Università degli Studi di Torino
Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Triennale in Matematica

TESI DI LAUREA

METODI DI FATTORIZZAZIONE: UN APPROCCIO ALGORITMICO

Relatore:
Prof. Umberto Cerruti

Candidato:
Paolo Lella

A. A. 2005-06

Ringraziamenti

Desidero ringraziare innanzitutto il professor Umberto Cerruti, per avermi seguito durante la preparazione e la stesura di questo lavoro.

Un ringraziamento ovviamente va ai miei genitori e a mio fratello, che mi hanno supportato nei momenti di massimo impegno, durante questi tre anni e nell'ultimo periodo, in cui le scadenze si avvicinavano e il lavoro da svolgere era ancora tanto.

Un grazie speciale a Giulia, perché c'è sempre e quando ho bisogno di aiuto è sempre pronta a tendermi una mano.

Grazie alla pallavolo e ai miei compagni di squadra.

Grazie infine a Elena, Laura e Nicola, con i quali ho condiviso in modo più stretto la vita universitaria di questi tre anni.

Indice

Introduzione

Le tecniche per la fattorizzazione degli interi rivestono un ruolo particolare nella matematica e nella società contemporanea. Ogni giorno milioni di persone effettuano pagamenti via internet trasmettendo il numero della propria carta di credito in assoluta sicurezza, senza temere di essere derubati da qualcuno che riesca ad intercettare le informazioni trasmesse sul web. Questo perché ogni comunicazione riservata viene protetta con un sistema crittografico; i crittosistemi attualmente utilizzati vengono detti sistemi a chiave pubblica, tra i quali il più conosciuto e usato è il protocollo RSA. Questo sistema si basa sulle proprietà dei gruppi finiti e su un'evoluzione del Piccolo Teorema di Fermat: presi due primi p e q si lavora con potenze di elementi di $\mathbb{Z}_{pq}$ e per infrangere il sistema bisogna essere in grado di calcolare efficacemente l'inverso di tali potenze, cosa praticamente impossibile senza conoscere p e q, quando si tratta di interi di circa 30 cifre. La sua sicurezza risiede quindi nell'attuale difficoltà di determinare i fattori di un intero, prodotto di due numeri primi molto grandi. Al momento non esistono algoritmi di fattorizzazione computazionalmente efficienti, ma la ricerca prosegue e la difficoltà potrebbe rivelarsi momentanea: il giorno che un matematico scoprirà una tecnica efficiente, i sistemi crittografici come l'RSA risulteranno inutili ed inutilizzabili e la trasmissione di informazioni riservate tramite internet non sicura.

In questo lavoro vengono presentate alcune idee ed alcuni metodi che hanno portato allo sviluppo delle tecniche moderne di fattorizzazione. Il capitolo 1 è dedicato ai metodi classici, che hanno aperto la strada in questo campo, basati su semplici proprietà matematiche, se non addirittura sulla semplice intuizione. Nel capitolo 2, si affrontano invece tecniche più raffinate, basate su strumenti matematici più particolari, come le frazioni continue, di cui è data una rapida descrizione nell'appendice A. L'obiettivo è arrivare ad esporre con chiarezza uno dei metodi più efficaci al momento: il crivello quadratico di Pomerance. Questo metodo ha riportato grandi successi con interi anche di centinaia di cifre ed è stato determinante per la fattorizzazione dei numeri di Mersenne $M_n = 2^n - 1$, con $n \leqslant 257$.

Visti gli importanti risvolti computazionali dell'argomento, l'appendice B è dedicata all'implementazione dei metodi presentati. Si tratta di algoritmi di facile lettura e comprensione, con in ogni caso una certa attenzione rivolta alla ricerca di soluzioni efficienti nella loro semplicità. Il linguaggio scelto è il C++, affrontato durante il percorso di studi. Tra gli aspetti positivi di questo linguaggio, c'è la possibilità di estendere con semplicità le operazioni base dai tipi di dato fondamentali a strutture più complesse e destinate ad un impiego specifico. È il caso degli interi multiprecisione, di cui è proposta un'implementazione nell'appendice C. Per affrontare da un punto di vista computazionale il problema della fattorizzazione infatti, i tipi di dato fondamentali risultano essere limitanti. La libreria proposta consente invece di lavorare con interi di qualsiasi dimensione, tenendo ovviamente conto che più i numeri sono grandi, maggiore è il tempo richiesto per portare a termine le operazioni.

Capitolo 1

Metodi classici

1.1 Un approccio ingenuo

Il metodo più ovvio e immediato per trovare i divisori di un intero, o equivalentemente per provare la primalità di un numero, è tentare di dividerlo per tutti i numeri primi che lo precedono: se non si trovano divisori propri, il numero sarà primo, altrimenti si sarà determinata la scomposizione in fattori. Riflettiamo ora sull'efficienza di questo sistema ai fini della fattorizzazione di un numero.

La prima osservazione è che dobbiamo conoscere tutti i primi che lo precedono: per ottenere ciò possiamo servirci del crivello di Eratostene. Il funzionamento è molto semplice: si scrivono tutti gli interi da 2 al limite n prefissato, poi si parte da 2, lo si considera primo e si cancellano tutti i suoi multipli; a questo punto si passa al 3 e si cancellano i multipli...dopo ogni eliminazione si prende il più piccolo intero non cancellato, che risulterà essere primo e si eliminano i multipli... Questo processo non viene iterato fino a n, ma solamente fino all'ultimo intero $\leqslant \sqrt{n}$, in quanto tutti gli elementi non cancellati successivi saranno primi. Questo perché ogni intero d, con $\sqrt{n} < d < n$, se composto, ha almeno un divisore $< \sqrt{n}$, infatti

$$d = a \cdot b \text{ e } d < n \Longrightarrow a \leqslant \sqrt{d} < \sqrt{n} \text{ oppure } b \leqslant \sqrt{d} < \sqrt{n}$$

e quindi d è già stato cancellato. Per vedere una sua implementazione, si faccia riferimento all'appendice B.1.

Alternativamente alla ricerca dei numeri primi mediante il crivello, è possibile tentare di dividere il nostro intero n per 2, per 3 e in seguito per tutti gli interi della forma $6k \pm 1$. Il numero complessivo di tentativi da effettuare è superiore rispetto ai tentativi svolti con il crivello (vedi tabella 1.1.1), perché nella successione $6k \pm 1, k = 1, 2, ...$ compaiono molti multipli che prima erano stati eliminati. Il vantaggio invece è che in questo caso non è necessaria la precomputazione dei primi; inoltre è molto semplice determinare la successione con un ciclo, partendo da 5 si aggiunge alternativamente

	10^2	10^3	10^4	10^5	10^6	10^7
Crivello	25	168	1229	9592	78498	664579
$6k \pm 1$	34	334	3334	33334	333334	3333334

Tabella 1.1.1: Confronto tra i tentativi di divisione

2 e 4: inizializzando un parametro $d = 2$, con l'istruzione $d = 6 - d$ ad ogni passo otteniamo l'addendo cercato.

La divisione per tentativi, implementata nell'appendice B.2, risulta efficiente se il numero da scomporre ha fattori abbastanza piccoli, ma decisamente inadeguata se ci troviamo di fronte ad un numero grande, prodotto di primi grandi, in quanto il calcolatore esegue molte operazioni senza mai trovare un divisore. La soluzione di "forza bruta" è quindi da scartare in favore di metodi dalla ricerca più mirata.

1.2 L'approccio di Fermat

L'idea di Fermat è molto semplice: dato un numero n, se troviamo due interi x e y, tali che la differenza dei loro quadrati è proprio n, abbiamo che

$$n = x^2 - y^2 = (x - y)(x + y)$$

cioè, se $x - y \neq 1$, abbiamo trovato due fattori propri di n. Due fattori di questo tipo esistono se n è composto: mettiamoci nel caso $n = a \cdot b$, se consideriamo

$$x = \frac{a+b}{2} \quad \text{e} \quad y = \frac{a-b}{2},$$

otteniamo

$$x^2 - y^2 = \frac{(a+b)^2}{4} - \frac{(a-b)^2}{4} = \frac{a^2 + 2ab + b^2 - a^2 + 2ab - b^2}{4} = a\,b = n.$$

Per costruzione abbiamo che $x \geqslant \sqrt{n}$, quindi è utile controllare preliminarmente se n è un quadrato perfetto. Se questo controllo dà esito negativo comincia la ricerca: in pratica cerchiamo due interi a e b tali che si annulli la quantità

$$S = x^2 - y^2 - n = a^2 - b^2 - n.$$

Quali valori assegnamo a x e y all'inizio del processo? Sappiamo che $x > \sqrt{n}$ quindi si pone $x = \lceil \sqrt{n} \rceil$[1] e $y = 1$. Analizziamo i passi successivi ponendoci in un caso generico, $x = r$ e $y = t$:

1. calcoliamo la somma $S = r^2 - t^2 - n$.

[1]Per definizione, $\forall\, a \in \mathbb{R}, \lceil a \rceil = \inf_{x \in \mathbb{Z}} \{x : x \geqslant a\}$

2. Se $S = 0$ abbiamo trovato gli interi x e y cercati.

3. Se $S \neq 0$ abbiamo due casi:

   - $S > 0$, cioè x prevale su y, quindi aumentiamo y di 1 unità;
   - $S < 0$, quindi è y a prevalere e aumentiamo x;

   quindi ricalcoliamo la somma S, tornando al punto 1.

Per calcolare S non è necessario calcolare ogni volta i quadrati di x e y. Vediamo perché: supponiamo di essere nel caso $x = r$, $y = t$ e che $S = r^2 - t^2 - n$ sia positivo; secondo quanto detto prima dobbiamo aumentare y, da t a $t + 1$, prima di ricalcolare S

$$\begin{aligned} S' &= x^2 - y^2 - n = r^2 - (t+1)^2 - n = \\ &= r^2 - t^2 - 2t - 1 - n = (r^2 - t^2 - n) - 2t - 1 = \\ &= S - 2t - 1. \end{aligned}$$

Con procedimento analogo si trova che se $S < 0$

$$S' = S + 2r + 1.$$

In pratica prima si ricalcola S a partire dai vecchi valori di x o y e in seguito si incrementa uno dei due a seconda del caso in cui mi trovo.

Per i valori di inizializzazione scelti, l'algoritmo (appendice B.3) funziona molto bene quando n è il prodotto di due numeri abbastanza prossimi tra loro. Se i due fattori hanno invece ordine di grandezza diversa, l'algoritmo eseguirà un gran numero di tentativi prima di trovare un divisore; per questo motivo è consigliabile inserire un limite massimo di iterazioni, che consente di interrompere la ricerca dopo un certo numero di fallimenti. L'algoritmo ovviamente raggiunge il limite di iterazioni possibili, prima di dare risposta negativa, se come input inseriamo un numero primo, conviene quindi preliminarmente sottoporre l'intero che si vuole fattorizzare ad un test probabilistico di primalità per essere sicuri che si tratti di un numero composto. Si può ottenere un buon metodo di fattorizzazione combinando la divisione per tentativi e quest'ultima tecnica: preso un intero composto n si tenta di dividerlo per tutti i primi sotto un limite prefissato B, trovando, se ci sono, i fattori di piccola grandezza; a questo punto ci si può aspettare che i fattori rimanenti siano prossimi tra loro e che quindi possano essere individuati con l'algoritmo di Fermat.

Esempio 1.2.1. Proviamo a fattorizzare con questo metodo l'intero $N = 1162201201$. Come detto per prima cosa, proviamo a dividere N per tutti i primi minori di 100 (il nostro B in questo caso). Così facendo troviamo 3 fattori, precisamente:

$$N = 1162201201 = 7 \cdot 23 \cdot 71 \cdot 101671.$$

Ora passiamo ad applicare il metodo di Fermat alla parte resistente, non fattorizzata, $R = 101671$. Inizializziamo i parametri x e y:

$$x = \left\lceil \sqrt{101671} \right\rceil = 319, \quad y = 1.$$

Iterata	x	y	S
1	319	1	89
2	319	2	86
3	319	3	81
4	319	4	74
5	319	5	65
6	319	6	54
7	319	7	41
8	319	8	26
9	319	9	9
10	319	10	-10
11	320	10	629
12	320	11	608
13	320	12	585
14	320	13	560

Iterata	x	y	S
15	320	14	533
16	320	15	504
17	320	16	473
18	320	17	440
19	320	18	405
20	320	19	368
21	320	20	329
22	320	21	288
23	320	22	245
24	320	23	200
25	320	24	153
26	320	25	104
27	320	26	53
28	320	27	0 ✓

Quindi da $R = 101671 = (320 - 27)(320 + 27) = 293 \cdot 347$, otteniamo

$$N = 1162201201 = 7 \cdot 23 \cdot 71 \cdot 293 \cdot 347.$$

1.3 Le idee di Pollard

1.3.1 Il metodo $p - 1$

Consideriamo un intero n e supponiamo che p sia un suo divisore primo. Per il Piccolo Teorema di Fermat, $\forall\ a = 1, 2, \ldots, p-1,\ a^{p-1} \equiv 1 \bmod p$, quindi

$$\forall\, E \text{ tale che } p - 1 \mid E, \quad a^E \equiv 1 \bmod p$$

cioè in questo caso

$$p \mid a^E - 1 \quad \text{e} \quad p \mid n.$$

Calcolando quindi il massimo comune divisore tra n e elementi della forma $a^E - 1$ potremmo trovare un divisore proprio di n. Così a prima vista non sembra di aver trovato un metodo vantaggioso, anzi richiede meno operazioni la divisione per tentativi. Se però supponiamo che n abbia un divisore primo p, tale che $p - 1$ sia scomponibile in fattori di piccole dimensioni, allora elevando un elemento a per un certo E prodotto di piccoli interi, troviamo in pochi passi un divisore di n.

Come esponente E dobbiamo dunque scegliere un prodotto di tanti interi piccoli, diciamo minori di un certo limite B: possiamo considerare quindi

$E = B!$ oppure il minimo comune multiplo di tutti gli interi da 2 a B. Nel primo caso il calcolo è molto semplice però si considerano fattori inutili, sovrabbondanti, nel secondo invece si considerano i fattori indispensabili ma è richiesta una precomputazione. Infatti per calcolare il minimo comune multiplo si può procedere in questo modo: conoscendo $\text{mcm}(1, \ldots, B)$, se $B+1$ è una potenza di un primo, cioè $B+1 = q^\alpha$, allora $\text{mcm}(1, \ldots, B+1) = q \cdot \text{mcm}(1, \ldots, B)$, altrimenti $\text{mcm}(1, \ldots, B) = \text{mcm}(1, \ldots, B+1)$, e per fare ciò bisogna conoscere i primi e le loro potenze (per esempio con una applicazione del crivello di Eratostene).

Scelto come valore di partenza un certo C, consideriamo la successione:

$$a_{i+1} \equiv {a_i}^{i+1} \bmod n, \quad \text{con } a_1 = C,$$

e periodicamente calcoliamo $\text{MCD}(a_i - 1, n)$. Fissato un limite di iterazioni, se raggiunto questo limite il massimo comune divisore è sempre risultato 1, oppure se questo, al k-esimo controllo, risulta essere n, si può provare a cambiare il valore iniziale C e ricominciare da capo.

Esempio 1.3.1. Proviamo a fattorizzare con il metodo $p-1$ di Pollard l'intero $N = 17113$. Partiamo da $a = 5$ e calcoliamo le potenze successive.

i	$a^{i!} - 1 \pmod N$	$(a^{i!} - 1, N)$
1	4	1
2	24	1
3	15624	1
4	728	1
5	11035	1
6	11162	1
7	4077	1
8	1479	1
9	1199	109 ✓

Infatti $N = 17113 = 109 \cdot 157$. Osserviamo inoltre che $p - 1 = 108 = 2^2 \cdot 3^3$ e che quindi 9 è il più piccolo intero per cui il suo fattoriale è divisibile per 108.

1.3.2 Il metodo ρ

Supponiamo anche in questo caso di avere un intero n composto e un suo divisore primo p. L'idea di base è di trovare p, determinando il periodo di una successione ciclica modulo p, infatti se individuiamo una congruenza

$$x_i \equiv x_j \bmod p$$

a questo punto

$$p \mid n \quad \text{e} \quad p \mid x_i - x_j$$

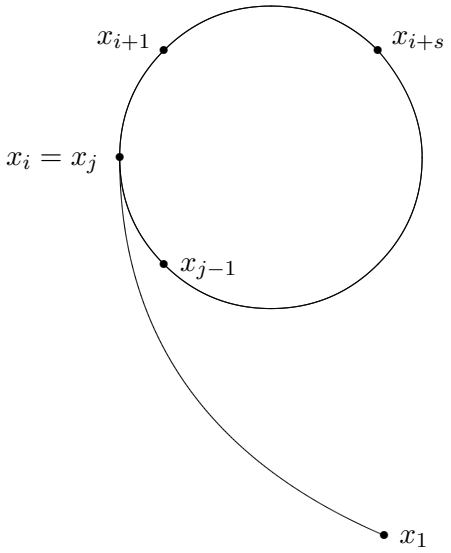

Figura 1.3.1: ρ

e possiamo trovare p calcolando $\text{MCD}(n, x_i - x_j)$.

La prima cosa che ci serve è quindi una successione ciclica. Definita una successione ricorsiva

$$x_i = F(x_{i-1}, \ldots, x_{i-m}) \bmod n,$$

i termini x_k possono assumere al più n valori distinti, dal momento che $\mathbb{Z}_n$ è composto da n elementi. Ogni elemento di una sequenza di lunghezza m può quindi assumere n valori diversi, pertanto ci sono al massimo n^m sequenze diverse composte da m valori, cioè dopo $n^m + 1$ passi troviamo sicuramente due sequenze $\{x_{i-1}, \ldots, x_{i-m}\}$ e $\{x_{j-1}, \ldots, x_{j-m}\}$ identiche. Dato che gli x_k sono definiti a partire dagli m valori precedenti avremo $x_i = x_j$, cioè la successione ricorsiva è ciclica, eventualmente tralasciando alcuni valori iniziali, detti coda della successione. Da qui prende il nome di metodo ρ, in quanto una rappresentazione grafica dei valori ricorda questa lettera greca (figura 1.3.1).

Nel nostro caso prendiamo come successione:

$$x_i \equiv f(x_{i-1}) \bmod n$$

dove $f(t)$ è un polinomio irriducibile, per esempio $f(t) = t^2 + c$. Consideriamo parallelamente la successione

$$y_i \equiv x_i \bmod p, \quad \text{e quindi} \quad y_{i+1} \equiv f(y_i) \bmod p$$

ricordando che $p \mid n$. Se determiniamo un'equivalenza $y_j = y_i$, questa varrà anche per le x,

$$x_j \equiv x_i \bmod p \tag{1.3.1}$$

e ci sono buone possibilità che $x_j \neq x_i$. A questo punto calcolando $\mathrm{MCD}(n, x_j - x_i)$ si determina un fattore non triviale di n.

Il problema è che non conosciamo p, quindi non siamo in grado di determinare un'equivalenza del tipo (1.3.1). Osserviamo però che determinata una coppia di indici (i, j) che verifica la (1.3.1), per ogni intero positivo a, vale la seguente

$$x_{j+a} \equiv x_{i+a} \bmod p. \tag{1.3.2}$$

In pratica chiamato L il periodo della successione, per ogni coppia (i, j) tale che x_i e x_j sono fuori dalla coda, vale la (1.3.2) se $L \mid j - i$, cioè se $j = i + kL$, per un certo k. Per determinare il periodo dobbiamo confrontare ogni elemento con tutti i suoi precedenti, ricerca che con un periodo molto grande risulta inefficiente; pertanto ci limitiamo a valutare la seguente relazione

$$x_{2i} \equiv x_i \bmod p.$$

Questo processo dà esito positivo quando si trova un i multiplo di L: il più piccolo fuori dalla coda risulta essere $k\lceil A/k \rceil$, dove A è la lunghezza dell'antiperiodo (della coda). Non potendo determinare direttamente l'equivalenza procediamo per tentativi e calcoliamo ripetutamente il massimo comune divisore tra n e $x_{2i} - x_i$, non ad ogni passo ma periodicamente; infatti dato che l'algoritmo di Euclide richiede numerosi calcoli, è più conveniente "salvare" le differenze in un prodotto (modulo n) e effettuare il controllo MCD, ad esempio, ogni 50 iterazioni. Come per il metodo $p - 1$, se si ottiene n come massimo comune divisore si può provare a ricominciare la ricerca cambiando il valore iniziale di x.

Esempio 1.3.2. Applichiamo il metodo ρ al numero $N = 22703$, prendendo come primo valore della successione 7 e come polinomio ricorrente $t^2 + 1$.

i	x_i	x_{2i}	$x_{2i} - x_i \pmod N$	$(x_{2i} - x_i, N)$
1	50	2501	2451	1
2	2501	20815	18314	1
3	11677	7574	18600	1
4	20815	21311	496	1
5	174	21336	21162	1
6	7574	11882	4308	1
7	17699	13802	18806	1
8	21311	1480	2872	1
9	7910	16335	8425	1
10	21336	15116	16483	311 ✓

Quindi $N = 22703$ si scompone in due fattori: $73 \cdot 313$.

Capitolo 2

Metodi avanzati

In questo secondo capitolo affronteremo alcune tecniche moderne, tra le più efficaci al momento, in grado di scomporre interi di 50 cifre binarie. L'idea di base è comune e si tratta di un'evoluzione della fattorizzazione di Fermat. Abbiamo visto nella sezione 1.2 che per scomporre n si cercano due interi tali che la differenza dei quadrati valga proprio n. Kraïtchik propose di ridursi all'ambiente $\mathbb{Z}_n$ e di cercare una soluzione della congruenza

$$x^2 \equiv y^2 \bmod n. \tag{2.0.1}$$

Se n è primo, $\mathbb{Z}_n$ è un campo, struttura algebrica nella quale un'equazione di secondo grado ha al più due soluzioni. Pertanto preso un certo $y \in \mathbb{Z}_n$ le soluzioni della (2.0.1) sono $x \equiv \pm y \bmod n$, soluzioni che possiamo definire triviali. Se n è composto invece non abbiamo solo queste due soluzioni, ed è grazie a questa proprietà che la ricerca di soluzioni della (2.0.1) ha interesse nel campo della fattorizzazione. Supponiamo che l'intero n sia prodotto di due primi p e q, con $p \neq q$. Preso un elemento y possiamo risolvere nei due campi $\mathbb{Z}_p$ e $\mathbb{Z}_q$ l'equazione (2.0.1) ottenendo:

$$\begin{aligned} u^2 \equiv y^2 \bmod p \quad &\Longrightarrow \quad u \equiv \pm y \bmod p, \\ v^2 \equiv y^2 \bmod q \quad &\Longrightarrow \quad v \equiv \pm y \bmod q. \end{aligned}$$

L'equazione in $\mathbb{Z}_n$ dà luogo allora a 4 soluzioni, che ridotte modulo p e modulo q, corrispondono alle quattro combinazioni possibili delle soluzioni in $\mathbb{Z}_p$ e $\mathbb{Z}_q$:

1. $$x \equiv y \bmod n \quad \Longrightarrow \quad \begin{cases} u \equiv y \bmod p \\ v \equiv y \bmod q \end{cases}$$
2. $$x \equiv -y \bmod n \quad \Longrightarrow \quad \begin{cases} u \equiv -y \bmod p \\ v \equiv -y \bmod q \end{cases}$$

3.
$$x \equiv z \bmod n \quad \Longrightarrow \quad \begin{cases} u \equiv y \bmod p \\ v \equiv -y \bmod q \end{cases}$$

4.
$$x \equiv -z \bmod n \quad \Longrightarrow \quad \begin{cases} u \equiv -y \bmod p \\ v \equiv y \bmod q \end{cases}$$

Una volta risolta l'equazione (2.0.1), non abbiamo garanzie di successo nella fattorizzazione, però sappiamo che

$$x^2 - y^2 = (x+y)(x-y) \equiv 0 \bmod n, \quad \text{cioè } n \,|\, (x+y)(x-y).$$

A questo punto abbiamo il 50% di possibilità che i fattori primi di n siano distribuiti nei due fattori $x+y$ e $x-y$. Se si verifica questa evenienza troviamo due fattori calcolando $(x+y, n)$ e $(x-y, n)$. Questo non accade sempre, perché $x-y$ e $x+y$ potrebbero risultare dei fattori triviali. I tre metodi seguenti rappresentano tre proposte alternative per la ricerca di soluzioni della (2.0.1).

2.1 SQUFOF

Il metodo SQUFOF (SQUare FOrms Factorization) ideato da D. Shanks si serve delle frazioni continue. Nell'appendice A vengono spiegati gli elementi basilari che permettono di dimostrare la formula su cui si basa il metodo. Considerando l'espansione della frazione continua di $\sqrt{n}$ abbiamo che:

$$p_{k-1}^2 - n\, q_{k-1}^2 = (-1)^k\, Q_k, \tag{2.1.1}$$

dove p_j e q_j sono rispettivamente numeratore e denominatore del convergente j-esimo di $\sqrt{n}$, mentre Q_k viene calcolato mediante l'algoritmo di Lagrange (teorema A.0.3). Passando in modulo n, otteniamo

$$p_{k-1}^2 \equiv (-1)^k\, Q_k \pmod{n}. \tag{2.1.2}$$

Si intuisce che se riuscissimo a trovare un indice k pari tale per cui Q_k sia un quadrato, cioè $Q_k = S^2$, avremmo trovato una soluzione della (2.0.1), infatti

$$p_{k-1}^2 \equiv Q_k \pmod{n}, \text{ cioè } p_{k-1}^2 \equiv S^2 \pmod{n},$$

che ci permette di cercare un divisore di n, calcolando $(p_{k-1} - S, n)$ o $(p_{k-1} + S, n)$. Anche in questo caso nell'algoritmo (la cui implementazione si trova nell'appendice B.6) è consigliabile inserire un limite massimo di tentativi in modo da poter interrompere l'esecuzione dopo un certo numero di fallimenti.

Esempio 2.1.1. Proviamo a scomporre con SQUFOF l'intero $N = 2623$.

k	p_{k-1}	Q_k	$p_{k-1} - S$	$(p_{k-1} - S, N)$
2	205	57	—	—
4	461	58	—	—
6	3329	66	—	—
8	11421	74	—	—
10	521832	41	—	—
12	2640094	13	—	—
14	41719672	9	41719669	61 ✓

$$N = 2623 = 43 \cdot 61.$$

2.2 CFRAC

Il metodo CFRAC (Continued FRACtion Algorithm) ideato da Morrison e Brillhart riprende l'idea di Shanks e ne propone un'importante evoluzione. Con l'algoritmo SQUFOF, per cercare un fattore di un intero n, dovevamo sperare di trovare un elemento della successione dei Q_k di posto pari che fosse un quadrato perfetto. Morrison e Brillhart pensarono di cercare un quadrato combinando insieme più elementi della successione. Supponiamo per esempio che per gli indici i e j

$$\left[(-1)^i\, Q_i\right] \cdot \left[(-1)^j\, Q_j\right] = (-1)^{i+j}\, Q_i\, Q_j = S^2 \pmod{n},$$

abbiamo allora trovato una soluzione della (2.0.1), infatti

$$(p_{i-1}^2) \cdot (p_{j-1}^2) = (p_{i-1}\, p_{j-1})^2 \equiv S^2 \pmod{n}.$$

Come possiamo determinare una combinazione di questo tipo? Innanzitutto consideriamo una sistema di primi β_i, ponendo $\beta_0 = -1$. Questo insieme diventa una base di fattori che supponiamo avere cardinalità t.

$$\mathcal{B} = \{\beta_0, \beta_1, \ldots, \beta_i, \ldots\}, \qquad \left|\, \mathcal{B} \,\right| = t.$$

Tra tutti i primi, possiamo limitarci a quelli per cui il simbolo di Legendre $\left(\frac{n}{\beta}\right)$ vale 1 (o eventualmente $\left(\frac{n}{\beta}\right) = 0$, ma in questo caso $\beta \,|\, n$ e possiamo escludere questo caso supponendo di aver sottoposto preliminarmente n alla divisione per tentativi fino ad un certo limite L, più grande dei primi della base), infatti se $P \,|\, Q_k$ la (2.1.1) diventa:

$$p_{k-1}^2 - n\, q_{k-1}^2 \equiv 0 \pmod{\beta},$$

da cui

$$n \equiv \left(\frac{p_{k-1}}{q_{k-1}}\right)^2 \pmod{\beta}.$$

A questo punto con l'algoritmo A.0.3 generiamo i Q_k e cerchiamo di scomporli sulla base $\mathcal{B}$. A noi interessano quelli che si fattorizzano completamente; in tal caso possiamo identificare $(-1)^k Q_k$ con un vettore di lunghezza t contenente gli esponenti della scomposizione in fattori su $\mathcal{B}$.

$$(-1)^k Q_k = (-1)^k \prod_{i=1}^{t-1} \beta_i^{v_{k\,i}} \longleftrightarrow \boldsymbol{v}_k = (v_{k\,0} = k, v_{k\,1}, \ldots, v_{k\,(t-1)}).$$

Per stabilire se un elemento è un quadrato è sufficiente sapere se gli esponenti, cioè gli elementi del vettore $\boldsymbol{v}$ sono pari. Pertanto consideriamo al posto di $\boldsymbol{v}$ il suo ridotto in $\mathbb{Z}_2$:

$$\boldsymbol{\nu} = (v_0 \bmod 2, v_1 \bmod 2, \ldots, v_{t-1} \bmod 2) = (\nu_0, \nu_1, \ldots, \nu_{t-1}).$$

Il nostro obiettivo diventa quindi cercare una dipendenza lineare, cioè un insieme di vettori $\{\boldsymbol{\nu}_1, \boldsymbol{\nu}_2, \ldots, \boldsymbol{\nu}_h\}$ tali che

$$\sum_{j=1}^{h} \boldsymbol{\nu}_j = \boldsymbol{\nu}_1 + \boldsymbol{\nu}_2 + \cdots + \boldsymbol{\nu}_h = (0, 0, \ldots, 0).$$

In questo caso considerando i Q_j abbiamo

$$\prod_{j=1}^{h} Q_j = \prod_{j=1}^{h} \left(\prod_{i=0}^{t-1} \beta_i^{v_{j\,i}} \right) = \prod_{i=0}^{t-1} \beta_i^{\sum_{j=1}^{h} (v_{j\,i})}$$

ed inoltre sappiamo che

$$\forall\ i = 0, \ldots, t-1,\ \sum_{j=1}^{h} v_{j\,i} \pmod 2 = \left(\sum_{j=1}^{h} \boldsymbol{\nu}_j \right)_i = \sum_{j=1}^{h} \nu_{i\,j} = 0 \pmod 2.$$

Quindi in conclusione il prodotto dei Q_j è un quadrato perfetto:

$$\prod_{j=1}^{h} Q_j = y^2.$$

Andando a considerare i numeratori dei convergenti p_{j-1} corrispondenti abbiamo che

$$\prod_{j=1}^{h} p_{j-1}^2 = \left(\prod_{j=1}^{h} p_{j-1} \right)^2 = x^2 \equiv y^2 \pmod n.$$

Per essere sicuri di generare una dipendenza lineare è sufficiente trovare $t+1$ elementi della successione dei Q_k che fattorizzano su $\mathcal{B}$ perché $t+1$ vettori su una base di dimensione t sono sicuramente linearmente indipendenti.

Esempio 2.2.1. Proviamo a fattorizzare nuovamente $N = 2623$ con questo metodo (implementato nell'appendice B.7). Per prima cosa cerchiamo una base di primi β_i tali che il simbolo di Legendre $\left(\frac{N}{\beta_i}\right)$ valga 1, ottenendo

$$\mathcal{B} = \{-1, 2, 3, 11, 13\}.$$

A questo punto cominciamo a generare i Q_k. Per essere sicuri di trovare una dipendenza lineare dobbiamo fattorizzare completamente 6 elementi dal momento che $\left|\mathcal{B}\right| = 5$.

k	p_{k-1}	Q_k	fattorizzazione
1	51	22	$2 \cdot 11$
3	256	39	$3 \cdot 13$
6	3329	66	$2 \cdot 3 \cdot 11$
7	4046	27	3^3
9	15467	3	3
11	1059131	39	$3 \cdot 13$

Consideriamo i vettori degli esponenti ridotti modulo 2:

	-1	2	3	11	13
$\boldsymbol{\nu}_1$	1	1	0	1	0
$\boldsymbol{\nu}_3$	1	0	1	0	1
$\boldsymbol{\nu}_6$	0	1	1	1	0
$\boldsymbol{\nu}_7$	1	0	1	0	0
$\boldsymbol{\nu}_9$	1	0	1	0	0
$\boldsymbol{\nu}_{11}$	1	0	1	0	1

Si riconoscono alcune combinazioni linearmente dipendenti: $\boldsymbol{\nu}_1+\boldsymbol{\nu}_6+\boldsymbol{\nu}_7 = \mathbf{0}$, $\boldsymbol{\nu}_1 + \boldsymbol{\nu}_6 + \boldsymbol{\nu}_9 = \mathbf{0}$, $\boldsymbol{\nu}_7 + \boldsymbol{\nu}_9 = \mathbf{0}$, $\boldsymbol{\nu}_3 + \boldsymbol{\nu}_{11} = \mathbf{0}$. Proviamo a fattorizzare N a partire da $\boldsymbol{\nu}_1 + \boldsymbol{\nu}_6 + \boldsymbol{\nu}_9 = \mathbf{0}$:

$$\begin{aligned} x^2 &= (p_0 \cdot p_5 \cdot p_9)^2, \qquad x = 51 \cdot 3329 \cdot 15467 = 2625971793, \\ y^2 &= Q_1 \cdot Q_6 \cdot Q_9 = 22 \cdot 66 \cdot 3 = (2 \cdot 3 \cdot 11)^2, \qquad y = 66. \end{aligned}$$

$$(x - y, N) = (2625971727, 2623) = (2623, 2491) = 1.$$

Questa combinazione non dà esito positivo, proviamo allora con $\boldsymbol{\nu}_7+\boldsymbol{\nu}_9 = \mathbf{0}$:

$$\begin{aligned} x^2 &= (p_6 \cdot p_8)^2, \qquad x = 4046 \cdot 15467 = 62579482, \\ y^2 &= Q_7 \cdot Q_9 = 27 \cdot 3 = 3^4, \qquad y = 9. \end{aligned}$$

$$(x - y, N) = (62579473, 2623) = (2623, 2562) = 61.$$

Facendo un confronto con l'esempio 2.1.1 ci accorgiamo della maggiore efficienza del metodo CFRAC. Con il metodo SQUFOF infatti abbiamo dovuto attendere la quattordicesima iterazione prima di trovare un fattore di N mentre adesso è stata sufficiente l'undicesima (e alla nona, la combinazione usata era già disponibile).

2.3 Il crivello quadratico

Il crivello quadratico (Quadratic Sieve) ideato da Pomerance è al momento tra i metodi più performanti nel campo della fattorizzazione. Esso ha alcuni aspetti comuni al CFRAC, infatti anche in questo caso avremo una base di fattori sulla quale cercheremo degli elementi completamente scomponibili per determinare una dipendenza lineare. La differenza riguarda invece il sistema di ricerca degli elementi scomponibili: nel CFRAC si procedeva con la divisione per tentativi, procedimento che però con interi molto grandi risulta eccessivamente dispendioso. Pomerance propone di utilizzare, per ottenere la scomposizione degli interi, un crivello che non richiede il calcolo di divisioni.

2.3.1 Il crivello

Per prima cosa, definito $A = \lfloor \sqrt{n} \rfloor$[1], consideriamo gli interi

$$Q(k) = (A+k)^2 - n = r^2 - n,$$

dove n è l'intero che cerchiamo di fattorizzare e $k = 1, 2, \ldots$. Occupiamoci quindi della base di fattori. Anche in questo caso, supponendo di aver sottoposto n alla divisione per tentativi per i primi al di sotto di un certo limite, consideriamo come elementi della base i primi p_i tali che $\left(\frac{n}{p_i}\right) = 1$. Infatti se $p_i \mid Q(k)$

$$r \cdot r - n \equiv 0 \pmod{p_i} \implies n \equiv r^2 \pmod{p_i}.$$

$$\mathcal{B} = \{p_1, p_2, \ldots, p_t\}, \quad \forall\, i = 1, 2, \ldots, t, \ \left(\frac{n}{p_i}\right) = 1$$

Se n è un residuo quadratico di p_i, quindi esistono due elementi z e $-z$ tali che

$$n \equiv z^2 \pmod{p_i} \text{ oppure } n \equiv (-z)^2 \pmod{p_i},$$

cioè r è congruo a z o a $-z$. La proprietà che risulta ancora più importante in questo contesto però è l'inversa:

$$A + k = r \equiv \pm z \pmod{p_i} \implies p_i \mid Q(k).$$

Quindi preso un elemento della base di fattori p, risolvendo la congruenza

$$z^2 \equiv n \pmod{p}, \tag{2.3.1}$$

abbiamo un criterio per stabilire se p divide $Q(k)$ senza dover effettuare il tentativo di divisione. Ovviamente p potrebbe dividere $Q(k)$ più di una volta, pertanto dobbiamo risolvere anche le congruenze del tipo

$$z^2 \equiv n \pmod{p^e}.$$

[1] Per definizione, $\forall\, a \in \mathbb{R}, \lfloor a \rfloor = \sup_{x \in \mathbb{Z}} \{x : x \leqslant a\}$

L'idea di Pomerance ha una seconda grande qualità, infatti determinato un k tale che $p \mid Q(k)$, possiamo conoscere tutti gli altri $Q(h)$ divisibili per p senza applicare il criterio trovato con la congruenza (2.3.1). Supponendo di aver determinato k tale che $Q(k) \equiv 0 \pmod{p}$, abbiamo che

$$\begin{aligned} Q(k + c\,p) &= (A + k + c\,p)^2 - n = \\ &= (A + k)^2 - n + 2c\,(A + k)\,p + c^2\,p^2 \equiv Q(k) \pmod{p}, \end{aligned}$$

quindi anche tutti i $Q(h)$, tali che $h - k$ è un multiplo di p, sono divisibili per p.

Esempio 2.3.1. Applichiamo ad un esempio quanto detto finora. Consideriamo $n = 9361$ e $p = 29$. Per prima cosa dobbiamo verificare che n sia un residuo quadratico ed infatti $\left(\frac{n}{p}\right) = \left(\frac{9361}{29}\right) = 1$. A questo punto risolviamo la congruenza (2.3.1), ottenendo

$$z^2 = 9361 \pmod{29} \quad \Longrightarrow \quad z = 9,\ z = 20.$$

In questo caso $A = \lfloor \sqrt{9361} \rfloor = 96$, quindi cominciamo la ricerca da $r = A + 1 = 97$:

$$r = 97 \equiv 10 \pmod{29} \quad \Longrightarrow \quad 29 \nmid Q(1).$$

Il primo k che soddisfa la congruenza è 11:

$$r = A + 11 = 107 \equiv 20 \pmod{29} \implies 29 \mid Q(11) = 2088, \quad 2088 = 72 \cdot 29.$$

Verifichiamo ancora che anche $Q(11 + 29) = Q(40)$ è divisibile per 29:

$$Q(11 + 29) = Q(40) = 9135 = 315 \cdot 29.$$

2.3.2 La dipendenza lineare

Per ottenere gli elementi completamente fattorizzati generiamo una successione di $Q(k)$, applichiamo il crivello quadratico, quindi scartiamo gli interi non scomposti su $\mathcal{B}$ ed infine associamo i restanti ad un vettore:

$$Q(k) = p_1^{v_{k\,1}} \cdot p_2^{v_{k\,2}} \cdot \cdots \cdot p_t^{v_{k\,t}} \ \longleftrightarrow \ \boldsymbol{v}_k = (v_{k\,1}, v_{k\,2}, \ldots, v_{k\,t}).$$

Come per il metodo CFRAC, ci interessa il vettore ridotto modulo 2:

$$\boldsymbol{\nu}_k = (\nu_{k\,1} = v_{k\,1} \bmod 2, \nu_{k\,2} = v_{k\,2} \bmod 2, \ldots, \nu_{k\,t} = v_{k\,t} \bmod 2).$$

Arrivati a questo punto ci troviamo con $t+1$ vettori su una base di cardinalità t, in pratica abbiamo una matrice con $t + 1$ righe e t colonne:

$$\boldsymbol{\Lambda} = \begin{vmatrix} \nu_{1\,1} & \nu_{1\,2} & \cdots & \nu_{1\,t} \\ \nu_{2\,1} & \nu_{2\,2} & \cdots & \nu_{2\,t} \\ \vdots & \vdots & \ddots & \vdots \\ \nu_{(t+1)\,1} & \nu_{(t+1)\,2} & \cdots & \nu_{(t+1)\,t} \end{vmatrix}$$

Determinare una dipendenza lineare significa ottenere una riga composta da soli 0, scambiando e sommando tra loro righe della matrice. A noi interessa anche sapere quali vettori hanno dato luogo al vettore nullo e per fare ciò ci serviamo del metodo delle eliminazioni successive di Gauss. Per prima cosa aggiungiamo alla matrice $\mathbf{\Lambda}$ una matrice identità di dimensione $t+1 \times t+1$:

$$\left|\begin{array}{cccc|cccc} \nu_{1\,1} & \nu_{1\,2} & \dots & \nu_{1\,t} & 1 & 0 & \dots & 0 \\ \nu_{2\,1} & \nu_{2\,2} & \dots & \nu_{2\,t} & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \nu_{(t+1)\,1} & \nu_{(t+1)\,2} & \dots & \nu_{(t+1)\,t} & 0 & 0 & \dots & 1 \end{array}\right|.$$

A questo punto cominciamo con le eliminazioni successive: partendo dalla prima colonna, scorriamo i vettori cercando una prima componente non nulla. Supponiamo di averla trovata nella j-esima riga. A questo punto effettuiamo uno scambio di righe tra la prima e la j-esima, quindi riprendiamo l'esame della colonna ed ogni qual volta incontriamo una componente uguale a 1, sostituiamo tale riga con la stessa sommata con la prima riga (aggiornata). Alla fine della scansione tutte le prime componenti saranno nulle eccetto che per il primo vettore. Passiamo quindi alla seconda colonna ripetendo la ricerca della componente non nulla a partire però dalla seconda riga. Iteriamo questo procedimento per le t colonne ed alla fine avremo sicuramente generato una dipendenza lineare. Per sapere quali vettori sono risultati linearmente dipendenti, non dobbiamo far altro che osservare la riga corrispondente al vettore nullo nella matrice identità. Infatti sommando la j-esima riga con la m-esima, viene generata una riga con componente pari a 1 nelle posizioni j ed m e componente 0 altrove, quindi nella riga affiancata al vettore nullo avrò componente 1 nelle posizioni corrispondenti ai vettori linearmente dipendenti.

Esempio 2.3.2. Chiariamo il procedimento con un esempio. Cerchiamo di determinare una dipendenza lineare tra i vettori $\boldsymbol{\nu}_1 = (0,0,1)$, $\boldsymbol{\nu}_2 = (1,0,1)$, $\boldsymbol{\nu}_3 = (1,1,0)$ e $\boldsymbol{\nu}_4 = (0,1,1)$.

$$\left|\begin{array}{ccc|cccc} 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right| \quad 1 \leftrightharpoons 2 \quad \left|\begin{array}{ccc|cccc} 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right|$$

$$\left|\begin{array}{ccc|cccc} 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right| \quad 3 \leftarrow 1+3 \quad \left|\begin{array}{ccc|cccc} 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right|$$

$$\left|\begin{array}{ccc|cccc} 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right| \quad 2 \leftrightharpoons 3 \quad \left|\begin{array}{ccc|cccc} 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 \end{array}\right|$$

$$\left|\begin{array}{cccccccc} 1 & 0 & 1 & & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & & 0 & 0 & 0 & 1 \end{array}\right| \quad 4 \leftarrow 2+4 \quad \left|\begin{array}{cccccccc} 1 & 0 & 1 & & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & & 0 & 1 & 1 & 1 \end{array}\right|$$

Nell'ultima riga abbiamo a sinistra il vettore nullo e a destra la componente 1 nelle posizioni 2,3,4 da cui ricaviamo che

$$\boldsymbol{\nu}_2 + \boldsymbol{\nu}_3 + \boldsymbol{\nu}_4 = (0,0,0).$$

Una volta determinato l'insieme di vettori linearmente dipendenti

$$\sum_{j=1}^{q} \boldsymbol{\nu}_{k_j} = (0,\ldots,0),$$

definiamo

$$x^2 = r_{k_1}^2 \cdot r_{k_2}^2 \cdot \cdots \cdot r_{k_q}^2 = \left(\prod_{j=1}^{q} r_{k_j}\right)^2, \quad \text{con } r_{k_j} = A + k_j,$$

$$y^2 = Q(k_1) \cdot Q(k_2) \cdot \cdots \cdot Q(k_q) = \prod_{j=1}^{q} Q(k_j) =$$

$$= \prod_{j=1}^{q} \left(\prod_{i=1}^{t} p_i^{v_{j\,i}}\right) = \prod_{i=1}^{t} p_i^{\sum_{j=1}^{q}(v_{j\,i})} = \left(\prod_{i=1}^{t} p_i^{\frac{\sum_{j=1}^{q}(v_{j\,i})}{2}}\right)^2,$$

determinando una soluzione della (2.0.1) con

$$x = \prod_{j=1}^{q} r_{k_j} \quad \text{e} \quad y = \prod_{i=1}^{t} p_i^{\frac{\sum_{j=1}^{q}(v_{j\,i})}{2}}.$$

In conclusione possiamo quindi cercare un fattore di n calcolando $(x-y, n)$.

Esempio 2.3.3. Applichiamo il crivello quadratico (implementato nell'appendice B.8) a $N = 6901$. Controlliamo preliminarmente che non abbia divisori minori di 50. A questo punto determiniamo una base di 4 fattori andando a valutare il simbolo di Legendre:

$$\mathcal{B} = \{2,3,5,11\},$$

e inoltre risolviamo la congruenza (2.3.1) per ogni p_i.

$$\begin{array}{llll} z^2 \equiv N \pmod{p_1} = 1 \pmod 2 & \Longrightarrow & z = 1 \\ z^2 \equiv N \pmod{p_2} = 1 \pmod 3 & \Longrightarrow & z = 1,2 \\ z^2 \equiv N \pmod{p_3} = 1 \pmod 5 & \Longrightarrow & z = 1,4 \\ z^2 \equiv N \pmod{p_4} = 4 \pmod{11} & \Longrightarrow & z = 2,9 \end{array}$$

Dopo aver generato la successione dei $Q(k)$ partendo da $A = \lfloor\sqrt{6901}\rfloor = 83$, applichiamo il crivello fino ad ottenere 5 fattorizzazioni complete:

k	$r_k = A + k$	$r_k \bmod 2$	$r_k \bmod 3$	$r_k \bmod 5$	$r_k \bmod 11$
3	86	0	2	1	9
18	101	1	2	1	2
36	119	1	2	4	9
216	299	1	2	4	2
326	409	1	1	4	2

k	$Q(k)$	scomposizione	$\boldsymbol{v}_k$	$\boldsymbol{\nu}_k$
3	495	$3^2 \cdot 5 \cdot 11$	$(0,2,1,1)$	$(0,0,1,1)$
18	3300	$2^2 \cdot 3 \cdot 5^2 \cdot 11$	$(2,1,2,1)$	$(0,1,0,1)$
36	7260	$2^2 \cdot 3 \cdot 5 \cdot 11^2$	$(2,1,1,2)$	$(0,1,1,0)$
216	82500	$2^2 \cdot 3 \cdot 5^4 \cdot 11$	$(2,1,4,1)$	$(0,1,0,1)$
326	160380	$2^2 \cdot 3^6 \cdot 5 \cdot 11$	$(2,6,1,1)$	$(0,0,1,1)$

Costruiamo ora la matrice per determinare i vettori linearmente dipendenti:

$$\left|\begin{array}{ccccccccc} 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right|.$$

Applicando le trasformazioni consentite otteniamo la matrice

$$\left|\begin{array}{ccccccccc} 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{array}\right|$$

da cui ricaviamo 3 combinazioni linearmente dipendenti

$$\boldsymbol{\nu}_3 + \boldsymbol{\nu}_{18} + \boldsymbol{\nu}_{36} = \boldsymbol{\nu}_{18} + \boldsymbol{\nu}_{216} = \boldsymbol{\nu}_3 + \boldsymbol{\nu}_{326} = (0,0,0,0).$$

Analizziamo i tre casi separatamente:

- $\boldsymbol{\nu}_3 + \boldsymbol{\nu}_{18} + \boldsymbol{\nu}_{36} = (0,0,0,0)$. Calcoliamo per prima cosa x:

  $$x = r_3 \cdot r_{18} \cdot r_{36} = 86 \cdot 101 \cdot 119 = 1033634.$$

  Passiamo ora ad y, sommando per prima cosa i vettori $\boldsymbol{v}_k$ coinvolti

  $$\boldsymbol{v}_3 + \boldsymbol{v}_{18} + \boldsymbol{v}_{36} = (4,4,4,4) = 2\,(2,2,2,2),$$

  da cui

  $$y = 2^2 \cdot 3^2 \cdot 5^2 \cdot 11^2 = 108900.$$

  Siamo pronti per vedere se questa combinazione ci fornisce un fattore di 6901:

  $$(x - y, N) = (924734, 6901) = 6901,$$

  pertanto il tentativo è fallito.

- $\boldsymbol{\nu}_3 + \boldsymbol{\nu}_{326} = (0,0,0,0)$. Ripetiamo le stesse operazioni:

$$x = r_3 \cdot r_{326} = 86 \cdot 409 = 35174,$$

$$\boldsymbol{v}_3 + \boldsymbol{v}_{326} = (2,8,2,2) = 2\,(1,4,1,1),$$

$$y = 2 \cdot 3^4 \cdot 5 \cdot 11 = 8901,$$

$$(x - y, N) = (26273, 6901) = 67.$$

- $\boldsymbol{\nu}_{18} + \boldsymbol{\nu}_{216} = (0,0,0,0)$

$$x = r_{18} \cdot r_{216} = 101 \cdot 299 = 30199,$$

$$\boldsymbol{v}_{18} + \boldsymbol{v}_{216} = (4,2,6,2) = 2\,(2,1,3,1),$$

$$y = 2^2 \cdot 3 \cdot 5^3 \cdot 11 = 16500,$$

$$(x - y, N) = (13699, 6901) = 103.$$

Siamo quindi riusciti a scomporre in fattori $N = 6901 = 67 \cdot 103$.

Appendice A

Frazioni continue: nozioni di base

Definizione A.0.1. Si definisce frazione continua l'espressione

$$a_0 + \cfrac{b_0}{a_1 + \cfrac{b_1}{a_2 + \cfrac{b_2}{a_3 + \cdots}}}, \text{ con } a_i, b_i \in \mathbb{C}.$$

Se $\forall\ i,\ b_i = 1$, si parla di frazioni continue semplici e si scrive

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cdots}}} = [a_0, a_1, a_2, a_3, \ldots].$$

Infine una frazione continua semplice viene detta aritmetica se $\forall\ i,\ a_i \in \mathbb{Z}$ e viene indicata con f.c.a..

Esempio A.0.4. Consideriamo un elemento $\frac{a}{b} \in \mathbb{Q},\ a, b \in \mathbb{Z}$ e sviluppiamone la frazione continua. Per prima cosa effettuiamo la divisione con resto di a rispetto a b:

$$a = q_0\, b + r_0 \ \Rightarrow\ \frac{a}{b} = q_0 + \frac{r_0}{b} = q_0 + \frac{1}{\frac{b}{r_0}}.$$

Adesso ripetiamo la divisione con resto con b e r_0, ottenendo

$$b = q_1\, r_0 + r_1 \ \Rightarrow\ \frac{b}{r_0} = q_1 + \frac{1}{\frac{r_0}{r_1}} \ \Rightarrow\ \frac{a}{b} = q_0 + \cfrac{1}{q_1 + \cfrac{1}{\frac{r_0}{r_1}}}.$$

Si prosegue in questo modo fino a quando non si trova un resto nullo e il risultato finale è

$$\frac{a}{b} = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{\ldots + \cfrac{1}{q_n}}}} = [q_0, q_1, q_2, \ldots, q_n].$$

Esempio A.0.5. Nel caso di un elemento irrazionale $\alpha \in \mathbb{R} \setminus \mathbb{Q}$

$$\alpha = [\alpha] + (\alpha - [\alpha]) = [\alpha] + \cfrac{1}{\cfrac{1}{\alpha - [\alpha]}}.$$

Definiamo quindi $\alpha_1 = \frac{1}{\alpha - [\alpha]}$ osservando che $\alpha_1 \in \mathbb{R} \setminus \mathbb{Q}$ e $\alpha_1 > 1$. Possiamo quindi sviluppare α_1 come abbiamo sviluppato α. Generallizzando, al passo k-esimo, otteniamo

$$\begin{aligned}
\alpha_{k-1} &= [\alpha_{k-1}] + \cfrac{1}{\cfrac{1}{\alpha_{k-1} - [\alpha_{k-1}]}}, \\
\alpha_k &= \frac{1}{\alpha_{k-1} - [\alpha_{k-1}]}, \\
a_k &= [\alpha_k].
\end{aligned}$$

Si intuisce che la f.c.a. di un numero irrazionale è infinita, quindi

$$\alpha = [a_0, a_1, \ldots, a_k, \ldots].$$

Se però ci accontentiamo di una frazione continua semplice, possiamo ottenere un'espansione finita fermandoci al k-esimo passo

$$\alpha = [a_0, a_1, \ldots, a_{k-1}, \alpha_k].$$

Definizione A.0.2. Si definisce k-esimo convergente di α la frazione ottenuta considerando i primi a_k termini dell'espansione in frazione continua.

$$c_k = [a_0, a_1, \ldots, a_{k-1}, a_k]$$

Teorema A.0.1. *Consideriamo una f.c.a.* $a = [a_0, a_1, \ldots, a_k, \ldots]$. *Allora*

$$\forall\ k, \quad c_k = \frac{p_k}{q_k},$$

dove $p_0 = a_0$, $q_0 = 1$, $p_1 = a_0\, a_1 + 1$, $q_1 = a_1$ *e* $\forall\ k > 1$*:*

$$p_k = a_k\, p_{k-1} + p_{k-2}, \qquad q_k = a_k\, q_{k-1} + q_{k-2}.$$

Dimostrazione. Per dimostrare il teorema, operiamo per induzione sulla lunghezza dell'espressione $c_k = [a_o, a_1, \ldots, a_k]$. Verifichiamo che vale per c_0 (lunghezza 1) e c_1 (lunghezza 2):

$$c_0 = [a_0] = a_0 = \frac{a_0}{1} = \frac{p_0}{q_0},$$
$$c_1 = [a_0, a_1] = a_0 + \frac{1}{a_1} = \frac{a_0\, a_1 + 1}{a_1} = \frac{p_1}{q_1}.$$

Ora supponiamo valga per espressioni di lunghezza $k+1$ e verifichiamo che allora vale anche per $k+2$.

$$c_{k+1} = [a_0, a_1, \ldots, a_k, a_{k+1}] = \left[a_0, a_1, \ldots, a_k + \frac{1}{a_{k+1}}\right].$$

La seconda espressione ha lunghezza $k+1$ quindi per l'ipotesi induttiva ho che

$$\left[a_0, a_1, \ldots, a_k + \frac{1}{a_{k+1}}\right] = \frac{P_k}{Q_k} = \frac{A_k\, P_{k-1} + P_{k-2}}{A_k\, Q_{k-1} + Q_{k-2}}$$

dove $A_k = a_k + \frac{1}{a_{k+1}}$. Inoltre, essendo invariati tutti gli elementi precedenti A_k ho che $P_{k-1} = p_{k-1}$, $Q_{k-1} = q_{k-1}$, $P_{k-2} = p_{k-2}$ e $Q_{k-2} = q_{k-2}$. Quindi

$$c_{k+1} = \frac{(a_k + \frac{1}{a_{k+1}})p_{k-1} + p_{k-2}}{(a_k + \frac{1}{a_{k+1}})q_{k-1} + q_{k-2}} = \frac{(a_k\, a_{k+1} + 1)p_{k-1} + a_{k+1}\, p_{k-2}}{(a_k\, a_{k+1} + 1)q_{k-1} + a_{k+1}\, q_{k-2}} =$$
$$= \frac{a_{k+1}(a_k\, p_{k-1} + p_{k-2}) + p_{k-1}}{a_{k+1}(a_k\, q_{k-1} + q_{k-2}) + q_{k-1}} = \frac{a_{k+1}\, p_k + p_{k-1}}{a_{k+1}\, q_k + q_{k-1}} = \frac{p_{k+1}}{q_{k+1}}. \qquad \square$$

Proprietà A.0.2. *Consideriamo una f.c.a. $a = [a_0, a_1, \ldots]$ ed i numeratori e denominatori dei suoi convergenti p_k e q_k. Allora*

$$p_k\, q_{k-1} - q_k\, p_{k-1} = (-1)^{k-1}.$$

Dimostrazione. Procediamo anche in questo caso per induzione. Per $k = 1$

$$p_1\, q_0 - q_1\, p_0 = (a_1\, a_0 + 1) \cdot 1 - a_1 \cdot a_0 = 1^0 = 1^{1-1}.$$

Supponiamolo ora vero per un k generico e verifichiamolo per $k+1$:

$$p_{k+1}\, q_k - q_{k+1}\, p_k = (a_{k+1}\, p_k + p_{k-1})\, q_k - (a_{k+1}\, q_k + q_{k-1})\, p_k =$$
$$= p_{k-1}\, q_k - q_{k-1}\, p_k = -(p_k\, q_{k-1} - q_k\, p_{k-1}) =$$
$$= -(-1)^{k-1} = (-1)^k. \qquad \square$$

Per quanto riguarda la fattorizzazione ed i metodi SQUFOF e CFRAC, sono importanti le espansioni in frazioni continue aritmetiche delle radici quadrate. Vediamo qui un algoritmo per calcolare i termini della f.c.a. di una irrazionalità quadratica.

Teorema A.0.3 (Algoritmo di Lagrange)**.** *Consideriamo*

$$\alpha = \frac{P + \sqrt{d}}{Q}, \text{ con } Q, P, d \in \mathbb{Z},\ Q \neq 0,\ \sqrt{d} \notin \mathbb{Z},\ Q \mid d - P^2.$$

Non è restrittivo suppore che Q divida $d - P^2$ perché ogni espressione $\frac{a+\sqrt{b}}{c}$ può essere ricondotta ad un'altra che verifica questa condizione.

$$\frac{a + \sqrt{b}}{c} \longrightarrow \frac{a\,|c| + \sqrt{b\,c^2}}{c\,|c|},\ c\,|c| \ \Big|\ b\,c^2 - (a\,|c|)^2.$$

Si può calcolare la frazione continua di $\alpha = [a_0, a_1, \ldots, a_k, \ldots]$, partendo da $P_0 = P$ e $Q_0 = Q$, con il seguente algoritmo, al k-esimo passo:

$$\begin{aligned} \alpha_k &= \frac{P_k + \sqrt{d}}{Q_k}, \\ a_k &= [\alpha_k], \\ P_{k+1} &= a_k\,Q_k - P_k, \\ Q_{k+1} &= \frac{d - {P_{k+1}}^2}{Q_k}. \end{aligned}$$

Dimostrazione. Innanzitutto dobbiamo verificare che i Q_k sono tutti interi non nulli che dividono $d - {P_k}^2$. Operiamo induttivamente su k prendendo come base dell'induzione Q_0, scelto proprio in questo modo. Supponiamo che Q_k verifichi le condizioni e vediamo che allora valgono anche per Q_{k+1}.

$$Q_{k+1} \in \mathbb{Z} \iff Q_k \ \Big|\ d - {P_{k+1}}^2.$$
$$d - {P_{k+1}}^2 = d - (a_k\,Q_k - P_k)^2 = d - {P_k}^2 - {a_k}^2{Q_k}^2 + 2\,a_k\,Q_k.$$

$$\left.\begin{array}{l} Q_k \mid d - {P_k}^2 \text{ (per ipotesi)} \\ Q_k \mid -{a_k}^2{Q_k}^2 + 2\,a_k\,Q_k \end{array}\right\} \Rightarrow Q_k \mid d - {P_k}^2 - {a_k}^2{Q_k}^2 + 2\,a_k\,Q_k = d - {P_{k+1}}^2.$$

Quindi per come è definito, Q_{k+1} è un intero diverso da 0 ed infine

$$Q_{k+1} = \frac{d - {P_{k+1}}^2}{Q_k},\ Q_k\,Q_{k+1} = d - {P_{k+1}}^2 \Rightarrow Q_{k+1} \mid d - {P_{k+1}}^2.$$

L'ultima parte della dimostrazione consiste nel verificare che la frazione continua trovata con l'algoritmo sia effettivamente uguale ad α Per quanto visto nell'esempio A.0.5 dobbiamo dimostrare che

$$\alpha_k - a_k = \frac{1}{\alpha_{k+1}}.$$

$$\alpha_k - a_k = \frac{P_k + \sqrt{d}}{Q_k} - \frac{P_{k+1} + P_k}{Q_k} = \frac{\sqrt{d} - P_{k+1}}{Q_k},$$

avendo esplicitato a_k da $P_{k+1} = a_k\, Q_k - P_k$. Ora da $Q_k\, Q_{k+1} = d - {P_{k+1}}^2$, esplicitiamo Q_k ed andiamo a sostituire:

$$\alpha_k - a_k = \frac{Q_{k+1}\,(\sqrt{d} - P_{k+1})}{(\sqrt{d} + P_{k+1})(\sqrt{d} - P_{k+1})} = \frac{1}{\frac{P_{k+1}+\sqrt{d}}{Q_{k+1}}} = \frac{1}{\alpha_{k+1}}. \qquad \square$$

Possiamo ora dimostrare il teorema su cui si basano SQUFOF e CFRAC.

Teorema A.0.4. *Consideriamo $\sqrt{d}$, d intero positivo non quadrato. Si tratta di un'irrazionalità quadratica con $P = P_0 = 0$ e $Q = Q_0 = 1$. Possiamo quindi servirci dell'algoritmo di Lagrange per calcolare la frazione continua. Allora*

$${p_{k-1}}^2 - d\,{q_{k-1}}^2 = (-1)^k\, Q_k.$$

Dimostrazione. Consideriamo la frazione continua semplice di $\sqrt{d}$ finita e composta da $k+1$ termini. Abbiamo quindi

$$\sqrt{d} = \alpha = \frac{\alpha_k\, p_{k-1} + p_{k-2}}{\alpha_k\, q_{k-1} + q_{k-2}}.$$

Andiamo a sostituire a α_k l'espressione $\frac{P_k+\sqrt{d}}{Q_k}$ ottenuta con l'algoritmo di Lagrange, moltiplicando numeratore e denominatore per Q_k:

$$\frac{(P_k + \sqrt{d})\, p_{k-1} + Q_k\, p_{k-2}}{(P_k + \sqrt{d})\, q_{k-1} + Q_k\, q_{k-2}} = \sqrt{d}$$

$$P_k\, p_{k-1} + \sqrt{d}\, p_{k-1} + Q_k\, p_{k-2} = \sqrt{d}\, P_k\, q_{k-1} + d\, q_{k-1} + \sqrt{d}\, Q_k\, q_{k-2}$$

Eguagliando parte razionale e parte irrazionale, otteniamo il sistema

$$\begin{cases} P_k\, q_{k-1} + Q_k\, q_{k-2} = p_{k-1} \\ P_k\, p_{k-1} + Q_k\, p_{k-2} = d\, q_{k-1} \end{cases}.$$

Determiniamo ora con il metodo di Cramer la soluzione Q_k del sistema

$$Q_k = \frac{\begin{vmatrix} q_{k-1} & p_{k-1} \\ p_{k-1} & d\, q_{k-1} \end{vmatrix}}{\begin{vmatrix} q_{k-1} & q_{k-2} \\ p_{k-1} & p_{k-2} \end{vmatrix}} = \frac{d\,{q_{k-1}}^2 - {p_{k-1}}^2}{q_{k-1}\, p_{k-2} - p_{k-1}\, q_{k-2}} =$$

$$= \frac{-({p_{k-1}}^2 - d\,{q_{k-1}}^2)}{(-1)^{k-1}} = (-1)^k\,({p_{k-1}}^2 - d\,{q_{k-1}}^2). \qquad \square$$

Appendice B

Implementazione degli algoritmi

B.1 Crivello di Eratostene

Input:

- n intero.

Output:

- p, numero di primi trovati;
- vettore di lunghezza p dei primi $\leqslant n$.

```
long* Crivello (long n, int& p)
{ /* Inizializzazione del vettore di ricerca */
  long* sieve;
  sieve = new long[n];
  sieve[0] = 0;     // Nel primo indirizzo, quanti primi trovati.
  for (long i=2 ; i<=n ; ++i)
     sieve[i-1] = i;

  /* Scansione del vettore */
  long i = 2;
  while (i*i<=n)
     {
      if (sieve[i-1]!=0)
         {
          ++sieve[0];              // Trovato un nuovo primo!
          Elimina_Mult(n,sieve,i); // Cancellazione dei multipli.
         }
      ++i;
     }
```

```
   while (i<=n)
     {
      if (sieve[i-1]!=0)
         ++sieve[0];              // Conta i primi rimanenti.
      ++i;
     }

  long *primi = new long[sieve[0]];
  p = sieve[0];

  /* Vettore dei primi */
  int j = 0;
  for (int i = 1; i < n; ++i)
     {
      if (sieve[i] != 0)
         {
          primi[j] = sieve[i];
          ++j;
         }
     }

  delete [] sieve;

  return primi;
}


void Elimina_Mult(long n, long* v, long a)
{
  for (long i=a ; i*a<=n ; ++i)
     v[i*a-1]=0;
}
```

Una piccola accortezza per ridurre il numero di operazioni è stata inserita nella funzione `Elimina_Mult`: il contatore del ciclo `for` viene inizializzato al valore `a`, cioè al valore del nuovo primo trovato, in quanto tutti i multipli da $a \cdot 2$ a $a \cdot (a-1)$ sono già stati eliminati in precedenza.

B.2 Divisione per tentativi

Input:

- n, intero da fattorizzare;
- $*primi$, vettore degli elementi con i quali tentare la divisione;
- p, numero di elementi del vettore $*primi$;

Output:

- vettore di lunghezza p degli esponenti massimi degli elementi di $*primi$ che dividono n;
- R, parte resistente, non fattorizzata, di n.

```
int* Dividi (long n, long* primi, int p, long& R)
{
    int* esp = new int[p];

    int i = 0;
    R = n;
    while (i < p && R != 1)
       {
        esp[i] = 0;

        while (R%primi[i] == 0)
           {
            ++esp[i];
            R = R/primi[i];
           }

        ++i;
       }

    while (i < p)
       {
        esp[i] = 0;
        ++i;
       }

    return esp;
}
```

B.3 Fattorizzazione alla Fermat

Input:

- n, intero da fattorizzare;
- max, numero massimo di tentativi (se superato l'algoritmo fallisce).

Output:

- il divisore più piccolo trovato (se fallisce restituisce n).

```
long DividiFermat (long n, long max)
{
   long a = long(sqrt(n));
   if (a*a == n)
      return a;
   else
      {
       long x = a+1;
       long y = 1;

       long sum = x*x - y*y - n;
       long counter = 0;

       while (sum != 0 && counter < max)
          {
           if (sum > 0)
              {
               sum = sum - 2*y - 1;
               ++y;
              }
           else
              {
               sum = sum + 2*x + 1;
               ++x;
              }
           ++counter;
          }

       if (sum == 0)
          return (x-y);
       else
          return n;
      }
}
```

B.4 Il metodo $p-1$

Input:

- n, intero da fattorizzare;
- c, valore iniziale della successione;
- max, numero massimo di tentativi (se superato l'algoritmo fallisce).

Output:

- il divisore trovato (se fallisce restituisce n).

```
long PollardP (long n, long c, long max)
{
 long a = c;
 long div = 1;

 for (int i = 1 ; i <= max ; ++i)
    {
     a = pow_modN(a,i,n);

     if (i%20 == 0)
        {
         div = MCD(a-1,n);

         if (div > 1)
            return div;
        }
    }

 return n;
}
```

Nell'algoritmo viene utilizzata la funzione `pow_modN`, che calcola in maniera efficiente le potenze modulo n, sfruttando l'espressione binaria dell'esponente, mediante la tecnica delle quadrature: per esempio

$$a^{77} = a^{32} \cdot a^8 \cdot a^4 \cdot a,$$

dove

$$a^{32} = (a^{16})^2 = ((a^8)^2)^2 = \ldots = ((((a^2)^2)^2)^2)^2.$$

B.5 Il metodo ρ

Input:

- n, intero da fattorizzare;
- c, valore iniziale della successione;
- max, numero massimo di tentativi (se superato l'algoritmo fallisce).

Output:

- il divisore trovato (se fallisce restituisce n).

```
long PollardRHO (long n, long c, long max)
{
 long x = c;
 long y = c;
 long prod = 1;
 long div = 1;

 for (int i = 1 ; i <= max ; ++i)
    {
     x = (x*x + 1)%n;
     y = (y*y + 1)%n;
     y = (y*y + 1)%n;

     prod = (prod*(y-x))%n;

     if (i%20 == 0)
        {
         div = MCD(prod,n);

         if (div > 1)
            return div;
        }
    }

  return n;
}
```

B.6 SQUFOF

Input:

- n, intero da fattorizzare;
- max, numero massimo di tentativi di ricerca di un quadrato (se superato, l'algoritmo fallisce).

Output:

- il divisore trovato (se fallisce restituisce n).

```
long SQUFOF (long n, long max)
{
     long d = long(sqrt(n));

     long p0 = 1,  p1 = d, p;
     int P = 0, Q = 1;

     long a = d;
     int i = 1;
     while (i <= max)
        {
         P = a*Q - P;
         Q = (n - P*P)/Q;
         a = long((P + d)/Q);

         if (i%2 == 0 && IsSquare(Q))
            {
             long temp = p - long(sqrt(Q));
             long div = MCD(temp,n);

             if (div > 1 && div < n)
                return div;
            }

         p = a*p1 + p0;
         p0 = p1;
         p1 = p;
         ++i;
        }

    return n;
}

bool IsSquare(long a)
// Restituisce true se a e' un quadrato perfetto, altrimenti false
```

B.7 CFRAC

Input:

- n, intero da fattorizzare;
- dim, cardinalità della base di fattori;
- max, numero massimo di tentativi di ricerca degli elementi Q_k (se superato, l'algoritmo fallisce).

Output:

- il divisore trovato (se fallisce, restituisce n).

```
long CFRAC (long n, int dim, long max)
{ /* Ricerca di una base */
  int L = 25,l;
search_base:
  long* primi = Crivello (L,l);
  long BASE[dim-1];
  int j = 0;
  for (int i = 0 ; i < l && j < dim-1 ; ++i)
     {
      if (Legendre(n,primi[i]) == 1)
         {
          BASE[j] = primi[i];
          ++j;
         }
     }

  if (j != dim-1)
     {
      L = 2*L;
      goto search_base;
     }

  /* Ricerca di dim+1 vettori fattorizzati */
  long pk[dim+1];
  int **ESP, **ESPmod2;
  ESP = new int*[dim+1];
  ESPmod2 = new int*[dim+1];
  for (int i = 0; i < dim+1 ; ++i)
     {
      ESP[i] = new int[dim];
      ESPmod2[i] = new int[dim];
     }

  long d = long(sqrt(n));
  long p0 = 1, p1 = d, p = d;
```

```
long P = 0, Q = 1;

long a = d;
int i = 1, k = 0;
while (i <= max && k < dim+1)
  {
   P = a*Q - P;
   Q = (n - P*P)/Q;
   a = long((P + d)/Q);

   long R;   // Scomposizione in fattori.
   int* esp = Dividi(Q,BASE,dim-1,R);
   if (R == 1)  // Trovato un elemento fattorizzato
      {
       pk[k] = p;
       ESP[k][0] = i;
       ESPmod2[k][0] = i%2;
       for (int h = 1 ; h < dim ; ++h)
          {
           ESP[k][h] = esp[h-1];
           ESPmod2[k][h] = esp[h-1]%2;
          }
       ++k;
      }

   p = a*p1 + p0;
   p0 = p1;
   p1 = p;
   ++i;
  }

if (i == max && j != dim+1) // Fallimento. Non trovati
   {                        // sufficienti Qk.
    for (int h = 0 ; h < dim+1; ++h)
       {
        delete [] ESP[h];
        delete [] ESPmod2[h];
       }
    delete [] ESP;
    delete [] ESPmod2;

    return n;
   }

/*  Dip. lin. Eliminazione di Gauss */
int **gauss;
gauss = new int*[dim+1];
```

```
for (int i = 0 ; i <= dim ; ++i)
   {
    gauss[i] = new int[dim+1];
    for (int j = 0 ; j <= dim ; ++j)
       {
        if (j==i)
           gauss[i][j] = 1;
        else
           gauss[i][j] = 0;
       }
   }

 int start = 0;
 for (int i = 0 ; i < dim ; ++i)
    {
     int incr = 0;
     int t = start;
     while (t <= dim && ESPmod2[t][i] == 0)
        ++t;          // Ricerca della componente non nulla.

     if (t <= dim)
        {
         if (t != start)
            {               // Scambio delle righe.
             swaprow(ESPmod2,start,t);
             swaprow(gauss,start,t);
            }
         ++incr;
        }

     ++t;
     while (t <= dim)
        {
         if (ESPmod2[t][i] == 1) // Annullamento delle altre
            {                    // componenti della colonna.
             addrow(ESPmod2,t,start,dim);
             addrow(gauss,t,start,dim+1);
            }
         ++t;
        }
     start = start + incr;
    }

 /* Calcolo delle dipendenze lineari */
 int h = dim;
 bool ok = true;
 while (h >= 0 && ok)
    {
     ok = IsNULL(ESPmod2,h,dim);
```

```
        if (ok)  // Tentativo di fattorizzazione
           {
            long x = 1, y = 1;
            long espy[dim];
            for (int j = 0; j < dim ; ++j)
               espy[j] = 0;

            for (int i = 0 ; i <= dim ; ++i)
               {
                if (gauss[h][i] == 1)
                   {
                    x = x*pk[i];
                    for (int j = 0; j < dim ; ++j)
                       espy[j] = espy[j]+ESP[i][j];
                   }
               }

            for (int j = 1; j < dim ; ++j)
               y = y*long(pow(BASE[j-1],espy[j]/2));

            long div = MCD(x-y,n);
            if (div > 1 && div < n)
               {
                for (int h = 0 ; h < dim+1; ++h)
                   {
                    delete [] ESP[h];
                    delete [] ESPmod2[h];
                    delete [] gauss[h];
                   }
                delete [] ESP;
                delete [] ESPmod2;
                delete [] gauss;
                return div;
               }
           }
        --h;
       }

   for (int h = 0 ; h < dim+1; ++h)
      {
       delete [] ESP[h];
       delete [] ESPmod2[h];
       delete [] gauss[h];
      }
   delete [] ESP;
   delete [] ESPmod2;
   delete [] gauss;
   return n;
}
```

B.8 Il crivello quadratico

Input:

- n, intero da fattorizzare;
- dim, cardinalità della base di fattori;
- max, numero di elementi a cui applicare il crivello.

Output:

- il divisore trovato (se fallisce, restituisce n).

```
long QS (long n, int dim, long max)
{
  /* Ricerca della base */
  int l, L = 25;

search_base:
  long* primi = Crivello (L,l);
  long BASE[dim];
  int j = 0;
  for (int i = 0 ; i < l && j < dim ; ++i)
     {
      if (Legendre(n,primi[i]) == 1)
         {
          BASE[j] = primi[i];
          ++j;
         }
     }

  if (j != dim)
     {
      L = 2*L;
      goto search_base;
     }

  /* Ricerca di dim+1 vettori fattorizzati */
  long A = long(sqrt(n)) + 1;
  int **esp;
  esp = new int*[max];
  for (int i = 0 ; i < max ; ++i)
     {
      esp[i] = new int[dim];
      for (int j = 0 ; j < dim ; ++j)
         esp[i][j] = 0;
     }
```

```
/* Crivello Quadratico */
for (int i = 0 ; i < dim ; ++i)
   {
    int p = BASE[i];
    int e = 2*int(2*log(BASE[dim-1])/log(p));
    for (int j = 1 ; j <= e && Legendre(n,p) == 1; ++j)
       {
        long res = solve(n,p);

        bool found = false;
        for (int k = 0 ; k < p && k < max && !found ; ++k)
           {
            if ((A+k)%p == res)
               {
                for (int h = k ; h < max ; h = h + p)
                    esp[h][i] = j;

                found = true;
               }
           }

        if (p != 2)
           {
            res = p - res;

            found = false;
            for (int k = 0 ; k < p && !found ; ++ k)
               {
                if ((A+k)%p == res)
                   {
                    for (int h = k ; h < max ; h = h + p)
                        esp[h][i] = j;

                    found = true;
                   }
               }
           }
        p = p*BASE[i];
       }
   }

/* Scelta degli elementi scomposti */
long r[dim+1];
int **ESPmod2, **ESP;
ESPmod2 = new int*[dim+1];
ESP = new int*[dim+1];
for (int i = 0 ; i <= dim ; ++i)
   {
    ESPmod2[i] = new int[dim];
```

```
    ESP[i] = new int[dim];
    for (int j = 0 ; j < dim ; ++j)
       {
        ESPmod2[i][j] = 0;
        ESP[i][j] = 0;
       }
   }

int k = 0;
int i = 0;
while (i < max && k <= dim)
   {
    long q = (A+i)*(A+i) - n;
    long fact = 1;
    for (int j = 0 ; j < dim ; ++j)
       fact = fact*long(pow(BASE[j],esp[i][j]));

    if (q == fact)
       {
        r[k] = A+i;
        for (int j = 0 ; j < dim ; ++j)
           {
            ESP[k][j] = esp[i][j];
            ESPmod2[k][j] = (esp[i][j])%2;
           }
        ++k;
       }
    ++i;
   }

 for (int h = 0 ; h < max ; ++h)
    delete [] esp[h];
 delete [] esp;

if (i == max && j != dim+1)    // Fallimento. Non trovati
   {                           // sufficienti Qk.
    for (int h = 0 ; h < dim + 1 ; ++h)
       {
        delete [] ESP[h];
        delete [] ESPmod2[h];
       }
    delete [] ESP;
    delete [] ESPmod2;
    return n;
   }

/*  Dip. lin. Eliminazione di Gauss */
int **gauss;
gauss = new int*[dim+1];
```

```
for (int i = 0 ; i <= dim ; ++i)
   {
    gauss[i] = new int[dim+1];
    for (int j = 0 ; j <= dim ; ++j)
       {
        if (j==i)
           gauss[i][j] = 1;
        else
           gauss[i][j] = 0;
       }
   }

 int start = 0;
 for (int i = 0 ; i < dim ; ++i)
    {
     int incr = 0, t = start;
     while (t <= dim && ESPmod2[t][i] == 0)
        ++t;          // Ricerca della componente non nulla.

     if (t <= dim)
        {
         if (t != start)
            {               // Scambio delle righe.
             swaprow(ESPmod2,start,t);
             swaprow(gauss,start,t);
            }
         ++incr;
        }
     ++t;
     while (t <= dim)
        {
         if (ESPmod2[t][i] == 1) // Annullamento delle altre
            {                    // componenti della colonna.
             addrow(ESPmod2,t,start,dim);
             addrow(gauss,t,start,dim+1);
            }
         ++t;
        }
     start = start + incr;
    }

 /* Calcolo delle dipendenze lineari */
 int h = dim;
 bool ok = true;
 while (h >= 0 && ok)
    {
     ok = IsNULL(ESPmod2,h,dim);
```

```
        if (ok)  // Tentativo di fattorizzazione.
           {
            long x = 1, y = 1;
            long espy[dim];
            for (int j = 0 ; j < dim ; ++j)
               espy[j] = 0;

            for (int i = 0 ; i <= dim ; ++i)
               {
                if (gauss[h][i] == 1)
                   {
                    x = x*r[i];
                    for (int t = 0 ; t < dim ; ++t)
                       espy[t] = espy[t] + ESP[i][t];
                   }
               }

            for (int i = 0 ; i < dim ; ++i)
               y = y*long(pow(BASE[i],espy[i]/2));

            long div = MCD(x-y,n);
            if (div > 1 && div < n)
               {
                for (int h = 0 ; h < dim + 1 ; ++h)
                   {
                    delete [] ESP[h];
                    delete [] ESPmod2[h];
                    delete [] gauss[h];
                   }
                delete [] ESP;
                delete [] ESPmod2;
                delete [] gauss;
                return div;
               }
           }
        --h;
       }

   for (int h = 0 ; h < dim + 1 ; ++h)
      {
       delete [] ESP[h];
       delete [] ESPmod2[h];
       delete [] gauss[h];
      }
   delete [] ESP;
   delete [] ESPmod2;
   delete [] gauss;
   return n;
}
```

Negli algoritmi `CFRAC` e `QS` inoltre sono utilizzate le seguenti funzioni ausiliarie:

```
long Legendre(long a, long p)
// Restituisce il simbolo di Legendre (a/p)

long solve (long a, long p)
// Restituisce, se esiste, la piu' piccola soluzione
// dell'equazione z^2 = a mod p.

void swaprow(int** M, int i, int j)
// Scambia la riga i e la riga j della matrice M

void addrow(int** M, int i, int j, int dim)
// Sostituisce la riga i con la somma delle righe i e j

bool IsNULL(int** M, int i, int dim)
// Restituisce true se la riga i della matrice M
// e' composta da soli 0, altrimenti false
```

Appendice C

Implementazione degli interi multiprecisione

In quest'ultima parte è presentata una soluzione elementare per lavorare con interi multiprecisione. Ogni intero è conservato nella sua rappresentazione in base 256 in un vettore di elementi di 1 byte (`unsigned char`) che possono contenere appunto le cifre da 0 a 255. La costante `max_lenght` indica la lunghezza massima del vettore di un intero, quindi in questo caso il numero più grande con cui si può lavorare è $256^{100}-1$. Gli algoritmi per lavorare con i dati `MP` sono un tentativo di compromesso tra l'efficienza computazionale e la leggibilità degli stessi.

C.1 MultiPrecision.h

```
#ifndef _MULTI_PRECISION_INT_
#define _MULTI_PRECISION_INT_

#include <iostream.h>
#include <stdlib.h>
#include <math.h>

const int max_lenght = 100;

typedef unsigned char UCHAR;

class MultiPrecision
{
   private:
       /* Campi Privati */
       UCHAR* number;
       char segno;
       int L;
```

```
    /* Funzioni Private */
    MultiPrecision (int lenght, UCHAR val);
    UCHAR cifra (int i) const;
    int Segno (void);
    void Opposto (void);

public:
    /* Costruttori - Distruttori */
    MultiPrecision (void);
    MultiPrecision (const MultiPrecision& p);
    MultiPrecision (const int& p);
    MultiPrecision (const char* p);
    ~MultiPrecision (void);

    /* Overload dell'assegnamento */
    MultiPrecision& operator= (const MultiPrecision& p);

    /* Operatori binari */
    MultiPrecision operator+ (const MultiPrecision& p);
    MultiPrecision operator+ (const int& p);
    MultiPrecision operator- (const MultiPrecision& p);
    MultiPrecision operator- (const int& p);
    MultiPrecision operator* (const MultiPrecision& p);
    MultiPrecision operator* (const int& p);
    MultiPrecision operator/ (const MultiPrecision& p);
    MultiPrecision operator/ (const int& p);
    MultiPrecision operator% (const MultiPrecision& p);
    MultiPrecision operator% (const int& p);

    /* Operatori booleani */
    bool operator== (const MultiPrecision& p);
    bool operator== (const int& p);
    bool operator!= (const MultiPrecision& p);
    bool operator!= (const int& p);
    bool operator> (const MultiPrecision& p);
    bool operator> (const int& p);
    bool operator< (const MultiPrecision& p);
    bool operator< (const int& p);
    bool operator>= (const MultiPrecision& p);
    bool operator>= (const int& p);
    bool operator<= (const MultiPrecision& p);
    bool operator<= (const int& p);

    /* Operatori di cast */
    operator char (void);
    operator short (void);
    operator int (void);
    operator long (void);
    operator float (void);
```

```
        /* Funzioini generiche */
        int Log (void) {return L;}

/* Funzioni friend: Operatori input - output */
friend ostream& operator<< (ostream& os, const MultiPrecision& p);
friend istream& operator>> (istream& is, MultiPrecision& p);

/* Funzioni friend: Operatori binari */
friend MultiPrecision operator+ (const int& q, const MultiPrecision& p);
friend MultiPrecision operator- (const int& q, const MultiPrecision& p);
friend MultiPrecision operator* (const int& q, const MultiPrecision& p);
friend MultiPrecision operator/ (const int& q, const MultiPrecision& p);
friend MultiPrecision operator% (const int& q, const MultiPrecision& p);

/* Funzioni friend: Operatori unari */
friend MultiPrecision operator++ (MultiPrecision& p);
friend MultiPrecision operator-- (MultiPrecision& p);
friend MultiPrecision operator++ (MultiPrecision& p, int notused);
friend MultiPrecision operator-- (MultiPrecision& p, int notused);

/* Funzioni friend: Operatori booleani */
friend bool operator== (const int& q, const MultiPrecision& p);
friend bool operator!= (const int& q, const MultiPrecision& p);
friend bool operator> (const int& q, const MultiPrecision& p);
friend bool operator>= (const int& q, const MultiPrecision& p);
friend bool operator< (const int& q, const MultiPrecision& p);
friend bool operator<= (const int& q, const MultiPrecision& p);
};

typedef MultiPrecision MP;


/* Funzioni per lavorare con MP */
int MAX (int a, int b);
MP sqr(MP a);
MP pow(MP a, int exp);
MP sqrt(MP a);

#endif
```

C.2 MultiPrecision.cpp

In questa sezione non viene riportato il listato nella sua totalità perché troppo lungo. Vengono presentate le procedure fondamentali ed alcuni modelli validi per più di una funzione.

```
#include <iostream.h>
#include <istream.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "MultiPrecision.h"
```

Inizialmente presentiamo le funzioni private, spiegandone l'effettiva utilità e non riportando l'implementazione.

```
MultiPrecision::MultiPrecision(int lenght, UCHAR val)
// Inizializza un intero con lenght cifre in base 256,
// tutte inizializzate al valore val.
// Es. MultiPrecision(4,1) = (1111).

UCHAR MultiPrecision::cifra (int i) const
// Restituisce il valore della cifra i-esima.
// Se la cifra i-esima non esiste restituisce 0.

int MultiPrecision::Segno (void)
// Restituisce il segno dell'intero.

void MultiPrecision::Opposto (void)
// Cambia segno all'intero.
```

Passiamo alle funzioni pubbliche, partendo ovviamente da allocatori, deallocatori e operatore di assegnamento.

```
MultiPrecision::MultiPrecision(void)
{
   L = 0;
   segno = 1;
}

MultiPrecision::MultiPrecision(const MultiPrecision& p)
{
   L = p.L;
   segno = p.segno;
   number = new UCHAR[L];

   for (int i = 0 ; i < L ; ++i)
      number[i] = p.number[i];
}
```

```
MultiPrecision::MultiPrecision(const int& p)
{
   if (p == 0)
      {
       L = 1;
       segno = 1;

       number = new UCHAR[1];
       number[0] = 0;
      }
   else
      {
       int temp;
       if (p > 0)
          {
           segno = 1;
           temp = p;
          }
        else
          {
           segno = -1;
           temp = (-1)*p;
          }

       UCHAR* aux;
       aux = new UCHAR[max_lenght];

       int l = 0;
       while (temp != 0)
          {
           aux[l] = temp%256;
           temp = temp/256;
           ++l;
          }

       L = l;
       number = new UCHAR[L];
       for (int i=0 ; i<L ; ++i)
          number[i] = aux[i];

       delete [] aux;
      }
}

MultiPrecision::MultiPrecision (const char* p)
{
   *this = MultiPrecision ();

   int l = strlen(p);
```

```
   if (l>0)
      {
       *this = MultiPrecision (1,0);
       int S = 1;
       int i = 0;

       if (p[0] == '-' && l > 1)
          {
           S = -1;
           ++i;
          }

       while(i < l)
          {
           *this = (*this)*10 + (p[i] - 48);
           ++i;
          }

       segno = S;
      }
}

MultiPrecision::~MultiPrecision(void)
{
   if (L != 0)
      delete [] number;
}

MultiPrecision& MultiPrecision::operator= (const MultiPrecision& p)
{
   if (&p != this)
      {
       segno = p.segno;

       if (L != p.L)
          {
           delete [] number;
           L = p.L;
           number = new UCHAR[L];
          }

       for (int i = 0 ; i < L ; ++i)
          number[i] = p.number[i];
      }

   return *this;
}
```

Ci occupiamo ora degli operatori binari che operano su una coppia di dati

MP, più avanti spiegheremo cosa succede quando si considerano un elemento MP ed uno int.

```
MultiPrecision MultiPrecision::operator+ (const MultiPrecision& p)
{
   if (L != 0 && p.L != 0)
      {
       if (segno == p.segno)
          {
           UCHAR* temp;
           int l = MAX(L,p.L);
           temp = new UCHAR[l+1];

           int resto = 0;
           for (int i = 0 ; i <= l ; ++i)
              {
               int sum = cifra(i) + p.cifra(i) + resto;

               temp[i] = sum%256;
               resto = sum/256;
              }

           MultiPrecision r;

           if (temp[l] == 0)
              r = MultiPrecision (l,0);
           else
              r = MultiPrecision (l+1,0);

           r.segno = segno;
           for (int i = 0 ; i < r.L ; ++i)
              r.number[i] = temp[i];

           delete [] temp;
           return r;
          }
       else
          {
           MultiPrecision A,B;
           int S;
           if (segno == 1)
              {
               MultiPrecision temp = p;
               temp.Opposto();
               if ((*this) >= temp)
                  {
                   S = 1;
                   A = *this;
                   B = temp;
                  }
```

```
    else
       {
        S = -1;
        A = temp;
        B = *this;
       }
   }
else
   {
    MultiPrecision temp = *this;
    temp.Opposto();
    if (temp > p)
       {
        S = -1;
        A = temp;
        B = p;
       }
    else
       {
        S = 1;
        A = p;
        B = temp;
       }
   }

UCHAR* aux = new UCHAR[A.L];
MultiPrecision C (B.L,0);

for (int i=0 ; i<A.L ; ++i)
   {
    if (A.cifra(i) < B.cifra(i))
       --A.number[i+1];

    if (i < B.L)
       C.number[i] = 256 - B.cifra(i);

    aux[i] = A.cifra(i) + C.cifra(i);
   }

int counter = A.L;
while (counter > 0 && aux[counter-1] == 0)
   --counter;

if (counter == 0)
   {
    delete [] aux;
    return MultiPrecision (1,0);
   }
else
```

```
            {
             MultiPrecision r (counter,0);
             r.segno = S;
             for (int i=0 ; i<counter ; ++i)
                r.number[i] = aux[i];

             delete [] aux;
             return r;
            }
         }
      }
}

MultiPrecision MultiPrecision::operator- (const MultiPrecision& p)
{
   MultiPrecision r = p;
   r.Opposto();

   return (*this + r);
}

MultiPrecision MultiPrecision::operator* (const MultiPrecision& p)
{
   if (L != 0 && p.L != 0)
      {
       UCHAR* temp;
       int l = L + p.L;
       temp = new UCHAR[l];

       int resto = 0;
       for (int i = 0 ; i < l ; ++i)
          {
           int sum = resto;
           for (int j = 0 ; j < p.L ; ++j)
              sum = sum + cifra(i-j)*p.cifra(j);

           temp[i] = sum%256;
           resto = sum/256;
          }

       int counter = l;
       while (counter > 0 && temp[counter-1] == 0)
          --counter;

       if (counter == 0)
          {
           delete [] temp;
           return MultiPrecision (1,0);
          }
```

```
        else
           {
            MultiPrecision r (counter,0);
            r.segno = segno*p.segno;
            for (int i=0 ; i<counter ; ++i)
               r.number[i] = temp[i];

            delete [] temp;
            return r;
           }
      }
}

MultiPrecision MultiPrecision::operator/ (const MultiPrecision& p)
{
   if (L != 0 && p.L != 0)
      {
       MultiPrecision A = *this, B = p;

       if (B != 0)
          {
           int S;

           if (*this < 0)
              A.Opposto();

           if (B > 0)
              S = 1;
           else
              {
               B.Opposto();
               S = -1;
              }

           MultiPrecision r = A;
           MultiPrecision q (1,0);
           int I = A.L - B.L;
           if (I == 0)
              I = 1;
           while (I > 0 && r > 0)
              {
               MultiPrecision s (I,0);
               s.number[I-1] = 1;

               while(r >= s*B)
                  {
                   r = r - s*B;
                   q = q + s;
                  }
```

```
            --I;
           }

        q.segno = S;
        if (*this < 0)
           {
            q.Opposto();

            if (r != 0)
               {
                if (B > 0)
                   q = q - 1;
                else
                   q = q + 1;
               }
           }

        return q;
       }
    }
}

MultiPrecision MultiPrecision::operator% (const MultiPrecision& p)
{
   if (L != 0 && p.L != 0)
      {
       MultiPrecision A = *this;
       MultiPrecision B = p;
       if (B != 0)
          {
           if (*this < 0)
              A.Opposto();

           if (B < 0)
              B.Opposto();

           MultiPrecision r = A;
           int I = A.L - B.L;
           if (I == 0)
              I = 1;
           while (I > 0 && r > 0)
              {
               MultiPrecision s (I,0);
               s.number[I-1] = 1;

               while(r >= s*B)
                   r = r - s*B;
```

```
            --I;
           }

         if (*this < 0 && r != 0)
            r = B - r;

         return r;
        }
     }
}

/* Operatori booleani */
bool MultiPrecision::operator== (const MultiPrecision& p)
{
   if (segno != p.segno)
      return false;

   if (L != p.L)
      return false;

   for (int i = L-1 ; i >= 0 ; --i)
      {
       if (number[i] != p.number[i])
          return false;
      }

   return true;
}

bool MultiPrecision::operator!= (const MultiPrecision& p)
{
   return(!(*this == p));
}

bool MultiPrecision::operator> (const MultiPrecision& p)
{
   if (segno != p.segno)
      {
       if (segno == 1)
          return true;
       else
          return false;
      }
   else
      {
       if (segno == 1)
          {
           if (L > p.L)
              return true;
```

```
            else if (L == p.L)
               {
                for (int i = L-1 ; i>=0 ; --i)
                   {
                    if (cifra(i) > p.cifra(i))
                       return true;
                    else if (cifra(i) < p.cifra(i))
                       return false;
                    }
               }
            }
         else
            {
             if (L < p.L)
                return true;
             else if (L == p.L)
                {
                 for (int i = L-1 ; i>=0 ; --i)
                    {
                     if (cifra(i) < p.cifra(i))
                        return true;
                     else if (cifra(i) > p.cifra(i))
                        return false;
                    }
                }
            }
      }

   return false;
}

bool MultiPrecision::operator< (const MultiPrecision& p)
{
   if (segno != p.segno)
      {
       if (segno == 1)
          return false;
       else
          return true;
      }
   else
      {
       if (segno == 1)
          {
           if (L < p.L)
              return true;
           else if (L == p.L)
              {
               for (int i = L-1 ; i>=0 ; --i)
```

```
                {
                 if (cifra(i) < p.cifra(i))
                    return true;
                 else if (cifra(i) > p.cifra(i))
                    return false;
                 }
             }
          }
       else
          {
           if (L > p.L)
              return true;
           else if (L == p.L)
              {
               for (int i = L-1 ; i>=0 ; --i)
                  {
                   if (cifra(i) > p.cifra(i))
                      return true;
                   else if (cifra(i) < p.cifra(i))
                      return false;
                  }
              }
          }
      }

   return false;
}

bool MultiPrecision::operator>= (const MultiPrecision& p)
{
    return (!(*this < p));
}

bool MultiPrecision::operator<= (const MultiPrecision& p)
{
   return (!(*this > p));
}
```

Se il secondo argomento dell'operatore è di tipo `int`, si utilizza l'allocatore di `MP` a partire da un `int` e si chiama lo stesso operatore che però prende come argomento due elementi `MP`. Vediamo un esempio sulla somma:

```
MultiPrecision MultiPrecision::operator+ (const int& p)
{
   MultiPrecision r = p;

   return (*this + r);
}
```

Facciamo un esempio di operatori di cast che permettono di passare da un elemento di tipo `MP` ad uno dei tipi di dato fondamentale del C++.

```
MultiPrecision::operator int (void)
{
   if (L != 0)
      {
       int r = 0;
       int l;

       if (L > 4)
          l = 4;
       else
          l = L;

       for (int i = l ; i > 0 ; --i)
          r = r*256 + number[i-1];

       if (segno == -1)
          r = r*(-1);

       return r;
      }
}
```

Rimangono ora da vedere le funzioni `friend` tra cui si trovano gli operatori `>>` e `<<` che consentono l'inserimento da file o tastiera e la scrittura su file o a video.

```
/* Operatori input - output */
istream& operator>> (istream& is, MultiPrecision& p)
{
   char BUFFER[max_lenght];
   is >> BUFFER;
   p = MultiPrecision (1,0);
   int l = strlen(BUFFER);

   if (l>0)
      {
       int S = 1;
       int i = 0;

       if (BUFFER[0] == '-' && l > 1)
          {
           S = -1;
           ++i;
          }

       while(i < l)
          {
           p = p*10 + (BUFFER[i] - 48);
           ++i;
          }
```

```
       p.segno = S;
      }

   return is;
}

ostream& operator<< (ostream& os, const MultiPrecision& p)
{
   if (p.L != 0)
      {
       MultiPrecision temp = p;

       if (temp == 0)
          os << 0;
       else
          {
           char BUFFER[max_lenght];

           if(temp < 0)
              {
               temp.Opposto();
               os << "-";
              }

           int l = 0;
           while (temp > 0)
              {
               BUFFER[l] = char(temp%10) + 48;
               temp = temp/10;
               ++l;
              }

           for (int i = l ; i > 0 ; --i)
              os << BUFFER[i-1];
          }
      }

   return os;
}
```

Prima abbiamo visto gli operatori che prendono come argomento due elementi `MP`, oppure come primo argomento un `MP` e come secondo un `int`. Il caso inverso non può essere contemplato tra le funzioni membro, si ricorre pertanto ad una funzione `friend`. Forniamo ora un esempio di com'è definito un operatore binario che come primo argomento considera un `int` e come secondo un `MP`. Anche in questo caso si utilizza si richiama l'operatore membro.

```
MultiPrecision operator+ (const int& q, const MultiPrecision& p)
{
   MultiPrecision r = q;

   return (r+p);
}
```

Terminate le funzioni della classe `MultiPrecision`, ci occupiamo delle funzioni per calcolare le potenze e la radice quadrata, adattate a questo caso.

```
MP sqr (MP a)
{
  return a*a;
}

MP pow(MP a, int exp)
{
   MP r = 1;
   MP base = a;

   while (exp>0)
     {
      if (exp%2==0)
         {
          base = sqr(base);
          exp = exp/2;
         }
      else
         {
          r = r * base;
          base = sqr(base);
          exp = exp/2;
         }
     }

   return r;
}

MP sqrt(MP a)
{
   if (a >= 0)
      {
       if (a == 0)
          return 0;

       int k = a.Log();
       if (k%2 == 0)
          k = k/2;
       else
          k = k/2 + 1;
```

```
        MP c = 256;
        MP root = 0;
        while (k >= 0)
           {
            MP x = pow(c,k);

            while (sqr(root + x) <= a)
               {
                root = root + x;
               }

            --k;
           }
        return root;
       }
}
```

Bibliografia

[1] David M. Bressoud, *Factorization and Primality Testing*, Springer-Verlag, 1989.

[2] Hans Riesel, *Prime Numbers and Computer Methods for Factorization - Second Edition*, Birkhäuser, 1994.

[3] Richard Crandall, Carl Pomerance, *Prime Numbers - A Computational Perspective*, Springer-Verlag, 2000.

[4] Alessandro Languasco, Alessandro Zaccagnini, *Introduzione alla Crittografia*, Hoepli, 2004.