ALESSANDRO LANGUASCO
ALESSANDRO ZACCAGNINI

MANUALE DI CRITTOGRAFIA

Complementi

EDITORE ULRICO HOEPLI MILANO

Indice

Prefazione ai complementi

In questi complementi raccogliamo una certa quantità di informazioni che sono di ausilio a quanto presentato nel testo cartaceo. Alcune sono delle integrazioni, altre sono delle parti descrittive di alcuni aspetti storici e di sviluppo di algoritmi o crittosistemi. Altre parti riguardano aspetti non direttamente pertinenti alla crittografia in quanto tale, ma che riportiamo, per la comodità dei lettori, per completezza. Riguardano le classi di complessità degli algoritmi di decisione, alcuni elementi di algebra lineare, elementi di teoria analitica dei numeri ed un capitolo di letture ulteriori che riteniamo possano risultare utili a chi voglia saperne di più. Infine inseriamo anche un capitolo descrittivo del software di calcolo PARI/GP; software che permette di utilizzare facilmente un'aritmetica intera a lunghezza arbitraria consentendo così di realizzare brevi script funzionanti che esemplificano gli algoritmi descritti nel volume.

Commenti, critiche, passaggi oscuri, eventuali errori di stampa possono essere segnalati agli indirizzi qui sotto:

Prof. Alessandro Languasco, email: languasco@math.unipd.it; webpage: www.math.unipd.it/~languasc.

Prof. Alessandro Zaccagnini, e-mail: alessandro.zaccagnini@unipr.it; webpage: people.math.unipr.it/alessandro.zaccagnini.

Legenda dei complementi

$C(x)$	funzione che conta i numeri di Carmichael $n \leq x$, p. 17
$P(z)$	$\prod_{p \leq z} p$, p. 11
$[a]_{\mathcal{R}}$	classe di equivalenza di a secondo $\mathcal{R}$, p. 26
$\mathcal{A}/\mathcal{R}$	insieme quoziente secondo $\mathcal{R}$, p. 26
$O(\cdot)$	notazione di Bachmann-Landau, p. 27
$\Psi(x,y)$	funzione che conta gli interi $n \leq x$ che non hanno fattori primi $> y$, p. 15
$\mathrm{li}(x)$	integrale logaritmico, p. 12
$\mu(n)$	funzione μ di Möbius, p. 11
$\omega(n)$	funzione che conta il numero di fattori primi distinti dell'intero n, p. 14
$\pi(N)$	funzione che conta i numeri primi $\leq N$, p. 8
$\pi(x;q,a)$	funzione che conta i numeri primi $\leq x$ congrui ad a modulo q, p. 12
$\psi(N)$	funzione ψ di Chebyshev, p. 8
$\theta(N)$	funzione θ di Chebyshev, p. 8
$\zeta(s)$	funzione ζ di Riemann, p. 13
$a \mathcal{R} b$	equivalenza di a e b secondo $\mathcal{R}$, p. 26
$f(x) \sim g(x)$	f asintotica a g, p. 11

Complementi A
Alcune aggiunte

Includiamo qui alcuni complementi ai vari capitoli pubblicati.

A.1 Complementi al DES

Per completezza riportiamo qui tutte le funzioni S_j.

$$S_1(b_1,b_2,b_3,b_4,b_5,b_6) = \begin{pmatrix} 14 & 4 & 13 & 1 & 2 & 15 & 11 & 8 & 3 & 10 & 6 & 12 & 5 & 9 & 0 & 7 \\ 0 & 15 & 7 & 4 & 14 & 2 & 13 & 1 & 10 & 6 & 12 & 11 & 9 & 5 & 3 & 8 \\ 4 & 1 & 14 & 8 & 13 & 6 & 2 & 11 & 15 & 12 & 9 & 7 & 3 & 10 & 5 & 0 \\ 15 & 12 & 8 & 2 & 4 & 9 & 1 & 7 & 5 & 11 & 3 & 14 & 10 & 0 & 6 & 13 \end{pmatrix}$$

$$S_2(b_1,b_2,b_3,b_4,b_5,b_6) = \begin{pmatrix} 15 & 1 & 8 & 14 & 6 & 11 & 3 & 4 & 9 & 7 & 2 & 13 & 12 & 0 & 5 & 10 \\ 3 & 13 & 4 & 7 & 15 & 2 & 8 & 14 & 12 & 0 & 1 & 10 & 6 & 9 & 11 & 5 \\ 0 & 14 & 7 & 11 & 10 & 4 & 13 & 1 & 5 & 8 & 12 & 6 & 9 & 3 & 2 & 15 \\ 13 & 8 & 10 & 1 & 3 & 15 & 4 & 2 & 11 & 6 & 7 & 12 & 0 & 5 & 14 & 9 \end{pmatrix}$$

$$S_3(b_1,b_2,b_3,b_4,b_5,b_6) = \begin{pmatrix} 10 & 0 & 9 & 14 & 6 & 3 & 15 & 5 & 1 & 13 & 12 & 7 & 11 & 4 & 2 & 8 \\ 13 & 7 & 0 & 9 & 3 & 4 & 6 & 10 & 2 & 8 & 5 & 14 & 12 & 11 & 15 & 1 \\ 13 & 6 & 4 & 9 & 8 & 15 & 3 & 0 & 11 & 1 & 2 & 12 & 5 & 10 & 14 & 7 \\ 1 & 10 & 13 & 0 & 6 & 9 & 8 & 7 & 4 & 15 & 14 & 3 & 11 & 5 & 2 & 12 \end{pmatrix}$$

$$S_4(b_1,b_2,b_3,b_4,b_5,b_6) = \begin{pmatrix} 7 & 13 & 14 & 3 & 0 & 6 & 9 & 10 & 1 & 2 & 8 & 5 & 11 & 12 & 4 & 15 \\ 13 & 8 & 11 & 5 & 6 & 15 & 0 & 3 & 4 & 7 & 2 & 12 & 1 & 10 & 14 & 9 \\ 10 & 6 & 9 & 0 & 12 & 11 & 7 & 13 & 15 & 1 & 3 & 14 & 5 & 2 & 8 & 4 \\ 3 & 15 & 0 & 6 & 10 & 1 & 13 & 8 & 9 & 4 & 5 & 11 & 12 & 7 & 2 & 14 \end{pmatrix}$$

$$S_5(b_1,b_2,b_3,b_4,b_5,b_6) = \begin{pmatrix} 2 & 12 & 4 & 1 & 7 & 10 & 11 & 6 & 8 & 5 & 3 & 15 & 13 & 0 & 14 & 9 \\ 14 & 11 & 2 & 12 & 4 & 7 & 13 & 1 & 5 & 0 & 15 & 10 & 3 & 9 & 8 & 6 \\ 4 & 2 & 1 & 11 & 10 & 13 & 7 & 8 & 15 & 9 & 12 & 5 & 6 & 3 & 0 & 14 \\ 11 & 8 & 12 & 7 & 1 & 14 & 2 & 13 & 6 & 15 & 0 & 9 & 10 & 4 & 5 & 3 \end{pmatrix}$$

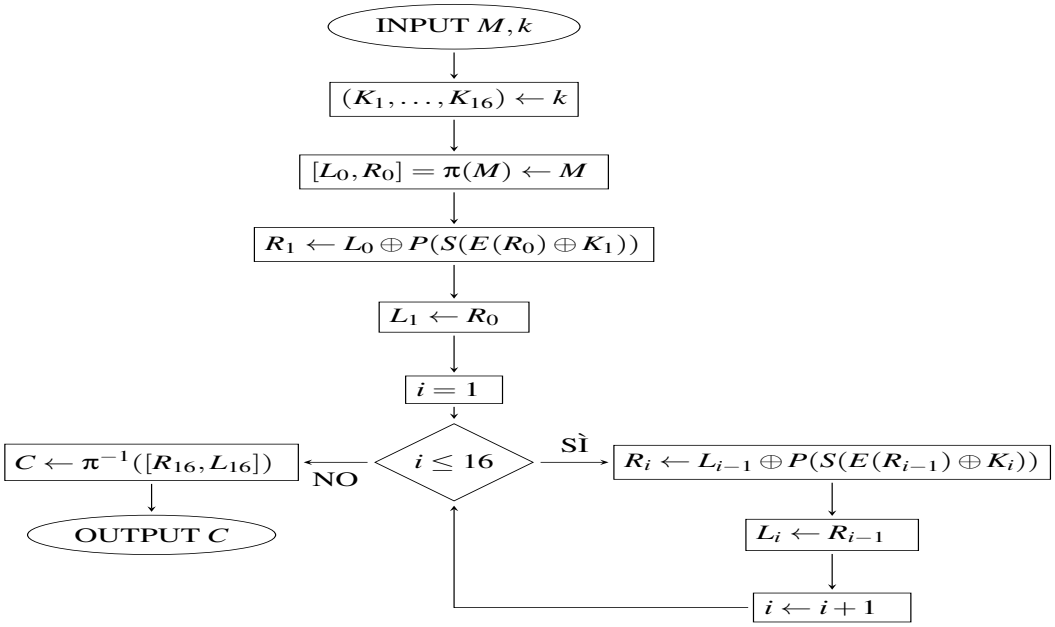

Figura A.1 - Schema di funzionamento della codifica del DES.

$$S_6(b_1,b_2,b_3,b_4,b_5,b_6) =$$
$$\begin{pmatrix} 12 & 1 & 10 & 15 & 9 & 2 & 6 & 8 & 0 & 13 & 3 & 4 & 14 & 7 & 5 & 11 \\ 10 & 15 & 4 & 2 & 7 & 12 & 9 & 5 & 6 & 1 & 13 & 14 & 0 & 11 & 3 & 8 \\ 9 & 14 & 15 & 5 & 2 & 8 & 12 & 3 & 7 & 0 & 4 & 10 & 1 & 13 & 11 & 6 \\ 4 & 3 & 2 & 12 & 9 & 5 & 15 & 10 & 11 & 14 & 1 & 7 & 6 & 0 & 8 & 13 \end{pmatrix}$$

$$S_7(b_1,b_2,b_3,b_4,b_5,b_6) =$$
$$\begin{pmatrix} 4 & 11 & 2 & 14 & 15 & 0 & 8 & 13 & 3 & 12 & 9 & 7 & 5 & 10 & 6 & 1 \\ 13 & 0 & 11 & 7 & 4 & 9 & 1 & 10 & 14 & 3 & 5 & 12 & 2 & 15 & 8 & 6 \\ 1 & 4 & 11 & 13 & 12 & 3 & 7 & 14 & 10 & 15 & 6 & 8 & 0 & 5 & 9 & 2 \\ 6 & 11 & 13 & 8 & 1 & 4 & 10 & 7 & 9 & 5 & 0 & 15 & 14 & 2 & 3 & 12 \end{pmatrix}$$

$$S_8(b_1,b_2,b_3,b_4,b_5,b_6) =$$
$$\begin{pmatrix} 13 & 2 & 8 & 4 & 6 & 15 & 11 & 1 & 10 & 9 & 3 & 14 & 5 & 0 & 12 & 7 \\ 1 & 15 & 13 & 8 & 10 & 3 & 7 & 4 & 12 & 5 & 6 & 11 & 0 & 14 & 9 & 2 \\ 7 & 11 & 4 & 1 & 9 & 12 & 14 & 2 & 0 & 6 & 10 & 13 & 15 & 3 & 5 & 8 \\ 2 & 1 & 14 & 7 & 4 & 10 & 8 & 13 & 15 & 12 & 9 & 0 & 3 & 5 & 6 & 11 \end{pmatrix}$$

Uno schema di funzionamento alternativo a quanto già visto nel testo cartaceo è quello presentato in figura A.1.

A.2 Complementi a AKS

Negli ultimi anni sono state sviluppate delle varianti di AKS che ottimizzano la trattazione in vari punti; tra queste, oltre al già citato lavoro di Granville [35],

ricordiamo anche quello di Bernstein [10].

Il fermento suscitato da questa scoperta ha portato molti esperti a lavorare sul tipo di approccio di Agrawal, Kayal e Saxena. Il risultato più importante è stato ottenuto da Lenstra e Pomerance (si vedano [24, §4.5.3] e [59]). Viene dimostrata la validità di una variante dell'algoritmo AKS avente complessità computazionale $O\big((\log n)^{6+\varepsilon}\big)$ (usando gli algoritmi FFT, si veda l'osservazione 4.1.12) che si ritiene essere la migliore stima ottenibile per la complessità computazionale degli algoritmi deterministici di primalità. Tale algoritmo però coinvolge aspetti matematici più avanzati di quelli utilizzati qui, mentre la versione che abbiamo presentato, sebbene abbia una complessità computazionale maggiore (ma pur sempre polinomiale), utilizza solamente concetti algebrici che sono patrimonio del curriculum di una laurea di primo livello in matematica e risultati elementari di teoria dei numeri per la valutazione della complessità computazionale.

Un'altra linea di ricerca si è concentrata sulle varianti probabilistiche del metodo AKS. I risultati più importanti sono dovuti indipendentemente a Berrizbeitia [13], Bernstein [12], Cheng [19] e a Mihăilescu e Avanzi [9]. Essi hanno trovato una variante dell'algoritmo AKS che in ogni iterazione ha probabilità $> 1/2$ di distinguere un intero primo da uno composto con complessità $O\big((\log n)^{4+\varepsilon}\big)$ usando gli algoritmi FFT. Ripetendone l'applicazione per k volte si ottiene quindi un algoritmo che ha probabilità di fallire $< 2^{-k}$ e che ha complessità $O\big(k(\log n)^{4+\varepsilon}\big)$. Il problema è chiaramente dato dal fatto che un tale algoritmo può non terminare e ciò dipende dal dover eseguire, in un passo dell'algoritmo stesso, una ricerca di radici dell'unità in una data estensione finita di $\mathbb{Z}_n$. Per maggiori dettagli su AKS e altri algoritmi di primalità si vedano anche Granville [35], Dietzfelbinger [29] e Schoof [85].

Tali ricerche sono anche dovute al fatto che al momento non esistono implementazioni "praticamente" efficienti di AKS e quindi si è cercato, e si cerca tuttora, di migliorare il più possibile l'applicazione "pratica" di queste idee al fine di poter dimostrare la primalità di interi di "forma generale" sempre più grandi. In questo ambito, recentemente, Cao e Liu [18] hanno analizzato le richieste di memoria necessarie ad AKS per lavorare con numeri di 1024 bit ottenendo che sarebbero necessari un miliardo di gigabyte; una quantità assolutamente eccessiva per le attuali memorie di computer.

Infine, ricordiamo che Bernstein [11] ha recentemente scritto un esaustivo, sebbene schematico, resoconto dello sviluppo cronologico degli algoritmi di primalità.

A.3 Dimostrazione della formula di Binet (4.2.2)

La (4.2.2) può essere facilmente dimostrata mediante il principio di induzione (invitiamo il lettore a svolgere autonomamente tale ragionamento), ma esistono

anche altre dimostrazioni. Una di tipo immediato è la seguente. Definiti $\phi_1 = (1+\sqrt{5})/2$ e $\phi_2 = (1-\sqrt{5})/2$, osserviamo che $\phi_1+\phi_2 = 1$, $\phi_1-\phi_2 = \sqrt{5}$ e $\phi_1\phi_2 = -1$. Sfruttando tali relazioni, per ogni $k \in \mathbb{Z}$, $k \geq 1$, si ha che

$$f_{k+1} - \phi_2 f_k = f_k + f_{k-1} - \phi_2 f_k = (1-\phi_2)f_k + f_{k-1} = \phi_1(f_k - \phi_2 f_{k-1}),$$

dove f_k è il k-esimo numero di Fibonacci. Riapplicando la relazione precedente $k-1$ volte e ricordando che $f_1 = f_2 = 1$, abbiamo che

$$f_{k+1} - \phi_2 f_k = \phi_1^2(f_{k-1} - \phi_2 f_{k-2}) = \ldots = \phi_1^{k-1}(f_2 - \phi_2 f_1) = \phi_1^{k-1}(1-\phi_2) = \phi_1^k \tag{A.3.1}$$

per ogni $k \in \mathbb{Z}$ con $k \geq 1$. In modo del tutto analogo si dimostra che

$$f_{k+1} - \phi_1 f_k = \phi_2^2(f_{k-1} - \phi_1 f_{k-2}) = \ldots = \phi_2^{k-1}(f_2 - \phi_1 f_1) = \phi_2^{k-1}(1-\phi_1) = \phi_2^k \tag{A.3.2}$$

per ogni $k \in \mathbb{Z}$, $k \geq 1$. Sottraendo (A.3.2) da (A.3.1) otteniamo $f_k(\phi_1 - \phi_2) = \phi_1^k - \phi_2^k$ da cui, dividendo per $\phi_1 - \phi_2 = \sqrt{5}$, si ottiene immediatamente la (4.2.2).

A.4 L'algoritmo della divisione con resto

Uno degli algoritmi più utili è quello della divisione con resto. Indichiamo qui un algoritmo per la divisione con resto che sfrutta le potenzialità offerte dall'architettura delle macchine. Per semplicità, diamo un esempio in cui si sfrutta la base 16, mentre nelle applicazioni pratiche è più frequente il caso di numeri scritti con 16 o 32 bit. L'idea è quella di sfruttare il fatto che è molto facile determinare quoziente e resto della divisione di un intero N per 2^{16}, diciamo, se questo è memorizzato in byte contigui: in questa situazione il resto si trova nei due byte meno significativi di N, ed il quoziente negli altri.

Se volessimo invece dividere per $2^{16}-3$, saremmo costretti apparentemente a scrivere una routine molto meno efficiente. Ma è chiaro che i risultati delle divisioni per $M = 2^{16}$ e per $m = 2^{16}-3$ non saranno molto diversi; poniamo in generale $d = M - m$, con $|d| < \frac{1}{2}M$, e definiamo

$$\begin{cases} q_0 = \left\lfloor \dfrac{N}{M} \right\rfloor \\ r_0 = d \cdot \left\lfloor \dfrac{N}{M} \right\rfloor + (N \bmod M) \end{cases} \qquad \begin{cases} q_{k+1} = q_k + \left\lfloor \dfrac{r_k}{M} \right\rfloor \\ r_{k+1} = d \cdot \left\lfloor \dfrac{r_k}{M} \right\rfloor + (r_k \bmod M). \end{cases}$$

Quando $\lfloor r_k/M \rfloor = 0$ l'algoritmo termina: c'è ancora da osservare che $r_k < M$, e potrebbe accadere che $r_k \geq m$, che è contrario alla definizione di resto. In questo caso si aumenta di 1 il valore di q_k e si sottrae m da r_k, ripetendo se necessario fino a che $r_k < m$.

In un certo senso, se $d > 0$ questo algoritmo calcola un valore del quoziente approssimato per difetto, ed un valore del resto approssimato per eccesso, e ad ogni iterazione si correggono queste quantità fino ad ottenere i valori esatti. La figura A.2 illustra un esempio pratico di applicazione di questo algoritmo.

Il funzionamento del metodo dipende dal fatto che dopo ciascuna iterazione si ha $q_k \cdot m + r_k = N$: infatti, per $k = 0$ si ha

$$q_0 \cdot m + r_0 = \left\lfloor \frac{N}{M} \right\rfloor \cdot m + d \cdot \left\lfloor \frac{N}{M} \right\rfloor + \left(N - M \left\lfloor \frac{N}{M} \right\rfloor \right) = (m + d - M) \cdot \left\lfloor \frac{N}{M} \right\rfloor + N = N.$$

Un calcolo molto simile mostra che la stessa cosa vale per $k \geq 1$. Inoltre, se $d > 0$ si vede subito che

$$0 \leq r_{k+1} = d \cdot \left\lfloor \frac{r_k}{M} \right\rfloor + \left(r_k - M \left\lfloor \frac{r_k}{M} \right\rfloor \right) = r_k - m \cdot \left\lfloor \frac{r_k}{M} \right\rfloor < r_k,$$

e quindi l'algoritmo converge. Se $d < 0$, invece, è possibile che qualche r_k sia negativo, ed è necessario cambiare leggermente la dimostrazione e la condizione di arresto delle iterazioni: si veda lo pseudocodice della figura A.2. È opportuno notare che questo algoritmo è efficiente se $|d|$ è piccolo, perché altrimenti i valori di r_k con k piccolo possono essere grandi in valore assoluto.

A.5 Dimostrazione alternativa del teorema di Wilson 11.1.4

Il teorema di Gauss 11.3.1 ci dà la possibilità di dimostrare un'altra volta il teorema di Wilson, mostrando che un enunciato analogo vale in ogni gruppo ciclico finito.

Dimostrazione alternativa del teorema di Wilson. Dato che ogni elemento $m \in \mathbb{Z}_p^*$ può essere scritto come g^n per un opportuno valore di $n \in \{1, \ldots, p-1\}$, dove g è un generatore, se $p \geq 3$ abbiamo

$$\prod_{m=1}^{p-1} m = \prod_{n=1}^{p-1} g^n = g^{p(p-1)/2} = g^{(p-1)^2/2} \cdot g^{(p-1)/2} = 1 \cdot (-1).$$

Infatti, se p è dispari allora $(p-1)^2/2$ è un multiplo di $p-1$; inoltre $g^{(p-1)/2}$ è una delle due soluzioni dell'equazione $x^2 \equiv 1 \bmod p$, e quindi vale ± 1 per il corollario 10.2.16, ma se valesse 1 allora g avrebbe ordine $\leq (p-1)/2$. □

q	r	x	r'
0	6561	410	$410 \cdot 2 + 1 = 821$
410	821	51	$51 \cdot 2 + 5 = 107$
461	107	6	$6 \cdot 2 + 11 = 23$
467	23	1	$1 \cdot 2 + 7 = 9$
468	9	0	

q	r	x	r'
0	19A1	19A	$19A \cdot 2 + 1 = 335$
19A	335	33	$33 \cdot 2 + 5 = 6B$
1CD	6B	6	$6 \cdot 2 + B = 17$
1D3	17	1	$1 \cdot 2 + 7 = 9$
1D4	9	0	

DIVISIONE CON RESTO

```
1  d ← M − m
2  q ← 0
3  r ← N
4  x ← ⌊r/M⌋
5  repeat
6      q += x
7      r ← x·d + (r % M)
8      x ← ⌊r/M⌋
9  until (r < 0)||(r ≥ max(M,m))
10 while r ≥ m
11     q += 1
12     r −= m
13 endwhile
```

Figura A.2 - Calcolo del quoziente con resto di $N = 6561$ ed $m = 14$. Scegliamo $M = 16$ e quindi $d = 2$. Si osservi che alla fine di ogni ciclo si ha sempre $N = m \cdot q + r$, e che r è uguale al resto della divisione solo dopo l'ultimo ciclo. A sinistra il calcolo è eseguito una volta in base 10 ed una seconda volta in base 16. In questo secondo caso è evidente il modo in cui si sta sfruttando l'architettura delle macchine.
Nello pseudocodice a fianco, si tenga presente che se $d < 0$ allora i valori di r possono essere negativi, ma per noi la funzione $r \,\%\, M$ restituisce sempre un valore nell'insieme $\{0, \ldots, M-1\}$. Si osservi che questa convenzione *non* è necessariamente quella usata da tutti i compilatori.
Le ultime linee sono necessarie per assicurare che il resto calcolato sia effettivamente minore di m, dato che il ciclo nelle linee 5–9 garantisce solo che questo sia minore di M. Se $\frac{1}{2}M < m < M$, allora questo ciclo può essere eseguito al più una volta.

Complementi B
Distribuzione dei numeri primi

I risultati di questo capitolo, a stretto rigore, non sono necessari per la comprensione degli argomenti presentati nel testo: il motivo per cui sono stati qui inclusi è che quasi tutte le applicazioni crittografiche moderne si basano sulla scelta di uno o più numeri primi "grandi," ed è ragionevole mostrare che i numeri primi sono piuttosto numerosi, e quindi che le applicazioni di cui sopra sono davvero realizzabili nella pratica. Indicheremo con log il *logaritmo naturale*, e con p un numero primo.

B.1 Euristica

Ricordiamo la formula di Stirling 6.6.1 nella forma semplice $\log(N!) = N\log N + O(N)$. Se si calcola la massima potenza di p che divide $N!$ per ogni $p \leq N$, il primo membro può essere riscritto come

$$\log(N!) = \sum_{p \leq N} \log p \sum_{m \geq 1} \left\lfloor \frac{N}{p^m} \right\rfloor . \tag{B.1.1}$$

Infatti, scelto $p \leq N$, l'esponente α_p di p nella fattorizzazione di $N!$ è dato dalla somma

$$\alpha_p = \left\lfloor \frac{N}{p} \right\rfloor + \left\lfloor \frac{N}{p^2} \right\rfloor + \left\lfloor \frac{N}{p^3} \right\rfloor + \dots$$

Il primo addendo proviene dagli interi $\leq N$ che sono divisibili per p e quindi contribuiscono 1 ad α_p, il secondo proviene dagli interi $\leq N$ che sono divisibili per p^2 e quindi contribuiscono ancora un'unità ad α_p, e così via. Si noti che la somma interna nella (B.1.1) è finita poiché l'addendo vale zero non appena $p^m > N$. Se trascuriamo le parti frazionarie ed i termini con $m > 1$, e facciamo le opportune semplificazioni, troviamo la *formula di Mertens*

$$\sum_{p \leq N} \frac{\log p}{p} = \log N + O(1). \tag{B.1.2}$$

Per la precisione, per la dimostrazione completa della formula di Mertens è necessaria un'informazione sulla funzione più importante in questo campo, e cioè

quella che conta i numeri primi $\leq N$, che indicheremo con $\pi(N)$:

$$\pi(N) \stackrel{\text{def}}{=} \operatorname{card}(\{p \leq N\colon\ p \text{ è primo}\}) = \sum_{p \leq N} 1.$$

L'informazione necessaria, che otterremo fra breve, è una maggiorazione per $\pi(N)$ dell'ordine di grandezza corretto: più precisamente, dimostreremo che esiste una costante positiva C tale che per ogni $N \geq 3$ si ha

$$\pi(N) \leq \frac{CN}{\log N}. \tag{B.1.3}$$

Introduciamo le funzioni di Chebyshev θ e ψ definite rispettivamente da

$$\theta(N) \stackrel{\text{def}}{=} \sum_{p \leq N} \log p = \log \prod_{p \leq N} p,$$
$$\psi(N) \stackrel{\text{def}}{=} \log \operatorname{lcm}\{1, 2, 3, \ldots, N\} = \sum_{p \leq N} \left\lfloor \frac{\log N}{\log p} \right\rfloor \log p.$$

Per dimostrare l'ultima uguaglianza chiamiamo p^m la più alta potenza di p che divide $\operatorname{lcm}\{1, 2, 3, \ldots, N\}$, e poniamo $m_0 = \lfloor (\log N)(\log p)^{-1} \rfloor$. Dato che $p^{m_0} \leq N$ si ha $m \geq m_0$. Inoltre, poiché $p^{m_0+1} > N$, nessun intero $\leq N$ è divisibile per p^{m_0+1}, e quindi $m \leq m_0$. Questo ci dà anche la maggiorazione

$$\psi(N) = \sum_{p \leq N} \left\lfloor \frac{\log N}{\log p} \right\rfloor \log p \leq \log N \sum_{p \leq N} 1 = \pi(N) \log N. \tag{B.1.4}$$

Per quello che riguarda θ, per prima cosa osserviamo che per ogni $y \in (1, N]$ si ha

$$\theta(N) \geq \sum_{y < p \leq N} \log p \geq \log y \big(\pi(N) - \pi(y)\big) \qquad \text{da cui} \qquad \pi(N) \leq \frac{\theta(N)}{\log y} + \pi(y).$$

Dato che $\pi(y) \leq y$, possiamo scegliere $y = \sqrt{N}$ ottenendo $\pi(N) \leq 3\theta(N)/\log N$. Consideriamo ora il coefficiente binomiale $M = \binom{2N+1}{N}$. Poiché M compare due volte nello sviluppo di $(1+1)^{2N+1}$, si ha $2M < 2^{2N+1}$ da cui $M < 2^{2N}$. Osserviamo che se $p \in (N+1, 2N+1]$ allora $p \mid M$, poiché divide il numeratore del coefficiente binomiale, ma non il denominatore. Questo ci permette di concludere che

$$\theta(2N+1) - \theta(N+1) \leq \log M < 2N \log 2. \tag{B.1.5}$$

Supponiamo di aver dimostrato che $\theta(n) < 2n \log 2$ per $1 \leq n \leq n_0 - 1$ con $n_0 \geq 4$, osservando che questa relazione è banale per $n = 1,\ 2,\ 3$. Se n_0 è pari allora

$\theta(n_0) = \theta(n_0-1) < 2(n_0-1)\log 2 < 2n_0 \log 2$. Se n_0 è dispari, $n_0 = 2N+1$ e quindi

$$\begin{aligned}\theta(n_0) &= \theta(2N+1) = \theta(2N+1) - \theta(N+1) + \theta(N+1)\\ &< 2N\log 2 + 2(N+1)\log 2 = 2n_0 \log 2,\end{aligned}$$

per la (B.1.5) e per l'ipotesi induttiva, e la disuguaglianza (B.1.3) segue con $C = 6\log 2$. Per dare una *minorazione* per $\pi(N)$ dello stesso ordine di grandezza, abbiamo bisogno di un risultato di Nair [63] che si potrà anche esprimere come

$$\psi(m) = \log \operatorname{lcm}\{1, 2, 3, \ldots, m\} \geq m \log 2. \tag{B.1.6}$$

Lemma B.1.1 *Sia $m \in \mathbb{N}$ e sia $m \geq 7$. Allora* $\operatorname{lcm}\{1,\ldots,m\} \geq 2^m$.

Dim. Indichiamo con d_m la quantità $\operatorname{lcm}\{1,\ldots,m\}$. Sia inoltre n un intero, $1 \leq n \leq m$, e

$$I(n,m) = \int_0^1 t^{n-1}(1-t)^{m-n}\mathrm{d}t.$$

Calcoliamo $I(n,m)$ in due modi diversi. Il primo è usando l'espansione binomiale di $(1-t)^{m-n}$ data da $(1-t)^{m-n} = \sum_{j=0}^{m-n}\binom{m-n}{j}(-t)^j$. In tal modo abbiamo che

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

che è chiaramente un numero razionale. È anche immediato verificare che il suo denominatore divide d_m. Pertanto

$$d_m I(n,m) \in \mathbb{N}. \tag{B.1.7}$$

Usiamo adesso l'integrazione per parti su $I(n,m)$ usando $(1-t)^{m-n}$ come fattore differenziale. Otteniamo che

$$\begin{aligned}I(n,m) &= -\frac{t^{n-1}(1-t)^{m-n+1}}{m-n+1}\bigg|_0^1 + \frac{n-1}{m-n+1}\int_0^1 t^{n-2}(1-t)^{m-n+1}\mathrm{d}t\\ &= \frac{n-1}{m-n+1}\int_0^1 t^{n-2}(1-t)^{m-n+1}\mathrm{d}t = \frac{n-1}{m-n+1}I(n-1,m).\end{aligned}$$

Ripetendo lo stesso calcolo $n-1$ volte giungiamo a

$$\begin{aligned}I(n,m) &= \frac{(n-1)(n-2)\cdots 2\cdot 1}{(m-n+1)(m-n+2)\cdots(m-1)}\,I(1,m)\\ &= \frac{(n-1)(n-2)\cdots 2\cdot 1}{(m-n+1)(m-n+2)\cdots(m-1)}\int_0^1 (1-t)^{m-1}\mathrm{d}t\\ &= \frac{(n-1)(n-2)\cdots 2\cdot 1}{(m-n+1)(m-n+2)\cdots(m-1)m} = \frac{1}{n\binom{m}{n}}.\end{aligned} \tag{B.1.8}$$

Confrontando (B.1.7) e (B.1.8), è chiaro che $n\binom{m}{n} \mid d_m$ per ogni n intero, $1 \leq n \leq m$. Pertanto, come caso particolare, abbiamo che $m\binom{2m}{m} \mid d_{2m}$. Osservando che vale l'identità

$$(2m+1)\binom{2m}{m} = (m+1)\binom{2m+1}{m+1},$$

possiamo scrivere che $(2m+1)\binom{2m}{m} \mid d_{2m+1}$. Dato che $(m, 2m+1) = 1$ e che $d_{2m} \mid d_{2m+1}$, concludiamo

$$m(2m+1)\binom{2m}{m} \mid d_{2m+1}.$$

Ma, considerando lo sviluppo binomiale di $2^{2m} = (1+1)^{2m}$ e osservando che $\binom{2m}{m}$ è il massimo coefficiente ivi presente, abbiamo immediatamente che $(2m+1)\binom{2m}{m} \geq 2^{2m}$. Quindi

$$d_{2m+1} \geq m2^{2m}. \tag{B.1.9}$$

Applicando la (B.1.9) abbiamo che

$$d_{2m+1} \geq 2 \cdot 2^{2m} = 2^{2m+1} \quad \text{per ogni} \quad m \geq 2$$

e che

$$d_{2m+2} \geq d_{2m+1} \geq 4 \cdot 2^{2m} = 2^{2m+2} \quad \text{per ogni} \quad m \geq 4.$$

Le ultime due disequazioni implicano che $d_m \geq 2^m$ per ogni $m \geq 9$. Il lemma segue osservando che $d_8 = 840 > 2^8$ e $d_7 = 420 > 2^7$. □

Se $N \geq 7$ la (B.1.4) e la (B.1.6) insieme implicano

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

Riassumendo: si può dimostrare che esistono due costanti positive $c < C$ tali che per ogni $N \geq 2$ si ha

$$\frac{cN}{\log N} < \pi(N) < \frac{CN}{\log N}. \tag{B.1.10}$$

Valori espliciti sono specificati, ad esempio, in [7, teorema 4.6] e sono $c = 1/6$ e $C = 6$.

In definitiva, i numeri primi sono piuttosto numerosi: utilizzando le relazioni ottenute qui sopra, si trova che l'ordine di grandezza di p_n, l'n-esimo numero primo, è $n \log n$. Mediante un procedimento simile all'integrazione per parti (il

suo analogo discreto: si veda il teorema B.2.4), è possibile dedurre dalla formula di Mertens (B.1.2) che per $N \to +\infty$ si ha

$$\sum_{p \le N} \frac{1}{p} = \log\log N + O(1). \tag{B.1.11}$$

Quest'ultima relazione è rilevante per l'analisi di complessità del crivello di Eratostene descritto nel §5.1. Il crivello suggerisce una formula per determinare $\pi(N)$ quando N è grande, senza dover conoscere individualmente i singoli numeri primi. Utilizzando le idee di Eratostene, Legendre scoprì una formula che permette di calcolare $\pi(x)$ iterativamente. Prima di scriverla, introduciamo due nuove funzioni:

$$\begin{aligned} P(z) &\stackrel{\text{def}}{=} \prod_{p \le z} p = \exp(\theta(z)); \\ \mu(n) &\stackrel{\text{def}}{=} \begin{cases} 1 & \text{se } n = 1; \\ 0 & \text{se esiste } p \text{ primo tale che } p^2 \mid n; \\ (-1)^k & \text{se } n = p_1 \cdots p_k \text{ con } p_1 < p_2 < \cdots < p_k. \end{cases} \end{aligned} \tag{B.1.12}$$

La seconda è detta funzione di Möbius. La formula di Legendre è dunque

$$\pi(x) - \pi\left(x^{1/2}\right) + 1 = \sum_{d \mid P\left(x^{1/2}\right)} \mu(d) \left\lfloor \frac{x}{d} \right\rfloor. \tag{B.1.13}$$

La dimostrazione è molto semplice: ci sono esattamente $\lfloor x \rfloor$ interi $\le x$ (il termine $d = 1$). Ogni primo $p \le x^{1/2}$ divide $\lfloor x/p \rfloor$ di questi interi: questi vengono sottratti poiché $\mu(p) = -1$ per ogni primo p. Ma ora abbiamo indebitamente sottratto due volte tutti i numeri che sono divisibili per 2 o più primi distinti, e così via. La formula di Legendre ha un indubbio interesse storico, ma scarsa utilità pratica perché ha troppi addendi: recentemente è stato annunciato il valore esatto di $\pi\left(10^{25}\right)$, calcolato per mezzo di un procedimento combinatorio molto più efficiente, descritto nel lavoro di Deléglise e Rivat [28].

B.2 Risultati quantitativi

Aggiungiamo senza dimostrazione alcuni risultati relativi alla distribuzione dei numeri primi, per prima cosa per il loro interesse intrinseco, e poi perché sono rilevanti per la determinazione della complessità di alcuni algoritmi di fattorizzazione. Scriveremo $f(x) \sim g(x)$ per $x \to +\infty$ per indicare che vale la relazione

$$\lim_{x \to +\infty} \frac{f(x)}{g(x)} = 1.$$

Teorema B.2.1 (dei numeri primi) *Poniamo*

$$\mathrm{li}(x) \stackrel{\text{def}}{=} \int_2^x \frac{\mathrm{d}t}{\log t}.$$

Per $x \to +\infty$ si ha

$$\pi(x) = \mathrm{li}(x) + O\big(x\exp\{-\sqrt{\log x}\}\big).$$

Si osservi che per $x \to +\infty$ si ha $\mathrm{li}(x) \sim x(\log x)^{-1}$, ma la relazione espressa dal teorema dei numeri primi è più precisa dal punto di vista numerico.

Il teorema dei numeri primi fu dimostrato indipendentemente da Hadamard e da de la Valleé Poussin nel 1896. Fu Gauss che per primo congetturò la validità di questo teorema, circa un secolo prima. Per motivi tecnici, la dimostrazione passa attraverso le funzioni di Chebyshev θ e ψ. Un semplice calcolo mostra che per $x \to +\infty$ si ha $\theta(x) \sim \psi(x) \sim \pi(x)\log x$, e quindi $\theta(x) \sim x$. È piuttosto curioso il fatto che la vera difficoltà del teorema dei numeri primi, nella forma più semplice $\pi(x) \sim x(\log x)^{-1}$, è la dimostrazione che il limite di $\pi(x)(\log x)/x$ esiste, mentre è del tutto elementare dimostrare che, se esiste, allora vale 1. Quest'ultima è una conseguenza quasi immediata della (B.1.11).

Una versione più generale del risultato qui sopra riguarda i numeri primi nelle progressioni aritmetiche.

Teorema B.2.2 (dei numeri primi nelle progressioni aritmetiche) *Fissato arbitrariamente $A > 0$ e posto*

$$\pi(x;q,a) \stackrel{\text{def}}{=} \mathrm{card}(\{p\colon\ p \text{ è primo, } p \le x,\ p \equiv a \bmod q\}),$$

esiste una costante $C = C(A) > 0$ tale che, uniformemente per $1 \le q \le (\log x)^A$, $a \in \mathbb{Z}$ tale che $(a,q) = 1$, si ha

$$\pi(x;q,a) = \frac{1}{\varphi(q)}\,\mathrm{li}(x) + O\big(x\exp\{-C\sqrt{\log x}\}\big).$$

In particolare, fissati $a \in \mathbb{Z}$ e $q \ge 1$, se $(a,q) = 1$ allora circa $1/\varphi(q)$ dei numeri primi nell'intervallo $[1,x]$ sono $\equiv a \bmod q$. Osserviamo che in entrambi i casi è stato dimostrato un risultato leggermente più forte, ma più complicato da enunciare. Si congettura inoltre che debba valere un risultato ancora più forte, che, per semplicità, enunciamo solo nel primo caso.

Congettura B.2.3 (Riemann) *Per $x \to +\infty$ si ha*

$$\pi(x) = \mathrm{li}(x) + O\big(x^{1/2}\log x\big).$$

Siamo molto lontani dal poter dimostrare un risultato del genere, che in un certo senso è ottimale. Infatti, è stato provato che l'esponente $1/2$ non può essere abbassato. Una generalizzazione di questa congettura, valida anche per i numeri primi nelle progressioni aritmetiche, implica la congettura 11.6.3.

Fu Riemann in un articolo del 1859 - il suo unico lavoro in teoria dei numeri - ad indicare la strada per affrontare questi problemi: in effetti, Eulero aveva introdotto la funzione, che oggi chiamiamo zeta di Riemann, definita da

$$\zeta(s) \stackrel{\text{def}}{=} \sum_{n\geq 1} \frac{1}{n^s} = 1 + \frac{1}{2^s} + \frac{1}{3^s} + \frac{1}{4^s} + \cdots \tag{B.2.1}$$

e l'aveva studiata per valori *reali* di $s > 1$. In particolare aveva dimostrato l'identità

$$\begin{aligned} \zeta(s) &= \prod_p \left(1 - \frac{1}{p^s}\right)^{-1} \\ &= \left(1 + \frac{1}{2^s} + \frac{1}{2^{2s}} + \frac{1}{2^{3s}} + \cdots\right) \cdot \left(1 + \frac{1}{3^s} + \frac{1}{3^{2s}} + \frac{1}{3^{3s}} + \cdots\right) \cdots \end{aligned} \tag{B.2.2}$$

dove il prodotto è fatto su tutti i numeri primi presi in ordine crescente. L'identità espressa dalle (B.2.1)-(B.2.2) è importante perché in una delle due espressioni i numeri primi non compaiono esplicitamente. La dimostrazione si dà moltiplicando formalmente fra loro le infinite serie a destra nella (B.2.2) ed usando il teorema di fattorizzazione unica 11.1.2. Non è difficile dimostrare dalla (B.2.1) che

$$\lim_{s\to 1+} \zeta(s) = +\infty,$$

e questo implica, come osservò lo stesso Eulero, che esistono infiniti numeri primi. Infatti, se l'insieme dei numeri primi fosse finito, allora il prodotto a destra nella (B.2.2) avrebbe un limite finito per $s \to 1+$.

Riemann mostrò che la chiave per capire la distribuzione dei numeri primi sta nel considerare ζ come una funzione della variabile *complessa* $s = \sigma + it$. In particolare mostrò che ζ ha un prolungamento meromorfo a $\mathbb{C} \setminus \{1\}$ e che in $s = 1$ la funzione zeta ha un polo semplice con residuo 1. Riemann, inoltre, determinò altre importanti proprietà della funzione ζ ed in particolare mostrò che l'errore che si commette nel teorema dei numeri primi quando si approssima $\pi(x)$ con $\mathrm{li}(x)$ dipende in modo cruciale dalla posizione degli zeri complessi di ζ.

La dimostrazione rigorosa di tutte le proprietà della funzione zeta scoperte da Riemann (tutte meno una, per la precisione) fu portata a termine da von Mangoldt, Hadamard e de la Valleé Poussin, e culminò con la dimostrazione del teorema dei numeri primi. L'unica proprietà non ancora dimostrata, per l'appunto, è la congettura di Riemann B.2.3, probabilmente il più importante problema aperto della teoria dei numeri, se non dell'intera matematica.

B.2.1 Fattori primi distinti di un numero

Sia $\omega(n)$ il numero di fattori primi *distinti* dell'intero n. Non è particolarmente difficile provare che $\omega(n) = O(\log n)$ ma, usando le formule del paragrafo precedente, si dimostra la relazione più precisa

$$\limsup_{n\to+\infty} \frac{\omega(n)\log(\log n)}{\log n} = 1.$$

Questo dà una limitazione al numero di chiamate ricorsive degli algoritmi di fattorizzazione dei §§5.3-5.4. Si noti che, in un senso preciso, di solito un intero n ha approssimativamente $\log(\log n)$ fattori primi distinti. Anche questa è una semplice conseguenza della stima di Mertens B.1.11.

B.2.2 Formula di sommazione di Abel

La formula di somma di Abel, anche detta di sommazione parziale o di somma per parti, è un utile strumento che consente di esprimere somme mediante integrali.

Teorema B.2.4 *Data una successione di numeri complessi $(a_n)_{n\in\mathbb{N}}$ ed una funzione $\phi : (0,+\infty) \to \mathbb{C}$ di classe $\mathcal{C}^1((0,+\infty))$, sia*

$$A(x) = \sum_{n\le x} a_n.$$

Allora, per $x \ge 1$ si ha

$$\sum_{n\le x} a_n\phi(n) = A(x)\phi(x) - \int_1^x A(t)\phi'(t)dt.$$

Tralasciamo la dimostrazione, che il lettore interessato può trovare in [7, teorema 4.2], per concentrarci sulle sue conseguenze per il calcolo della complessità del crivello di Eratostene, si veda il §5.1, anche se per tale scopo sarebbe sufficiente semplicemente osservare che

$$\sum_{p\le x} \log^2 p \le \pi(x)\log^2 x < Cx\log x.$$

Applicando il teorema B.2.4 al caso $a_n = 1$ se n è primo, $a_n = 0$ se n è composto e $\phi(x) = \log^2 x$, abbiamo

$$\sum_{p\le x} \log^2 p = \sum_{2\le n\le x} a_n \log^2 n = \pi(x)\log^2 x - \int_2^x \frac{2\pi(t)\log t}{t} dt.$$

Per la relazione (B.1.10) si ha allora che

$$\sum_{p\le x}\log^2 p < Cx\log x - 2c\int_2^x dt = Cx\log x - 2c(x-2) = O(x\log x).$$

L'approccio che utilizza la formula di somma parziale, se combinato con il teorema B.2.1, consente però di dimostrare un risultato più forte, ossia che esiste $C>0$ tale che

$$\sum_{p\le x}\log^2 p = x\log x - x + O\big(x\exp\{-C\sqrt{\log x}\}\big)$$

per x sufficientemente grande.

B.3 Numeri senza fattori primi grandi

Abbiamo visto nella descrizione del crivello quadratico nel §5.4.1 che è importante la distribuzione degli interi privi di fattori primi "grandi": infatti il tempo di esecuzione (più correttamente, la complessità dell'algoritmo) dipende dalla frequenza con cui si trovano interi che si scompongono completamente in fattori tutti appartenenti alla base di fattori $\mathcal{B}$. Definiamo quindi

$$\Psi(x,y) \stackrel{\text{def}}{=} \operatorname{card}(\{n\le x\colon\ p\mid n \Rightarrow p\le y\}).$$

L'obiettivo è il conteggio degli interi $n\le x$ che non hanno fattori primi "grandi," dove la grandezza dei fattori primi è misurata dal parametro y. È possibile dimostrare che il comportamento di questa funzione dipende in modo cruciale dal valore di $u := (\log x)/\log y$, quando $x\ge 1$, $y\ge 2$. Non è facile enunciare un risultato valido per ogni valore di u: il più significativo è forse la relazione $\Psi(x,y) = xu^{-(1+o(1))\log u}$, che vale uniformemente in un'ampia regione di valori di u, e cioè $u\le y^{1-\varepsilon}$, dove $\varepsilon>0$ è fissato, y ed u tendono ad infinito.

Non è possibile includere la dimostrazione in questa sede, ma non è difficile ottenere informazioni su $\Psi(x,y)$ in alcuni casi particolari interessanti. Consideriamo per primo il caso in cui y è limitato; poniamo per brevità $k=\pi(y)$, ed indichiamo i k numeri primi $\le y$ con $p_1,\dots,p_k$. Se n è uno degli interi contati da $\Psi(x,y)$, allora $n = p_1^{\alpha_1}\cdots p_k^{\alpha_k}$, dove $\alpha_i\in\mathbb{N}$, e quindi $\alpha_1\log p_1+\cdots+\alpha_k\log p_k \le \log x$. Stiamo contando il numero dei punti a coordinate intere nel "tetraedro" $T_k(x) = \{(x_1,\dots,x_k)\in\mathbb{R}^k\colon x_1\ge 0,\dots,x_k\ge 0, x_1\log p_1+\cdots+x_k\log p_k\le\log x\}$. Poiché $T_k(x)$ è un insieme convesso, il numero di questi punti non differisce molto dal suo volume, che vale $\left(k!\prod_{p\le y}\log p\right)^{-1}(\log x)^k$. Abbiamo dunque dimostrato che

$$\Psi(x,y)\sim\left(\pi(y)!\prod_{p\le y}\log p\right)^{-1}(\log x)^{\pi(y)}. \tag{B.3.1}$$

Per estendere questo risultato a valori di y "piccoli" rispetto ad x, ma non necessariamente limitati, possiamo usare un'idea di Rankin basata sul prodotto (B.2.2), o meglio, su una parte finita del prodotto stesso. Per ogni $\sigma > 0$ si ha

$$\begin{aligned}\Psi(x,y) = \sum_{\substack{n\le x\\ p|n\Rightarrow p\le y}} 1 &\le \sum_{\substack{n\le x\\ p|n\Rightarrow p\le y}} \Big(\frac{x}{n}\Big)^{\sigma} \le \sum_{\substack{n\ge 1\\ p|n\Rightarrow p\le y}} \Big(\frac{x}{n}\Big)^{\sigma}\\ &= x^{\sigma}\prod_{p\le y}\big(1-p^{-\sigma}\big)^{-1}. \end{aligned} \tag{B.3.2}$$

Questa relazione è interessante solo per $\sigma < 1$, poiché per $\sigma \ge 1$ il secondo membro è $\ge x$: vogliamo dunque scegliere σ in modo "ottimale" per ottenere una buona maggiorazione. Se $2 \le y \le \sqrt{\log x}$, prendiamo $\sigma = c(\log x)^{-1}$ dove $c > 0$ verrà scelta più avanti. Quindi $p^{\sigma} = \exp\big(\sigma \log p\big) = 1 + \sigma \log p + O\big(\sigma^2 (\log p)^2\big)$ e la (B.3.2) dà

$$\begin{aligned}\log\Psi(x,y) &\le c + \sum_{p\le y} \log\frac{p^{\sigma}}{p^{\sigma}-1} = c + \sigma\theta(y) - \sum_{p\le y}\log\Big(\sigma\log p\big(1+O(\sigma\log p)\big)\Big)\\ &= c + \sigma\theta(y) - \pi(y)\log\sigma - \sum_{p\le y}\log\log p + O(\sigma\theta(y))\\ &= c - \pi(y)\log c + \pi(y)\log\log x - \sum_{p\le y}\log\log p + O(\sigma y).\end{aligned}$$

Si osservi ora che la funzione $g(t) = t - A\log t$ ha un minimo per $t = A$: scelto dunque $c = \pi(y)$ si ottiene

$$\Psi(x,y) \le \left(\frac{e}{\pi(y)}\right)^{\pi(y)} \Big(\prod_{p\le y}\log p\Big)^{-1} (\log x)^{\pi(y)} \Big\{1 + O\Big(\frac{y^2}{\log x \log y}\Big)\Big\} \tag{B.3.3}$$

Per la formula di Stirling 6.6.1 si ha $n! \sim \sqrt{2\pi n}(n/e)^n$ e quindi la stima (B.3.3) non è molto più debole della formula asintotica (B.3.1). È importante cercare di estendere questo tipo di stime anche al caso in cui y è più grande: per esempio, prendendo $\sigma = 1 - (2\log y)^{-1}$ in (B.3.2) ed usando la formula di Mertens e la stima $p^{1-\sigma} = 1 + O((1-\sigma)\log p)$ si ottiene la maggiorazione universale, valida per $x \ge 1$, $y \ge 2$, $\Psi(x,y) \ll xe^{-u/2}\log y$, dove $u = (\log x)/\log y$.

B.4 Formule per i numeri primi

Abbiamo visto sopra che sarebbe molto comodo se esistesse un modo efficiente per generare numeri primi: non possiamo dedicare molto spazio a questa questione, ma è abbastanza semplice mostrare che, almeno nel senso più ingenuo del termine, non esistono "formule" per i numeri primi il cui costo sia inferiore al

crivello di Eratostene discusso nel §5.1. Ci limitiamo a segnalare questo semplice fatto: se $f \in \mathbb{Z}[x]$ assume valore primo per ogni intero, allora f è costante. Infatti, se poniamo $p = f(1)$, si ha $f(1+np) \equiv f(1) \equiv 0 \bmod p$ per ogni $n \in \mathbb{Z}$. Dunque $p \mid f(1+np)$ per ogni $n \in \mathbb{Z}$ e quindi $f(1+np) = \pm p$ poiché deve essere un numero primo, ma questo è assurdo se f non è costante, perché allora $|f(1+np)|$ dovrebbe tendere a $+\infty$ quando $n \to \infty$.

B.5 Pseudoprimi e numeri di Carmichael

Alford, Granville e Pomerance [5] hanno dimostrato che esistono infiniti numeri di Carmichael, e che questi sono piuttosto frequenti, nel senso che, per x sufficientemente grande, si ha

$$C(x) \stackrel{\text{def}}{=} \text{card}(\{n \leq x\colon n \text{ è di Carmichael}\}) \geq x^{2/7}.$$

Nel 2005 Harman [38] ha migliorato tale stima dimostrando $C(x) > x^{0.33}$. In effetti ci si aspetta che l'ordine di grandezza di $C(x)$ sia nettamente superiore. Nel 1956 Erdős [31], assumendo la validità di una congettura sulla distribuzione dei numeri primi nelle progressioni aritmetiche, ha dedotto che per ogni $\varepsilon > 0$ esiste $x_0(\varepsilon) > 0$ tale che si abbia $C(x) > x^{1-\varepsilon}$ per $x > x_0(\varepsilon)$. Per maggiori dettagli si consulti l'articolo di Granville e Pomerance [36]. Per quanto riguarda la ricerca della costruzione di una tabella contenente tutti i numeri di Carmichael minori o uguali di un certo limite, varie investigazioni sono state effettuate da Pinch. Attualmente il limite è 10^{21}; si consulti Pinch [66].

Conviene anche osservare che in questa discussione abbiamo considerato solo gli pseudoprimi relativi alla proprietà di Fermat (teorema 11.2.1). Come abbiamo visto nel §11.8 è possibile considerare il concetto di pseudoprimalità esteso a qualunque condizione necessaria, ma non sufficiente, per la primalità.

B.6 Congettura di Artin

Concludiamo questa discussione con l'enunciato della

Congettura B.6.1 (Artin) *Sia $g \in \mathbb{Z}$ un intero diverso da 0, -1 e che non sia un quadrato perfetto. Allora g è un generatore di $\mathbb{Z}_p^*$ per infiniti valori di p e, più precisamente,*

$$\lim_{x \to +\infty} \frac{\text{card}(\{p \leq x\colon p \text{ è primo e } g \text{ genera } \mathbb{Z}_p^*\})}{\pi(x)} > 0.$$

È evidente che se $g = -1$ oppure se $g = m^2$ allora g al massimo ha ordine 2 o, rispettivamente, $\frac{1}{2}(p-1)$, e quindi non può generare $\mathbb{Z}_p^*$. Oggi è noto che le eventuali eccezioni a questa congettura sono molto rare.

Complementi C
Classi di complessità dei problemi di decisione

Diamo qui una breve trattazione delle classi di complessità riguardanti i problemi di decisione. Per una esposizione più esauriente rimandiamo al libro di Aho, Hopcroft e Ullman [3].

Definizione C.0.1 (Problema di decisione) *Diciamo* problema di decisione *un problema le cui soluzioni (o output) consistono solamente di risposte SÌ o NO.*

I problemi di decisione possono essere classificati in base alla complessità computazionale degli algoritmi che li risolvono. Diremo che un problema appartiene ad una certa classe se il migliore algoritmo noto per risolvere tale problema ha le caratteristiche indicate dalla classe. Il problema di decisione cambia classe di appartenenza quando viene scoperto un algoritmo più efficiente. Le due principali classi sono P ed NP. Introdurremo anche la classe RP per spiegare che cosa significhi complessità "probabilisticamente polinomiale".

C.1 La classe P

Per prima cosa definiamo il concetto di algoritmo deterministico.

Definizione C.1.1 (Algoritmo deterministico) *Un algoritmo viene detto deterministico se il suo comportamento può essere completamente predetto dall'input.*

Vediamo adesso quale significato ha il dire che un problema è risolubile con complessità polinomiale.

Definizione C.1.2 (Classe P) *Un problema $\mathcal{P}$ appartiene alla classe di complessità P se esiste un polinomio f ed un algoritmo deterministico tale che per ogni istanza (ossia per ogni caso) del problema $\mathcal{P}$, avente lunghezza dell'input minore o uguale a N, l'algoritmo risponde correttamente con complessità computazionale $O(f(N))$.*

La classe P è interessante perché la maggior parte dei problemi in essa compresi possono essere risolti efficientemente nel mondo reale. Tuttavia esistono delle eccezioni a causa delle costanti comprese, ma non specificate, nelle stime di complessità computazionale. Ciò vuol dire che, nella pratica, usando un algoritmo polinomiale invece di uno avente complessità esponenziale, non è detto che la soluzione di un problema appaia dopo che è trascorso un tempo minore dall'inizio dell'esecuzione; per convincersi di ciò basta studiare la crescita delle funzioni N^{100} ed $e^{N/10000}$ e determinare per quali valori di N la prima è maggiore della seconda.

È solo per *N sufficientemente grande* che un algoritmo di complessità polinomiale è sicuramente più efficiente di un altro con complessità esponenziale: il problema, spesso, è stabilire in modo esplicito che cosa significa in concreto "sufficientemente grande."

Esempio C.1.3 *Da quanto abbiamo detto nel* §5.2 *sappiamo che la primalità è un problema di decisione appartenente a P perché l'algoritmo di Agrawal, Kayal & Saxena, nella versione di Lenstra & Pomerance, ha complessità* $O\big((\log n)^{6+\varepsilon}\big)$. *Si noti che la lunghezza dell'input (detta N nella definizione precedente) in questo caso è* $\log n$.

C.2 La classe NP

Definizione C.2.1 (Classe NP) *Un problema $\mathcal{P}$ appartiene alla classe di complessità NP se, per ogni istanza del problema $\mathcal{P}$, avendo illimitata potenza di calcolo non solo si può rispondere al problema ma, nel caso tale risposta sia SÌ, si può inoltre fornire una prova (detta certificato o* witness*) che può essere utilizzata per verificare, con complessità polinomiale, la correttezza della risposta.*

Un modo alternativo per definire NP è quello di dire che:

Un problema $\mathcal{P}$ appartiene alla classe di complessità NP se esiste un algoritmo non-deterministico che lo risolve con complessità polinomiale; per algoritmo non-deterministico si intende un algoritmo che, ottenuto un input extra (un certificato), può trovare e verificare la soluzione del problema.

In modo più specifico un problema di decisione appartiene a NP se esiste un algoritmo per cui, per ogni possibile input la cui risposta è SÌ, esiste un certificato che consente all'algoritmo di rispondere SÌ con complessità polinomiale. D'altra parte, se la risposta corretta è NO, non esistono certificati con cui l'algoritmo stesso possa erroneamente rispondere SÌ.

Un modo oramai standard di riferire alla classe NP è quello di dire che sono i problemi di decisione per cui la risposta può essere verificata con complessità polinomiale.

Osservazione C.2.2 *NP significa* non-deterministico polinomiale *e NON significa "non polinomiale"!!*

Esempio C.2.3 *Il problema della somma di sottoinsiemi (*subset sum problem*) è il seguente: dato un sottoinsieme dei numeri naturali $A = \{a_1, a_2, \ldots, a_n\}$ e un numero naturale s, determinare se esiste o meno un sottoinsieme B di A per cui la somma dei suoi elementi sia s. Tale problema appartiene alla classe NP perché, nel caso venga fornito come certificato un sottoinsieme di A di cui si afferma che la somma degli elementi è uguale a s, la verifica richiede complessità polinomiale. Si noti che non sono noti né algoritmi deterministici né algoritmi probabilistici che risolvano tale problema con complessità polinomiale.*

Si definisce anche la classe co-NP che è analoga alla NP in cui si sostituisce la risposta positiva con la risposta negativa.

Esempio C.2.4 *La fattorizzazione di interi definita come segue: "dato n e k, n ha fattori primi nell'intervallo $[2,k]$?" appartiene ad NP. Infatti determinato in qualche modo un fattore m, il certificato è m stesso e la verifica consiste nell'eseguire la divisione n/m (che ha complessità polinomiale). Inoltre la fattorizzazione appartiene anche a co-NP. Infatti se la risposta è NO, dovrà essere fornita come certificato la completa fattorizzazione di n insieme a certificati polinomiali della primalità di ogni fattore. Da ciò si può verificare con complessità polinomiale se n ha fattori primi minori o uguali a k.*

Esempio C.2.5 *Come certificato di non-primalità di un intero n è sufficiente esibire una base $a \in \mathbb{Z}$ tale che $a^{(n-1)/2} \not\equiv \left(a \mid n\right) \mod n$, per il teorema di Eulero* 11.7.10*. La verifica della congruenza ed il calcolo del simbolo di Legendre hanno complessità polinomiale.*

Osservazione C.2.6 *Si ha chiaramente che $P \subset NP$.*

Non è invece per nulla chiaro se esistano problemi che appartengano a NP e non appartengano a P. Ciò conduce ad enunciare la seguente:

Congettura C.2.7 *$P \neq NP$.*

Una dimostrazione che provi o confuti la congettura C.2.7 ha notevole importanza teorica tanto da essere indicata tra i problemi fondamentali della matematica; si veda per esempio: http://www.claymath.org/millennium-problems.

C.3 Problemi NP-completi

Per prima cosa dobbiamo definire che cosa intendiamo per *riduzione (con complessità polinomiale) di un problema di decisione* ad un altro.

Dati due problemi $\mathcal{P}_1$ e $\mathcal{P}_2$, diciamo che $\mathcal{P}_1$ si riduce con complessità polinomiale a $\mathcal{P}_2$ se esiste un algoritmo deterministico polinomiale nella lunghezza dell'input di $\mathcal{P}_1$ che, data una istanza P_1 di $\mathcal{P}_1$, costruisce una istanza P_2 di $\mathcal{P}_2$ per cui le risposte ai rispettivi problemi di decisione sono identiche.

Definizione C.3.1 (Problemi NP-completi) *Un problema $\mathcal{P} \in NP$ è detto NP-completo, se ogni altro problema della classe NP si può ridurre a $\mathcal{P}$ con complessità polinomiale.*

Esempio C.3.2 *Supponiamo che $\mathcal{P}_1$ sia il problema di decidere se la funzione polinomiale $p(x)$ di grado 2 a coefficienti interi abbia radici distinte e che $\mathcal{P}_2$ sia il problema di decidere se un numero reale x sia positivo. $\mathcal{P}_1$ si riduce polinomialmente a $\mathcal{P}_2$ perché, se $p(x) = ax^2 + bx + c$, per decidere se $p(x)$ ha radici distinte è sufficiente considerare se $\Delta = b^2 - 4ac$ è positivo. Quindi $\mathcal{P}_1(p)$ ha la stessa risposta di $\mathcal{P}_2(\Delta)$ ed il calcolo di Δ ha complessità polinomiale.*

Si noti che se esiste un algoritmo che risolve un problema NP-completo con complessità polinomiale, allora P = NP. Per tale ragione si pensa che per i problemi NP-completi non esistano algoritmi aventi complessità polinomiale. In tal senso, i problemi NP-completi sono i problemi "più difficili" della classe NP.

C.4 Altre classi: BPP, RP, ZPP

Definizione C.4.1 (Classe BPP) *La classe BPP* (bounded probabilistic polynomial-time) *è la classe di problemi di decisione che possono essere risolti con complessità polinomiale usando un algoritmo che può compiere scelte casuali durante la sua esecuzione* (randomized algorithm) *con al più una probabilità esponenzialmente piccola di errore su ogni input.*

Se un algoritmo deterministico viene fatto lavorare più volte sullo stesso input, esso esegue sempre lo stesso insieme di operazioni e produce sempre lo stesso output. Invece un algoritmo probabilistico sceglierà ogni volta differenti cammini di esecuzione e potrà fornire differenti risposte, come spieghiamo qui sotto.

Un'altra classe riguardante algoritmi probabilistici è la seguente.

Definizione C.4.2 (Classe ZPP) *Alla classe ZPP* (zero-error probabilistic polynomial-time) *appartengono i problemi di decisione che possono essere risolti*

con errore pari a zero ma per cui l'algoritmo può, con bassa probabilità, avere complessità non polinomiale (ossia la complessità computazionale "attesa" è polinomiale).

L'ultima classe che definiamo è invece analoga a BPP.

Definizione C.4.3 (Classe RP) *Un problema $\mathcal{P} \in RP$ se esiste un algoritmo che, includendo una scelta casuale, produce una risposta con complessità polinomiale. Inoltre la risposta NO è corretta, mentre si ha solo una probabilità $> 1/2$ che la risposta SÌ sia corretta.*

Alternativamente si può osservare che la classe RP è analoga alla BPP tranne per il fatto che in RP l'algoritmo che risolve il problema può commettere errori solo sugli input per cui la risposta deve essere SÌ.

Si noti inoltre che se $\mathcal{P} \in$ RP allora, per ogni $\varepsilon > 0$ esiste un algoritmo per cui la probabilità che la risposta SÌ sia corretta è $> 1 - \varepsilon$; infatti basta iterare un numero sufficientemente grande di volte l'algoritmo della definizione.

Definizione C.4.4 (Classe co-RP) *Un problema $\mathcal{P} \in co\text{-}RP$ se esiste un algoritmo che, includendo una scelta casuale, produce una risposta con complessità polinomiale. Inoltre la risposta SÌ è corretta, mentre si ha solo una probabilità $> 1/2$ che la risposta NO sia corretta.*

In altre parole un problema di decisione appartiene a co-RP se il suo problema "complementare" appartiene a RP. Si noti che il problema complementare della primalità è il decidere se un intero è composto.

Esempio C.4.5 *Come esempio consideriamo il problema: dato n un intero, n è composto? Sappiamo che l'algoritmo di pseudoprimalità di Miller & Rabin prova con complessità polinomiale il fatto che n sia composto (cioè la risposta SÌ è corretta) mentre la risposta NO (ossia n è primo) ha una probabilità $> 3/4$ di essere corretta. Quindi questo problema appartiene a co-RP. E quindi la primalità appartiene a RP. Si noti che, usando l'algoritmo di Agrawal, Kayal & Saxena si ottiene che in realtà questi problemi appartengono a P.*

C.5 Relazioni tra le classi

Concludiamo schematizzando qualche relazione tra le classi di complessità. La situazione è la seguente:

$$\text{P} \subset \text{ZPP} \subset \text{RP} \subset \text{NP}$$

$$\text{P} \subset \text{ZPP} \subset \text{co-RP} \subset \text{BPP}.$$

Non vi è alcuna relazione nota tra BPP e NP. Tuttavia, se P=NP, allora BPP=P e le relazioni precedenti divengono banali.

Complementi D
Complementi matematici

D.1 Funzioni aritmetiche

Definizione D.1.1 *Si dice* funzione aritmetica *un'applicazione* $f\colon \mathbb{N}^* \to \mathbb{C}$. *Una funzione aritmetica* f *si dice* moltiplicativa *se* $f(1) = 1$ *e per ogni* n, $m \in \mathbb{N}^*$ *con* $(n,m) = 1$ *si ha* $f(nm) = f(n)f(m)$. *Se questo vale per ogni* n, $m \in \mathbb{N}^*$, *allora* f *si dice* completamente moltiplicativa.

Esempio D.1.2 *Sono funzioni aritmetiche moltiplicative la funzione* φ *di Eulero, la funzione* R_f *definita nell'esercizio* 10.2.20*, la funzione* μ *di Möbius definita in* (B.1.12)*, ecc. Sono funzioni aritmetiche completamente moltiplicative il simbolo di Legendre* $\left(\cdot \mid p\right)$ *per ogni numero primo* p *fissato, e le funzioni* $n \to n^k$, *dove* k *è un qualsiasi numero complesso fissato.*

Anche i caratteri di Dirichlet della definizione 11.8.9 sono funzioni aritmetiche completamente moltiplicative. Ne vediamo qualche proprietà sotto forma di esercizi guidati: per una discussione completa delle loro proprietà si veda Apostol [7, capitolo 6] oppure Davenport [27, §§4-5].

Esercizio D.1.3 *Sia* $\chi\colon \mathbb{Z} \to \mathbb{C}$ *un carattere di Dirichlet modulo* n. *Dimostrare che* $\chi(1) = 1$, *e che se* $(n,h) = 1$ *allora* $\chi(h) \neq 0$.

Risposta: Dato che χ non è identicamente nullo, sia $h \in \mathbb{Z}$ tale che $\chi(h) \neq 0$; dunque $\chi(h) = \chi(h \cdot 1) = \chi(h)\chi(1)$, da cui $\chi(1) = 1$. Se $(n,h) = 1$ allora esiste $k \in \mathbb{Z}$ tale che $hk \equiv 1 \bmod n$. Dato che χ è periodico con periodo n, si ha che $\chi(hk) = \chi(1) = 1$, e d'altra parte, $\chi(hk) = \chi(h)\chi(k)$: quindi $\chi(h)$ non può essere uguale a 0. ■

Esercizio D.1.4 *Sia* $\chi\colon \mathbb{Z} \to \mathbb{C}$ *un carattere di Dirichlet modulo* n. *Dimostrare che* $(n,h) = 1$ *allora esiste* $m \in \mathbb{N}^*$ *tale che* $\chi(h)^m = 1$.

Risposta: Se $(n,h) = 1$ allora $h^{\varphi(n)} \equiv 1 \bmod n$ per il teorema di Eulero 11.2.7. Dunque $\chi\big(h^{\varphi(n)}\big) = \big(\chi(h)\big)^{\varphi(n)} = \chi(1) = 1$, e si può quindi prendere $m = \varphi(n)$. ■

D.2 Relazioni di equivalenza

Nel capitolo 10 abbiamo visto una particolare *relazione di equivalenza*. Per completezza, ne diamo la definizione formale in generale e qualche proprietà importante: si veda anche Lang [52, Appendice 2.G].

Definizione D.2.1 (Relazione di equivalenza) *Dato un insieme qualsiasi $\mathcal{A}$, diciamo che un sottoinsieme $\mathcal{R}$ del prodotto cartesiano $\mathcal{A} \times \mathcal{A}$ è una* relazione di equivalenza *se:*

- *per ogni $a \in \mathcal{A}$ si ha $(a,a) \in \mathcal{R}$ (proprietà riflessiva);*
- *se $(a,b) \in \mathcal{R}$ allora $(b,a) \in \mathcal{R}$ (proprietà simmetrica);*
- *se $(a,b) \in \mathcal{R}$ e $(b,c) \in \mathcal{R}$ allora $(a,c) \in \mathcal{R}$ (proprietà transitiva).*

Le relazioni di equivalenza più note e importanti sono l'uguaglianza, il parallelismo fra rette nello spazio euclideo, la congruenza o la similitudine fra triangoli. Si noti che è più comune indicare l'equivalenza con $a \mathcal{R} b$ piuttosto che scrivendo $(a,b) \in \mathcal{R}$, come sarebbe, formalmente, più corretto.

Definizione D.2.2 (Classe di equivalenza; insieme quoziente) *Dato un insieme $\mathcal{A}$ con una relazione di equivalenza $\mathcal{R}$, e dato $a \in \mathcal{A}$, indichiamo con*

$$[a]_{\mathcal{R}} = \{b \in \mathcal{A} \colon a \mathcal{R} b\}$$

l'insieme degli elementi di $\mathcal{A}$ che sono in relazione con a. Questo insieme si dice classe di equivalenza *di a. L'insieme di tutte le classi di equivalenza si dice* insieme quoziente *e si indica con $\mathcal{A}/\mathcal{R}$.*

Notiamo che le classi di equivalenza non possono essere vuote a causa della riflessività delle relazioni di equivalenza: in altre parole, per ogni $a \in \mathcal{A}$ si ha $a \in [a]_{\mathcal{R}}$. Inoltre, esse sono disgiunte, come mostra il lemma seguente.

Lemma D.2.3 *Dato un insieme $\mathcal{A}$ con una relazione di equivalenza $\mathcal{R}$, e dati due elementi a, $b \in \mathcal{A}$, si ha*

$$[a]_{\mathcal{R}} = [b]_{\mathcal{R}} \qquad \textit{oppure} \qquad [a]_{\mathcal{R}} \cap [b]_{\mathcal{R}} = \emptyset.$$

Dim. Se $[a]_{\mathcal{R}} \cap [b]_{\mathcal{R}} \neq \emptyset$, prendiamo $c \in [a]_{\mathcal{R}} \cap [b]_{\mathcal{R}}$. Per definizione, $a \mathcal{R} c$ e $b \mathcal{R} c$, e quindi $a \mathcal{R} b$ per le proprietà simmetrica e transitiva. Se $d \in [a]_{\mathcal{R}}$ segue che $d \mathcal{R} b$, e in definitiva $d \in [b]_{\mathcal{R}}$, cioè $[a]_{\mathcal{R}} \subseteq [b]_{\mathcal{R}}$. Per la simmetria, possiamo scambiare a con b, trovando $[b]_{\mathcal{R}} \subseteq [a]_{\mathcal{R}}$, e quindi $[a]_{\mathcal{R}} = [b]_{\mathcal{R}}$. □

L'insieme $\mathbb{Z}_n$ è il quoziente di $\mathbb{Z}$ rispetto alla relazione di congruenza modulo n, e può essere alternativamente denotato, come viene fatto su qualche altro testo, con $\mathbb{Z}/n\mathbb{Z}$. Le classi di congruenza modulo n sono progressioni aritmetiche di ragione n, e sono evidentemente disgiunte.

D.3 La Notazione di Bachmann-Landau

Useremo il simbolo di Bachmann-Landau $O(\cdot)$ con il seguente significato: siano f e g funzioni definite per $x \geq x_0$, dove x_0 è un opportuno numero reale. Se g è non negativa per $x \geq x_0$ scriviamo $f(x) = O(g(x))$ se esiste $C \in \mathbb{R}^+$ tale che per tutti gli $x \geq x_0$ si ha

$$|f(x)| \leq Cg(x).$$

Se la costante C non è uniforme, ma dipende dai parametri A, B, ..., scriveremo $f(x) = O_{A,B,\ldots}(g(x))$. Scriveremo anche $f(x) = h(x) + O(g(x))$ quando $f(x) - h(x) = O(g(x))$.

D.4 Frazioni continue

Diamo qualche rudimento della teoria delle frazioni continue, che possono essere utilizzate per dimostrare i risultati contenuti nel §5.3.2 ed anche per l'analisi della complessità dell'algoritmo di Euclide che qui è accennata nel §4.2.1. Ed è proprio da una interpretazione alternativa dell'algoritmo di Euclide che prendiamo spunto per iniziare la nostra discussione. Possiamo disporre i risultati parziali indicati nella figura 4.4 nel modo seguente:

$$\frac{43}{35} = 1 + \cfrac{1}{4 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{2}}}}$$

in maniera che siano evidenziati i valori dei quozienti successivi. Questa scrittura, che per comodità tipografica abbrevieremo nel seguito con $[1,4,2,1,2]$, si dice *frazione continua*. In generale, dato un intero non negativo a_0 ed N interi strettamente positivi a_1, ..., a_N, considereremo le proprietà della frazione continua $x = [a_0, a_1, \ldots, a_N]$ definita da

$$x = [a_0, a_1, \ldots, a_N] \stackrel{\text{def}}{=} a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\cdots + \cfrac{1}{a_{N-1} + \cfrac{1}{a_N}}}}}.$$

Una definizione rigorosa può essere data per ricorrenza ponendo

$$[a_0] \stackrel{\text{def}}{=} a_0, \qquad [a_0, a_1, \ldots, a_N] \stackrel{\text{def}}{=} a_0 + \frac{1}{[a_1, \ldots, a_N]} \quad \text{per } N \geq 1.$$

Si noti che queste definizioni possono essere date, più in generale, supponendo che $a_1, \ldots, a_N$ siano numeri reali positivi, non necessariamente interi. Dunque, dalla struttura della frazione continua è chiaro che se $N > 1$ vale l'identità

$$[a_0, a_1, \ldots, a_N] = \Big[a_0, a_1, \ldots, a_{N-1} + \frac{1}{a_N}\Big]. \tag{D.4.1}$$

Ci serviranno anche tutte le frazioni continue che si ottengono da questa tagliando via una parte della "coda" di x. Chiameremo dunque *n-esimo convergente* della frazione continua $[a_0, a_1, \ldots, a_N]$ la frazione continua $x_n := [a_0, a_1, \ldots, a_n]$, per $n = 0, \ldots, N$. Costruiamo ora la coppia di successioni finite (p_n) e (q_n):

$$\begin{cases} p_0 = a_0, \\ q_0 = 1, \end{cases} \quad \begin{cases} p_1 = a_0 a_1 + 1, \\ q_1 = a_1, \end{cases} \quad \begin{cases} p_n = a_n p_{n-1} + p_{n-2}, \\ q_n = a_n q_{n-1} + q_{n-2}, \end{cases} \quad \text{per } n = 2, \ldots, N.$$

Si noti la non casuale somiglianza con le formule (4.2.1) del §4.2. È importante osservare che ciascun p_n o q_n dipende solo da $a_0, \ldots, a_n$ e *non* dai successivi coefficienti $a_{n+1}, \ldots, a_N$. Vogliamo dimostrare che

$$x_n = [a_0, a_1, \ldots, a_n] = \frac{p_n}{q_n} \qquad \text{per } n = 0, 1, \ldots, N.$$

Per $n = 0$, per $n = 1$ e per $n = 2$ questo è immediato. Supponiamo dunque di averlo dimostrato per tutti gli interi $n = 0, 1, \ldots, m$ con $2 \le m \le N-1$ e per *tutte* le possibili scelte di $a_0 \in \mathbb{N}$ e dei numeri reali positivi $a_1, \ldots, a_m$; per la (D.4.1), possiamo scrivere la frazione continua $[a_0, \ldots, a_m, a_{m+1}]$ in termini di un'altra frazione continua più corta di un'unità, e cioè $\Big[a_0, \ldots, a_m + \frac{1}{a_{m+1}}\Big]$. Questo fatto ci permette di sfruttare l'ipotesi induttiva, che riguarda per l'appunto tutte le frazioni continue con $m+1$ coefficienti. In definitiva abbiamo

$$\begin{aligned} x_{m+1} = [a_0, \ldots, a_m, a_{m+1}] &= \Big[a_0, \ldots, a_m + \frac{1}{a_{m+1}}\Big] \\ &= \frac{(a_m + \frac{1}{a_{m+1}})p_{m-1} + p_{m-2}}{(a_m + \frac{1}{a_{m+1}})q_{m-1} + q_{m-2}} = \frac{(a_m p_{m-1} + p_{m-2})a_{m+1} + p_{m-1}}{(a_m q_{m-1} + q_{m-2})a_{m+1} + q_{m-1}} \\ &= \frac{a_{m+1}p_m + p_{m-1}}{a_{m+1}q_m + q_{m-1}} = \frac{p_{m+1}}{q_{m+1}}. \end{aligned}$$

Sempre per induzione si possono dimostrare le due formule

$$\begin{cases} p_n q_{n+1} - q_n p_{n+1} = (-1)^{n+1} & \text{per } n = 0, \ldots, N-1, \\ p_n q_{n+2} - q_n p_{n+2} = (-1)^{n+1} a_{n+2} & \text{per } n = 0, \ldots, N-2. \end{cases} \tag{D.4.2}$$

Queste sono di fondamentale importanza: infatti, equivalgono alle relazioni

$$\frac{p_n}{q_n} - \frac{p_{n+1}}{q_{n+1}} = \frac{(-1)^{n+1}}{q_n q_{n+1}}, \qquad \frac{p_n}{q_n} - \frac{p_{n+2}}{q_{n+2}} = \frac{(-1)^{n+1} a_{n+2}}{q_n q_{n+2}}.$$

La prima implica che tutti i convergenti di indice pari sono più piccoli del convergente immediatamente successivo, che a sua volta è più grande di quello che lo segue. La seconda implica che un convergente di indice pari è più piccolo del convergente di indice pari che lo segue immediatamente, mentre ogni convergente di indice dispari è più grande del successivo convergente di indice dispari. In breve, abbiamo dimostrato che valgono le disuguaglianze

$$\frac{p_0}{q_0} < \frac{p_2}{q_2} < \frac{p_4}{q_4} < \cdots \leq x = x_N \leq \cdots < \frac{p_5}{q_5} < \frac{p_3}{q_3} < \frac{p_1}{q_1}.$$

Come conseguenza immediata, notiamo in particolare la disuguaglianza

$$\left| x - \frac{p_n}{q_n} \right| \leq \frac{1}{q_n q_{n+1}} \leq \frac{1}{q_n^2},$$

valida per $n = 0, 1, \ldots, N-1$. Un'altra conseguenza importante della prima delle relazioni (D.4.2) è che $(p_n, q_n) = 1$, cioè che le frazioni prodotte in questo modo sono sempre ridotte ai minimi termini.

Infine, dimostriamo per induzione che, per $n = 2, \ldots, N$, si ha $q_n \geq 2^{(n-1)/2}$. Infatti $q_2 = 1 + a_1 \geq 2$, dato che a_1 è un intero positivo, e $q_3 = a_3 q_2 + q_1 \geq 2 + 1 = 3 > 2^{3/2}$. A questo punto, se la disuguaglianza voluta è vera per q_{n-2} e per q_{n-1}, abbiamo $q_n = a_n q_{n-1} + q_{n-2} \geq 2^{(n-2)/2} + 2^{(n-3)/2} \geq 2^{(n-1)/2}$. Questo fatto è estremamente importante perché garantisce che la lunghezza della frazione continua associata al numero razionale p/q (che supponiamo ridotto ai minimi termini) è $O(\log q)$: infatti $q = q_N \geq 2^{(N-1)/2}$, da cui ricaviamo $\log_2 q \geq \frac{1}{2}(N-1)$, e cioè $N \leq 2\log_2 q + 1$. Il numero N, la lunghezza della frazione continua, è proprio uguale al numero di passi necessari ad eseguire l'algoritmo di Euclide sulla coppia (p, q), coerentemente con quanto abbiamo visto nel §4.2.1.

Da quanto sopra si può essenzialmente dedurre il seguente risultato

Teorema D.4.1 (Hardy & Wright [37], teorema 161) *Ogni razionale positivo si può scrivere come frazione continua finita.*

Un'altra proprietà molto importante delle frazioni continue è detta "legge della miglior approssimazione" che fu per la prima volta dimostrata da Lagrange nel 1770; per la dimostrazione si veda Hardy & Wright [37], §10, teorema 184.

Teorema D.4.2 (Legendre) *Sia $x \in \mathbb{R}$. Se*

$$\left| x - \frac{p}{q} \right| < \frac{1}{2q^2}$$

allora p/q è un convergente di x.

Essenzialmente esso significa che, non appena si ottiene un'approssimazione razionale a x sufficientemente buona, allora la frazione p/q è in realtà un convergente della frazione continua di x. Questo fatto è usato nell'attacco di Wiener a RSA, si veda il teorema 3.3.6. Inseriamo anche un algoritmo per il calcolo delle frazioni continue.

Algoritmo D.4.3 (Calcolo della frazione continua, versione I) *Sia $x \in \mathbb{R}$.*

1) *Poniamo $a_0 = \lfloor x \rfloor$; quindi $x = a_0 + \xi_0$ e $\xi_0 \in [0,1)$;*
2) *se $\xi_0 \neq 0$, poniamo $a_1' = \xi_0^{-1}$ e $a_1 = \lfloor a_1' \rfloor$; quindi $a_1' = a_1 + \xi_1$ e $\xi_1 \in [0,1)$;*
3) *per $n \geq 2$, si proceda come segue:*
4) *se $\xi_{n-1} \neq 0$, poniamo $a_n' = \xi_{n-1}^{-1}$ e $a_n = \lfloor a_n' \rfloor$; quindi $a_n' = a_n + \xi_n$ e $\xi_n \in [0,1)$;*
5) *se $\xi_n = 0$ allora si dia in output $[a_0, a_1, \ldots, a_n]$; altrimenti si torni al punto 4).*

L'algoritmo continua indefinitamente a meno che non si ottenga $\xi_N = 0$; in tal caso $x = [a_0, a_1, \ldots, a_N]$. Si osservi inoltre che $a_n' = \xi_{n-1}^{-1} > 1$ e quindi $a_n \geq 1$ per $n \geq 1$. I convergenti si calcolano usando le formule (D.4.2).

Nel caso in cui x sia un razionale non intero, possiamo interpretare quanto sopra come una variante dell'algoritmo di Euclide 4.2.

Algoritmo D.4.4 (Calcolo della frazione continua, versione II) *Sia $x \in \mathbb{Q} \setminus \mathbb{Z}$, $x = h/k$, $h,k \in \mathbb{Z}$, $(h,k) = 1$ e $k > 1$.*

1) *Sia $h/k = a_0 + \xi_0$ cioè $h = a_0 k + \xi_0 k$; pertanto $a_0 k$ è il quoziente e $\xi_0 k = k_1 < k$ è il resto della divisione intera di h per k;*
2) *se $\xi_0 \neq 0$, poniamo $a_1' = \xi_0^{-1} = k/k_1$ e $k/k_1 = a_1 + \xi_1$ cioè $k = a_1 k_1 + \xi_1 k_1$ in cui $\xi_1 k_1 = k_2 < k_1$; pertanto $a_1 k_1$ è il quoziente e $\xi_1 k_1$ è il resto della divisione intera di k per k_1;*
3) *per $n \geq 2$, si proceda come segue:*
4) *se $\xi_{n-1} \neq 0$, poniamo $a_n' = \xi_{n-1}^{-1} = k_{n-1}/k_n$ e $k_{n-1}/k_n = a_n + \xi_n$ cioè $k_{n-1} = a_n k_n + \xi_n k_n$ in cui $\xi_n k_n = k_{n+1} < k_n$; pertanto $a_n k_n$ è il quoziente e $\xi_n k_n$ è il resto della divisione intera di k_{n-1} per k_n;*
5) *si procede in questo modo finché non si ha $\xi_N = 0$; in tal caso $k_{N-1} = a_N k_N$ e l'algoritmo termina.*

Quindi il calcolo della frazione continua di $x \in \mathbb{Q} \setminus \mathbb{Z}$ si esegue con l'algoritmo di Euclide e l'algoritmo in tale situazione termina certamente dopo $N = O(\log k)$ passi, come abbiamo visto precedentemente. Anche in questo caso, i convergenti si calcolano usando le formule (D.4.2).

Le frazioni continue hanno una teoria molto bella ed un'infinità di applicazioni matematiche. Una introduzione molto semplice si trova nel capitolo 6 di Conway e Guy [22], mentre la teoria generale è esposta nel capitolo 10 di Hardy e Wright

[37]. Se ne trovano trattazioni in italiano nei libri di Davenport [26] e di Olds [64]. Knuth [45] fornisce nel §4.5.3 un'applicazione allo studio della complessità dell'algoritmo di Euclide.

D.5 Alcuni cenni di algebra lineare

Raccogliamo in questa sezione alcuni fatti di base riguardanti l'algebra lineare.

D.5.1 Matrici

Una *matrice* è una tabella rettangolare di numeri (talvolta di simboli) formata da *m righe* e *n colonne*:

$$\begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix} = (a_{ij}),\ \ i = 1,\dots,m \ \text{ e } \ j = 1,\dots,n,\ \ a_{ij} \in \mathbb{R}.$$

L'insieme delle matrici con m righe e n colonne in cui gli elementi sono numeri reali, dette anche matrici di dimensione $m \times n$, si denota con $M_{m,n}(\mathbb{R})$. Chiaramente non è obbligatorio che gli elementi a_{ij} debbano appartenere a $\mathbb{R}$, ma è possibile, per esempio, scrivere matrici in cui gli $a_{ij} \in \mathbb{Z}$, $a_{ij} \in \mathbb{Z}_r$, $a_{ij} \in \mathbb{Q}$ oppure $a_{ij} \in \mathbb{C}$. Per semplicità, in quanto segue considereremo $a_{ij} \in \mathbb{R}$ a meno che non si specifichi diversamente. Chiameremo i numeri a_{ij} *coefficienti* o *entrate* ed indicheremo con R_i *riga i-esima* e con C_j *colonna j-esima* della matrice M. Usualmente è comodo indicare con $(a_1 \ \dots \ a_n)$ la matrice *riga* e con $\begin{pmatrix} a_1 \\ \vdots \\ a_m \end{pmatrix}$ la matrice *colonna*.

Nel caso in cui $m = n$, la matrice viene detta *quadrata* di *ordine n* e la speciale matrice quadrata

$$\begin{pmatrix} a_{11} & \dots & \dots & \dots \\ \dots & a_{22} & \dots & \dots \\ \vdots & & & \vdots \\ \dots & \dots & \dots & a_{nn} \end{pmatrix}$$

viene detta *diagonale* della matrice di dimensione $n \times n$.

Una matrice diagonale importante è la

$$\begin{pmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \dots & 1 \end{pmatrix}$$

che viene detta matrice *identica* e viene denotata con I_n. Analogamente è comodo indicare con Δ_n o con $\mathrm{diag}(a_1,\ldots,a_n)$, la matrice diagonale

$$\begin{pmatrix} a_1 & 0 & \ldots & 0 \\ 0 & a_2 & \ldots & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \ldots & a_n \end{pmatrix}.$$

Un concetto che troveremo utile più avanti è quello di sottomatrice: data una matrice A di dimensione $m \times n$ e dati $m' \leq m$ e $n' \leq n$, ogni matrice ottenuta dall'intersezione di m' righe e n' colonne di A è una *sottomatrice* di A.

L'insieme delle matrici può essere dotato di operazioni; in particolare $M_{m,n}(\mathbb{R})$ risulterà essere un gruppo commutativo rispetto alla somma e, nel caso $n = m$, un gruppo non commutativo rispetto al prodotto. Inoltre risulterà anche essere uno spazio vettoriale rispetto ad $\mathbb{R}$.

Definizione D.5.1 *Date $A,B \in M_{m,n}(\mathbb{R})$, con $A = (a_{ij})$ e $B = (b_{ij})$, e $\lambda \in \mathbb{R}$ definiamo*

$$\textit{la somma} \qquad A+B = (c_{ij}) \quad \textit{con} \quad c_{ij} = a_{ij} + b_{ij}$$

$$\textit{e il prodotto per scalare} \qquad \lambda A = (c_{ij}) \quad \textit{con} \quad c_{ij} = \lambda a_{ij}.$$

In questo ambito viene usualmente indicato con il sostantivo *scalare* un elemento dell'insieme numerico soggiacente all'insieme delle matrici. Per esempio, nella definizione precedente i numeri reali sono detti scalari perché stiamo considerando matrici a coefficienti reali.

Le operazioni si somma matriciale e di prodotto per uno scalare godono delle stesse proprietà delle operazioni tra numeri reali: sono *commutative*, *associative* ed esistono la matrice *nulla*

$$0 = \begin{pmatrix} 0 & \ldots & 0 \\ \vdots & & \vdots \\ 0 & \ldots & 0 \end{pmatrix}$$

e la matrice *opposta* $-A = (-a_{ij})$, $i = 1,\ldots,m$, $j = 1,\ldots,n$. Con tali operazioni $M_{m,n}(\mathbb{R})$ è uno spazio vettoriale su $\mathbb{R}$ di dimensione mn.

Più interessante è il prodotto righe per colonne che definiamo nel modo seguente.

Definizione D.5.2 (Prodotto righe per colonne) *Siano A una matrice di dimensione $m \times k$ e B una matrice di dimensione $k \times n$ (osservare che* il numero di

colonne di A è uguale al numero di righe di B). *Data una riga R_i di A e una colonna C_j di B definiamo il* prodotto di R_i per C_j *come*

$$R_i \cdot C_j = (a_{i1} \ \dots \ a_{ik}) \begin{pmatrix} b_{1j} \\ \vdots \\ b_{kj} \end{pmatrix} = \sum_{l=1}^{k} a_{il} b_{lj}.$$

Definiamo allora il prodotto righe per colonne come

$$AB = (c_{ij}) \ \ con \ \ c_{ij} = R_i \cdot C_j, \quad i = 1, \dots, m \ \ e \ \ j = 1, \dots, n.$$

Allora AB è una matrice di dimensione $m \times n$.

È chiaro che se si può fare il prodotto AB non è detto si possa fare il prodotto BA. Il caso più generale in cui si può fare sia AB che BA è quello di A matrice di dimensione $m \times n$ e B matrice di dimensione $n \times m$; in tal caso abbiamo che AB è di dimensione $m \times m$ e BA è di dimensione $n \times n$. In particolare, se A e B sono entrambe di dimensione $n \times n$, allora è certamente possibile fare AB e BA.

Osserviamo che il prodotto righe per colonne *non è commutativo*, ovvero, in generale, $AB \neq BA$; è addirittura possibile avere $AB = 0$ con $A, B \neq 0$.

Esempio D.5.3 *Siano* $A = \begin{pmatrix} 2 & 4 \\ 1 & 2 \end{pmatrix}$, $B = \begin{pmatrix} 1 & -2 \\ -1 & 2 \end{pmatrix}$. *Svolgendo il prodotto righe per colonne si ha che* $AB = \begin{pmatrix} -2 & 4 \\ -1 & 2 \end{pmatrix}$ *e* $BA = 0$.

Osserviamo inoltre che

$$AI = IA = A \quad \text{e} \quad A0 = 0A = 0 \quad \text{per ogni matrice } A$$

non appena le dimensioni di A, I e 0 permettano di fare i prodotti e che

$$\Delta_1 \Delta_2 = \Delta_2 \Delta_1 = \operatorname{diag}(a_1 b_1, \dots, a_n b_n)$$

se $\Delta_1 = \operatorname{diag}(a_1, \dots, a_n)$ e $\Delta_2 = \operatorname{diag}(b_1, \dots, b_n)$.

Diamo adesso la definizione di matrice *invertibile*.

Definizione D.5.4 *Una matrice A di dimensione $n \times n$ è invertibile se esiste B di dimensione $n \times n$ tale che $AB = BA = I$. In tal caso B è la matrice inversa di A e viene denotata con A^{-1}.*

Proposizione D.5.5 *Siano A e B matrici di dimensione $n \times n$ invertibili. Allora AB è invertibile e $(AB)^{-1} = B^{-1}A^{-1}$.*

Dim. Calcoliamo il prodotto di AB con $B^{-1}A^{-1}$. Abbiamo allora che il prodotto a destra diviene $(AB)(B^{-1}A^{-1}) = ABB^{-1}A^{-1} = AIA^{-1} = I$. Dobbiamo fare anche la verifica per il prodotto a sinistra ottenendo $(B^{-1}A^{-1})(AB) = B^{-1}(A^{-1}A)B = B^{-1}IB = I$. □

Osserviamo che una matrice invertibile non può avere righe o colonne *nulle*, come si può verificare facilmente.

Come ultima definizione diamo il concetto di matrice *trasposta*.

Definizione D.5.6 *Data una matrice $A = (a_{ij})$, $m \times n$, definiamo la matrice trasposta $A^T = (t_{ij})$, $n \times m$, come la matrice avente $(t_{ij}) = (a_{ji})$.*

In pratica A^T si ottiene da A scrivendo come righe di A^T le colonne di A e come colonne di A^T le righe di A. Non è difficile dimostrare che date le matrici A di dimensione $m \times k$ e B di dimensione $k \times n$ si ha $(AB)^T = B^T A^T$. Controlliamo solamente che le dimensioni di tali matrici siano coerenti:

$$\begin{gathered} AB \text{ è di dimensione } m \times n \quad \Rightarrow \quad (AB)^T \text{ è di dimensione } n \times m; \\ B^T \text{ è di dimensione } n \times k,\ A^T \text{ è di dimensione } k \times m \quad \Rightarrow \\ B^T A^T \text{ è di dimensione } n \times m. \end{gathered}$$

Altre informazioni sulle matrici e sulla loro struttura rispetto al prodotto possono essere trovate nel capitolo 3, §12 (pagine 55-60) e nel §14 (pagine 64-72) del libro di Lang [52].

D.5.2 Determinante di una matrice

Un concetto fondamentale nella teoria delle matrici è quello di *determinante*.

Definizione D.5.7 (Determinante) *Data una matrice quadrata A di dimensione $n \times n$, denotiamo con A_{ij} la sottomatrice di dimensione $(n-1) \times (n-1)$ di A ottenuta sopprimendo la riga i-esima R_i e la colonna j-esima C_j. Diamo la definizione di determinante di A nel modo seguente:*

- $n = 1$*: se* $A = (a)$ *allora* $\det A = a$*;*
- $n = 2$*: se* $A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ *allora* $\det A = ad - bc$*;*
- $n \geq 3$*:* $\det A = \sum_{j=1}^{n} (-1)^{j+1} a_{1j} \det A_{1j}$.

Tale definizione di determinante è *ricorsiva* in quanto il determinante di una matrice di dimensione $n \times n$ si esprime per mezzo del determinante di matrici di dimensione $(n-1) \times (n-1)$, e così via. Si verifica facilmente che

- $\det I = 1$;

– $\det\Delta = a_{11}\cdots a_{nn}$, dove con Δ indichiamo una matrice diagonale.

È chiaro che se la prima riga di A è nulla, allora $\det A = 0$. È anche chiaro che se A ha una riga nulla allora $\det A = 0$. Questo è ovvio se $n = 1,2$, mentre se $n \geq 3$ basta osservare che sviluppando $\det A$ per mezzo della definizione ricorsiva si perviene necessariamente ad un'espressione per $\det A$ in termini di determinanti di matrici aventi la prima riga nulla. Vale il seguente risultato, di immediata verifica nel caso di matrici diagonali.

Teorema D.5.8 *Siano A,B matrici di dimensione $n \times n$. Allora abbiamo che* $\det(AB) = \det A \det B$.

Abbiamo la seguente importante caratterizzazione delle matrici invertibili.

Teorema D.5.9 *Una matrice A di dimensione $n \times n$ è invertibile se e solo se* $\det A \neq 0$. *Inoltre, se A è invertibile, abbiamo che* $\det(A^{-1}) = (\det A)^{-1}$.

Nel caso in cui i coefficienti della matrice appartengano a $\mathbb{Z}_r$ si può dimostrare che A è matrice invertibile se e solo se $\det A \in \mathbb{Z}_r^*$, ossia se e solo se $(\det A, r) = 1$.

Estendendo la nozione di combinazione lineare di due righe, diciamo che una combinazione lineare di righe è un'espressione del tipo $\sum_{i=1}^{k} \lambda_i R_i$, $\lambda_i \in \mathbb{R}$. Un'analoga definizione vale per le colonne.

È possibile dimostrare le seguenti proprietà del determinante:

1) scambiando due righe il determinante cambia segno;
2) moltiplicando una riga per λ il determinante viene moltiplicato per λ;
3) sommando ad una riga R_i una combinazione lineare di righe diverse da R_i il determinante non cambia;
4) se una riga è combinazione lineare di altre righe il determinante è nullo.

Il teorema seguente fornisce utili formule per il calcolo del determinante.

Teorema D.5.10 (Laplace) *Sia A una matrice di dimensione $n \times n$. Allora*

1) per ogni $i = 1,\ldots,n$ si ha che $\det A = \sum_{j=1}^{n} (-1)^{i+j} a_{ij} \det A_{ij}$;

2) per ogni $j = 1,\ldots,n$ si ha che $\det A = \sum_{i=1}^{n} (-1)^{i+j} a_{ij} \det A_{ij}$.

La 1) prende il nome di *sviluppo secondo la riga i-esima* del determinante, mentre la 2) è lo *sviluppo secondo la colonna j-esima*. Tali sviluppi consentono di scegliere la riga o colonna più favorevole per il calcolo del determinante. Un fattore importante per tale scelta è il numero di coefficienti nulli in una riga o colonna; per esempio, si verifica facilmente che se A ha una colonna nulla allora $\det A = 0$.

Osservazione D.5.11 *Nel caso in cui si abbia da calcolare il* **determinante di una matrice** *di dimensione* $\mathbf{3 \times 3}$ *(e solo in questo caso) si può usare come alternativa la* Regola di Sarrus*:*

$$\det \begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} = (aei + bfg + cdh) - (ceg + afh + bdi).$$

Un metodo per ricordare tale regola è quello di osservare che, se si riscrivono le prime due colonne della matrice a fianco della terza, il determinante viene dato dalla somma dei prodotti degli elementi posti sulle tre diagonali principali a cui vengono sottratti i prodotti degli elementi posti sulle tre diagonali secondarie:

$$\begin{pmatrix} a & & b & & c & & a & & b \\ & \searrow & & \swarrow\!\!\searrow & & \swarrow\!\!\searrow & & \swarrow & \\ d & & e & & f & & d & & e \\ & \swarrow & & \swarrow\!\!\searrow & & \swarrow\!\!\searrow & & \searrow & \\ g & & h & & i & & g & & h \end{pmatrix} .$$

Come ultimo fatto riportiamo anche il seguente teorema che lega il valore del determinante di una matrice con il valore del determinante della matrice trasposta.

Teorema D.5.12 *Sia A una matrice quadrata; allora* $\det A^T = \det A$.

Per quanto riguarda una presentazione dei determinanti di matrici e delle loro proprietà, si faccia riferimento al capitolo 6, §§24-30 (pagine 117-148), del libro di Lang [52].

D.5.3 Matrici elementari

In riferimento a quanto abbiamo visto nel §2.3.1 riguardo ai crittosistemi a pacchetto, introduciamo le matrici elementari.

Definizione D.5.13 *Le matrici elementari di dimensione n sono le seguenti matrici quadrate:*

1) *matrice di scambio* E_{ij}*: è la matrice che si ottiene dalla matrice identica* I_n *mediante lo scambio della riga i-esima con la riga j-esima* ($R_i \leftrightarrow R_j$)*,* $1 \leq i, j \leq n$*;*
2) *matrice di moltiplicazione (per scalare)* $E_i(\lambda)$*,* $\lambda \neq 0$ *e* $\lambda \in \mathbb{R}$*: è la matrice che si ottiene da* I_n *mettendo al posto della riga i-esima la riga stessa moltiplicata per* λ ($R_i \leftarrow \lambda R_j$)*,* $1 \leq i \leq n$*;*

3) *matrice di combinazione lineare* $E_{ij}(\lambda)$, $i \neq j$ *e* $\lambda \in \mathbb{R}$*: è la matrice ottenuta da* I_n *mediante l'operazione* $R_i \leftarrow R_i + \lambda R_j$ *(ossia si sostituisce alla riga i-esima la combinazione lineare della stessa riga con la riga j-esima moltiplicata per lo scalare* λ*).*

Data $A \in M_{m,n}(\mathbb{R})$, è facile verificare che le matrici elementari agiscono nel modo seguente (indichiamo con C_i, R_i, rispettivamente, la colonna e la riga i-esima di A):

1) se E_{ij} è di dimensione $m \times m$ allora

$$E_{ij}A = \text{ matrice ottenuta da } A \text{ mediante } R_i \leftrightarrow R_j;$$

2) se E_{ij} è di dimensione $n \times n$ allora

$$AE_{ij} = \text{ matrice ottenuta da } A \text{ mediante } C_i \leftrightarrow C_j;$$

3) se $E_i(\lambda)$ è di dimensione $m \times m$ allora

$$E_i(\lambda)A = \text{ matrice ottenuta da } A \text{ mediante } R_i \leftarrow \lambda R_i;$$

4) se $E_i(\lambda)$ è di dimensione $n \times n$ allora

$$AE_i(\lambda) = \text{ matrice ottenuta da } A \text{ mediante } C_i \leftarrow \lambda C_i;$$

5) se $E_{ij}(\lambda)$ è di dimensione $m \times m$ allora

$$E_{ij}(\lambda)A = \text{ matrice ottenuta da } A \text{ mediante } R_i \leftarrow R_i + \lambda R_j;$$

6) se $E_{ij}(\lambda)$ è di dimensione $n \times n$ allora

$$AE_{ij}(\lambda) = \text{ matrice ottenuta da } A \text{ mediante } C_j \leftarrow C_j + \lambda C_i;$$

si noti lo scambio di indici rispetto al caso precedente.

Abbiamo quindi che la moltiplicazione a sinistra per una matrice elementare agisce sulle righe, mentre la moltiplicazione a destra agisce sulle colonne (la verifica è lasciata al lettore). Osservando che

1) $\det E_{ij} = -1$, se $(i \neq j)$;
2) $\det E_i(\lambda) = \lambda$;
3) $\det E_{ij}(\lambda) = 1$ $(i \neq j)$;

è evidente che le matrici elementari sono *invertibili* e le rispettive matrici inverse sono:

1) $E_{ij}^{-1} = E_{ij}$;
2) se $\lambda \neq 0$, $E_i(\lambda)^{-1} = E_i(\frac{1}{\lambda})$;
3) $E_{ij}(\lambda)^{-1} = E_{ij}(-\lambda)$.

Complementi E
Introduzione a PARI/GP

E.1 Introduzione

In questo capitolo daremo una breve introduzione ad uno strumento di calcolo utile a chi deve occuparsi di teoria dei numeri e quindi anche a chi vuole valutare alcuni algoritmi crittografici senza dover preventivamente costruire autonomamente gli oggetti necessari (interi di lunghezza arbitraria, funzioni aritmetiche di base, ecc.)

E.2 Generalità su PARI/GP

PARI/GP è una combinazione di una libreria di aritmetica estesa (e molto altro . . .) (`libpari`) e di un linguaggio di scripting (`gp`). È stato sviluppato da H. Cohen (Univ. Bordeaux 1) ed è attualmente mantenuto da K. Belabas (Univ. Bordeaux 1) con l'aiuto di alcuni volontari.

In breve alcune sue caratteristiche fondamentali sono:

1) è un software free su licenza GNU, General Public License;
2) è multipiattaforma (Mac OS X, MS Windows, Linux, UniX);
3) dispone di una shell (`gp`) utilizzabile come calcolatore "avanzato". In tale modalità è anche disponibile un linguaggio di scripting (`GP`) la cui sintassi è analoga a quella del C e risulta quindi di facile utilizzo per la maggioranza degli utenti;
4) il programmi scritti nel linguaggio di scripting possono essere automaticamente tradotti in C con il software `gp2c` in modo da aumentarne la velocità di esecuzione;
5) le notazioni sono di tipo "usuale" per chi si occupa di matematica;
6) è sviluppato e costantemente aggiornato da specialisti del settore.

E.3 Primi passi con PARI/GP

E.3.1 Documentazione

La documentazione di PARI/GP si trova sul sito web http://pari.math.u-bordeaux.fr e contiene:

1) **Installation Guide:** guida all'installazione di PARI/GP su un computer UniX;
2) **Tutorial:** una guida stringata alle principali caratteristiche di PARI/GP;
3) **User's Guide:** il manuale vero e proprio;
4) **Reference Card:** un elenco delle principali funzioni.

Consigliamo chi vuole avvicinarsi a PARI/GP di cominciare leggendo il tutorial e, se vuole, queste brevi note. Un'altra pratica referenza è lo schematico tutorial di Ash [8].

E.3.2 Primi passi

Dopo aver installato il software, per avviare la sessione di PARI/GP è sufficiente dare il comando `gp` al prompt del sistema operativo. La risposta che otterremo sarà analoga a quanto riportato qui sotto; il riconoscimento che PARI/GP è pronto ad accettare i comandi sarà la presenza di un prompt del tipo `gp >`. Le risposte di PARI/GP che cominciano con il simbolo `%n`, dove n sta per un numero, indicano il fatto che PARI/GP ha assegnato ad una variabile interna (identificata con % ed un numero progressivo) il valore indicato.

```
[riemann:~] languasc& gp
GP/PARI CALCULATOR Version 2.7.1 (released)
i386 running darwin (x86-64/GMP-5.1.2 kernel) 64-bit version
compiled: May 17 2014, gcc version 4.8.2 (GCC)
threading engine: single
(readline v6.2 enabled, extended help enabled)
Copyright (C) 2000-2014 The PARI Group
PARI/GP is free software, covered by the GNU General Public
License,  and comes  WITHOUT ANY WARRANTY WHATSOEVER.
Type ? for help, \q to quit.
Type ?12 for how to get moral (and possibly technical) support.
parisize = 8000000, primelimit = 500000
```

Osservazione E.3.1 *Si notino le dichiarazioni finali* `parisize = 8000000`, `primelimit = 500000`. *Esse indicano la quantità di memoria allocata (in*

byte) per il funzionamento di PARI/GP (`parisize`*) e che sono stati precalcolati tutti i numeri primi minori o uguali a* `primelimit`*.*

Per incrementare la quantità di memoria allocata si può eseguire Gp mediante il comando `gp -sNUMERO` *mentre, durante l'esecuzione di Gp, si può utilizzare il comando* `allocatemem` *che raddoppia la quantità di memoria preesistente.*

Per quanto riguarda `primelimit`*, esso può essere incrementato eseguendo Gp con il comando* `gp -pNUMERO`*. È importante ricordare che i primi calcolati in questa fase sono utilizzati da molte delle funzioni aritmetiche che vedremo in seguito (ed anche da altre:* `forprime`*, per esempio).*

Volendo fare qualche semplice esempio di utilizzo di PARI/GP notiamo che:

1) i commenti sono identificati da una doppia barra:

```
gp > \\ questo e' un commento
```

2) l'assegnazione di un valore ad una variabile avviene con il comando =

```
gp > x=2399393949
%1 = 2399393949
```

3) la stampa a video del valore di una variabile avviene usando il comando `print`:

```
gp > print(x)
2399393949
```

4) le operazioni di base hanno una sintassi analoga a quella usuale in matematica:

```
gp > x^3+77848848
%2 = 13813530083041663919441098197
gp > x*x
%3 = 5757091322497814601
gp > sqrt(x)
%4 = 48983.608983005733608881237291311990353
```

5) Abbiamo già usato la funzione (predefinita) `sqrt`. Altre funzioni predefinite che per noi saranno particolarmente importanti sono:
   (a) `factor(x)`: essa restituisce una matrice contenente la fattorizzazione completa del valore assunto dalla variabile `x` (nella prima colonna sono indicati i fattori primi p mentre nella seconda sono indicate le potenze α tali che $p^\alpha \parallel x$):


```
gp > factor(x)
%5 =
[3 1]
[193 1]
[4144031 1]
```

(b) `gcd(x,y)`: essa restituisce il valore del massimo comun divisore dei valori assunti dalle variabili `x` e `y`:

```
gp > gcd(x,6390809)
%11 = 193
```

6) in PARI/GP si ha accesso automatico ad una aritmetica intera a lunghezza arbitraria:

```
gp > x^100
%12 = 10236604580854796111054922719965911328088305080092010545849729312707202530054916325155314566497163742396846038235743464498265682423459321621679335231992877111693142793110061959289164984455088334070111401568120466629497211092294230465348653872418392493135399314695432024362566168086675228719473530277598914942769990271413462887220446688310035286604369736441315344873385600030821547372491384725211458557336245812157448383506112151580607425992894560905551739751241946859721981687258027593392585736124912622877120034263747960088955242309913265206309186598542440313477600485374856197993225066986199524654477882683937936607247635970824112034652460413043404148044422616789204595861226840694051333729395824553739823215305135635721488436762624392056143998551486163680671700095861232582730793985063823685166339881718630413431593845989513601153804904150875680848493573995998073569066093679350553567073035608381459180730814498128154834616323761173000 1
```

E.3.3 Help in PARI/GP

La sorgente principale di informazioni sull'utilizzo di PARI/GP è la sua Guida per l'Utente (User's Guide). Una versione più stringata, ma spesso sufficiente per comprendere il funzionamento di una particolare funzione, si può anche ottenere con l'help in linea. Per visualizzare l'help in linea è sufficiente battere il comando `?` per ottenere la lista sotto riportata.

```
gp > ?
```



```
Help topics: for a list of relevant subtopics, type ?n for n in
  0: user-defined functions (aliases, installed and user
   functions)
  1: Standard monadic or dyadic OPERATORS
  2: CONVERSIONS and similar elementary functions
  3: TRANSCENDENTAL functions
  4: NUMBER THEORETICAL functions
  5: Functions related to ELLIPTIC CURVES
  6: Functions related to general NUMBER FIELDS
  7: POLYNOMIALS and power series
  8: Vectors, matrices, LINEAR ALGEBRA and sets
  9: SUMS, products, integrals and similar functions
 10: GRAPHIC functions
 11: PROGRAMMING under GP
 12: The PARI community
Also:
  ? functionname (short on-line help)
  ?\    (keyboard shortcuts)
  ?.    (member functions)
Extended help (if available):
  ??   (opens the full user's manual in a dvi previewer)
  ??  tutorial / refcard / libpari (tutorial/reference card/
   libpari manual)
  ??  keyword    (long help text about "keyword" from the
   user's manual)
  ??? keyword    (a propos: list of related functions).
```

Nel caso in cui si vogliano ottenere informazioni sulle funzioni della teoria dei numeri implementate in PARI/GP va eseguito il comando `?4`. In tal caso si otterrà la seguente risposta.

```
gp > ?4
addprimes      bestappr       bestapprPade   bezout
bigomega       binomial       chinese        content
contfrac       contfracpnqn   core           coredisc
dirdiv         direuler       dirmul         divisors
eulerphi       factor         factorback     factorcantor
factorff       factorial      factorint      factormod
ffgen          ffinit         fflog          ffnbirred
fforder        ffprimroot     fibonacci      gcd
gcdext         hilbert        isfundamental  ispolygonal
ispower        ispowerful     isprime        isprimepower
```



```
ispseudoprime issquare      issquarefree  istotient
kronecker     lcm           logint        moebius
nextprime     numbpart      numdiv        omega
partitions    polrootsff    precprime     prime
primepi       primes        qfbclassno    qfbcompraw
qfbhclassno   qfbnucomp     qfbnupow      qfbpowraw
qfbprimeform  qfbred        qfbsolve      quadclassunit
quaddisc      quadgen       quadhilbert   quadpoly
quadray       quadregulator quadunit      randomprime
removeprimes  sigma         sqrtint       sqrtnint
stirling      sumdedekind   sumdigits     zncoppersmith
znlog         znorder       znprimroot    znstar
```

Per accedere alle informazioni relative ad una specifica funzione va inserito il comando `?NOMEFUNZIONE`. Per esempio, nel caso desiderassimo leggere la descrizione della funzione che realizza un test di primalità, sarebbe sufficiente scrivere `?isprime` ed otterremmo seguente risposta:

```
gp > ?isprime
isprime(x,{flag=0}): true(1) if x is a (proven) prime number,
false(0) if not. If flag is 0 or omitted, use a combination of
algorithms. If flag is 1, the primality is certified by the
Pocklington-Lehmer Test. If flag is 2, the primality is
certified using the APRCL test.
```

Si noti che `isprime` può usare differenti algoritmi per rispondere al nostro quesito. La scelta tra gli algoritmi disponibili viene fatta mediante una variabile di comodo (in gergo chiamata `flag`). Questo tipo di soluzione è abitualmente usata in PARI/GP.

Alcuni testi di pseudoprimalità vengono invece eseguiti dalla seguente funzione:

```
gp > ?ispseudoprime
ispseudoprime(x,{flag}): true(1) if x is a strong pseudoprime,
false(0) if not. If flag is 0 or omitted, use BPSW test,
otherwise use strong Rabin-Miller test for flag randomly
chosen bases.
```

Usualmente PARI/GP indica con lo stesso nome funzioni diverse o che hanno argomenti di tipo diverso. Per esempio il massimo comun divisore di interi e di polinomi viene calcolato con la stessa funzione `gcd`.

Nel caso non si sia soddisfatti delle stringate indicazioni dell'help in linea, si può ottenere la pagina del manuale corrispondente ad una determinata funzione.

Per esempio, per leggere la pagina del manuale corrispondente alla documentazione relativa alla funzione `isprime` è sufficiente scrivere `??isprime`.

```
gp > ??isprime
isprime(x,{flag = 0}):
true (1) if x is a (proven) prime number, false (0)  otherwise.
This can be very  slow  when  x is indeed prime and has
more than 1000 digits, say. Use ispseudoprime to quickly
check for pseudo primality. See also factor. If flag  =  0, use
a combination of Baillie-PSW pseudo-primality test (see
ispseudoprime), Selfridge "p-1" test if x-1 is smooth enough,
and Adleman-Pomerance-Rumely-Cohen-Lenstra (APRCL)
for general x. If  flag  =  1,  use  Selfridge-Pocklington-
Lehmer  "p-1"  test and output a primality  certificate as
follows: return 0 if x is composite,1 if x is small enough that
passing Baillie-PSW test guarantees its primality (currently
x < 10^{13}),  2 if x is a large prime whose primality could
only sensibly be proven (given the algorithms implemented
in PARI) using the APRCL test. Otherwise (x is  large and
x-1 is smooth) output a three column matrix as a primality
certificate. The first column contains the prime factors  p of
x-1, the second the  corresponding elements a_p as in
Proposition 8.3.1 in GTM 138, and the third the output of
isprime(p,1). The algorithm  fails if one of the pseudo-prime
factors is not prime,  which is exceedingly unlikely  (and
well worth a bug report)..... If flag = 2, use APRCL.
```

Nel caso in cui nel nostro sistema sia installato anche il software di composizione matematica TEX tale pagina verrà proposta direttamente in versione stampabile.

Per rimarcare nuovamente la differenza concettuale tra algoritmi di primalità e di fattorizzazione, riportiamo l'help in linea della funzione `factor` (di cui abbiamo precedentemente visto un esempio di uso):

```
gp > ?factor
factor(x,{lim}): factorization of x. lim is optional and  can
be  set whenever x is of (possibly recursive)  rational type.
If  lim is set return partial factorization, using primes up
to lim.
```

La pagina del manuale di PARI/GP è molto più dettagliata e ne riportiamo solo un estratto:


```
gp > ??factor
factor(x,{lim}): General factorization function, where x is a
rational (including integers), a complex number with rational
real and imaginary parts, or a rational function (including
polynomials). The result is a two-column matrix: the first
contains the irreducibles dividing x (rational or Gaussian
primes, irreducible polynomials), and the second the
exponents. By convention, 0 is factored as 0^1....
```

E.4 Programmare in PARI/GP: alcuni comandi di base

Diamo alcuni esempi di programmazione in PARI/GP esaminando alcuni problemi di base. Useremo qui la funzione `lift` che ha il compito di consentire di passare da un elemento dell'insieme $\mathbb{Z}_n$ ad uno di $\mathbb{Z}$ (in generale gli elementi di questi due insiemi sono di tipo diverso ed altrimenti non potrebbero essere confrontati).

```
gp > ?lift
lift(x,{v}): if v is omitted, lifts elements of Z/nZ to Z, of
Qp to Q, and of K[x]/(P) to K[x]. Otherwise lift only
polmods  with main variable v.
```

E.4.1 Leggere un file

Il comando `\r` consente di leggere il contenuto di un file.

Esempio E.4.1 *Creiamo un file `pm.gp` che contiene le seguenti linee (una funzione che ci consente di decidere se a è un residuo quadratico modulo p, si veda il teorema di Eulero* 11.7.10*):*

```
  {quadres(a, p) = return (lift(Mod(a,p)^((p-1)/2)))}
```

La funzione `quadres` va racchiusa in un costrutto $\{\ldots\}$ *per indicarne l'inizio e la fine. Per leggere questo file in PARI/GP è sufficiente usare `\r` :*

```
gp > ?quadres
 ***  quadres: unknown identifier.
gp > \rpm        \\ anche  \rpm.gp va bene
gp > ?quadres
quadres(a, p) = return (lift(Mod(a,p)^((p-1)/2)));
gp > quadres(2,101)
%1 = -1       \\ 2 non e' residuo quadratico modulo 101
```


Nel caso il file `pm.gp` *venga modificato è sufficiente scrivere* `\r` *per ricaricarlo (nel caso si ometta il nome del file viene ricaricato l'ultimo file in ordine di tempo). Per esempio modificando*

```
return (lift(Mod(a,p)^((p-1)/2)));
```

in `pm.gp` *con*

```
return (lift(Mod(a,p)^((p-1)/2))-p);
```

otteniamo una funzione che non ha particolare significato teorico-numerico, ma esemplifica l'uso di `\r`*:*

```
gp > \r
gp > quadres(2,101)
%2 = -102 .
```

E.4.2 Argomenti di funzioni e loro passaggio

Le funzioni di PARI/GP possono avere vari argomenti. Per esempio, la funzione

```
gp > {add(a, b, c, d)= return (a + b + c + d)}
gp > add(1,2,3,4)
%3 = 10
```

ha quattro argomenti. Se, nella chiamata della funzione, alcuni di essi vengono omessi, sono automaticamente posti uguali a 0.

```
gp > add(1,2)
%4 = 3
```

Se si vuole cambiare il valore di default di qualche variabile, è sufficiente includere tale informazione nella dichiarazione della funzione.

```
{add(a, b=-1, c=2, d=1)= return (a + b + c + d)}
```

In tal modo il valore di default per b è -1, per c è 2 e per d è 1. Tali valori verranno assegnati alle variabili b, c, d nel caso in cui nella chiamata della funzione `add` non sia specificato il valore dell'argomento corrispondente.

```
gp > add(1,2)
%6 = 6
gp > add(1)
%7 = 3
gp > add(1,2,3)
%8 = 7
```


E.4.3 Variabili locali

Nel linguaggio di scripting di PARI/GP è, come abbiamo visto, possibile scrivere dei piccoli programmi in appositi file e richiamarli al momento del bisogno. Un'importante particolarità di tale linguaggio è che tutte le variabili, a meno che non sia specificato il contrario, vengono considerate globali. Ciò può fare incorrere in errori. Vediamo quindi come dichiarare variabili locali in PARI/GP.

Esempio E.4.2 *La funzione* `errata` *somma gli interi* $1, 2, \ldots, n$ *facendo un uso errato della variabile* `i`.

```
gp > {errata(n)=
  i=0;
  for(j=1,n, i=i+j);
  return(i)}
gp > errata(3)
%9 = 6
gp > i=4;
gp > errata(3);
gp > i
%10= 6          \\ ecco l'errore !!
```

Per usare correttamente le variabili si usa il comando `local`*: i è locale e dovrà essere dichiarata come tale.*

```
gp > {corretta(n)=
  local(i);
  i=0; for(j=1,n, i=i+j);
  return(i)}
gp > i=4;
gp > corretta(3)
%11 = 6
gp > i
%12 = 4
```

E.4.4 Come inserire i dati

Supponiamo, come spesso accade in realtà, di voler richiedere all'utente dei dati durante l'esecuzione di una funzione. In tale caso va usato il comando `input` che legge una espressione PARI/GP dalla tastiera o da un file. L'espressione viene valutata ed il risultato viene ritornato al programma. Si faccia attenzione al fatto che non necessariamente `input` deve ritornare una stringa ma il tipo di dato dipende

da come inseriamo il dato stesso. Per conoscere quale sia il tipo a cui appartiene un certo dato si utilizza il comando `type`. Ecco alcuni esempi:

```
gp > ?input
input(): read an expression from the input file or standard
input.
gp > s = input();
2+2
gp > s
%13 = 4   \\ attenzione: s non e‘ la stringa "2+2"
gp > s=input()
variabile
%14 = variabile
gp > type(s)
%15 = "t_POL" \\ PARI intende s come un polinomio
           \\ nella variabile ’’variabile’’
gp > s=input()
"ciao"
%16 = "ciao"
gp > type(s)
%17 = "t_STR"   \\ questa e‘ una stringa di caratteri
```

E.4.5 Scrivere in un file

Nel caso, non infrequente nella pratica, in cui il risultato di una certa funzione sia troppo lungo (o complesso) per essere visualizzato sullo schermo di un computer, si può salvare l'output di un programma PARI/GP in un file. A tale scopo si usa il comando `write`:

```
gp >  ?write
write(filename,{str}*): appends the remaining arguments (same
output as print) to filename.
gp > write("provafile", "Qui Quo e Qua")
```

Il comando `write` appende la linea "Qui Quo e Qua" al fondo del file `provafile` (se il file non esiste, viene creato).

I dati prodotti in una sessione di lavoro sono anche disponibili nel file di log `pari.log` che si attiva con il comando `\l` e contiene la trascrizione integrale della sessione di lavoro.

```
gp > \l
  log = 1 (on)
```



```
gp > 2+2
%29 = 4
gp > \l
  log = 0 (off)
  [logfile was "pari.log"]
```

E.5 Alcune famose asserzioni sui primi

L'utilizzo del calcolatore in teoria dei numeri consente di poter convincersi o meno della validità di una congettura e, in alcuni casi, di aiutare a risolvere uno specifico problema. Vediamo quattro famose asserzioni sui primi per cui il computer può facilmente "convincerci" della loro validità. Si noti che questi problemi sono molto noti per la loro difficoltà e, sebbene i loro enunciati non coinvolgano che concetti elementari, allo stato dell'arte non sono note né dimostrazioni né confutazioni. Usiamo questi famosi enunciati per esemplificare come è possibile utilizzare il linguaggio di scripting di PARI/GP per costruire piccoli programmi o funzioni.

– *Il polinomio x^2+1 assume infiniti valori primi.*
Poniamo

$$primepol(n) = \text{card}(\{x : x < n \text{ e } x^2+1 \text{ è primo}\}).$$

Con l'ausilio di un computer è facile ottenere che

$$primepol(10^2) = 19,\ primepol(10^3) = 112,$$

$$primepol(10^4) = 841,\ primepol(10^5) = 6656.$$

Pare proprio che la funzione in questione non sia limitata. Il codice in PARI/GP utile a calcolare `primepol(n)` è molto facile: è sufficiente usare un ciclo `for`. Sfruttiamo questa occasione per riportare la descrizione di tale costrutto:

```
gp > ??for
for(X = a,b,seq):
 the formal variable X going from a to b, the seq is eva-
 luated. Nothing is done if a > b. a and b must be in R.
```

Osserviamo inoltre che useremo il costrutto ++ (tipico del linguaggio di programmazione C) per realizzare l'operazione $counter \leftarrow counter + 1$. Allora possiamo realizzare la funzione `primepol(n)` nel seguente modo:

```
gp > { primepol(n) = local(s) ; s=0;
       for(x=1,n,if(isprime(x^2+1),s++)); return(s) }
```


Per valutare *primepol* in 100 è sufficiente, come si vede nel seguito, scrivere *primepol*(100).

```
gp > primepol(100)
%1 = 19
gp > primepol(1000)
%2 = 112
gp > primepol(10000)
%3 = 841
gp > primepol(10^5)
%4 = 6656
```

– *Ogni intero pari $n > 2$ è somma di due numeri primi (congettura di Goldbach, 1742).*
La congettura di Goldbach è uno dei problemi di teoria additiva dei numeri che sfida da più tempo i matematici. Allo stato attuale non ne sono note né dimostrazioni né confutazioni e, sebbene i vari tentativi di giungere ad una dimostrazione abbiano utilizzato anche strumenti piuttosto sofisticati di analisi complessa, al momento si sa provare solamente che ogni intero pari sufficientemente grande è somma di un primo e di un intero avente al più due divisori primi (teorema di Chen, 1966).
Dal punto di vista numerico, la congettura di Goldbach è stata verificata per tutti gli interi minori o uguali di $4 \cdot 10^{18}$ (si veda [65]). Non avendo a disposizione una grande potenza di calcolo, ci accontentiamo di usare PARI/GP per verificare la congettura per un intero dato. In questo caso usiamo la funzione `forprime` per realizzare il ciclo sui numeri primi minori di $n-2$ e per la verifica della primalità usiamo, come per il caso precedente, la funzione `isprime`. La descrizione di `forprime` è:

```
gp > ??forprime
forprime(X = a,b,seq):
  the formal variable X ranging over the prime numbers
  between a to b (including a and b if they are prime),
  the seq is evaluated. More precisely, the value  of  X
  is incremented to the smallest prime strictly larger
  than X at the end of each iteration. Nothing is done
  if a > b.  Note that a and b must be in R.
```

Un possibile programma PARI/GP per questo problema è quindi il seguente:

```
gp > goldbach(n) =
forprime(p=2,n,if(isprime(n-p),return([p,n-p])));
gp > goldbach(4)
```



```
%7 = [2, 2]
gp > goldbach(6)
%8 = [3, 3]
gp > goldbach(100)
%9 = [3, 97]
gp > goldbach(1000)
%10 = [3, 997]
gp > goldbach(570)
%11 = [7, 563]
```

Un altro problema interessante collegato ai numeri di Goldbach è quello di determinare in quanti modi un intero pari può essere scritto come somma di due primi, ossia calcolare

$$R(n) = \sum_{\substack{p_1 \le n \\ p_2 \le n \\ p_1 + p_2 = n}} 1.$$

Definite le funzioni:

```
{rep(n) = local(s);
    s=0;
    forprime(p=2, n, if(isprime(n-p), s++));
    return(s)}

{gold(n) = local(s);
    s = 0;
    for(p=2, n, if(isprime(p) && isprime(n-p), s++));
    return(s)}
```

possiamo provare a calcolare quante sono le rappresentazioni come somma di due numeri primi di alcuni interi pari. Si faccia attenzione al fatto che questo è un problema computazionalmente piuttosto oneroso perché si suppone che sia valida la seguente forma forte della congettura di Goldbach:

$$R(n) \sim \mathfrak{S}(n) \frac{n}{(\log n)^2} \quad \text{per } n \to +\infty.$$

Nella formula precedente abbiamo indicato con $\mathfrak{S}(n)$ la *serie singolare* del problema di Goldbach definita mediante il seguente prodotto sui primi:

$$\mathfrak{S}(n) = 2 \prod_{p>2} \left(1 - \frac{1}{(p-1)^2}\right) \prod_{\substack{p|n \\ p>2}} \left(\frac{p-1}{p-2}\right).$$

Abbiamo i seguenti risultati:


```
gp > for(n=1,9,print("n= ",n*10^5," R(n)= ",gold(n*10^5)))
n= 100000  R(n)= 1620
n= 200000  R(n)= 2834
n= 300000  R(n)= 7830
n= 400000  R(n)= 4974
n= 500000  R(n)= 6104
n= 600000  R(n)= 13986
n= 700000  R(n)= 9756
n= 800000  R(n)= 8866
n= 900000  R(n)= 19706
time =  7,224 ms.
```

La funzione `rep` è più veloce perché si basa su un insieme di primi precalcolati:

```
gp > for(n=1,9,print("n= ",n*10^5," R(n)= ",rep(n*10^5)))
n= 100000 R(n)= 1620
n= 200000 R(n)= 2834
n= 300000 R(n)= 7830
n= 400000 R(n)= 4974
n= 500000 R(n)= 6104
n= 600000 R(n)= 13986
n= 700000 R(n)= 9756
n= 800000 R(n)= 8866
n= 900000 R(n)= 19706
time =  1,532 ms.
```

Come si può notare immediatamente la distribuzione di $R(n)$ è piuttosto irregolare. Ciò è dovuto alla presenza di $\mathfrak{S}(n)$. Uno studio accurato del comportamento della serie singolare ci farebbe deviare troppo dallo scopo di questo testo e quindi non investighiamo ulteriormente questo problema.

– *Esistono infiniti numeri primi p tali che $p+2$ è primo anch'esso (congettura dei primi gemelli).*
La congettura dei primi gemelli è una importante congettura sulla distribuzione dei numeri primi ed è anch'essa né provata né confutata sebbene nel 2013 Y. Zhang [103] abbia dimostrato che esistono infiniti numeri primi consecutivi aventi distanza finita.
Poniamo $twins(n) = \text{card}(\{p : p \leq n \text{ e } p+2 \text{ è primo}\})$. Per mezzo di un computer è facile ottenere

$$twins(10^2) = 8,\ twins(10^3) = 35,\ twins(10^4) = 205,\ twins(10^5) = 1224.$$

Analogamente a quanto visto sopra, un programma PARI/GP per calcolare `twins(n)` è il seguente.


```
gp > {twins(n) = local(s); s=0;
    forprime(p=2,n,if(isprime(p+2),s++)); return(s)}
gp > twins(10^2)
%12 = 8
gp > twins(10^3)
%13 = 35
gp > twins(10^4)
%14 = 205
gp > twins(10^5)
%15 = 1224
```

– *La congettura di Hardy e Littlewood (1922)*
Un'altra importante congettura che riguarda la distribuzione dei numeri primi è stata formulata da Hardy e Littlewood nel 1922. Anche se non è altrettanto famosa delle precedenti ne parliamo qui per illustrare alcune caratteristiche importanti di PARI/GP, ed in particolare la possibilità di definire funzioni a partire da altre funzioni (che siano predefinite o definite dall'utente). Per semplicità, ci limiteremo alla formulazione originale della congettura di Hardy e Littlewood, anche se è possibile darne una versione più generale. Fissiamo dunque $k = 2$ oppure $k = 3$. Ci si chiede se sia vero che ogni numero intero sufficientemente grande (cioè da un certo punto in poi) sia una potenza k-esima perfetta oppure possa essere scritto come la somma di un numero primo e di una potenza k-esima.
Il nostro obiettivo è la definizione di una funzione `sum_p_and_pow(n,k)` che restituisca 1 se n è somma di un numero primo e di una k-esima potenza, e -1 altrimenti. L'idea, quindi, è simile a quella impiegata sopra: dati n e k si ripete un ciclo nel quale si calcola $n - i^k$ e si chiede se questa quantità sia un numero primo. In caso affermativo si ritorna il valore 1, mentre se si esce dal ciclo (cioè se $n - i^k$ *non è* un numero primo per nessuno dei valori ammissibili di i) si controlla se n sia una k-esima potenza perfetta; se lo è si ritorna il valore 1, altrimenti -1, per indicare che n è "eccezionale".
Per comodità, definiamo la funzione `root(n,k)` che calcola la radice k-esima intera dell'intero n ed anche la funzione `HL(x,k)` che conta i numeri interi $\leq x$ che soddisfano la condizione di Hardy e Littlewood, mentre `except(x,k)` elenca le eccezioni alla congettura fino ad x. Abbiamo utilizzato la funzione predefinita `floor(x)` per calcolare $\lfloor x \rfloor$, ossia la parte intera di x, e `--` per realizzare l'operazione $counter \leftarrow counter - 1$.

```
root(n,k) = floor(n^(1/k))
{sum_p_and_pow(n,k) = local(s);
   s = root(n,k);
   for(i = 0, s, if(isprime(n - i^k), return(1)));
```



```
   if(n == s^k, return(1));
   return(-1)}

{HL(x,k) = local(t);
   t = 0;
   for(n = 1, x, if(sum_p_and_pow(n,k) > 0, t++));
   return(t)}

{except(x,k) =
   for(n = 1, x, if(sum_p_and_pow(n,k) < 0, print(n)))}
```

A causa di problemi di arrotondamento, in effetti è preferibile definire la funzione `root` in un modo diverso, e precisamente:

```
{root(n, k) = local(s);
   s = round(0.5 + n^(1/k));
   if(s^k > n, s--);
   return(s)}
```

Infine osserviamo che la soluzione più efficiente dal punto di vista computazionale (anche nel caso del problema di Goldbach) utilizza una variante del crivello di Eratostene.
Così come per la congettura di Goldbach discussa qui sopra, anche in questo caso si pensa (ma nessuno è ancora riuscito a dimostrare) che per n grande, diverso da un quadrato o da un cubo perfetto, il numero di soluzioni dell'equazione $n = p + m^k$ sia piuttosto grande. Per esercizio, vediamo come modificare le funzioni date prima per ottenere anche questo dato: la funzione `sum_p_and_pow(n,k)` ha lo scopo di determinare se n ha almeno una rappresentazione nella forma desiderata, e il ciclo si arresta appena ne viene trovata una. Ora, invece, vogliamo determinarle tutte. Definiamo dunque

```
{HL_rep(n,k) = local(s, t);
   s = root(n,k);
   t = 0;
   for(i = 0, s, if(isprime(n - i^k), t++));
   return(t)}
```

Per esempio, `HL_rep(100001,2)=37`. Le considerazioni fatte a proposito dell'irregolarità della funzione $R(n)$ per il problema di Goldbach sono valide anche in questo caso.

E.6 Esempi in PARI/GP

Elenchiamo alcuni altri esempi di calcoli teorico-numerici in PARI/GP.

E.6.1 Distribuzione dei numeri primi

Rimanendo su problemi classici della teoria dei numeri, proviamo ad utilizzare PARI/GP per fare qualche calcolo sulla distribuzione dei numeri primi negli interi e nelle progressioni aritmetiche.

Una possibile domanda è

Domanda. *Quanti sono i numeri primi $\leq x$?*

Sia

$$\pi(x) = \text{card}(\{p \text{ primo tale che } p \leq x\}),$$

Sappiamo grazie al teorema B.2.1 che $\pi(x) \sim \text{li}(x)$ per $x \to +\infty$. Vogliamo calcolare effettivamente quanti sono i numeri primi minori o uguali di x nel caso in cui x si abbastanza piccolo. Con PARI/GP è sufficiente contare le iterazioni di un ciclo `forprime`.

```
gp > pi(x, c=0) = forprime(p=2,x,c++); c;
```

Inoltre, per calcolare alcuni valori di $\pi(x)$ ed alcune sue approssimazioni, si può, per esempio, costruire un ciclo `for` come segue. Visto che vogliamo lavorare con primi un po' più grandi di quanto sono usualmente predefiniti, utilizziamo una nuova sessione di PARI/GP.

```
$gp -p1000000
gp > pi(x, c=0) = forprime(p=2,x,c++); c;
gp >  for(n=1,9,print("n= ",n*10^5,"  pi(n)= ", pi(n*10^5),
"  n/(log(n)-1)= ",floor((n*10^5)/(log(n*10^5)-1))))

n= 100000  pi(n)= 9592  n/(log(n)-1)= 9512
n= 200000  pi(n)= 17984  n/(log(n)-1)= 17847
n= 300000  pi(n)= 25997  n/(log(n)-1)= 25836
n= 400000  pi(n)= 33860  n/(log(n)-1)= 33615
n= 500000  pi(n)= 41538  n/(log(n)-1)= 41246
n= 600000  pi(n)= 49098  n/(log(n)-1)= 48761
n= 700000  pi(n)= 56543  n/(log(n)-1)= 56185
n= 800000  pi(n)= 63951  n/(log(n)-1)= 63530
n= 900000  pi(n)= 71274  n/(log(n)-1)= 70809
```

Seguendo tale idea scriviamo un semplice programma PARI/GP che conta il numero di primi della forma $4x-1$ in cui x è minore o uguale ad un parametro n.


```
arprogprimes(n, s=0) = for(x=1, n, if(isprime(4*x-1), s++)); s
```

Nei valori elencati precedentemente (corrispondenti al caso $n = 12$) sono presenti otto numeri primi ed infatti si ha

```
gp > arprogprime(12)
%16 = 8
```

Vediamo ora alcune verifiche numeriche di teoremi presentati nel testo cartaceo.

– Verifica del teorema di Wilson 11.1.4. Definiamo una funzione che contenga la congruenza presente nella tesi del teorema 11.1.4. L'operatore di confronto == restituisce 1 (vero) nel caso due quantità siano uguali e 0 (falso) nel caso siano diverse.

```
gp > wilson(n) = Mod((n-1)!,n) == Mod(-1,n)
gp > wilson(5)
%9 = 1  \\ vero
gp > wilson(10)
%10 = 0  \\ falso
gp > wilson(389)
%11 = 1
gp > wilson(2001)
%12 = 0
```

– Uso del teorema cinese del resto 10.1.3. La funzione `chinese` è predefinita in PARI/GP. La sua descrizione ed il suo uso sono i seguenti:

```
gp > chinese(x,{y}): x,y being both intmods (or  polmods)
computes z in the same residue classes as x and y.
gp > chinese(Mod(2,3),Mod(3,5))
%13 = Mod(8, 15)
gp > chinese(Mod(8,15),Mod(2,7))
%14 = Mod(23, 105)
```

Se vogliamo risolvere un caso del problema dei generali cinesi dobbiamo "annidare" due volte la funzione `chinese` perché essa ammette solo due argomenti. È chiaramente anche possibile scrivere un programmino PARI/GP che utilizzi la funzione predefinita `chinese` e la estenda ad un sistema costituito da un numero arbitrario di congruenze lineari:

```
gp > chinese(chinese(Mod(2,7),Mod(4,11)),Mod(11,13))
%3 = Mod(37, 1001)
```


– Calcolo della funzione φ di Eulero definita in 10.2.4. La funzione `eulerphi` è predefinita in PARI/GP. La sua descrizione ed il suo uso sono i seguenti:

```
gp > ??eulerphi
eulerphi(x):
Euler's phi (totient) function of |x|, in other words
 |(Z/xZ)^*|.
gp >  eulerphi(2689*3^2*11^3)
%7 = 19514880
```

– Uso dell'algoritmo di Euclide 10.2. Come abbiamo visto nei capitoli precedenti, l'algoritmo di Euclide fornisce anche una relazione intera tra il massimo comun divisore di due interi e gli interi stessi. In letteratura tale relazione viene anche detta *formula di Bézout*. Essa si ottiene mediante l'algoritmo euclideo esteso e, per tale ragione, la funzione PARI/GP che realizza tale calcolo viene chiamata `gcdext`.

```
gp > ?gcdext
gcdext(x,y): returns [u,v,d] such that d=gcd(x,y) and
u*x+v*y=d.
gp > gcdext(1235,332)
%8 = [-25, 93, 1]
```

– Per realizzare l'esponenziazione modulare si può usare la funzione `Mod` combinata con l'operatore `^`. Vediamo alcuni esempi di esponenziazione modulare in PARI/GP.

```
gp > ?Mod
Mod(a,b): creates 'a modulo b'.
gp > Mod(11,2689)^2212
%10 = Mod(1493, 2689)
gp > Mod(1122348,3388827279)^223312
%12 = Mod(3029574879, 3388827279)
```

– La fattorizzazione di polinomi può essere calcolata mediante un'altra utile funzione di PARI/GP: `factormod(x,q)`. Questa funzione consente di fattorizzare il polinomio `x` in $\mathbb{Z}_q$ e, per esempio, può essere usata per verificare quanto detto nel capitolo 12 sull'irriducibilità su $\mathbb{F}_2$ del polinomio $x^8+x^4+x^3+x+1$.

```
gp > ??factormod
factormod(x,p,{flag=0}): factors the polynomial
x modulo the prime p, using Berlekamp. flag is optional,
```



```
and can be 0: default or 1: only
the degrees of the irreducible factors are given.

gp > factormod(x^4+x^2+1,2)
%5 = [Mod(1, 2)*x^2 + Mod(1, 2)*x + Mod(1, 2) 2]
      \\ x^4+x^2+1= (x^2+x+1)^2 su Z/2Z
gp > factormod(x^8 + x^4 + x^3 + x + 1,2)
%2 = [Mod(1, 2)*x^8 + Mod(1, 2)*x^4 +
      Mod(1, 2)*x^3 + Mod(1, 2)*x + Mod(1, 2) 1]
      \\ il polinomio x^8+x^4+x^3+x+1 e' irriducibile
      \\ su Z/2Z
```

E.6.2 Calcoli riguardanti RSA in PARI/GP

Vediamo in PARI/GP un esempio numerico dei calcoli riguardanti RSA la cui descrizione è data nel §3.3. Useremo l'operatore di negazione `!` applicato a `=` (che restituisce 0 se due quantità sono diverse e 1 se sono uguali), un ciclo `while`

```
gp > ??while
while(a,seq): while a is nonzero evaluate the expression
sequence seq. The test is made before evaluating the
seq, hence in particular if a is initially equal to zero the
seq will not be  evaluated at all
```

e le seguenti due funzioni.

```
gp >?nextprime
nextprime(x): smallest pseudoprime >= x.

gp > ?random
random({N=2^31}): random object, depending on the type of N...
```

– Generazione di p e q.

```
p=nextprime(random(10^50))
%2 = 22005109792022134749492304074618983912551252046861
gp > isprime(%2)
%3 = 1
gp > q=nextprime(random(10^60))
%5 = 46115964053880060094485305924606274945238607476053
53656267853
gp > isprime(%5)
%6 = 1
```


– Calcolo di n.

```
gp > n=p*q
%7 = 10147868521705768912906357852166732520259672369610
974 69715018803977036992478064186314786954326020738832 3
859433
```

– Calcolo di $\varphi(n)$.

```
gp > phin=(p-1)*(q-1)
%8 = 10147868521705768912906357852166732520259672369610
513537509627234059633049586664630809434253820153428341 5
544720
```

– Calcolo di e.

```
gp >  e=random(n)
%9 = 732201225334401894703078938632653293576002219538603
748236461978823051711778253268198635241742987529750559 66
96
gp > while(gcd(e,phin)!=1,e=e+1)
gp > e
%10 = 732201225334401894703078938632653293576002219538 60
374823646197882305171177825326819863524174298752975055 9
6697
```

– Calcolo di $d \equiv e^{-1} \bmod \varphi(n)$.

```
gp > d = lift(Mod(e,phin)^(-1));
gp > (e*d)%phin
%9 = 1
gp > d
%13 = 76764121908683995844052792587466690021105376758 93
917356633460599617954173796955225752040474866426557585
37193
```

– Calcolo della lunghezza massima del blocco di testo codificabile.

```
gp > log(n)/log(30)
%11 = 73.796497398886431714670067961393849992
```

Usando l'alfabeto di 30 caratteri dato dalla figura 1.2 possiamo quindi codificare in un blocco unico un testo avente meno di 73 caratteri. Codifichiamo "PADOVA":

– Calcolo dell'equivalente numerico di PADOVA;

```
gp > m=15*30^5+0*30^4+3*30^3+14*30^2+21*30^1+0*30^0
%12 = 364594230
```

– Definizione della funzione di cifratura.

```
gp > E(x)=lift(Mod(x,n)^e)
```

– Definizione della funzione di decifratura.

```
gp > D(x)=lift(Mod(x,n)^d)
```

– Codifica di PADOVA.

```
gp > messaggio_segreto = E(m)
%18 = 843919672864917640604980536647298759645143299704 48
784756924600474099016421406142932701141490671708700 27
254030
```

– Decodifica.

```
gp > D(messaggio_segreto)
%14 = 364594230
```

La procedura è stata sviluppata correttamente poiché l'equivalente numerico della decodifica è uguale a quello di PADOVA.

Il seguente programma automatizza il procedimento sopra descritto. Siccome non siamo in grado di sapere a priori quanto è lungo il messaggio, nel caso in cui esso sia più corto del massimo blocco di testo codificabile, concateniamo al fondo del messaggio una serie di spazi bianchi.

```
{alfabeto=["A","B","C","D","E","F","G","H","I","J","K",
  "L","M","N","O","P","Q","R","S","T","U","V","W","X","Y","Z",
   ";",".","'"," "];
}
{da_lettera_a_numero(lettera)=local(j);
     for(j=1,30,if(alfabeto[j]==lettera,return(j-1)));
     error("input non valido.")
}
{da_numero_a_messaggio(num,s="")= local(m,i);
     i=floor(log(n)/log(30))-1;
     while(i>0, s = concat(s,alfabeto[num\(30^i)+1]);
             num = num % (30)^i;i--);
             \\ n \ 30 e' la divisione intera di num per 30
```



```
     s = concat(s,alfabeto[num+1]);
     return(s)
}
{da_messaggio_a_numero(w,mod,num=0)= local(l,i,j);
     l=length(w);
     i=floor(log(mod)/log(30));
     M=Vecsmall(w);
          for(j=1,l, num = num +
             30^(i-j)*da_lettera_a_numero(Strchr(M[j])));
     if(i>l, for(j=l+1,i,num = num +
             30^(i-j)*da_lettera_a_numero(" ")));
         return(num);
}
{genera_chiave_rsa(len,p,q,n,e,d)=
 p=2;q=2;
     until (((isprime(p) && isprime(q))),
               p = nextprime(random(10^(len\2+1)));
               q = nextprime(random(10^(len\2+3)));
     );
     n = p*q; phin = (p-1)*(q-1);
     e = random(phin);
     while(gcd(e,phin)!=1,e=e+1);
     d = lift(Mod(e,phin)^(-1));
     blocco=floor(log(n)/log(30));
     print("La lunghezza massima del blocco codificabile e': ",
               blocco," caratteri");
     return([n,e,d]);
}
{rsa_cifra(messaggio, n, e) = local(l,i,j);
     l=length(messaggio);
     i=floor(log(n)/log(30));
     if (l>i, print("Errore: il messaggio e' di ", l,
          " caratteri ma non deve essere piu' di ", i,
                      " caratteri");
          return);
     lift(Mod(da_messaggio_a_numero(messaggio,n),n)^e);
}
{rsa_decifra(segreto, n, d)  =
             da_numero_a_messaggio(lift(Mod(segreto,n)^d));
}
```


Abbiamo quindi definito una serie di programmi e funzioni utili per i calcoli necessari in RSA. Vediamo un esempio di uso di `rsa.gp`.

```
gp > \r rsa
gp > setrand(1) \\ resetta il generatore di numeri casuali
%16 = 1
gp > rsa=genera_chiave_rsa(200)
\\ ritorna [n, e, d], 200 e' il numero di cifre richieste
per n

La lunghezza massima del blocco codificabile e': 137 caratteri
%99 = [635559057449398771439982117009717036351708300505 9
775897628703850555062412432868324414420436369435835343
756735897936754143118096452830820331122689466211435706 0
452245883770311935644336122028491810454286661,

428427147238060736405548986161631227996928483557784263 5
111059898255305928661292291448073056122973580422724152 6
803279243618682296768093840454253385602650754096031162 1
377509854724643306509358084717038988821,

270328031808773797759775221627596259210722368958053829 3
090842135633385953640043028566449700370983017834590994 1
807185185952045344477957742081456602759979036088691378 8
531522411410502205654299829152942410557]

gp >  n = rsa[1]; e = rsa[2]; d = rsa[3];

gp > chiave_pubblica = [n,e]
%101 = [635559057449398771439982117009717036351708300505 9
775897628703850555062412432868324414420436369435835343
756735897936754143118096452830820331122689466211435706 0
452245883770311935644336122028491810454286661,

428427147238060736405548986161631227996928483557784263 5
111059898255305928661292291448073056122973580422724152 6
803279243618682296768093840454253385602650754096031162 1
377509854724643306509358084717038988821]

gp > msg = "PADOVA"
%37 = "PADOVA"
```



```
gp > codifica = rsa_cifra(msg,n,e)
%104 = 3000061220448092896528151289744873968253801825 77
31678629530012217141226239170664133538801074543534934 09
61198074710202056307412386844051573263449382065084743 98
35393493714052553502967802931428424391131234 5

gp >  rsa_decifra(codifica,n,d)
%32 = "PADOVA              "

gp > msg="PROGETTO LAUREE SCIENTIFICHE"
%34 = "PROGETTO LAUREE SCIENTIFICHE"

gp > codifica=rsa_cifra(msg,n,e)
%107 = 10143994644493586434706695344087543382483313413
79930622072355094678250527268616664138068150504541737 9
74161532067038630815264000591434934776784686587947241 1
31618653157601158447476403893347768024189157501 48

gp > rsa_decifra(codifica,n,d)
%36 = "PROGETTO LAUREE SCIENTIFICHE                    "
```

E.6.3 Attacco di Wiener

Non è difficile scrivere uno script PARI/GP che svolga le verifiche richieste dall'Esempio 3.3.8. Si consiglia di usare i comandi `contfrac` per generare la frazione continua e `contfracpnqn` per calcolare i convergenti della frazione continua data.

E.6.4 Altri esempi in PARI/GP

Presentiamo qui alcuni script PARI/GP riguardanti alcuni algoritmi discussi nei capitoli precedenti. Useremo il costrutto `/*...*/` per inserire commenti lunghi.

– algoritmo di primalità di Agrawal, Kayal & Saxena 5.2.2. Lo script che presentiamo non ha alcuna pretesa di ottimizzazione del codice; è una diretta implementazione dell'algoritmo visto in questo testo. Peraltro, la questione di come implementarlo nel modo più efficiente è aperta; a chi è interessato consigliamo di leggere l'articolo di Crandall e Papadopulos [23].
In questo script PARI/GP (ed anche nei seguenti) useremo, al fine di verificare se un intero è pari o dispari, l'operatore `bittest(x,n)` il quale fornisce l'n-esimo bit dell'espansione binaria di x:


```
gp > ?bittest
bittest(x,n): gives bit number n (coefficient of 2^n) of
the integer x. Negative numbers behave as if modulo
big power of 2.
```

Chiaramente `bittest(x,0)` restituisce 0 se x è pari e 1 se x è dispari. Utilizzeremo anche gli operatori logici `||` (or), `&,&&` (and) ed il costrutto `until`

```
gp > ?until
until(a,seq): evaluate the expression sequence seq until a
is nonzero.
```

Uno script PARI/GP per l'algoritmo di Agrawal, Kayal & Saxena, in cui daremo in output 1 se il numero sotto test è primo e 0 se il numero sotto test è composto, è quindi:

```
{ /** Funzione ausiliaria che verifica che l'ordine di **
***    n modulo r sia piu' grande di ord       ******/
ordine(n, r, ord) = local (k, flag);
    flag=1;
    k=1;
    until ( k > ord,
     flag = flag && (lift(Mod(n,r)^k) !=1);
     if (flag == 0, return (flag));
     k=k+1;
    );
/**  flag = 0 significa che non e` stato raggiunto  **
**  un ordine di n mod r maggiore di ord; flag =1  **
**  significa che e` stato raggiunto   */
    return(flag)}

{aks(n)= local(b, r, good, ord, logn,log2n, l, m,
k,gcdbn);
/********** Controlli iniziali **************/
if((n <= 3)&(n>1), print(n, " e` primo"); return(1));
if( n<=1, error("Input non valido"));
if(!bittest(n,0), print(n, " e` pari"); return(0));
/****** Test per le potenze di primi dispari *****/
if(issquare(n), print(n," e` il quadrato di ",
                        floor(sqrt(n))); return(0));
logn=log(n);
```



```
l= ceil(exp(logn/ 3));
forprime(p=3,l,
     m = ceil(logn/log(p));
     forstep(k=3, m,2,
            if(n==p^k, print(n," e` uguale a ",p,
             " elevato a ",k); return(0))));
print(n," non e` una potenza prima");
/**   Ricerca del minimo r per cui n ha ordine  **
 **   maggiore di log_2^2 n in Zr           **/
log2n = logn/log(2);
ord= floor((log2n)^2);
/**   se n deve avere almeno ordine ord allora   **
 **   phi(r) >= ord e quindi r>=phi(r)+1 >= ord+1 **/
r = ord+1;
/**   n deve essere un elemento invertibile di Zr  **/
until (gcd(n,r) == 1, r=r+1);
good = ordine(n,r, ord);
/**   good=1 significa che r e` stato determinato  **/
until( good ,  until ((gcd(n,r) == 1), r=r+1);
             good = ordine(n,r,ord)
           );
print("r = ", r);
/**   condizioni per n <= r o con divisori <= r  **/
for( b=1, r, gcdbn=gcd(b,n);
     if( gcdbn > 1 &&  gcdbn < n,
         print(n," e` composto ed e` diviso da ",gcdbn);
          return(0) )
   );
if (n <= r,    print(n," e` primo"); return(1) );
/** Test polinomiale di AKS  **/
s=floor(sqrt(r)*log2n);  elle=n%r;
print("inizio verifiche condizioni polinomiali di aks");
for( b=1, s,
     if( Mod(x + Mod(b,n),x^r-1)^n  !=
                    Mod(x^elle + Mod(b,n),x^r-1),
            print(n," e` composto"); return(0))
   );
print(n," e` primo");
return(1)}
```

Il seguente esempio rende chiaro che l'aritmetica polinomiale rende in pra-

tica AKS poco efficiente.

```
gp> aks(1131114389)
1131114389 non e' una potenza prima
r = 907
inizio verifiche condizioni polinomiali di aks
1131114389 e' primo
%3 = 1
time =  last result computed in 19,392 ms.
gp> isprime(1131114389)
%11 = 1
time =  0 ms.
```

– Algoritmo di pseudoprimalità di Miller & Rabin 11.8.5. Riportiamo anche una semplice versione dell'algoritmo di Miller & Rabin perché, in mancanza di implementazioni efficienti dell'algoritmo AKS, costituisce una delle basi per i test di primalità effettivamente usati in pratica. Utilizziamo l'operatore di shift a destra di n bit `x>>n)` (che corrisponde alla divisione intera per 2^n) e l'istruzione `next` (di terminazione anticipata di un ciclo)

```
next({n=1}): interrupt execution of current instruction
sequence, and start another iteration from the n-th
innermost enclosing loops.
```

Nello script `MillerRabin(n,c)` in input è richiesto il numero n da testare ed il numero c di basi casuali su cui eseguire il test. Se non specificato diversamente viene posto $c = 1$. In output daremo 1 se il numero n è pseudoprimo forte in c basi scelte casualmente e 0 se il numero n è composto.

```
{  MillerRabin(n,c=1) = local(m,d,s,a,b,e,c1);
if((n <= 3)&(n>1), print(n, " e' primo"); return);
if( n<=1, error("Input non valido"));
if(!bittest(n,0), print(n, " e' pari"); return(0));
/*******************************************************
 INIZIALIZZAZIONE: Posto d = n-1, s = 0, e fino a
              che d e' pari si pone d = d/2 e s = s+1
*******************************************************/
m = n-1;
d = m;
s = 0;
while(!bittest(d,0), d=d>>1; s++);
/*******************************************************
```



```
     alla fine del while si ha  n-1=2^s * d, d dispari
*********************************************************/
c1=c;
while(c > 0,
/*********************************************************
TEST: usando un generatore di numeri pseudocasuali,
          si sceglie un a tale che 1 < a < n. Si
          pone e = 0, b = a^d (mod n). Se b == 1 o
          b == n-1 si pone c = c-1 e  si ripete il test.
*********************************************************/
     until(lift(a) > 1,  a = Mod(random(n),n));
     e = 0;
     b = Mod(lift(a),n)^d;
     if(b == 1 | b==m, c--; next);
/*********************************************************
   n e` pseudoprimo forte in base a;  ripartenza del ciclo
*********************************************************/
      while( b != 1 && b != m && e <= s-2,
                          b = Mod(lift(b),n)^2; e++);
      if(b != m, print(n, " e` composto");return(0));
/*********************************************************
QUADRATI :  Se b != (+-)1 (mod n) e e <= s-2,
          calcoliamo b = b^2 (mod n) e e = e + 1.
          Se b != n - 1, n e` composto, ritorno 0 e
          l'algoritmo termina, altrimenti n e`
          pseudoprimo forte in base a, decremento
          di c e ripartenza del ciclo
*********************************************************/
c--);
print(n," e` pseudoprimo forte in ", c1, "
basi a (casuali)");
return(1)}
```

Ed ecco un esempio di utilizzo:

```
gp> MillerRabin(1131114389,200000)
1131114389 e` pseudoprimo forte in 200000 basi a
(casuali)
%9 = 1
time = 1,980 ms.
```

– Algoritmo di fattorizzazione di Fermat 5.3.2

Definiamo una funzione `FermatFatt(n,R)`, dove n è un intero positivo dispari e R è un parametro fornito dall'utente che indica un intervallo in cui x varia per determinare $n = x^2 - y^2$; se R non viene indicato, per default si definisce $R = \lceil n^{1/3} \rceil$. Questo algoritmo è efficiente solo se y è piccolo rispetto a n.

```
{ FermatFatt(n,R=-1)=
    local(a,x,y,j);
    if( n<=1, error("Input non valido"));
    if(!bittest(n,0), print(n, " e` pari"); return);
    a=floor(sqrt(n));
    if(n==a^2,print(n," e` il quadrato di ",a);return);
    x=a+1;
    y=floor(sqrt(x^2-n));
    if(n==x^2-y^2,print(n," e` il prodotto di ",x-y,
                        " e ",x+y);return);
    if (R==-1, R=ceil(n^(1/3)));
    for (j=1,R,
        x++;
        y=floor(sqrt(x^2-n));
        if(n==x^2-y^2,print(n," e` il prodotto di ",x-y,
            " e ",x+y);return);
        );
    print (n" non e` del tipo x^2-y^2 per ogni x+j, j
         nell'insieme {0,...,R=", R ,"}");
    return}
```

Ed ecco un esempio di utilizzo:

```
gp> FermatFatt(1131114389)
1131114389 non e` del tipo x^2-y^2 per ogni x+j, j
                    nell'insieme {0,...,R=1042}
time =  2 ms.
```

– Algoritmo di fattorizzazione di Fermat-Lehman 5.3.2
Realizziamo la variante di Lehman al metodo di Fermat. Definiamo una funzione `FermatLehmanFatt(n)`, dove n è un intero positivo dispari. Si esegue una divisione per tentativi fino al livello $R = \lceil n^{1/3} \rceil$. Dopodiché si cerca una fattorizzazione della forma $4kn = x^2 - y^2$, dove k varia tra 1 e R.

```
{ FermatLehmanFatt(n)= local(a,b,t,s,m,R,R1,k);
    if( n<=1, error("Input non valido"));
    if(!bittest(n,0), print(n, " e` pari"); return);
```



```
    a=floor(sqrt(n));
    if(n==a^2,print(n," e` il quadrato di ",a);return);
    R=ceil(n^(1/3));
    R1=n^(1/6);
    /** trial division sui dispari fino al livello R **/
    forstep (m=3,R,2,if (n%m==0, print(m," divide ",n);
                return));
    /****** Ciclo di Fermat  *******/
    for (k=1,R,t=2*sqrt(k*n);
        for(a=ceil(t), floor(t+R1/(4*sqrt(k))),
                s=a^2-4*k*n;b=floor(sqrt(s));
                if(b^2==s,print(gcd(a+b,n)," divide ",n);
                print(gcd(a-b,n)," divide ",n);
                return
            );
        );
    );
    print(n," e` primo");
    return}
```

Ed ecco un esempio di utilizzo:

```
gp> FermatLehmanFatt(1131114389)
1131114389 e` primo
time = 5 ms.
```

– Metodo ρ di Pollard 5.3.4
Definiamo una funzione `PollardRhoFatt(n,seed, loops)` in cui *n* è il numero da fattorizzare, *seed* è un seme della successione pseudocasuale e *loops* indica il numero massimo di iterazioni consentito prima di richiedere un cambio di seed o loops.

```
{PollardRhoFatt(n,seed=2, loops=100)=
local(x,y,j,mcd);
x=seed; y=seed;
for (j=1,loops,
x=(x^2+1)%n;
y=(y^2+1)%n; y=(y^2+1)%n;
mcd = gcd(n,(y-x)%n);
if (mcd >1 && mcd < n, print(mcd, "e` un fattore di n");
     print("Numero di iterazioni necessarie: ",j);
     return() \\return(mcd)
```



```
    );
  );
  print("Nessun fattore determinato; prova a cambiare seed
  e/o incrementare loops");
  return(0)}
```

Ed ecco un esempio di utilizzo:

```
gp>   PollardRhoFatt(1131114389)
Nessun fattore determinato; prova a cambiare seed
e/o incrementare loops

gp> PollardRhoFatt(344342111992018112223311,35)
Nessun fattore determinato; prova a cambiare seed
e/o incrementare loops

gp> PollardRhoFatt(344342111992018112223311,3514)
Un fattore di n e': 14533
Numero di iterazioni necessarie: 4

gp> PollardRhoFatt(344342111992018112223311,35,100000)
Un fattore di n e': 14533
Numero di iterazioni necessarie: 110

gp> lungo = 251959084756578934940271832400483985714292821262040320277771378360436620207075955562640185258807844069182906412495150821892985591491761845028084891200728449926873928072877767359714183472702618963750149718246911650776133798590957000973304597488084284017974291006424586918171951187461215151726546322822168699875491824224336372590851418654620435767984233871847744479207399342365848238242811981638150106748104516603773060562016196762561338441436038339044149526344321901146575444541784240209246165157233507787077498171257724679629263863563732899121548314381678998850404453640235273819513786365658743912120103971228221207203570

gp>  PollardRhoFatt(lungo)
Un fattore di n e': 19
Numero di iterazioni necessarie: 3
```

– Metodo $p-1$ di Pollard 5.3.5

Definiamo una funzione `Pollardp1Fatt(n,B,s)` in cui n è il numero da fattorizzare, B è la grandezza della base di fattori e s indica il numero di basi a scelte casualmente; di default viene usata una unica base casuale.

```
{ Pollardp1Fatt(n,B=100,s=1)=
     local(m, a, d, g, i, lgn);
     if( n<=1, error("Input non valido"));
     if(!bittest(n,0), print(n, " e` pari"); return);
     lgn = log(n);
     print("livello base di fattori B = ", B);
     v=vector(B,i,i); \\print(v);
     k=lcm(v);
     print("esponente k=lcm(1,...,B) = ", k);
     for(i=1,s,
         /******* genero a finche' n non divide a *******/
        until(a, a=random()%n);
        print("base scelta numero ", i, ": a= ",a);
        g=gcd(a,n);
        if(g>1, print(g," e` un fattore di ",n); return);
        a = lift(Mod(a,n)^k);
        d = gcd(a-1, n);
        if(d>1 && d<n, print(d," e` un fattore di n'');
            return);
 );
     if(d==1 || d==n, print("Prova ad incrementare B"))}
```

Ed ecco un esempio di utilizzo:

```
gp> ? Pollardp1Fatt(1131114389,200,10)
livello base di fattori B = 200
esponente k=lcm(1,...,B) = 337293588832926264639465766794
8414074323943 827851572342 288470219172340180606773900
66992000
base scelta numero 1: a= 468907969
base scelta numero 2: a= 1016083457
base scelta numero 3: a= 282164227
base scelta numero 4: a= 597129616
base scelta numero 5: a= 455543176
base scelta numero 6: a= 943685859
base scelta numero 7: a= 41845718
base scelta numero 8: a= 1000046014
base scelta numero 9: a= 490226949
```



```
base scelta numero 10: a= 552921694
Prova ad incrementare B
time = 4 ms.


gp> Pollardp1Fatt(lungo,3,10)
livello base di fattori B = 3
esponente k=lcm(1,...,B) = 6
base scelta numero 1: a= 563198484
base scelta numero 2: a= 1811632248
base scelta numero 3: a= 1204133923
19 e' un fattore di n
time =  0 ms.
```

E.6.5 Alcuni esercizi

Esercizio E.6.1 *Determinare se* 3 *è un quadrato modulo* $p = 55950455273$.
Risposta: *Basta calcolare* $3^{(p-1)/2} \bmod p$. *Usando PARI/GP si ha:*

```
gp > p=55950455273
%1 = 55950455273
gp > p Mod(3,p)^((p-1)/2)
%2 = Mod(55950455272, 55950455273)
          \\ classe di -1 modulo 55950455273.
```

e quindi 3 *non è un quadrato modulo p.* ■

Esercizio E.6.2 *Scrivere un programma PARI/GP che scriva "Eccomi!!!!!" sei volte.*

Esercizio E.6.3 *Scrivere un programma PARI/GP per calcolare il gcd di due numeri scelti casualmente aventi 1000 cifre ciascuno.*

Esercizio E.6.4 *Calcolare, usando PARI/GP,* $\pi(16743)$.

Esercizio E.6.5 *Scrivere un programma PARI/GP che, dato in input un naturale n, produca come output una sequenza* `[x,y,z,w]` *di interi tali che* $x^2+y^2+z^2+w^2 = n$.

Esercizio E.6.6 *Scrivere, usando PARI/GP,* 2005 *come somma di tre quadrati.*

Complementi F
Letture ulteriori

Nella bibliografia sono elencati molti testi che possono integrare quanto qui detto. È comunque opportuno osservare che, per la maggior parte, la letteratura rilevante è in lingua inglese e si trova sparsa su decine di articoli apparsi su riviste specialistiche. Abbiamo citato testi in italiano tutte le volte che ci è stato possibile, ma, come si può vedere, si tratta di una minoranza dei casi. Ulteriori riferimenti bibliografici sono inseriti nei luoghi appropriati di questo testo, come, per esempio, in fondo al §D.4.

F.1 Capitolo 1

Il racconto breve "Lo scarabeo d'oro" di Poe [67] è una vivace descrizione di come si può violare un sistema crittografico monoalfabetico mediante un'analisi di frequenza. Per la trattazione dettagliata dei racconti di Poe [67] e Verne [95] si veda [100].

F.2 Capitolo 2

DES. La descrizione del DES appare in vari testi; alcuni di questi sono i libri di Buchmann [15], di Schneier [83] e di Menezes, van Oorschot e Vanstone [60]. Segnaliamo anche gli articoli di S. Landau [50], [51].
AES. In rete si può trovare del materiale su AES nel sito del NIST: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf Per una presentazione del procedimento di valutazione dei vari candidati al concorso AES si veda l'articolo di S. Landau [49]. In [51] lo stesso autore analizza il ruolo dell'Algebra nella progettazione di Rijndael e dei suoi competitori. La referenza principale è il testo *The design of Rijndael* di Daemen e Rijmen [25]. Altre presentazioni si trovano nei libri di Stallings [92] e di Stinson [93]. Una succinta interpretazione rivolta ai matematici è stata fatta da Lenstra [58].
Teoria di Shannon. I lavori originali di Shannon sono [87, 88]. Un buon riferimento bibliografico moderno è il libro di Stinson [93].

F.3 Capitolo 3

Problema di Diffie–Hellman. Per alcuni aspetti teorici si vedano i lavori di Canetti, Friedlander e Shparlinski [17] e dagli stessi autori in collaborazione anche con Konyagin, Larsen e Lieman [16]
RSA. Una panoramica di molte di queste varianti, dei loro punti di forza e debolezza si può trovare nel libro di Hinek [40]. È chiaro che la strategia di fattorizzare n non è l'unica possibile per attaccare RSA. Consigliamo al lettore interessato di leggere l'articolo di Boneh [14] e il libro di Hinek [40].

F.4 Capitolo 9

Strutture algebriche. Si vedano le Lezioni 16-24 e 27-28 del libro di Facchini [32], oppure i capitoli 14 e 15 del libro di Lang [52]. Per la teoria degli anelli, si veda in particolare Procesi [78].

Struttura di $\mathbb{Z}_n$ e di $\mathbb{Z}_n^*$

Si vedano i capitoli 3, 4, 6, 7 di Childs [20], il libro di Hardy e Wright [37], che contiene la maggior parte dei risultati teorici di cui abbiamo parlato qui: si vedano in particolare i capitoli 5-7. Il libro di Shanks [86] contiene una discussione dettagliata della struttura dei gruppi $\mathbb{Z}_m^*$ per qualunque valore di $m \in \mathbb{Z}$ nei §§23-38: si vedano in particolare i diagrammi nel §33. Nel §35 c'è la dimostrazione del teorema che riguarda i gruppi moltiplicativi ciclici del tipo $\mathbb{Z}_m^*$.

Algoritmo di Euclide

Una trattazione dettagliatissima dell'algoritmo di Euclide si può trovare in Knuth [45, §4.5.2], e la relativa analisi di complessità nel paragrafo successivo. Negli stessi paragrafi sono descritti algoritmi alternativi leggermente più efficienti. Si veda anche Shoup [89]. Per alcuni aspetti elementari si veda anche [101].

Polinomi e Congruenze

Per una discussione completa della soluzione di equazioni del tipo $f(x) \equiv 0 \bmod n$, dove $f \in \mathbb{Z}[x]$ ed $n \in \mathbb{N}^*$, si vedano i capitoli 7 ed 8 di Hardy e Wright [37]. Si veda anche il capitolo 5 di Apostol [7].

Pseudoprimi e numeri di Carmichael

Nel libro di Ribenboim [79] si possono trovare i risultati teorici su pseudoprimi, numeri di Carmichael ed ulteriori estensioni di questi concetti. Si vedano in particolare i §§2.II.C, 2.II.F, 2.III, 2.VIII, 2.IX. Per i generatori si veda il §2.II.A, per la crittografia il §2.XII.B, mentre la congettura di Artin è discussa nel §6.I. Ulteriori informazioni relative alla distribuzione degli pseudoprimi si possono trovare nell'articolo di Pomerance, Selfridge e Wagstaff [76].

I dati del §11.8.3 sono tratti dagli articoli di Pomerance, Selfridge e Wagstaff [76], Jaeschke [42], Zhang [104], Zhang e Tang [105] e Jiang e Deng [43].

Criteri di primalità

I più semplici si trovano in Hardy e Wright [37]. Si veda anche Ribenboim [79], l'articolo di Adleman, Pomerance e Rumely [1], Crandall e Pomerance [24, capitolo 4], Pomerance [75], Knuth [45, §4.5.4] e Shoup [89]. Si veda il lavoro di Agrawal, Kayal e Saxena [2] per la dimostrazione dell'esistenza di un algoritmo polinomiale per decidere la primalità di un intero. Questo articolo ha avuto un'eco notevolissima e generato numerosi tentativi di miglioramento, dei quali è impossibile dare un sunto significativo.

Dimostrazioni alternative di alcuni dei risultati presentati in questo capitolo si possono trovare negli articoli di Anderson, Benjamin e Rouse [6], di Mortola [61] e in [55], [56].

Algoritmi di base

Gli aspetti elementari sono trattati anche in [102].

Algoritmi di fattorizzazione

Si vedano l'articolo di Dixon [30], la monografia di Riesel [80] e il libro di Crandall e Pomerance [24], che contiene una dettagliata descrizione di tutti i più moderni algoritmi che riguardano i numeri primi (inclusi quelli trattati qui), tra cui i metodi di fattorizzazione mediante curve ellittiche ed il crivello con i campi di numeri (*Number Field Sieve*) che al momento attuale sembra essere il migliore in circolazione. Introduzioni più brevi agli stessi algoritmi si trovano negli articoli di Pomerance [68], [71] e [72], mentre il solo crivello quadratico è descritto in [70] ed in [74]. Si vedano anche il §V.5 di Koblitz [46], che tratta sia il crivello quadratico che il crivello con i campi di numeri, ed il §4.5.4 di Knuth [45].

Algoritmi per il logaritmo discreto

Si vedano i §§5.2.2. 5.2.3, 5.3 di Crandall e Pomerance [24], ed anche Pomerance [73].

Altri algoritmi

L'algoritmo di Gauss per la determinazione di un generatore di $\mathbb{Z}_p^*$ è descritto in Gauss [34], §§73-74. Si veda anche Ribenboim [79, §2.II.A]. Algoritmi per l'aritmetica modulo n, e per la moltiplicazione e l'esponenziazione veloce si trovano in Crandall e Pomerance [24] (rispettivamente nei capitoli 2 e 9) ed in Knuth [45, §4.3]. Algoritmi efficienti per l'aritmetica polinomiale si possono trovare in Knuth [45, §4.6]: in particolare, la discussione dell'esponenziazione si trova nel §4.6.3. Molti algoritmi importanti per la teoria computazionale dei numeri si trovano descritti in Shoup [89].

Curve ellittiche

Per una semplice introduzione si veda Husemöller [41], oppure il capitolo VI del libro di Koblitz [46], che contiene anche le applicazioni alla crittografia. Una presentazione più recente si trova nel libro di Washington [97]. Ulteriori dettagli, ed anche il caso delle curve iperellittiche si trovano nel testo di Cohen, Frey, et al., [21].

Crittografia

Si vedano i libri di Koblitz [46] e [47], e quello di Menezes, van Oorschot e Vanstone [60]. In particolare, di quest'ultimo si vedano i capitoli 1-3 e la bibliografia. Altri testi interessanti sono quelli di Wagstaff [96], Smart [91], Galbraith [33]

Si veda anche il capitolo 8 di Crandall e Pomerance [24] che dà una panoramica sui problemi legati all'utilizzazione dei numeri primi, in particolare alla crittografia. Una discussione più approfondita dei crittosistemi di ElGamal e di Massey-Omura si trova nel §IV.3 del libro di Koblitz [46]. Una presentazione dei concetti base della crittografia a chiave pubblica e dei suoi legami con la teoria dei numeri si trova negli articoli di Languasco e Perelli [53, 54].

Per la crittografia classica consigliamo il testo di Rouse Ball & Coxeter [81]. Una interessante e molto leggibile presentazione storica si trova in Singh [90] mentre l'enciclopedico lavoro di Kahn [44] va considerato come testo principale di riferimento. Nel libro di Winterbotham [98], che era un alto ufficiale dell'esercito britannico nella Seconda Guerra Mondiale, si parla degli effetti politici e

militari della decifrazione dei messaggi di Enigma. Per quanto riguarda la Guerra del Mediterraneo si consulti Santoni [82].

La descrizione di molti metodi classici, tra cui il famoso Purple utilizzato dai giapponesi durante la Guerra del Pacifico, si può anche trovare in Konheim [48].

Presentazioni rivolte agli studenti delle scuole superiori si possono trovare nell'articolo di Alberti [4], di Schoof [84], di Zaccagnini [99], [100] e in [57].

Teoria di Galois

Si vedano le note del corso di Murphy [62] ed il libro di Procesi [77].

Il Teorema dei Numeri Primi

Hardy e Wright [37, §§22.14-16] danno una dimostrazione elementare (che cioè non fa uso dell'analisi complessa) della relazione $\pi(x) \sim x(\log x)^{-1}$. Una breve descrizione delle principali idee necessarie si trova in Apostol [7, §4.10]. Un'altra dimostrazione elementare si può trovare nel capitolo 4 di Tenenbaum e Mendès France [94]. La dimostrazione dei teoremi B.2.1 e B.2.2 di trova in Davenport [27, §§7-22], e quella del solo teorema B.2.1 nel capitolo 13 di Apostol [7], in entrambi i casi con un uso molto pesante dell'analisi complessa.

Altri risultati sulla distribuzione dei numeri primi

Si vedano il libro di Hardy e Wright [37], in particolare il capitolo 22, il libro di Davenport [27], il libro di Apostol [7] e quello di Tenenbaum e Mendès France [94]. Il capitolo 1 di Crandall e Pomerance [24] contiene anche alcuni risultati interessanti dal punto di vista computazionale. La formula di Legendre (B.1.13) vi è dimostrata nell'esercizio 1.13. Per la funzione $\Psi(x,y)$ si vedano Hildebrand e Tenenbaum [39], Tenenbaum e Mendès France [94, §§4.5-4.6]. Una semplice argomentazione in sostegno di $\Psi(x,y) \approx xu^{-u}$ si trova nel §V.3 di Koblitz [46]. Il calcolo di $\pi\left(10^{18}\right)$ è stato portato a termine da Deléglise e Rivat [28]. A partire dai risultati di quest'ultimo articolo è nato un progetto dedicato al calcolo esatto di $\pi(x)$ per valori sempre più grandi di x: l'annuncio del risultato su $\pi\left(4 \cdot 10^{22}\right)$ ed altri aggiornamenti si possono trovare all'indirizzo http://numbers.computation.free.fr/Constants/Primes/Pix/pixproject.html. Nel maggio 2013 è stato annunciato il valore esatto di $\pi\left(10^{25}\right)$: gli autori sono J. Buethe, J. Franke, A. Jost e T. Kleinjung.

Ipotesi di Riemann

È brevemente menzionata in Apostol [7, §13.9]. Nel libro di Davenport [27, §8 e §18] si trovano l'enunciato e le conseguenze sulla distribuzione dei numeri primi, mentre nel §20 c'è la corrispondente trattazione dell'ipotesi generalizzata di Riemann. Ribenboim [79] parla della relazione dell'ipotesi di Riemann riguardo ai criteri di primalità nel §2.XI.B, e della sua relazione con la distribuzione dei numeri primi nel §4.I. Una breve discussione si trova anche in Riesel [80, capitolo 2]; la rilevanza dell'ipotesi generalizzata di Riemann per l'algoritmo di Miller-Rabin è spiegata nel capitolo 4. Tenenbaum e Mendès France [94] discutono l'ipotesi di Riemann nei §§2.4-2.5 e di nuovo all'inizio del capitolo 5. Il capitolo 1 di Crandall e Pomerance [24] spiega la rilevanza dell'ipotesi di Riemann e della sua generalizzazione in una grande quantità di problemi legati alla distribuzione dei numeri primi, inclusi alcuni, come la congettura 11.6.3, di diretto interesse per le applicazioni discusse in questo libro.

Altri riferimenti

Un'introduzione molto leggibile ai problemi di cui abbiamo parlato si trova in Pomerance [69].

Bibliografia complementi

[1] L.M. Adleman, C. Pomerance, R.S. Rumely, "On distinguishing prime numbers from composite numbers", *Annals of Math.*, 117, pagine 173–206, 1983.

[2] M. Agrawal, N. Kayal, N. Saxena, "PRIMES is in P", *Annals of Math.*, 160, pagine 781-793, 2004, http://annals.math.princeton.edu/wp-content/uploads/annals-v160-n2-p12.pdf.

[3] A.V. Aho, J.E. Hopcroft, J.D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, 1974.

[4] G. Alberti, "Aritmetica finita e crittografia a chiave pubblica - Un percorso didattico per gli studenti delle Scuole Medie Superiori", in A. Abbondandolo, M. Giaquinta, F. Ricci (a cura di), *Ricordando Franco Conti*, pagine 1–29, Scuola Normale Superiore, 2004.

[5] W.R. Alford, A. Granville, C. Pomerance, "There are infinitely many Carmichael numbers", *Annals of Math.*, 140, pagine 703–722, 1994.

[6] P.G. Anderson, A.T. Benjamin, J.A. Rouse, "Combinatorial Proofs of Fermat's, Lucas's, and Wilson's theorems", *Amer. Math. Monthly*, 112, pagine 266–268, 2005.

[7] T.M. Apostol, *Introduction to Analytic Number Theory,* Springer, 1976.

[8] R. Ash, "A Pari/GP Tutorial", http://www.math.uiuc.edu/~r-ash/GPTutorial.pdf, 2007.

[9] R. Avanzi, P. Mihăilescu, "Efficient "quasi"-deterministic primality test improving AKS draft", http://www.uni-math.gwdg.de/preda/mihailescu-papers/ouraks3.pdf, 2003.

[10] D. Bernstein, "Proving primality after Agrawal-Kayal-Saxena", prepubblicazione, 2003, http://cr.yp.to/papers.html#aks.

[11] D. Bernstein, "Distinguishing Prime Numbers from Composite Numbers: the state of the art in 2004", prepubblicazione, 2004, http://cr.yp.to/papers.html#prime2004.

[12] D. Bernstein, "Proving primality in essentially quartic random time", *Math. Comp.*, 76, pagine 389–403, 2007.

[13] P. Berrizbeitia, "Sharpening "PRIMES is in *P*" for a large family of numbers, *Math. Comp.*, 74, pagine 2043–2059, 2005.

[14] D. Boneh, "Twenty years of attacks on the RSA cryptosystem", *Notices of the A.M.S.*, 46, pagine 203–213, 1999.

[15] J. Buchmann, *Introduction to Cryptography,* Springer, seconda ed., 2004.

[16] R. Canetti, J. Friedlander, S. Konyagin, M. Larsen, D. Lieman, I. Shparlinski, "On the statistical properties of Diffie-Hellman distributions", *Israel J. Math.*, 120, pagine 23–46, 2000.

[17] R. Canetti, J. Friedlander, I. Shparlinski, "On certain exponential sums and the distribution of the Diffie-Hellman triples", *J. London Math. Soc.*, 59, pagine 799–812, 1999.

[18] Z. Cao, L. Liu, "Remarks on AKS Primality Testing Algorithm and A Flaw in the Definition of P", http://arxiv.org/abs/1402.0146, 2014.

[19] Q. Cheng, "Primality Proving via one round in ECPP and one iteration in AKS", *J. Cryptology*, 20, pagine 375–387, 2007, http://www.cs.ou.edu/~qcheng/paper/aksimp.pdf.

[20] L. Childs, *Algebra: un'introduzione concreta,* ETS, 1983.

[21] H. Cohen, G. Frey, R. Avanzi, C. Doche, T. Lange, K. Nguyen, F. Vercauteren (a cura di), *Handbook of Elliptic and Hyperelliptic Curve Cryptography,* Discrete Mathematics and its Applications, Chapman & Hall/CRC, 2006.

[22] J.H. Conway, R.K. Guy, *Il libro dei numeri,* Hoepli, 1999.

[23] R. Crandall, J. Papadopulos, "On the Implementation of AKS-Class Primality Tests", http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.95.58, 2003.

[24] R. Crandall, C. Pomerance, *Prime numbers. A computational perspective,* Springer, seconda ed., 2005.

[25] J. Daemen, V. Rijmen, *The design of Rijndael,* Springer, 2002.

[26] H. Davenport, *Aritmetica Superiore,* Zanichelli, 1994.

[27] H. Davenport, *Multiplicative Number Theory,* Springer, terza ed., 2000.

[28] M. Deléglise, J. Rivat, "Computing $\pi(x)$: The Meissel, Lehmer, Lagarias, Miller, Odlyzko method", *Math. Comp.*, 65, pagine 235–245, 1996.

[29] M. Dietzfelbinger, *Primality Testing in Polynomial Time,* Springer, 2004.

[30] J.D. Dixon, "Factorization and primality tests", *Amer. Math. Monthly*, 91, pagine 333–352, 1984.

[31] P. Erdős, "On pseudoprimes and Carmichael numbers", *Publ. Math. Debrecen*, 4, pagine 201–206, 1956.

[32] A. Facchini, *Algebra × Informatica,* decibel, 1986.

[33] S.D. Galbraith, *Mathematics of Public Key Cryptography,* Cambridge U.P., 2012.

[34] K.F. Gauss, *Disquisitiones Arithmeticae,* G. Fleischer, 1801, trad. inglese a cura di W.C. Waterhouse, Springer, 1986.

[35] A. Granville, "It is easy to determine whether a given integer is prime", *Bull. A.M.S.*, 42, pagine 3–38, 2005, http://www.ams.org/journals/bull/2005-42-01/S0273-0979-04-01037-7/S0273-0979-04-01037-7.pdf.

[36] A. Granville, C. Pomerance, "Two contradictory conjectures concerning Carmichael numbers", *Math. Comp.*, 71, pagine 883–908, 2002.

[37] G.H. Hardy, E.M. Wright, *An Introduction to the Theory of Numbers,* Oxford Science Publications, sesta ed., 2008, revisione a cura di D.R. Heath-Brown e J.H. Silverman; con una prefazione di A. Wiles.

[38] G. Harman, "On the number of Carmichael numbers up to x", *Bull. London Math. Soc.*, 37, pagine 641–650, 2005.

[39] A. Hildebrand, G. Tenenbaum, "Integers without large prime factors", *J. Théorie des Nombres Bordeaux*, 5, pagine 411-484, 1993.

[40] M.J. Hinek, *Cryptanalysis of RSA and its variants,* CRC press, 2009.

[41] D. Husemöller, *Elliptic Curves,* Springer, seconda ed., 2004.

[42] G. Jaeschke, "On strong pseudoprimes to several bases", *Math. Comp.*, 61, pagine 915–926, 1993.

[43] Y. Jiang, Y. Deng, "Strong pseudoprimes to the first eight prime bases", *Math. Comp.*, 83, pagine 2915–2924, 2014.

[44] D. Kahn, *The Codebreakers: The Comprehensive History of Secret Communication from Ancient Times to the Internet,* Scribner, seconda ed., 1996.

[45] D.E. Knuth, *The Art of Computer Programming. Vol. 2. Seminumerical Algorithms,* Addison Wesley, seconda ed., 1981.

[46] N. Koblitz, *A Course in Number Theory and Cryptography,* Springer, seconda ed., 1994.

[47] N. Koblitz, *Algebraic Aspects of Cryptography,* Springer, 1998.

[48] A.G. Konheim, *Computer security and cryptography,* John Wiley & Sons, 2007.

[49] S. Landau, "Communications Security for the Twenty-First Century: the Advanced Encryption Standard", *Notices of the A.M.S.*, 47, pagine 450–459, 2000, http://www.ams.org/notices/200004/fea-landau.pdf.

[50] S. Landau, "Standing the Test of Time: the Data Encryption Standard", *Notices of the A.M.S.*, 47, pagine 341–349, 2000, http://www.ams.org/notices/200003/fea-landau.pdf.

[51] S. Landau, "Polynomials in the Nation's Service: Using Algebra to Design the Advanced Encryption Standard", *Amer. Math. Monthly*, 111, pagine 89–117, 2004, http://www.jstor.org/stable/4145212.

[52] S. Lang, *Algebra Lineare,* Boringhieri, 1981.

[53] A. Languasco, A. Perelli, "Numeri Primi e Crittografia", in M. Emmer (a cura di), *Matematica e Cultura 2000*, pagine 227–233, Springer, 2000, trad. inglese in *Mathematics and Culture I*, Springer, 2003.

[54] A. Languasco, A. Perelli, "Crittografia e firma digitale", in M. Emmer, M. Manaresi (a cura di), *Matematica, Arte, Tecnologia, Cinema*, pagine 99–106, Springer, 2002, trad. inglese in *Mathematics, Art, Technology, and Cinema*, Springer, 2003.

[55] A. Languasco, A. Zaccagnini, "Alcune proprietà dei numeri primi, I", *Sito web Bocconi-Pristem*, pagina 26 pp., 2005, http://matematica-old.unibocconi.it/LangZac/home.htm.

[56] A. Languasco, A. Zaccagnini, "Alcune proprietà dei numeri primi, II", *Sito web Bocconi-Pristem*, pagina 32 pp., 2005, http://matematica-old.unibocconi.it/LangZac/home2.htm.

[57] A. Languasco, A. Zaccagnini, *Crittografia,* CLEUP, Padova, 2006, Progetto Lauree Scientifiche per il Veneto.

[58] H.W. Lenstra, "Rijndael for algebraists", http://www.math.berkeley.edu/~hwl/papers/rijndael0.pdf, 2002.

[59] H.W. Lenstra, C. Pomerance, "Primality testing with Gaussian periods", https://www.math.dartmouth.edu/~carlp/aks041411.pdf, 2011.

[60] A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography,* CRC Press, quinta ed., 2001, http://www.cacr.math.uwaterloo.ca/hac.

[61] S. Mortola, "Elogio delle dimostrazioni alternative", in A. Abbondandolo, M. Giaquinta, F. Ricci (a cura di), *Ricordando Franco Conti*, pagine 273–276, Scuola Normale Superiore, 2004.

[62] T. Murphy, *Finite Fields,* University of Dublin, 2012, http://www.maths.tcd.ie/pub/Maths/Courseware/FiniteFields/GF.pdf.

[63] M. Nair, "On Chebyshev-type inequalities for primes", *Amer. Math. Monthly*, 89, pagine 126–129, 1982.

[64] C.D. Olds, *Frazioni Continue,* Zanichelli, 1970.

[65] T. Oliveira e Silva, S. Herzog, S. Pardi, "Empirical verification of the even Goldbach conjecture and computation of prime gaps up to $4 \cdot 10^{18}$", *Math. Comp.*, 83, pagine 2033–2060, 2014.

[66] R. Pinch, "The Carmichael numbers up to 10 to the 21", in *Proceedings Conference on Algorithmic Number Theory, Turku, May 2007, (ed. A.M. Ernvall-Hytönen, M. Jutila, J. Karhumäki and A. Lepistö) Turku Centre for Computer Science General Publications*, volume 46, pagine 265–269, 2007, http://tucs.fi/publications/attachment.php?fname=G46.pdf.

[67] E.A. Poe, *The Gold Bug,* Random House, 1975, trad. it. *Lo scarabeo d'oro*, in *Racconti*, Einaudi, Torino. http://web.tiscali.it/no-redirect-tiscali/manuel_ger/ita/bug_ita.htm.

[68] C. Pomerance, "Recent developments in primality testing", *Math. Intellig.*, 3, pagine 97–105, 1981.

[69] C. Pomerance, "Alla ricerca dei numeri primi", *Le Scienze*, 174, pagine 86–94, febbraio 1983.

[70] C. Pomerance, "The quadratic sieve factoring algorithm", in *Advances in Cryptology, Proceedings of EUROCRYPT 84*, LNCS 209, pagine 169–182, Springer, 1985.

[71] C. Pomerance, "Factoring", in *Cryptology and computational number theory*, volume 42 di *Lect. Notes American Mathematical Society Short Course, Boulder, CO (USA), 1989, Proc. Symp. Appl. Math.*, pagine 27–47, 1990.

[72] C. Pomerance, "A tale of two sieves", *Notices A.M.S.*, 43, pagine 1473-1485, 1996.

[73] C. Pomerance, "Elementary thoughts on discrete logarithms", in J.P. Buhler, P. Stevenhagen (a cura di), *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography*, volume 44 di *Math. Sci. Res. Inst.*

Pub, pagine 385–396, Cambridge U.P., 2008, http://math.dartmouth.edu/~carlp/PDF/dltalk4.pdf.

[74] C. Pomerance, "Smooth numbers and the quadratic sieve", in J.P. Buhler, P. Stevenhagen (a cura di), *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography*, volume 44 di *Math. Sci. Res. Inst. Pub.*, pagine 69–81, Cambridge U.P., 2008, http://math.dartmouth.edu/~carlp/PDF/qs08.pdf.

[75] C. Pomerance, "Primality testing: variations on a theme of Lucas", in *Proceedings of the 13th Meeting of the Fibonacci Association, Congressus Numerantium*, volume 201, pagine 301–312, 2010, https://www.math.dartmouth.edu/~carlp/lucasprime3.pdf.

[76] C. Pomerance, J.L. Selfridge, S.S. Wagstaff, "The pseudoprimes to $25 \cdot 10^9$", *Math. Comp.*, 35, pagine 1003–1026, 1980.

[77] C. Procesi, *Elementi di Teoria di Galois,* decibel, 1977.

[78] C. Procesi, *Elementi di Teoria degli Anelli,* decibel, 1984.

[79] P. Ribenboim, *The New Book of Prime Numbers Records,* Springer, terza ed., 1996.

[80] H. Riesel, *Prime numbers and computer methods for factorization,* Birkhäuser, seconda ed., 1994.

[81] W.W. Rouse-Ball, H.S.M. Coxeter, *Mathematical Recreations and Essays,* Dover, tredicesima ed., 1987.

[82] A. Santoni, *Il vero traditore. Il ruolo documentato di Ultra nella guerra del Mediterraneo,* Mursia, 2005.

[83] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C,* Wiley, seconda ed., 1996.

[84] R. Schoof, "Fattorizzazione e criptosistemi a chiave pubblica", *Didattica delle Scienze*, 137, pagine 48–54, 1988, http://www.mat.uniroma2.it/~geo2/RSAschoof.pdf.

[85] R. Schoof, "Four primality testing algorithms", in *Algorithmic Number Theory: Lattices, Number Fields, Curves and Cryptography. MSRI Publications*, volume 44, pagine 101–126, Cambridge U.P., 2008, http://library.msri.org/books/Book44/files/05rene.pdf.

[86] D. Shanks, *Solved and Unsolved Problems in Number Theory,* Chelsea, quarta ed., 1993.

[87] C.E. Shannon, "A mathematical theory of communication", *Bell System Tech. J.*, 27, pagine 379–423, 623–656, 1948.

[88] C.E. Shannon, "Communication Theory and Secrecy Systems", *Bell System Tech. J.*, 28, pagine 656–715, 1949.

[89] V. Shoup, *A Computational Introduction to Number Theory and Algebra,* Cambridge U.P., 2009, http://www.shoup.net/ntb/ntb-v2.pdf.

[90] S. Singh, *Codici & Segreti,* Rizzoli, 1999.

[91] N.P. Smart, *Cryptography, An Introduction,* McGraw-Hill, 2002, la terza edizione è disponibile in rete all'indirizzo: http://www.cs.bris.ac.uk/~nigel/Crypto_Book/.

[92] W. Stallings, *Crittografia e sicurezza delle reti,* Mc-Graw Hill, seconda ed., 2007.

[93] D. Stinson, *Cryptography: Theory and Practice,* Chapman & Hall/CRC, terza ed., 2005.

[94] G. Tenenbaum, M. Mendès France, *The Prime Numbers and their Distribution,* A.M.S., 2000.

[95] J. Verne, *Viaggio al centro della Terra,* Einaudi, 1989.

[96] S.S. Wagstaff, Jr., *Cryptanalysis of number theoretic ciphers,* Chapman & Hall/CRC, 2003.

[97] L.C. Washington, *Elliptic Curves: Number Theory and Cryptography,* Chapman & Hall/CRC, seconda ed., 2008.

[98] F.W. Winterbotham, *"Ultra Secret - La macchina che decifrava i messaggi segreti dell'Asse",* Mursia, 1994.

[99] A. Zaccagnini, "L'importanza di essere primo", in A. Abbondandolo, M. Giaquinta, F. Ricci (a cura di), *Ricordando Franco Conti*, pagine 343–354, Scuola Normale Superiore, 2004.

[100] A. Zaccagnini, "Cryptographia ad usum Delphini", Quaderno n. 459, Dipartimento di Matematica dell'Università di Parma, febbraio 2007, http://people.math.unipr.it/alessandro.zaccagnini/psfiles/papers/CryptoDelph.pdf.

[101] A. Zaccagnini, Algoritmo di Euclide, numeri di Fibonacci e frazioni continue, in P. Vighi (a cura di), *Progettare Lavorare Scoprire*, pagine 153–162, Dipartimento di Matematica, Università di Parma, 2010, http://people.math.unipr.it/alessandro.zaccagnini/psfiles/papers/art_euclide.pdf.

[102] A. Zaccagnini, Riesame critico delle operazioni elementari, in M. Belloni e A. Zaccagnini (a cura di), *Uno sguardo matematico sulla realtà — Laboratori PLS 2010–2014*, pagine 71–91, Dipartimento di Matematica e Informatica, Università di Parma, 2014, PLS – Parma.

[103] Y. Zhang, "Bounded gaps between primes", *Annals of Math.*, 126, pagine 1121–1174, 2014, http://annals.math.princeton.edu/2014/179-3/p07.

[104] Z. Zhang, "Finding strong pseudoprimes to several bases", *Math. Comp.*, 70, pagine 863–872, 2001.

[105] Z. Zhang, M. Tang, "Finding strong pseudoprimes to several bases. II", *Math. Comp.*, 72, pagine 2085–2097, 2003.

Indice analitico complementi

