod m_2 \\ \quad \vdots \\ x \equiv a_s \bmod m_s. \end{cases}$$

Inoltre se $a_1, \ldots, a_s \in \mathbb{Z}/M\mathbb{Z}$ allora

$$\mathfrak{T}(x) = O(s \log^2 M).$$

Dimostrazione. Per $i = 1, \ldots, s$, poniamo $M_i = M/m_i$. Si ha che $(M_i, m_i) = 1$ quindi possiamo calcolare l'inverso aritmetico N_i di M_i modulo m_i. Infine poniamo

$$x = \sum_{i=1}^{s} a_i N_i M_i \bmod M.$$

Si verifica subito che x è (l'unica) soluzione del sistema di congruenze che, se $0 \leq a_1, \ldots, a_s \leq M$.

Infine notiamo che M può essere calcolato in $O(s \log^2 M)$ operazioni bit. Mentre per ogni $i = 1, \ldots, s$, M_i, N_i e $a_i N_i M_i \bmod M$ possono essere calcolati in $O(\log^2 M)$ operazioni bit. Infine $\sum_{i=1}^{s} a_i N_i M_i \bmod M$ può essere calcolata con s somme in $\mathbb{Z}/M\mathbb{Z}$ e questo richiede $O(s \log M)$. □

Corollario 1 *Se $m, n \in \mathbb{N}$ e $(m, n) = 1$, allora*

$$\varphi(mn) = \varphi(n)\varphi(m).$$

Dimostrazione. Basta considerare l'applicazione

$$\begin{gathered} U(\mathbb{Z}/mn\mathbb{Z}) \to U(\mathbb{Z}/m\mathbb{Z}) \times U(\mathbb{Z}/n\mathbb{Z}) \\ x \mapsto (x \bmod m, x \bmod n). \end{gathered}$$

È immediato verificare che è iniettiva. Inoltre se $(\alpha, \beta) \in U(\mathbb{Z}/m\mathbb{Z}) \times U(\mathbb{Z}/n\mathbb{Z})$, per il teorema cinese dei resti esiste un unico $x \in U(\mathbb{Z}/nm\mathbb{Z})$ tale che $x \equiv \alpha \bmod m$ e $x \equiv \beta \bmod n$. Infine, contando gli elementi, si ottiene il risultato. □

Per concludere il capitolo consideriamo la seguente generalizzazione del piccolo teorema di Fermat dovuta da Eulero.

Proposizione 11 *Sia $m \in \mathbb{N}$ e sia $a \in U(\mathbb{Z}/m\mathbb{Z})$. Allora*

$$a^{\varphi(m)} \equiv 1 \bmod m.$$

Inoltre se $n_1, n_2 \in Z$ con $n_1 \equiv n_2 \bmod \varphi(m)$, allora

$$a^{n_1} \equiv a^{n_2} \bmod m.$$

Dimostrazione. Se $m = p^\alpha$ allora per il piccolo teorema di Fermat sappiamo che $a^{p-1} = 1 + kp$. Elevando questa identità alla potenza $p^{\alpha-1}$ otteniamo

$$\begin{aligned} a^{p^{\alpha-1}(p-1)} &= (1+kp)^{p^{\alpha-1}} = \textstyle\sum_{j=0}^{p^{\alpha-1}} \binom{p^{\alpha-1}}{j} p^j k^j \\ &= 1 + \textstyle\sum_{j=1}^{p^{\alpha-1}-1} \binom{p^{\alpha-1}}{j} p^j k^j + (pk)^{p^{\alpha-1}} \\ &\equiv 1 \bmod p^\alpha \end{aligned}$$

in quanto $p^{\alpha-1} | \binom{p^{\alpha-1}}{j}$ se $j \neq 1, p^{\alpha-1}$. Quindi, essendo $\varphi(p^\alpha) = p^{\alpha-1}(p-1)$, otteniamo l'enunciato per potenze di numeri primi. Inoltre $p^\alpha \| m$ allora è chiaro che per la proposizione precedente

$$a^{\varphi(m)} = a^{\varphi(m/p^\alpha)\varphi(p^\alpha)} \equiv (a^{\varphi(p^\alpha)})^{\varphi(m/p^\alpha)} \equiv 1^{\varphi(m/p^\alpha)} \bmod p^\alpha.$$

Infine $n = p_1^{\alpha_1} \cdots p_s^{\alpha_s}$ è la fattorizzazione di n come prodotto di primi distinti, dal fatto che $a^{\varphi(m)} \equiv 1 \bmod p_i^{\alpha_i}$ deduciamo che $a^{\varphi(m)} \equiv 1 \bmod n$.

La seconda affermazione nell'enunciato si dimostra in modo analogo a quella nella seconda parte dell'enunciato del piccolo Teorema di Fermat □

Osservazione. Come nel caso del piccolo Teorema di Fermat, per calcolare $a^n \bmod m$ nel caso in cui n è molto grande rispetto a m si potrebbe dividere n per $\varphi(m)$ e

poi calcolare $a^{n \bmod \varphi(m} \bmod m$. Nelle circostanze pratiche però calcolare $\varphi(m)$ è ben più difficile che $a^n \bmod m$.

Esempio. Supponiamo di voler calcolare $5^{12345} \bmod 143$. Essendo $143 = 13*11$, $\varphi(143) = 120$, e $12345 \bmod 120 = 105$, basta calcolare $5^{105} \bmod 143$. Adesso $5^{105} \equiv 5^5 \equiv 5(25)^2 \equiv 1 \bmod 11$, mentre $5^{105} \equiv 5^9 \equiv 25^4 5 \equiv 5 \bmod 13$. Quindi se $x = 5^{12345} \bmod 143$, risulta

$$\begin{cases} x \equiv 1 \bmod 11 \\ x \equiv 5 \bmod 13 \end{cases}$$

Per il Teorema cinese dei resti, $x = 122$.

Esercizi.

1. Si determini un numero intero positivo x nell' intervallo $[60, 120]$ tale che

$$\begin{cases} x \equiv 1 \pmod 3 \\ x \equiv 4 \pmod 5 \\ x \equiv 2 \pmod 4. \end{cases}$$

2. Si determini un numero intero y nell' intervallo $[-80, 0]$ tale che

$$\begin{cases} y \equiv 2 \pmod 3 \\ y \equiv 2 \pmod 7 \\ y \equiv 4 \pmod 1 1. \end{cases}$$

3. Si calcolino tutte le soluzioni del seguente sistema di congruenze nell'intervallo $[1000, 6000]$.

$$\begin{cases} X \equiv 1 (\mathrm{mod} 6) \\ X \equiv 3 (\mathrm{mod} 7) \\ X \equiv 3 (\mathrm{mod} 11). \end{cases}$$

Risposta: $1081, 1543, 2005, 2467, 2929, 3391, 3853, 4315, 4777, 5239, 5701$.

4. Dimostrare che se $n = p_1 \cdots p_{20}$ è un intero privo di fattori quadratici, e $f(x) \in \mathbf{Z}/n\mathbf{Z}[x]$ ha grado 10, allora la congruenza $f(x) \equiv 0 \bmod n$ è risolvibile se e solo se lo sono le 20 congruenze

$$\begin{cases} f(x) \equiv 0 \bmod p_1 \\ \vdots \\ f(x) \equiv 0 \bmod p_{20}. \end{cases}$$

Dedurre che la prima congruenza $f(x) \equiv 0 \bmod n$ ha al più 10^{20} soluzioni. Sapreste dare un esempio in cui le soluzioni sono esattamente 10^{20}?

Soluzione. (Se) Per ogni $i = 1, \ldots, 20$, sia α_i una soluzione di $f(x) \equiv 0 \bmod p_i$ e sia $\alpha \in \mathbf{Z}/n\mathbf{Z}$ una soluzione del sistema di congruenze

$$\begin{cases} X \equiv \alpha_1 \bmod p_1 \\ \vdots \\ X \equiv \alpha_{20} \bmod p_{20}. \end{cases}$$

Si ha che $f(\alpha) \equiv f(\alpha_i) \equiv 0 \bmod p_i$. Quindi $f(\alpha)$ è divisibile per $p_1, \ldots, p_{20}$ e quindi per n.
(Solo se) Se $\alpha \in \mathbf{Z}/n\mathbf{Z}$ è una soluzione di $f(x) \equiv 0 \bmod n$, allora $\alpha \bmod p_i$ è una soluzione di $f(x) \equiv 0 \bmod p_i$. per ogni $i = 1, \ldots, 20$.
Se $f(x) = x(x-1)(x-2)\ldots(x-9)$ e $n = 11 \cdot 13 \cdot \cdots 89$ (il prodotto dei 20 primi tra 11 e 89). Allora f ha esattamente 10^{20} soluzioni modulo n. Infatti per ciascuna delle 10^{20} scelte $i_1, \ldots, i_{20} \in \{0, \ldots, 9\}$, il sistema di congruenza

$$\begin{cases} X \equiv i_1 \bmod p_1 \\ \vdots \\ X \equiv i_{20} \bmod p_{20} \end{cases}$$

da luogo (per il Teorema cinese dei resti) a esattamente una soluzione modulo n.

5. Si risolva il seguente sistema di equazioni di congruenze

$$\begin{cases} x^3 \equiv 1 \bmod 7 \\ x^2 \equiv 1 \bmod 5. \end{cases}$$

6. Si risolva il seguente sistema di equazioni di congruenze

$$\begin{cases} x^2 \equiv 4 \bmod 11 \\ x^3 \equiv 2 \bmod 5. \end{cases}$$

7. Sia m un intero dispari. Dopo aver dimostrato che ammette soluzione, si stimi il numero di operazioni bit necessarie a risolvere il seguente sistema

$$\begin{cases} X \equiv 1 \bmod m \\ X \equiv 2 \bmod m+1 \\ X \equiv 3 \bmod m+2. \end{cases}$$

Bibliografia

[1] Harold Davenport. *Multiplicative number theory*. Springer-Verlag, New York, second edition, 1980. Revised by Hugh L. Montgomery.

[2] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers*. The Clarendon Press Oxford University Press, New York, fifth edition, 1979.

[3] Neal Koblitz. *A course in number theory and cryptography*. Springer-Verlag, New York, second edition, 1994.

[4] Neal Koblitz. *Algebraic aspects of cryptography*. Springer-Verlag, Berlin, 1998. With an appendix by Alfred J. Menezes, Yi-Hong Wu and Robert J. Zuccherato.

[5] Karl Prachar. *Primzahlverteilung*. Springer-Verlag, Berlin, 1978. Reprint of the 1957 original.

[6] J. Barkley Rosser and Lowell Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois J. Math.*, 6:64–94, 1962.

[7] Bruce Schneier. *Applied Criptography*. John Wiley & Sons, Inc., seconda edizione edition, 1996.

[8] Douglas R. Stinson. *Cryptography*. CRC Press, Boca Raton, FL, 1995. Theory and practice.

[9] Gérald Tenenbaum. *Introduction to analytic and probabilistic number theory*. Cambridge University Press, Cambridge, 1995.

[10] Jan C. A. van der Lubbe. *Basic methods of cryptography*. Cambridge University Press, Cambridge, 1998. Translated from the 1994 Dutch original by Steve Gee.

Indice analitico