UNIVERSITÀ DEGLI STUDI ROMA TRE

FACOLTÀ DI SCIENZE M.F.N.

Sintesi della Tesi di Laurea in Matematica

di

Marta Leggio

Metodi e Algoritmi per la Moltiplicazione Veloce

Relatore

Prof. Francesco Pappalardi

ANNO ACCADEMICO 2003 - 2004

Luglio 2004

Classificazione AMS : 11Y16, 68Q25

Parole Chiave : Teoria computazionale dei numeri, moltiplicazione veloce di interi, algoritmi, complessità

Sintesi

Scopo di questa tesi è lo studio di metodi e l'implementazione di algoritmi che rendano veloce la moltiplicazione fra interi molto grandi.

Quando ci sediamo di fronte ad un computer, per gioco o per lavoro, diventa tutto più piacevole se ad esempio il caricamento di una pagina internet o l'esecuzione di un programma è veloce; il risultato dell'operazione non cambierebbe anche se dovessimo aspettare qualche secondo in più guardando il monitor, ma di fatto tutto sarebbe più efficiente se accompagnato dalla rapidità.

Veniamo alla moltiplicazione fra interi: quando si ha a che fare con numeri di decine, migliaia o addirittura milioni di cifre decimali non è così banale operare con essi, ad esempio moltiplicarli, anche se in linea teorica il prodotto in colonna che tutti conosciamo, il 'metodo classico', ci porterebbe dopo settimane o forse anni di conti al risultato corretto.

Si sfruttano allora nuovi metodi per la moltiplicazione diversi da quello classico che velocizzino l'operazione stessa.

Infatti, per moltiplicare due interi di n cifre ciascuno utilizzando il metodo classico sono necessarie al più $O(n^2)$ operazioni bit, mentre le strategie che vedremo faranno scendere la complessità prima a $O(n^{1.59})$ e poi a $O(n \log n \log\log n)$ operazioni bit, ossia moltiplicheremo più velocemente.

Questa utilizzata è la notazione O-grande, introdotta da Paul Bachmann nel 1894 e resa poi popolare da Edmund Landau negli anni seguenti.

Se f e g sono funzioni definite in $\mathbb{N}$ a valori in $\mathbb{R}$, la formula $f(n) = O(g(n))$ significa che esiste una costante C tale che $|f(n)| \leq C|g(n)| \quad \forall n \in \mathbb{N}$, e quando in una formula troviamo $O(g(n))$ allora questo rappresenta una

funzione $f(n)$ che verifica l'eguaglianza scritta.

Seguendo [1, 2], in tutto il nostro studio lavoreremo con polinomi a coefficienti in un anello commutativo unitario R. Adatteremo gli algoritmi per la moltiplicazione polinomiale alla moltiplicazione di numeri interi utilizzando polinomi a coefficienti interi.

Pertanto nella discussione degli algoritmi e nell'analisi della loro complessità dovremo specificare se parliamo di operazioni bit o di operazioni eseguite nell'anello, nozioni definite dettagliatamente in [15].

Dato un numero intero di n cifre decimali $c_1c_2\cdots c_n$, dove c_i indica l'i-esima cifra, e pertanto $0 \leq c_i < 10$, possiamo associargli il polinomio di grado $n-1$ il cui coefficiente del termine di grado i è c_{n-i}.

Viceversa, dato un polinomio $f(x)$ di grado $n-1$, l'intero di n cifre ad esso associato è $f(10)$.

Se n e m sono due interi con lo stesso numero di cifre, il nostro scopo è moltiplicarli velocemente. Consideriamo i polinomi $f(x)$ e $g(x)$ ad essi associati, applichiamo a questi uno dei metodi per la moltiplicazione veloce ottenendo il polinomio prodotto $h(x) = f(x)g(x)$ di grado $\deg(h) = \deg(f) + \deg(g)$, e da qui $n \cdot m = h(10)$.

Come vedremo in 1.3, calcolare $h(10)$ può essere tutt'altro che banale, ad esempio se il suo grado è 2^{20}.

Sfruttiamo allora un procedimento basato sulle divisioni con il resto per 10 (o per 10^k se lavoriamo in base 10^k) che permette di ricostruire l'intero associato al polinomio, e che riassumiamo nel seguente algoritmo:

- `Input :  Il polinomio` $f = \sum_{0 \leq i < n} f_i x^i \in \mathbb{Z}[x]$
- `Output :  L'intero` m `associato al polinomio`

1. `IF` $n = 1$ `RETURN` $m = f_0$

2. `FOR` $0 \leq i < n - 1$ `DO` $f_{i+1} = f_{i+1} + f_i/10, \quad f_i = f_i\%10$[†]

3. `RETURN` $m = f_{n-1}f_{n-2}\dots f_1 f_0$

Il primo metodo analizzato per la moltiplicazione polinomiale è quello classico, il meno efficiente ma il più intuitivo.

Tale metodo ha complessità quadratica, ossia saranno necessarie al più $O(n^2)$ operazioni nell'anello per moltiplicare due polinomi di grado minore di n.

Analizzeremo l'implemetazione in Linguaggio C di due strategie per moltiplicare polinomi a coefficienti interi in senso classico e li confronteremo sulla base dei programmi sviluppati, i Programmi A.1 e A.2.

Tutti gli algoritmi sono stati scritti in Linguaggio C, compilati ed eseguiti su un Notebook PC Asus CDC, processore Pentium 4, CPU 2.80 GHz, RAM 512 MB, HD 60 GB, sistema operativo Windows XP Home Edition, sempre alimentato a corrente e mai a batteria. É chiaro che compilando ed eseguendo gli stessi programmi su un computer con processore e memoria differenti si ottengono risultati diversi da quelli registrati.

Riportiamo un grafico per ogni metodo in cui riassumiamo tali risultati; in questo, come nei successivi grafici, in ascissa è il grado di ciascuno dei due polinomi da moltiplicare, che è la potenza di 2, 2^i con $i = 15, 16, 17, 18, 19, 20$, mentre nelle ordinate è il numero di secondi che il programma ha impiegato (o impiegherebbe) per la moltiplicazione.

[†]Con la notazione a/b si intende il quoziente nella divisione di a per b e con la notazione $a\%b$ si intende il resto.

I dati riportati sul primo metodo non sono stati effettivamente cronometrati ma calcolati stimando il numero di secondi necessario per eseguire una singola operazione, come spiegato nel dettaglio in 1.4.

I polinomi sono stati costruiti in maniera random, lavorando in base 100 e generando in modo casuale tra 0 e 99 ciascuno dei coefficienti. Il tempo impiegato da questa operazione non è stato considerato.

Diciamo inoltre che ogni programma, tranne il Programma A.1, è stato eseguito per 10 volte, e il numero di secondi riportato nel grafico è una media dei tempi registrati nelle 10 esecuzioni.

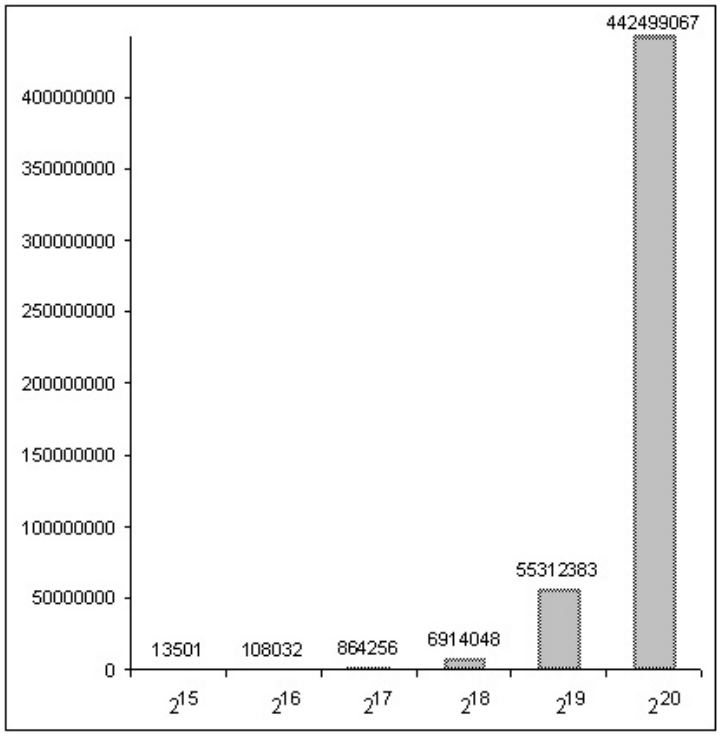

Figura 1: Primo Metodo per la Moltiplicazione in Senso Classico

Il secondo metodo, il Programma A.2, è molto più veloce del primo, e la differenza principale sta nel fatto che nel primo programma sono stati

concatenati 3 cicli `for` per il calcolo dei coefficienti del polinomio prodotto, mentre nel secondo programma solamente due.

Un'analisi più dettagliata è data in 1.4.

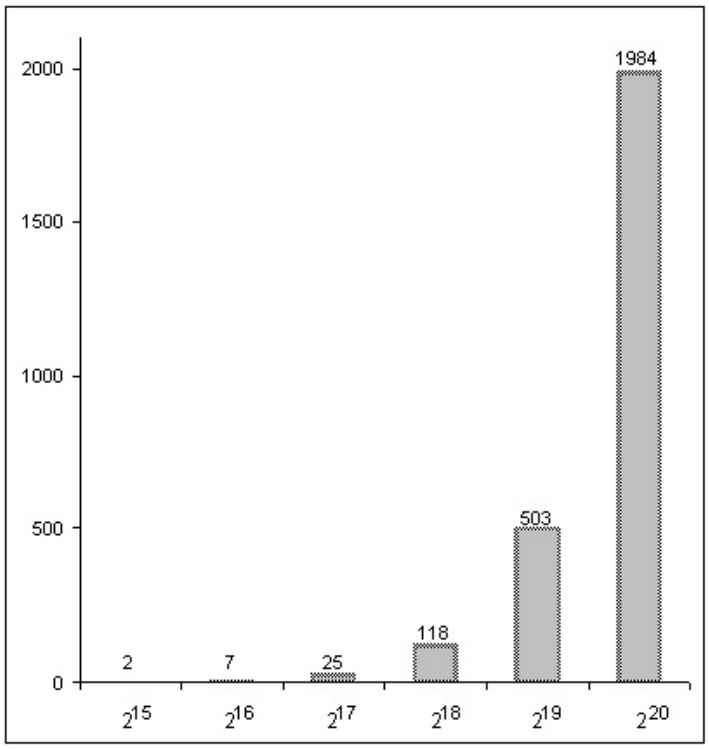

Figura 2: Secondo Metodo per la Moltiplicazione in Senso Classico

Nel Capitolo 2 basandoci su [1, 2, 3, 33] descriviamo l'algoritmo ricorsivo di Karatsuba, primo innovativo metodo che lavora su polinomi a coefficienti in un anello il cui grado è una potenza di 2 e fa scendere il costo della moltiplicazione a $O(n^{1.59})$ operazioni nell'anello.

Per capire l'idea alla base di questo metodo basta osservare che, per esempio, moltiplicare

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

richiede quattro moltiplicazioni ac, ad, bc, bd e una addizione $ad + bc$, ma si può ottenere più velocemente lo stesso risultato. Infatti se si calcolano ac, bd e $u = (a + b)(c + d)$, allora $ad + bc = u - ac - bd$ e sono state necessarie tre moltiplicazioni e quattro fra addizioni e sottrazioni.

In questo modo sembra che il numero delle operazioni sia aumentato a sette, ma osservando che l'addizione e la sottrazione sono più veloci della moltiplicazione e applicando ricorsivamente questo procedimento si diminuisce il tempo totale necessario per la moltiplicazione di partenza dei due polinomi.

Vediamo l'algoritmo associato a tale metodo:

- `Input:` $f, g \in R[x]$ `di grado minore di` n, n `potenza di` 2
- `Output:` $fg \in R[x]$

1. `IF` $n = 1$ `RETURN` $f \cdot g \in R$
2. $f = F_1 x^{n/2} + F_0$ `e` $g = G_1 x^{n/2} + G_0$ `con` $F_0, F_1, G_0, G_1 \in R[x]$ `di grado minore di` $n/2$
3. `Applica ricorsivamente lo stesso metodo per calcolare i prodotti` $F_0G_0, F_1G_1, (F_0 + F_1)(G_0 + G_1)$
4. `RETURN` $F_1G_1x^n + ((F_0 + F_1)(G_0 + G_1) - F_0G_0 - F_1G_1)x^{n/2} + F_0G_0$

Teorema 2.2.2. *Sia n una potenza di 2. L'algoritmo di Karatsuba per la moltiplicazione di polinomi di grado minore di n su un anello può essere fatto con al più $9n^{\log 3}$ o $O(n^{1.59})$ operazioni dell'anello.*

Teorema 2.2.3. *Sia n una potenza di 2. Utilizzando l'algortimo di Karatsuba, la moltiplicazione di interi di al più n cifre può essere fatta con al più $O(n^{1.59})$ operazioni bit.*

I risultati ottenuti con l'implementazione in Linguaggio C di tale metodo (decisamente più veloce dei due precedenti) sono riportati nel seguente grafico:

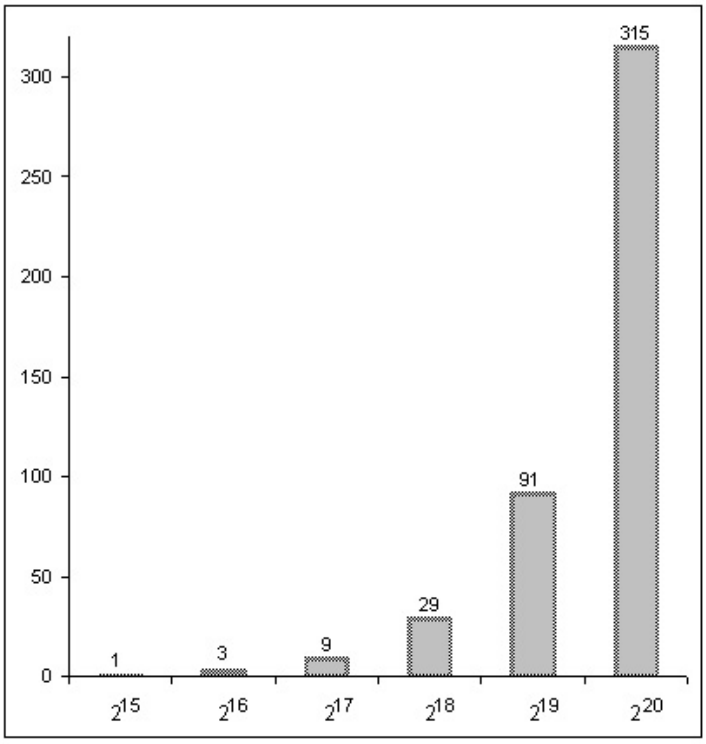

Figura 2.3: Moltiplicazione con il Metodo di Karatsuba in Linguaggio C

Confrontiamo quanto ottenuto con il metodo di Karatsuba implementato in Linguaggio C con il tempo utilizzato da GP/PARI CALCULATOR Versione 2.2.8, un software che nei suoi programmi sorgenti implementa proprio questo stesso algoritmo per la moltiplicazione.

Anche in PARI gli interi da moltiplicare sono stati generati in maniera casuale, e sono stati registrati i tempi medi ottenuti con 10 esecuzioni.

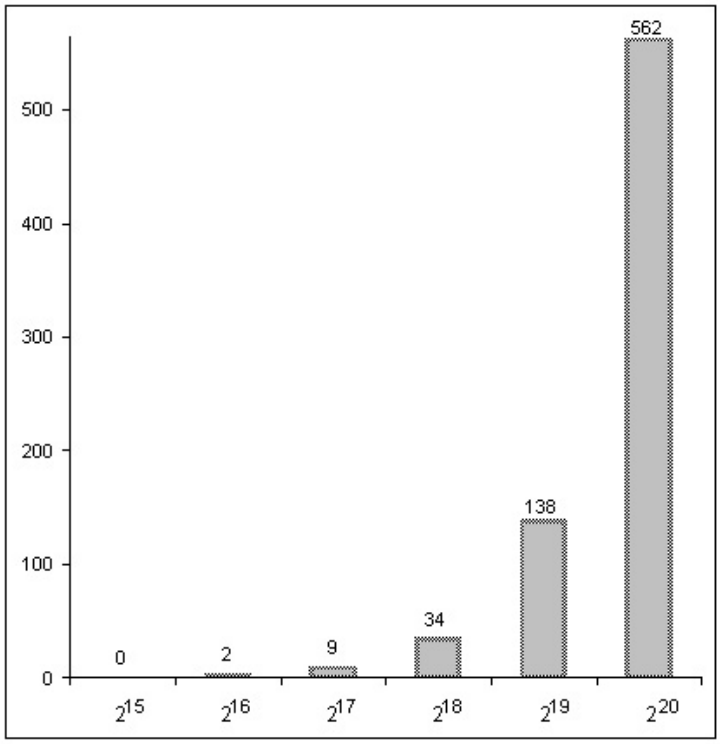

Figura 2.4: Moltiplicazione in PARI GP

Osserviamo, ad esempio, che se $n = 2^{19}$, ossia se moltiplichiamo due polinomi di grado minore di 2^{19}, il programma implementato in Linguaggio C con il metodo di Karatsuba è circa 270 mila volte più veloce del programma in Linguaggio C basato sul primo metodo classico, circa 6 volte più veloce del programma in C sul secondo metodo classico, e circa 2 volte più veloce dell'algoritmo di Karatsuba implementato nei sorgenti di PARI.

Nel Capitolo 3 introduciamo la Trasformata di Fourier Discreta (DFT) e la sua efficiente implementazione, la trasformata di Fourier Veloce (FFT).

Daremo la nozione di Convoluzione Veloce, basata su FFT, a partire dalla quale riusciremo a moltiplicare due polinomi a coefficienti in un anello R di grado minore di n con al più $O(n \log n)$ operazioni in R.

Per far ciò sarà necessario che R contenga radici primitive n-esime dell'unità.

Il nostro studio fa riferimento a [1, 2, 4, 5, 6, 17].

Definizione 3.2.3. *L'applicazione*

$$\begin{aligned} DFT_\omega \quad : \quad & R^n \longrightarrow R^n \\ & f \quad \longmapsto \quad \big(f(1), f(\omega), f(\omega^2), \ldots, f(\omega^{n-1})\big) \end{aligned}$$

che associa al polinomio f il vettore il cui elemento di posto i è $f(\omega^{i-1})$ si dice trasformata di Fourier discreta di f e si scrive $DFT_\omega(f)$.

Definizione 3.2.4. *La Convoluzione di due polinomi $f = \sum_{0 \le i < n} f_i x^i$ e $g = \sum_{0 \le j < n} g_j x^j \in R[x]$ è il polinomio*

$$h = f *_n g = \sum_{0 \le k < n} h_k x^k \in R[x]$$

dove

$$h_k = \sum_{i+j \equiv k \ mod \ n} f_i g_j \ \ per \ 0 \le k < n.$$

Sappiamo che un polinomio di grado m è univocamente determinato dal valore che assume in $m+1$ interi. Di conseguenza, per ottenere il prodotto $h(x) = f(x)g(x)$ di due polinomi f e g di grado $n-1$, è sufficiente calcolare h in $2n-1$ punti distinti ω_k, calcolando quindi $f(\omega_k) \cdot g(\omega_k)$, e useremo FFT per interpolare.

L'idea che sta alla base dei metodi di moltiplicazione di interi via FFT consiste nello scegliere come punti ω_k le radici complesse dell'unità.

Descriviamo allora l'algoritmo che consente di calcolare DFT in un generico anello R, come analizzato in 3.3:

- `Input :`
  - $n = 2^k \in \mathbb{N}$ `con` $k \in \mathbb{N}$
  - $f = \sum_{0 \leq i < n} f_i x^i \in R[x]$
  - `Le potenze` $1, \omega, \omega^2, \ldots, \omega^{n-1}$ `di una radice primitiva` n`-esima dell'unità` $\omega \in R$
- `Output :` $DFT_\omega(f) = (f(1), f(\omega), \ldots, f(\omega^{n-1})) \in R^n$

1. `IF` $n = 1$ `RETURN` f_0
2. $r_0 = \sum_{0 \leq j < n/2} (f_j + f_{j+n/2}) x^j, \quad r_1^* = \sum_{0 \leq j < n/2} (f_j - f_{j+n/2}) \omega^j x^j$
3. `Attraverso una chiamata ricorsiva calcola` r_0 `e` r_1^* `nelle potenze di` ω^2
4. `RETURN` $(r_0(1), r_1^*(1), r_0(\omega^2), r_1^*(\omega^2), \ldots, r_0(\omega^{n-2}), r_1^*(\omega^{n-2}))$

Teorema 3.3.2. *Sia n una potenza di 2 e $\omega \in R$ una radice primitiva n-esima dell'unità. Allora l'algoritmo appena descritto calcola correttamente DFT_ω usando al più $n \log n$ addizioni in R e al più $\frac{n}{2} \log n$ moltiplicazioni per una potenza di ω, per un totale di $\frac{3n}{2} \log n$ operazioni nell'anello.*

In 3.4 studiamo l'algoritmo di Convoluzione Veloce che utilizza FFT per calcolare il prodotto di due polinomi a coefficienti in R:

- `Input :`
  - $n = 2^k \in \mathbb{N}$ `con` $k \in \mathbb{N}$
  - $f, g \in R[x]$ `di grado minore di` n
  - `Una radice primitiva` n`-esima dell'unità` $\omega \in R$
- `Output :` $f * g \in R[x]$

1. `Calcola le potenze` $\omega^2, \ldots, \omega^{n-1}$
2. $\alpha = DFT_\omega(f), \beta = DFT_\omega(g)$
3. $\gamma = \alpha \cdot \beta$ `(prodotto componente per componente)`
4. `RETURN` $DFT_\omega^{-1}(\gamma) = \frac{1}{n} DFT_{\omega^{-1}}(\gamma)$

Teorema 3.4.2. *Sia R un anello che supporta FFT, $n = 2^k$ per qualche $k \in \mathbb{N}$. Allora la convoluzione in $R[x]/\langle x^n - 1\rangle$ coincide con la moltiplicazione di polinomi $f, g \in R[x]$ tali che $\deg(fg) < n$ e può essere fatta utilizzando al più $3n \log n$ addizioni in R, al più $\frac{3}{2} n \log n + n - 2$ moltiplicazioni per potenze di ω, n moltiplicazioni in R e n divisioni per n, per un totale di $\frac{9}{2} n \log n + O(n)$ operazioni nell'anello.*

Corollario 3.4.2. *Se R supporta FFT, allora i polinomi di grado minore di n in $R[x]$ possono essere moltiplicati con al più $18n \log n + O(n)$ operazioni in R.*

Fino ad ora abbiamo descritto l'algoritmo di moltiplicazione utilizzando la Trasformata di Fourier Veloce in un generico anello R e utilizzando $\omega \in R$ radice dell'unità.

Come vedremo in 3.4.1, per moltiplicare numeri interi, e dunque polinomi a coefficienti interi, lavoreremo in $\mathbb{C}$ con radici complesse dell'unità, seguendo l'idea di Xavier Gourdon e Pascal Sebah in [4].

Il programma che consente di moltiplicare polinomi a coefficienti interi, il Programma B.4, è stato implementato in Linguaggio C++, perchè questo mette a disposizione delle librerie che definiscono i numeri complessi e permettono di operare con essi.

Tale programma mostra come questo metodo sia il più veloce tra quelli implementati al computer per la moltiplicazione fra polinomi.

I numeri di secondi necessari per moltiplicare numeri a 2^k cifre, $k = 15, \ldots, 20$, sono riportati nel seguente grafico:

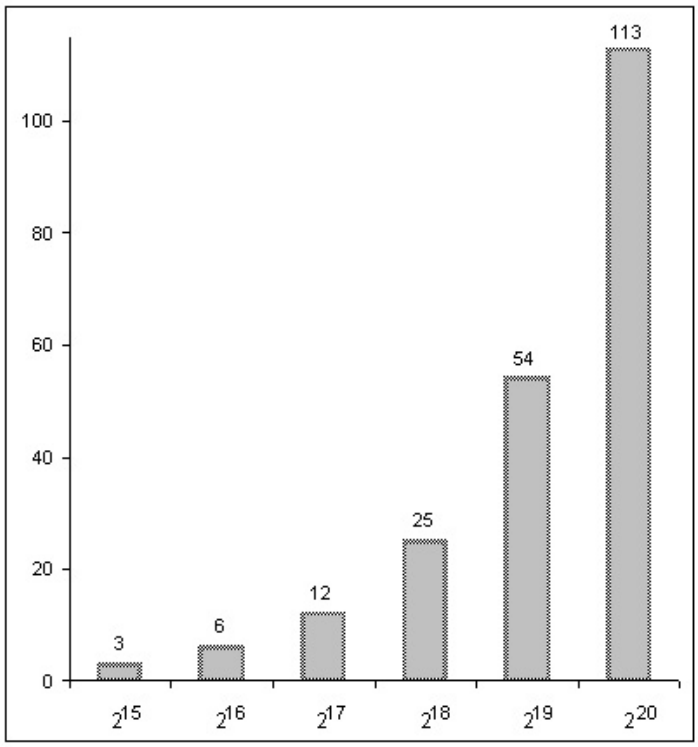

Figura 3.5: Moltiplicazione di polinomi a coefficienti interi via FFT

Osserviamo che se i polinomi che moltiplichiamo hanno grado minore di 2^{16} allora il programma in Linguaggio C basato sul metodo di Karatsuba è 2 volte più veloce del programma in Linguaggio C++ che implementa FFT, se hanno grado 2^{17} il distacco è minore ma Karatsuba è ancora circa 1.3 volte più veloce, mentre se i polinomi hanno grado 2^{18} la situazione si capovolge e il programma su FFT diventa circa 1.2 volte più veloce del programma su Karatsuba.

La differenza nella velocità di esecuzione dei due pogrammi è evidente per polinomi di grado 2^{20}: il programma su FFT è circa 3 volte più veloce del programma su Karatsuba.

Nel Capitolo 4 basandoci su [1, 2, 4, 25], trattiamo solo da un punto di vista teorico il metodo di Schönhage e Strassen, ancora basato sulla Trasformata di Fourier Veloce e sul calcolo della Convoluzione Negaciclica, così come definita in [2].

Abbiamo visto nel Capitolo 3 che per moltiplicare polinomi a coefficienti in un anello R di grado minore di n utilizzando il metodo basato su FFT, è necessario che R sia tale da contenere radici primitive n-esime dell'unità.

L'idea di Schönhage e Strassen è superare questo ostacolo lavorando nell'anello quoziente $R[x]/\langle x^n+1\rangle$ in cui $\omega = x \mod (x^n+1)$ è una radice primitiva $2n$-esima dell'unità.

In questo lavoro mostreremo che moltiplicare due polinomi il cui prodotto ha grado al più n modulo x^n+1 richiede al più $O(n \log n \log\log n)$ operazioni nell'anello.

Illustriamo l'algoritmo associato:

- Input: $f, g \in R[x]$ di grado minore di $n = 2^k$, con R anello commutativo tale che 2 è una unità in R
- Output: $h \in R[x]$ tale che $fg \equiv h \pmod{x^n + 1}$

1. IF $k \leq 2$ DO
   - Calcola fg utilizzando il metodo classico o con l'algoritmo di Karatsuba
   - RETURN $fg \pmod{x^n + 1}$
2. - $m = 2^{\lfloor k/2 \rfloor}$, $t = n/m$
   - $f', g' \in R[x, y]$ con $\deg_x(f'), \deg_x(g') < m$ tali che $f = f'(x, x^m)$, $g = g'(x, x^m)$
3. - $D = R[x]/\langle x^n + 1 \rangle$
   - IF $t = 2m$ THEN $\eta = x \pmod{x^{2m} + 1}$ ELSE $\eta = x^2 \pmod{x^{2m} + 1}$ (η è radice $2t$-esima primitiva dell'unità)
   - Siano $f^* = f' \pmod{x^{2m} + 1}$, $g^* = g' \pmod{x^{2m} + 1}$
   - Calcola $h^* \in D[y]$ con l'algoritmo di Convoluzione Veloce con $\omega = \eta^2$, in modo che $f^*(\eta y)g^*(\eta y) \equiv h^*(\eta y) \pmod{y^t - 1}$ utilizzando ricorsivamente l'algoritmo di convoluzione negaciclica per le moltiplicazioni in D

4. • `Sia` $h' \in R[x,y]$ `con` $deg_x(h') < 2m$ `tale che` $h^* \equiv h' \pmod{x^{2m}+1}$

   • $h = h'(x, x^m) \pmod{x^n + 1}$

5. `RETURN` h

Teorema 4.1.2. *L'algoritmo descritto calcola la Convoluzione Negaciclica Veloce usando al più* $\frac{9}{2}n \log n \log\log n + O(n \log n)$ *operazioni in* R.

Riassumiamo nella seguente tabella la complessità degli algortimi studiati e applicati alla moltiplicazione di interi con meno di n cifre, con n potenza di 2:

Metodo	Complessità
Classico	$2n^2$
Karatsuba	$O(n^{1.59})$
FFT	$O(n \log n (\log\log n)^2)$
Schönhage e Strassen basato su FFT	$O(n \log n \log\log n)$

Tabella 4.1: Algoritmi per la moltiplicazione veloce fra interi con meno di n cifre e loro complessità

Alla luce di quanto studiato ci chiediamo: si riuscirà a costruire un algoritmo per la moltiplicazione di interi di complessità lineare?

Concludiamo il nostro lavoro con due Appendici; nella prima riportiamo una breve storia del Calcolo Automatico in cui rivediamo le invenzioni e le idee che passo dopo passo hanno portato ai moderni computer ([7, 8, 9, 10, 11, 12, 13, 14]), e nella seconda trascriviamo i sorgenti dei programmi sviluppati.

Bibliografia

[1] Joachim von zur Gathen and Jürgen Gerhard. *Modern computer algebra*. Cambridge University Press, Cambridge, second edition, 2003.

[2] Richard Crandall and Carl Pomerance. *Prime numbers*. Springer-Verlag, New York, 2001. A computational prospective.

[3] Donald E. Knuth. *The art of computer programming. Vol. 2*. Addison-Wesley Publishing Co., Reading, Mass., second edition, 1981. Seminumerical algorithms, Addison-Wesley Series in Computer Science and Information Processing.

[4] Xavier Gourdon and Pascal Sebah. *FFT based multiplication of large numbers*. 2001. `http://numbers.computation.free.fr/Constants/Algorithms/fft.html`.

[5] M. Frigo and S. G. Johnson. FFTW, the fastest Fourier Transform in the west, 1997. `http:/www.fftw.org/`.

[6] Fast Fourier Transform. In *Wilkipedia, the free encyclopedia*. Da `http://en.wikipedia.org/wiki/Fast_Fourier_transform1`.

[7] Luca Fini. Chiedi all'esperto: architettura a 8, 16, 32, 64 bits. Da http://www.vialattea.net/esperti/inform/archit.html.

[8] Angela. Montanari. Antichi sistemi di numerazione. In *Storia delle matematiche elementari, una guida ad internet*. 1999. Da http://www.unife.it/altro/tesi/A.Montanari/computer.htm.

[9] Jeffrey Shallit. Discovery of a lost factoring machine. In *Talk for the American Mathematical Society, Orlando*.

[10] Roberto Mascia. La preistoria del computer. In *Storia dell'informatica*. 2000. Da http://www.itcbz.it/didattica/archivio/informatica/Generazioni/Preistoria/Preistoria.htm.

[11] L'abaco. In *'I racconti di Numeria'*. Argos, 1999. Da http://www.comune.roma.it/museomatematica/mouseABACO1.htm.

[12] Wilma e Lamberti Lamberto Di Palma. I bastoncini di Nepero. In *Le regole del regolo*. Bollati Boringhieri, 2000. Da http://www.sibiwin.it/matematica/mouseCALC2.htm.

[13] Paolo Giangrandi. Ingranaggi e calcolatori digitali. In *Numeri e macchine: breve storia degli strumenti di calcolo*. 2001. Da http:/www.dimi.uniud.it/cicloinf/mostra/pagina06.html.

[14] Paolo Bonavoglia. La macchina Enigma. In *Storia della crittografia, macchine cifranti*. Da http://digilander.libero.it/tuttisiti/crittografia.htm.

[15] Francesco Pappalardi. Prerequisiti di matematica. In *Note di crittografia a chiave pubblica, Fascicolo 1*. 2003.

Da http://www.mat.uniroma3.it/users/tfrance/corsi/CR1-2003/note-CR1.pdf.

[16] Daniel J. Bernstein. Multidigit multiplication for mathematicians. 2001. Da http://cr.yp.to/papers/m3.ps.

[17] Daniel J. Bernstein. Fast multiplication and its applications. 2003. Draft. Aimed at Buhler-Stevenhagen MSRI conference proceedings. Da http://cr.yp.to/papers/multapps.ps.

[18] Richard. P. Brent. The complexity of multiple-precision arithmetic. 1999. First appeared in *The complexity of computational problem solving*, edited by R. S. Anderssen and R. P. Brent, Universityy of Queensland Press, 1976, 126-175, Da http://web.comlab.ox.ac.uk/oucl/work/richard.brent/pub/pub032.html.

[19] Haitao Guo, Gary A. Sitton, and C. Sydney Burrus. The quick Fourier transform: an FFT based on symmetries. *IEEE Trans. Signal Process.*, 46(2):335–341, 1998.

[20] C. S. Burrus. Efficient Fourier transform and convolution algorithms. In *Advanced topics in signed processing*, Prentice Hall Signal Process. Ser., pages 199–245. Prentice Hall, Englewood Cliffs, NJ, 1988.

[21] Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus. Gauss and the history of the fast Fourier transform. *Arch. Hist. Exact Sci.*, 34(3):265–277, 1985.

[22] Chee Keng Yap. *Fundamental problems of algorithmic algebra*. Oxford University Press, New York, 2000.

[23] Richard Crandall and Barry Fagin. Discrete weighted transforms and large-integer arithmetic. *Math. Comp.*, 62(205):305–324, 1994.

[24] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms*. MIT Press, Cambridge, MA, second edition, 2001.

[25] Arnold Schönhage and Volker Strassen. Schnelle Multiplikation grosser Zahlen. *Computing (Arch. Elektron. Rechnen)*, 7:281–292, 1971.

[26] George U. Ramos. Roundoff error analysis of the fast Fourier transform. *Math. Comp.*, 25:757–768, 1971.

[27] James W. Cooley. How the FFT gained acceptance. In *A history of scientific computing (Princeton, NJ, 1987)*, ACM Press Hist. Ser., pages 133–140. ACM, New York, 1990.

[28] L. Auslander, E. Feig, and S. Winograd. The multiplicative complexity of the discrete Fourier transform. *Adv. in Appl. Math.*, 5(1):87–109, 1984.

[29] S. Winograd. On multiplication of polynomials modulo a polynomial. *SIAM J. Comput.*, 9(2):225–229, 1980.

[30] S. Winograd. On computing the discrete Fourier transform. *Math. Comp.*, 32(141):175–199, 1978.

[31] Donald E. Knuth. The analysis of algorithms. In *Actes du Congrès International des Mathématiciens (Nice, 1970), Tome 3*, pages 269–274. Gauthier-Villars, Paris, 1971.

[32] Arnold Schönhage. Asymptotically fast algorithms for the numerical multiplication and division of polynomials with complex coefficients. In *Computer algebra (Marseille, 1982)*, volume 144 of *Lecture Notes in Comput. Sci.*, pages 3–15. Springer, Berlin, 1982.

[33] Mircea Drăgan, Tudor Jebelean, Viorel Negru, and Dorin Ţepeneu. Parallel algorithms for practical multiprecision arithmetic using the Karatsuba method. *An. Univ. Timişoara Ser. Mat.-Inform.*, 38(2):51–56, 2000. The 2nd International Workshop on Symbolic and Numeric Algorithms on Scientific Computing (Timişoara, 2000).

[34] P. Duhamel and M. Vetterli. Fast Fourier transforms: a tutorial review and a state of the art. *Signal Process.*, 19(4):259–299, 1990.

[35] B.W. Kernighan and D.M. Ritchie. *Linguaggio C.* Gruppo Editoriale Jackson, 1985.

[36] H. Schildt. *Linguaggio C, la guida completa.* McGraw-Hill, 1995.

[37] Marco Liverani. *Programmare in C, guida al linguaggio attraverso esercizi svolti e commentati.* Esculapio, 2001.