Leonardo Web con cui sono state realizzate le animazioni a corredo di questo testo.

Molti dei nostri studenti hanno inoltre avuto un ruolo fondamentale, aiutandoci soprattutto a correggere alcuni errori nei nostri appunti delle lezioni e a migliorare l'esposizione dei contenuti.

Ringraziamo infine le Università di Roma "La Sapienza" e di Roma "Tor Vergata" per l'ambiente stimolante che ci hanno fornito. Ringraziamo inoltre Columbia University e Hong Kong University of Science and Technology, che ci hanno ospitato generosamente, e in cui abbiamo sviluppato parte del materiale in questo libro.

È stato un vero piacere lavorare con McGraw-Hill. Ringraziamo in particolare Chiara Tartara, Rossana Cecchi, Alessandra Porcelli e Daniela Cipollone per il loro continuo incoraggiamento e supporto.

Infine desideriamo ringraziare le nostre famiglie, che ci hanno supportato con amore, affetto e infinita pazienza durante la scrittura di questo libro. Questo libro è dedicato soprattutto a loro, con amore e riconoscenza.

Avremmo voluto scrivere un libro perfetto. Ma siamo autori imperfetti, e quindi perfettamente consapevoli che questo libro conterrà errori ed imprecisioni, di cui siamo ovviamente gli unici responsabili. Saremo grati a chiunque vorrà segnalarceli per posta elettronica all'indirizzo

`algoritmi@algonet.uniroma2.it`

Roma, Aprile 2004

Camil Demetrescu
Irene Finocchi
Giuseppe F. Italiano

1

Un'introduzione informale agli algoritmi

Quelli che s'innamorano di pratica, sanza scienza, son come 'l nocchiere, ch'entra in navilio sanza timone o bussola, che mai ha certezza dove si vada.

(Leonardo Da Vinci)

Intuitivamente, un algoritmo è un insieme di istruzioni, definite passo per passo, in modo tale da poter essere eseguite meccanicamente, e tali da produrre un determinato risultato. Sempre in modo informale potremmo definire un algoritmo come una sequenza di passi di calcolo che, ricevendo in *ingresso* un valore (od un insieme di valori), restituisce in *uscita* un altro valore (od un insieme di valori), trasformando quindi i dati in ingresso in dati in uscita. Gli algoritmi non sono ristretti soltanto alle discipline informatiche o matematiche, e probabilmente sono esistiti ancor prima che si coniasse un termine speciale per indicarli. Tutti noi utilizziamo, più o meno consapevolmente, algoritmi nella nostra vita quotidiana. Ad esempio, l'algoritmo illustrato nella Figura 1.1 mostra come preparare del caffè a partire da una caffettiera, e da un'opportuna quantità di acqua e di polvere di caffè. Notiamo che l'algoritmo di Figura 1.1 è definito mediante una sequenza *finita* di

algoritmo `preparaCaffè`
1. Svita la caffettiera.
2. Riempi d'acqua il serbatoio della caffettiera.
3. Inserisci il filtro.
4. Riempi il filtro con la polvere di caffè.
5. Avvita la parte superiore della caffettiera.
6. Metti la caffettiera, così predisposta, su un fornello acceso.
7. Spegni il fornello quando il caffè è pronto.
8. Versa il caffè nella tazzina.

Figura 1.1 Algoritmo per preparare il caffè.

passi elementari, descritti in un linguaggio naturale, in modo tale da poter essere interpretati da un essere umano. Molti degli algoritmi che vedremo in questo libro sono concepiti per essere eseguiti sulla CPU di un sistema di elaborazione,

e quindi saranno descritti in *pseudocodice*, che ricorda linguaggi di programmazione come C, C++, o Java, ma contiene alcune frasi in italiano (o in inglese) piuttosto che direttamente in un linguaggio di programmazione.

In questo libro ci occuperemo del progetto di algoritmi per vari tipi di problemi, e della loro analisi matematica, analisi che condurremo anche indipendentemente da eventuali esperimenti sulle implementazioni degli algoritmi. Lo scopo di progettare algoritmi dovrebbe essere evidente: abbiamo bisogno di algoritmi ogni qualvolta dobbiamo scrivere un programma. La necessità di analizzare algoritmi sembrerebbe a prima vista meno ovvia, anche se alcune motivazioni possono facilmente rendere conto della sua importanza.

Innanzitutto, cosa ci può offrire l'analisi di un algoritmo rispetto alla valutazione sperimentale delle prestazioni di un programma? L'analisi sembrerebbe essere, almeno in linea di principio, più affidabile: nella sperimentazione, possiamo solo studiare il comportamento di un programma in un numero limitato di casi, mentre al contrario l'analisi può offrirci delle precise garanzie matematiche sulle prestazioni di un algoritmo per ogni possibile distribuzione dei dati di ingresso. Inoltre, l'analisi ci può aiutare a scegliere tra diverse soluzioni allo stesso problema. Un'analisi attenta può esserci utile nel decidere quale soluzione possa essere più adeguata ai nostri obiettivi, senza costringerci a costose implementazioni ed ai relativi test dei programmi prima di accorgerci che abbiamo scelto di percorrere una strada sbagliata. Possiamo anche predire le prestazioni di un programma software, prima ancora di scriverne le prime linee di codice. In grossi progetti software, è di vitale importanza avere una stima delle prestazioni già a livello progettuale: attendere che tutto il codice sia scritto e scoprire soltanto allora che qualcosa non raggiunge i requisiti prestazionali, potrebbe portare a conseguenze disastrose o per lo meno molto costose. Quando analizziamo un algoritmo, abbiamo la possibilità di scoprire eventuali problemi di prestazioni già in una prima fase di analisi preliminare, e possiamo quindi tentare di eliminarli direttamente in quella fase. Infine, durante l'analisi di un algoritmo, riusciamo ad avere un'idea migliore di quali siano le sue componenti più lente e di quali siano le sue componenti più veloci, e di conseguenza possiamo metterci al lavoro sulle componenti più lente già in questa fase per rendere tutta l'implementazione globalmente più veloce.

1.1 I numeri di Fibonacci

In questo paragrafo introdurremo alcune delle problematiche relative al progetto e all'analisi di algoritmi tramite un problema "giocattolo": il calcolo dei numeri di Fibonacci. Probabilmente non avremo mai bisogno di risolvere questo problema nella nostra vita professionale, ma lo consideriamo qui perché ha una buona valenza didattica, è abbastanza semplice da comprendere e, sorprendentemente, può essere risolto con molti approcci distinti.

L'isola dei conigli. Leonardo di Pisa (noto anche come Fibonacci) è un noto matematico italiano, vissuto intorno al 1200. Fibonacci si interessò di molte cose, e

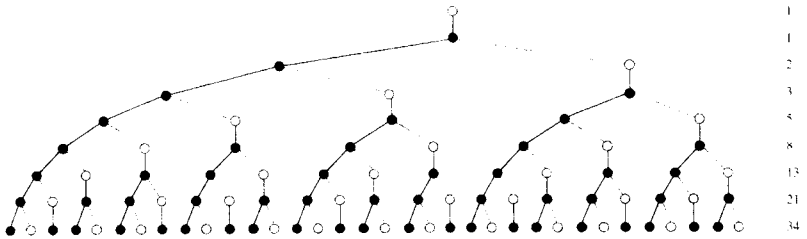

Figura 1.2 I conigli di Fibonacci. Ogni livello dell'albero rappresenta un anno. I nodi neri e bianchi rappresentano rispettivamente coppie di conigli adulti e giovani.

tra le altre studiò anche vari problemi che al giorno d'oggi definiremmo di "dinamica delle popolazioni". Ad esempio, studiò come si espande una popolazione di conigli, sotto opportune condizioni. Immaginiamo di fare il seguente esperimento: partiamo da una singola coppia di conigli in un'isola deserta, e proviamo a studiarne l'evoluzione nel corso degli anni. Come avviene spesso in matematica – e l'analisi di algoritmi utilizza strumenti matematici – rendiamo il problema più astratto, per avere un'idea delle sue caratteristiche generali senza rischiare di perderci troppo nei dettagli di basso livello. In particolare, assumiamo che:

(1) Una coppia di conigli genera due coniglietti ogni anno.
(2) I conigli cominciano a riprodursi soltanto al secondo anno dopo la loro nascita.
(3) I conigli sono immortali.

L'ultima ipotesi è sicuramente irrealistica, ma indubbiamente rende il problema più semplice da analizzare. Dopo averlo analizzato nella sua versione semplificata, potremmo tornare indietro e sostituire questa ipotesi con una più realistica, del tipo "I conigli vivono in media dieci anni". Così facendo, ci renderemmo conto che questo non cambierebbe troppo il risultato.

Proviamo quindi ad esprimere il numero di coppie di conigli in funzione del tempo, ovvero in funzione del numero di anni trascorsi dall'inizio dell'esperimento. Sia F_t il numero di coppie di conigli all'anno t:

$F_1 = 1$: Partiamo con una coppia
$F_2 = 1$: Nel secondo anno, la coppia è ancora troppo giovane per riprodursi
$F_3 = 2$: Nel terzo anno, la coppia iniziale dà alla luce un'altra coppia
$F_4 = 3$: Nel quarto anno nasce un'altra coppia di coniglietti
$F_5 = 5$: Nel quinto anno nascono i primi nipoti della coppia iniziale!

Questo fenomeno riproduttivo è illustrato in Figura 1.2. In generale, nell'anno n sono presenti tutti i conigli dell'anno precedente (F_{n-1} coppie) ed inoltre abbiamo una coppia di conigli per ogni coppia presente due anni prima (F_{n-2} coppie). Avremo quindi:

$$F_n = F_{n-1} + F_{n-2}$$

Il problema che vogliamo risolvere è, dato un intero $n \geq 1$, come calcolare il valore di F_n. In pratica, vogliamo trovare la soluzione della seguente *relazione di ricorrenza*:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{se } n \geq 3 \\ 1 & \text{se } n = 1, 2 \end{cases}$$

Faremo riferimento a tale relazione come *relazione di Fibonacci*. I primi numeri della sequenza di Fibonacci $\{F_n\}_{n \geq 1}$ sono i seguenti:

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
F_n	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610

Nei paragrafi seguenti affronteremo il problema del calcolo dei numeri di Fibonacci, presentando diverse soluzioni ed evidenziandone pregi e difetti.

1.2 Un algoritmo numerico

Per ottenere un primo algoritmo, cerchiamo una funzione matematica che calcoli direttamente i numeri di Fibonacci. A questo scopo, consideriamo innanzitutto funzioni che soddisfino la relazione di ricorrenza $F_n = F_{n-1} + F_{n-2}$. Proviamo con funzioni esponenziali della forma a^n con $a \neq 0$, e vediamo per quali valori di a esse soddisfano la relazione di Fibonacci. Studiamo quindi l'equazione:

$$a^n = a^{n-1} + a^{n-2}$$

ovvero

$$a^n - a^{n-1} - a^{n-2} = 0$$

che è equivalente a

$$a^{n-2} \cdot (a^2 - a - 1) = 0$$

Per l'ipotesi $a \neq 0$, cerchiamo i valori di a che soddisfano l'equazione:

$$a^2 - a - 1 = 0$$

Questa equazione di secondo grado ammette due radici reali [1]:

$$\begin{aligned} \phi &= \frac{1+\sqrt{1+4}}{2} = \frac{1+\sqrt{5}}{2} \approx +1.618 \\ \hat{\phi} &= \frac{1-\sqrt{1+4}}{2} = \frac{1-\sqrt{5}}{2} \approx -0.618 \end{aligned}$$

[1] Il numero $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ è importante in matematica, ma anche nel mondo dell'arte, dove appare sin dall'antichità come rapporto dimensionale di particolare pregio estetico nei progetti architettonici. Si denota tale grandezza come *sezione aurea*, e si usa la lettera greca ϕ in onore dello scultore Fidia, che si dice ne abbia fatto un particolare uso nelle sue opere.

algoritmo `fibonacci1`$(intero\ n) \rightarrow intero$

1. **return** $\frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right)$

Figura 1.3 Algoritmo `fibonacci1` per il calcolo dell'n-esimo numero di Fibonacci.

Abbiamo quindi trovato due funzioni ϕ^n e $\hat{\phi}^n$ che soddisfano la relazione di Fibonacci. Purtroppo, però, nessuna delle due calcola i numeri di Fibonacci come vorremmo: infatti, ad esempio, $\phi^2 \neq F_2$. Il punto sembra essere che, nel definire ϕ^n e $\hat{\phi}^n$, non abbiamo considerato i casi base della definizione ricorsiva.

Per risolvere il problema, osserviamo che una qualunque combinazione lineare di funzioni che soddisfano la relazione di Fibonacci soddisfa anch'essa la relazione di Fibonacci. Proviamo quindi a vedere se un'opportuna combinazione lineare $c_1 \cdot \phi^n + c_2 \cdot \hat{\phi}^n$ di ϕ^n e $\hat{\phi}^n$ può darci la funzione F_n cercata. Imponendo che vengano soddisfatti i casi base della definizione ricorsiva $F_1 = 1$ ed $F_2 = 1$, otteniamo il sistema di due equazioni in due incognite c_1 e c_2:

$$\begin{cases} c_1 \cdot \phi + c_2 \cdot \hat{\phi} = 1 \\ c_1 \cdot \phi^2 + c_2 \cdot \hat{\phi}^2 = 1 \end{cases}$$

Risolvendo il sistema otteniamo:

$$c_1 = +\frac{1}{\sqrt{5}} \qquad c_2 = -\frac{1}{\sqrt{5}}$$

La soluzione della relazione di Fibonacci risulta pertanto essere:

$$F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right)$$

Tornando al nostro problema originario, ovvero calcolare l'n-esimo numero di Fibonacci, un primo algoritmo risulta quindi essere quello mostrato in Figura 1.3.

Osservando che i numeri coinvolti nel calcolo sono reali, sorge spontaneo chiedersi: quale deve essere l'accuratezza di ϕ e $\hat{\phi}$ per ottenere la riposta esatta? Ad esempio, usando tre cifre decimali, ovvero $\phi \approx 1.618$ e $\hat{\phi} \approx -0.618$, otteniamo:

`fibonacci1`$(3) = 1.99992$:	arrotondando siamo in grado di ottenere il valore corretto $F_3 = 2$.
`fibonacci1`$(16) = 986.698$:	arrotondando siamo in grado di ottenere il valore corretto $F_{16} = 987$.
`fibonacci1`$(18) = 2583.1$:	arrotondando otteniamo 2583, mentre invece il valore corretto è $F_{18} = 2584$.

```
algoritmo fibonacci2(intero n) -- intero
1.    if ( n ≤ 2 ) then return 1
2.    else return fibonacci2(n - 1) + fibonacci2(n - 2)
```

Figura 1.4 Algoritmo `fibonacci2` per il calcolo dell'n-esimo numero di Fibonacci.

Il limite dell'algoritmo `fibonacci1`, che utilizza direttamente la soluzione della relazione di Fibonacci, sembra evidente: siamo costretti a lavorare con numeri reali, che sono rappresentati nei calcolatori con una precisione limitata, e corriamo quindi il rischio di fornire risposte non corrette a causa degli errori di arrotondamento. Dato che i numeri di Fibonacci sono interi, non sarebbe più semplice progettare un algoritmo che lavora soltanto su interi? Vedremo vari esempi di algoritmi che lavorano soltanto con numeri interi nelle pagine seguenti.

1.3 Un algoritmo ricorsivo

La relazione di Fibonacci sembra essere naturalmente ricorsiva. Sembrerebbe quindi naturale risolvere il nostro problema con un algoritmo ricorsivo, così come probabilmente abbiamo già visto in un corso introduttivo di programmazione. Lo pseudocodice di questo approccio è mostrato in Figura 1.4. Una tipica domanda che ci porremo frequentemente in questo libro è: "Quanto tempo richiede questo algoritmo?" Per rispondere a questa domanda dobbiamo innanzitutto chiarire come misurare il tempo. La risposta più ovvia sembrerebbe "in secondi", ma sarebbe importante riuscire ad ottenere una risposta che sia indipendente dalle tecnologie, ossia una risposta che non cambia ogni volta che si mette in produzione una nuova CPU. Potremmo quindi pensare di misurare il tempo in termini di istruzioni macchina; dividendo per la velocità della macchina (istruzioni/secondo) otteremmo i tempi esatti. Tuttavia, è estremamente complicato inferire da un pezzo di codice o di pseudocodice il numero esatto di istruzioni che saranno generate da un particolare compilatore. Per avere una prima approssimazione, potremmo quindi decidere di misurare la velocità di un algoritmo in termini di linee di codice eseguite.

Analizziamo quindi quante linee di codice richiede l'algoritmo `fibonacci2`. Notiamo innanzitutto che ogni chiamata alla funzione `fibonacci2` coinvolge una oppure due linee di codice.

- Se $n \leq 2$, viene eseguita solamente una linea di codice, e più precisamente l'istruzione **if** $(n \leq 2)$ **return** 1.
- Se $n = 3$, vengono eseguite 2 linee di codice per la chiamata `fibonacci2(3)`, più una linea di codice per la chiamata `fibonacci2(2)` ed una per la chiamata `fibonacci2(1)`. Questo comporta un totale di 4 linee di codice.

- Se $n = 4$, vengono eseguite 2 linee di codice per la chiamata `fibonacci2(4)`, più 4 linee di codice per la chiamata `fibonacci2(3)` ed una per la chiamata `fibonacci2(2)`. In totale, 7 linee di codice.

Cominciamo ad intuire che le linee di codice della funzione `fibonacci2` si riproducono in modo molto simile ai conigli di Fibonacci! Oltre alle due linee in ogni chiamata, il numero di linee di codice mandate in esecuzione in occasione di una chiamata alla funzione `fibonacci2`(n) è dato dalla somma delle linee di codice mandate in esecuzione dalle due chiamate ricorsive:

$$T(n) = 2 + T(n-1) + T(n-2)$$

In generale, ogni algoritmo ricorsivo come `fibonacci2` può essere analizzato tramite una *relazione di ricorrenza*: il tempo richiesto da una routine è uguale al tempo speso all'interno della routine più il tempo speso per le chiamate ricorsive. Questo produce quasi automaticamente una relazione di ricorrenza, che possiamo risolvere per trovare il tempo di esecuzione. In questo caso particolare, la relazione di ricorrenza è molto simile alla definizione dei numeri di Fibonacci.

Con un po' di lavoro, possiamo risolvere l'equazione almeno in termini dei numeri di Fibonacci F_n. Possiamo rappresentare le chiamate ricorsive con una struttura ad un albero: l'*albero della ricorsione*. In dettaglio, usiamo un nodo, la radice dell'albero, per la prima chiamata, e generiamo un figlio per ogni chiamata ricorsiva. L'albero delle chiamate ricorsive a partire dalla chiamata `fibonacci2(8)` è mostrato in Figura 1.5. Ognuno dei venti nodi interni dell'albero per `fibonacci2(8)` richiede due linee di codice, mentre le ventuno foglie richiedono una linea ciascuna. Il numero totale di linee di codice mandato in esecuzione è quindi $20 \cdot 2 + 21 = 61$. Come facciamo ad analizzare il numero di

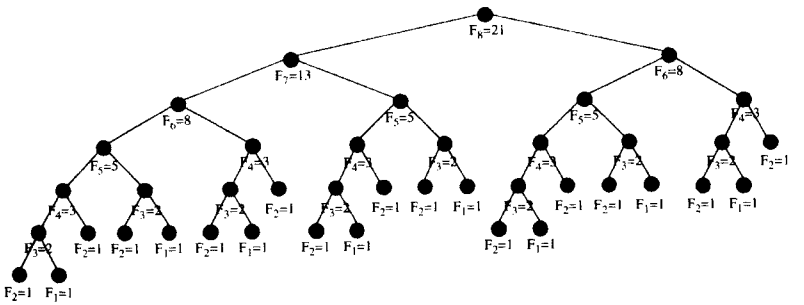

Figura 1.5 Chiamate ricorsive della funzione `fibonacci2` per il calcolo di F_8.

linee di codice mandato in esecuzione per una generica chiamata alla funzione `fibonacci2`(n)? I seguenti lemmi rispondono a questa domanda.

Lemma 1.1 *Sia T_n l'albero delle chiamate ricorsive della funzione* `fibonacci2`(n). *Il numero di foglie in T_n è esattamente uguale al numero di Fibonacci F_n.*

Dimostrazione. Dimostriamo il lemma per induzione. La base è banalmente verificata: se $n = 1$, $F_1 = 1$ e l'albero della ricorsione T_1 contiene un solo nodo e quindi una sola foglia. Anche per $n = 2$, $F_2 = 1$ e l'albero della ricorsione T_2 contiene un solo nodo ed una sola foglia.

Dimostriamo ora il passo induttivo. Sia $n > 2$ ed assumiamo induttivamente che l'enunciato del lemma sia verificato per ogni k, $2 \leq k \leq (n-1)$. Vogliamo dimostrare che il lemma vale anche per $k = n$. Se $n > 2$, sappiamo che l'albero della ricorsione T_n ha come sottoalbero sinistro T_{n-1} (l'albero delle chiamate ricorsive della funzione `fibonacci2`$(n-1)$) e come sottoalbero destro T_{n-2} (l'albero delle chiamate ricorsive della funzione `fibonacci2`$(n-2)$). Per l'ipotesi induttiva, il numero delle foglie di T_{n-1} è esattamente uguale a F_{n-1}, mentre il numero delle foglie di T_{n-2} è esattamente uguale a F_{n-2}. Il numero totale delle foglie in T_n sarà quindi uguale a $F_{n-1} + F_{n-2} = F_n$. □

Come conseguenza del Lemma 1.1, il numero totale di foglie contenute nell'albero della ricorsione della funzione `fibonacci2`(n) è dato esattamente da F_n. Nell'albero della ricorsione della funzione `fibonacci2`(n) ogni foglia è responsabile di una linea di codice (base della ricorsione), mentre ogni nodo interno dell'albero è responsabile di due linee di codice (chiamata ricorsiva). Per contare il numero di nodi interni, possiamo usare le proprietà di alberi in cui ogni nodo interno ha due figli. In particolare, il seguente lemma dimostra che il numero di nodi interni di un tale albero binario è sempre uguale al numero di foglie meno uno.

Lemma 1.2 *Sia T un albero binario in cui ogni nodo interno ha esattamente due figli. Allora il numero di nodi interni di T è sempre uguale al numero di foglie diminuito di uno.*

Dimostrazione. Ancora una volta la dimostrazione procederà per induzione. Sia n il numero di nodi dell'albero T. Esaminiamo la base dell'induzione: se $n = 1$, T ha solo una foglia e nessun nodo interno, quindi la condizione è banalmente verificata.

Supponiamo ora per ipotesi che la condizione valga per tutti gli alberi con meno di n nodi, e dimostriamo che deve valere anche per T. Se f e i sono il numero di foglie e di nodi interni di T, rispettivamente, vogliamo quindi dimostrare che $i = f - 1$. Sia $\widehat{T}$ un albero ottenuto da T rimuovendo una qualunque coppia di foglie con lo stesso padre. Questo albero avrà $i - 1$ nodi interni e $f - 2 + 1 = f - 1$ foglie. Per l'ipotesi induttiva, poiché $\widehat{T}$ ha meno nodi di T, si ha che $(i-1) = (f-1) - 1$. Sommando a entrambi i membri 1 si ottiene proprio l'uguaglianza $i = f - 1$ che si voleva dimostrare. □

Sia T_n l'albero di ricorsione della funzione `fibonacci2`(n). In base al Lemma 1.1 il numero di foglie di T_n è esattamente uguale a F_n, mentre per il Lemma 1.2 il numero di nodi interni di T_n è esattamente uguale a $(F_n - 1)$. Notiamo che ogni foglia dell'albero di ricorsione T_n corrisponde al caso base (linea 1) della funzione ricorsiva `fibonacci2`(n)): per ogni foglia in T_n viene quindi mandata in esecuzione una sola linea di codice, ed in particolare la linea 1. In maniera


```
algoritmo fibonacci3(intero n) → intero
1.  Sia Fib un array di n interi
2.  Fib[1] ← Fib[2] ← 1
3.  for i = 3 to n do
4.      Fib[i] ← Fib[i − 1] + Fib[i − 2]
5.  return Fib[n]
```

Figura 1.6 Algoritmo `fibonacci3` per il calcolo dell'n-esimo numero di Fibonacci.

analoga, ogni nodo interno dell'albero di ricorsione T_n corrisponde alla ricorsione nella funzione `fibonacci2`(n)): per ogni foglia in T_n vengono quindi mandate in esecuzione due linee di codice, la linea 1 e la linea 2.

In sintesi, vengono dunque mandate in esecuzione F_n linee di codice (una per ogni foglia dell'albero di ricorsione), più $(2 \cdot F_n - 2)$ linee di codice (due per ogni nodo interno dell'albero di ricorsione), per un totale di $(3 \cdot F_n - 2)$ linee di codice. L' algoritmo `fibonacci2`(n) sembra indubbiamente molto lento: il numero di linee di codice mandate in esecuzione cresce come i conigli di Fibonacci! Ad esempio, per $n = 8$ vengono mandate in esecuzione

$$3 \cdot F_8 - 2 = 3 \cdot (21) - 2 = 61$$

linee di codice. Per $n = 45$ l'algoritmo richiede

$$3 \cdot F_{45} - 2 = 3 \cdot 1.134.903.170 - 2 = 3.404.709.508$$

ovvero più di tre miliardi di linee di codice! Possiamo fare di meglio? La risposta è affermativa, e la vedremo nel prossimo paragrafo.

1.4 Un algoritmo iterativo

Perché l'algoritmo `fibonacci2` è lento? Perché continua a ricalcolare ripetutamente la soluzione dello stesso sottoproblema! Ad esempio, consideriamo l'albero delle chiamate ricorsive per il calcolo di F_8. La seconda volta che compare, ad esempio, F_4, perdiamo tempo a ricalcolarlo: in realtà, lo abbiamo già calcolato, e la risposta non cambierà di certo. Per fare di meglio, potremmo quindi risolvere ogni sottoproblema una volta soltanto, memorizzare questa soluzione, ed usarla nel seguito invece di ricalcolarla.

Questa semplice idea è alla base di una tecnica algoritmica, chiamata *programmazione dinamica*, che vedremo più in dettaglio nel Capitolo 10, e che può produrre algoritmi molto più complicati di questo. Nel nostro caso, la soluzione è molto semplice, come può desumersi dall'esame della Figura 1.6. L'algoritmo `fibonacci3` è un algoritmo iterativo, in cui utilizziamo dei cicli al posto della ricorsione. Per questo motivo, dobbiamo analizzarlo in modo un po' diverso da un

	`fibonacci2(58)`	`fibonacci3(58)`
Pentium IV 1700MHz	15820 sec. ($\simeq$ 4 ore)	0.7 milionesimi di secondo
Pentium III 450MHz	43518 sec. ($\simeq$ 12 ore)	2.4 milionesimi di secondo
PowerPC G4 500MHz	58321 sec. ($\simeq$ 16 ore)	2.8 milionesimi di secondo

Figura 1.7 Tempo richiesto su varie architetture dall'implementazione in C degli algoritmi `fibonacci2` e `fibonacci3` su input $n = 58$.

algoritmo ricorsivo. In pratica, per ogni linea di codice calcoliamo quante volte essa è eseguita, esaminando a quali cicli appartiene e quante volte tali cicli sono eseguiti.

In ogni caso, si mandano in esecuzione sempre tre linee di codice (righe 1, 2 e 5). La prima linea del ciclo (riga 3) viene eseguita $(n-1)$ volte, ad eccezione del caso $n = 1$. La seconda linea di codice del ciclo (riga 4) viene eseguita $(n-2)$ volte, ad eccezione del caso $n = 1$. Quindi, indicando con $T(n)$ il numero di linee di codice mandate in esecuzione dalla funzione `fibonacci3`(n), si ha

$$T(n) = n - 1 + n - 2 + 3 = 2n$$

ad eccezione del caso $T(1) = 4$.

Ad esempio, per $n = 45$ ci vogliono 90 passi, e l'algoritmo `fibonacci3` è all'incirca 38 milioni di volte più veloce dell'algoritmo `fibonacci2`, che invece richiede $3 \cdot F_{45} - 2 = 3.404.709.508$ di passi. Per $n = 58$ l'algoritmo `fibonacci3` richiede 116 passi, mentre l'algoritmo `fibonacci2` richiede $3 \cdot F_{58} - 2 = 1.773.860.189.635$, ovvero più di 1700 miliardi di passi! Quindi, nel caso $n = 58$, ci aspettiamo che l'algoritmo `fibonacci3` sia all'incirca 15 miliardi di volte più veloce dell'algoritmo `fibonacci2`. La Figura 1.7 evidenzia, per $n = 58$, le differenze di efficienza degli algoritmi `fibonacci2` e `fibonacci3` implementati in `C` su architetture disponibili in commercio nel momento in cui questo libro è stato scritto. I dati riportati confermano la nostra predizione teorica sul numero di linee di codice mandate in esecuzione: nelle varie architetture, l'implementazione di `fibonacci3` risulta essere in pratica da 18 a 22 miliardi di volte più veloce di quella di `fibonacci2`.

1.5 Occupazione di memoria

Il tempo di esecuzione non è l'unica misura che ci interessa, o l'unica metrica che può essere analizzata per valutare l'efficienza di un algoritmo. Anche il tempo richiesto per scrivere il codice (tempo di programmazione) e la lunghezza del codice prodotto sono parametri importanti, ma sono nozioni che saranno approfondite in maggior dettaglio nei corsi di ingegneria del software. In questo libro, analizzeremo invece frequentemente, oltre al tempo di esecuzione, anche la quantità di memoria utilizzata da un programma. Ci sono varie ragioni per preoccuparsi della quantità di memoria richiesta da un programma. Innanzitutto,

se un programma richiede molto tempo, possiamo sempre attendere una quantità sufficiente di tempo per ottenere i risultati dell'elaborazione. A titolo di esempio, se vogliamo calcolare F_{58} con l'algoritmo `fibonacci2` avendo a disposizione una CPU di 1700 MHz, possiamo attendere all'incirca 4 ore e 24 minuti, come riportato in Figura 1.7.

Nel caso in cui un programma richiede una quantità di memoria eccessiva, invece, non si ha neppure la garanzia di ottenere i risultati della sua elaborazione, anche attendendo tempi elevati. Ad esempio, il programma potrebbe richiedere più spazio di quello offerto dal disco rigido, oppure il continuo trasferimento di dati da memoria secondaria (disco rigido) a memoria principale (RAM) potrebbe non fare terminare affatto la sua esecuzione. Anche l'occupazione di memoria, quindi, sembra un parametro molto importante nell'analisi degli algoritmi.

Ancora una volta, anche per l'occupazione di memoria analizzeremo gli algoritmi in modo diverso a seconda che siano iterativi oppure ricorsivi. Per programmi iterativi, di solito è sufficiente esaminare la dichiarazione delle variabili e le chiamate di allocazione (come ad esempio la funzione `malloc()` in `C`). Per esempio, l'algoritmo `fibonacci3` dichiara solo un array di n numeri interi, e quindi possiamo concludere che la sua richiesta di memoria è proporzionale ad n. L'analisi di programmi ricorsivi è invece più complicata: lo spazio usato in un certo momento è lo spazio totale usato da tutte le chiamate ricorsive attive in quell'istante. Ad esempio, ogni chiamata ricorsiva dell'algoritmo `fibonacci2` richiede una quantità costante di spazio: una quantità costante per le variabili locali e per gli argomenti della funzione, ed una quantità costante per memorizzare l'indirizzo di ritorno. Le chiamate attive ad un certo istante formano un cammino nell'albero della ricorsione che abbiamo visto prima, dove l'argomento di ogni nodo è di una o due unità più piccolo dell'argomento nel nodo padre.

La lunghezza di ogni cammino nell'albero delle chiamate ricorsive è al più n, e quindi lo spazio richiesto dall'algoritmo ricorsivo è ancora proporzionale ad n, modulo fattori moltiplicativi costanti. Abbreviamo la frase "modulo fattori moltiplicativi costanti", usando la notazione asintotica "O" (O grande): diciamo che lo spazio richiesto dall'algoritmo `fibonacci2` è $O(n)$, ovvero "O grande di n".

L'algoritmo `fibonacci3` può essere modificato così da richiedere meno spazio. Basta osservare che in realtà ogni iterazione del ciclo utilizza soltanto i due valori precedenti di F_n, ovvero F_{n-1} ed F_{n-2}. Quindi possiamo usare soltanto due variabili al posto dell'intero array. Questa modifica richiede alcuni spostamenti perché tutto risieda al suo posto, come illustrato nella Figura 1.8. In questo codice, la variabile a rappresenta $Fib[i-2]$, b rappresenta $Fib[i-1]$ e c rappresenta $Fib[i]$. Le due assegnazioni dopo la somma preparano questi valori per l'iterazione successiva. Questo algoritmo richiede approssimativamente $4n$ linee di codice per calcolare F_n, e quindi è più lento dell'algoritmo `fibonacci3` per via del fattore moltiplicativo, ma richiede meno spazio: spazio costante invece di spazio $O(n)$.

```
algoritmo fibonacci4(intero n) → intero
1.   a ← 1, b ← 1
2.   for i = 3 to n do
3.     c ← a + b
4.     a ← b
5.     b ← c
6.   return b
```

Figura 1.8 Algoritmo `fibonacci4` per il calcolo dell'n-esimo numero di Fibonacci.

1.6 Notazione asintotica

Prima di descrivere algoritmi più efficienti per calcolare i numeri di Fibonacci, rendiamo la nostra analisi un po' più generale. Un problema con l'analisi che abbiamo condotto finora è definire che cos'è esattamente una linea di codice. Se spezziamo una linea di codice in due, non cambiamo certo la velocità del programma, ma cambiamo il numero di linee di codice eseguite. E se compriamo un computer più veloce, cambiamo la velocità del programma, ma non certo la sua analisi.

Per evitare di analizzare specificità e dettagli non necessari, usiamo la notazione asintotica. L'idea di fondo alla base della nozione di definizione asintotica è la seguente: scriviamo già i tempi in funzione dell'input n e trattiamo due funzioni come uguali, in prima istanza, se una è c volte l'altra, dove c è una constante che non dipende da n. Ad esempio, potremmo rimpiazzare $3F_n - 2$ con $O(F_n)$ e $2n$ e $4n$ con $O(n)$. In modo più formale, diciamo che $f(n) = O(g(n))$ se esistono due costanti positive c ed n_0 tali che $f(n) \leq cg(n)$ per ogni $n \geq n_0$, ovvero la funzione $f(n)$ cresce al più come $cg(n)$ a partire da un certo valore $n = n_0$ (si veda la Figura 1.9). Ad esempio, dato che $F_n = \frac{1}{\sqrt{5}}\left(\phi^n - \hat{\phi}^n\right)$, F_n cresce esponenzialmente in n, e quindi il fatto che $\phi < 2$ implica $F_n = O(2^n)$.

Perché usare la notazione asintotica? Innanzitutto, la utilizzeremo per semplificare le formule, ignorando i fattori costanti ed altri dettagli ininfluenti. Questo ci renderà la vita più semplice: non dovremo preoccuparci di tutti i dettagli di basso livello sul comportamento di un algoritmo. Inotre, cosa più rilevante, la notazione asintotica ci consentirà di confrontare due algoritmi in modo semplice. Ad esempio, gli algoritmi `fibonacci3` e `fibonacci4` richiedono entrambi l'esecuzione di $O(n)$ linee di codice. Analizzando esattamente il numero di linee di codice eseguite, un algoritmo è circa due volte più veloce dell'altro, ma questo rapporto non varia al variare di n. Dall'esame di altri fattori, come ad esempio il tempo necessario per l'allocazione dell'array nell'algoritmo `fibonacci3`, può venir fuori che i due algoritmi in realtà possano essere più vicini come prestazioni: per determinare quale dei due sia preferibile, servirà un'analisi più attenta e sperimentale. D'altronde, sappiamo che $4n$ è molto minore di $(3F_n - 2)$ per

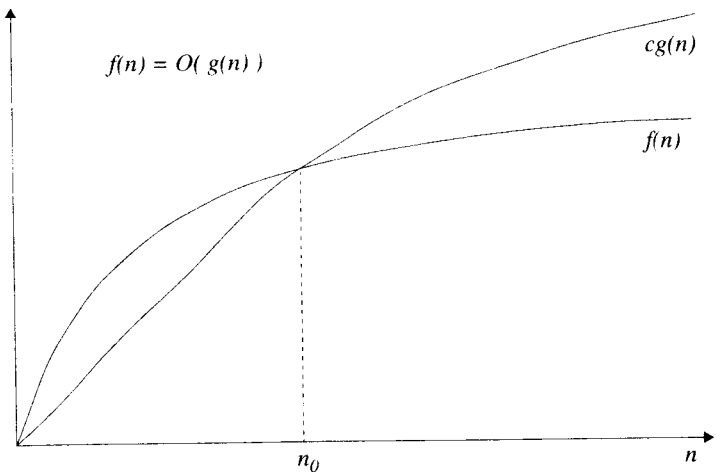

Figura 1.9 Notazione asintotica O.

ogni valore ragionevole di n, e questo non dipende dalla costante moltiplicativa 4: sarebbe vero anche per $8n$ o $15n$. Al crescere di n, il rapporto F_n/n diventa talmente grande da rendere sicuramente l'algoritmo `fibonacci3` preferibile all'algoritmo `fibonacci2`. Sostituire $4n$ con $O(n)$ è un'astrazione che ci consente di confrontarlo facilmente con altre funzioni senza tenere in considerazione dettagli poco rilevanti, quali la costante moltiplicativa 4.

1.7 Un algoritmo basato su potenze ricorsive

Forse sembra difficile crederlo, ma i precedenti algoritmi, `fibonacci3` e `fibonacci4`, non sono i più efficienti possibili. Sfruttando semplici proprietà delle matrici si può dimostrare il seguente lemma.

Lemma 1.3 *Sia* $A = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$. *Allora*

$$A^{n-1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}$$

Dimostrazione. La dimostrazione procede per induzione su n. Per convenzione, fissiamo $F_0 = 0$. Per $n = 2$ il passo base è banalmente verificato:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^1 = \begin{pmatrix} F_2 & F_1 \\ F_1 & F_0 \end{pmatrix}$$

```
algoritmo fibonacci5(intero n) → intero
1.  M ← ( 1 0 )
        ( 0 1 )
2.  for i = 1 to n − 1 do
3.      M ← M · ( 1 1 )
                ( 1 0 )
4.  return M[0][0]
```

Figura 1.10 Algoritmo `fibonacci5` per il calcolo dell'n-esimo numero di Fibonacci.

Assumiamo ora che valga induttivamente l'uguaglianza per $n-1$, ovvero:

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} = \begin{pmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{pmatrix}$$

Moltiplicando entrambi i membri dell'uguaglianza per la matrice A si ottiene

$$A^{n-1} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-1} = \begin{pmatrix} F_{n-1} & F_{n-2} \\ F_{n-2} & F_{n-3} \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} =$$

$$= \begin{pmatrix} F_{n-1} + F_{n-2} & F_{n-1} \\ F_{n-2} + F_{n-3} & F_{n-2} \end{pmatrix} = \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix}$$

□

Il Lemma 1.3 ci consente di definire un altro algoritmo iterativo basato su moltiplicazioni di matrici. Come può desumersi dall'esame della Figura 1.10, inizializziamo M alla matrice identità (potenza zero di A), e poi moltiplichiamo ripetutamente M per A fino ad ottenere la potenza $(n-1)$-esima. Quindi, secondo la formula precedente, l'elemento in alto a sinistra è esattamente F_n, il valore da restituire. Questo è esattamente quello che fa l'algoritmo `fibonacci5` in Figura 1.10.

Osserviamo che la moltiplicazione di matrici non è un'istruzione primitiva di linguaggi di programmazione come `C`, `C++` e `Java`, e quindi il nostro pseudocodice comincia ad allontanarsi da un linguaggio di programmazione reale. Questo ci consente di mantenere un buon livello di astrazione ignorando dettagli irrilevanti e focalizzando la nostra attenzione sugli aspetti essenziali del procedimento risolutivo.

È facile verificare che l'algoritmo `fibonacci5` richiede tempo $O(n)$ ed è quindi molto più veloce dell'algoritmo `fibonacci2`. Ci aspettiamo invece che sia più lento in pratica degli algoritmi `fibonacci3` o `fibonacci4`, a causa delle operazioni per la gestione delle matrici. La notazione asintotica nasconde le differenze tra questi algoritmi, e quindi bisogna essere cauti nel decidere chi di essi è in pratica il più veloce. Ancora una volta, `fibonacci5` ha bisogno di una


```
algoritmo fibonacci6(intero n) → intero
1.  M ← ( 1 0 )
        ( 0 1 )
2.  potenzaDiMatrice(M, n − 1)
3.  return M[0][0]

procedura potenzaDiMatrice(matrice M, intero n)
4.  if ( n > 1 ) then
5.      potenzaDiMatrice(M, n/2)
6.      M ← M · M
7.  if ( n è dispari ) then M ← M · ( 1 1 )
                                    ( 1 0 )
```

Figura 1.11 Algoritmo `fibonacci6` per il calcolo dell'n-esimo numero di Fibonacci.

quantità costante di memoria, ovvero spazio $O(1)$. Qual è dunque il vantaggio nell'usare moltiplicazione di matrici rispetto all'algoritmo `fibonacci4`?

In realtà, possiamo calcolare la matrice M^{n-1} molto più efficientemente. L'idea di base è la seguente: per calcolare, ad esempio, 3^8, si può moltiplicare 3 per otto volte (e.g., $3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3 = 6561$) oppure ricorrere a quadrati ripetuti (ovvero, $3^2 = 9$, $9^2 = 3^4 = 81$, $81^2 = 3^8 = 6561$). Con il metodo dei quadrati ripetuti si effettua un numero molto minore di moltiplicazioni. Con un po' di attenzione, lo stesso metodo può essere esteso a esponenti che non sono potenze di due e può anche essere adattato a matrici, come mostrato nella Figura 1.11. In pratica, tutto il tempo richiesto dall'algoritmo `fibonacci6` di Figura 1.11 è speso nella procedura `potenzaDiMatrice`, che calcola ricorsivamente la potenza n-esima della matrice M elevando a quadrato la sua potenza $(n/2)$-esima. Se n è dispari, l'arrotondamento di $n/2$ produce la potenza $(n-1)$-esima: possiamo aggiustare questo moltiplicando ancora una volta per A. Questo algoritmo è ricorsivo, e per analizzarlo usiamo di nuovo una relazione di ricorrenza. Se $n \leq 1$ l'algoritmo richiede un tempo di esecuzione costante. D'altro canto, il tempo relativo ad una chiamata a `potenzaDiMatrice` con argomento $n > 0$ è $O(1)$, più il tempo di una chiamata ricorsiva con argomento $n/2$:

$$T(n) = \begin{cases} O(1) + T(n/2) & \text{se } n > 1 \\ O(1) & \text{se } n \leq 1 \end{cases}$$

Dimostriamo ora il seguente lemma.

Lemma 1.4 *La relazione di ricorrenza*

$$T(n) = \begin{cases} O(1) + T(n/2) & \textit{se } n > 1 \\ O(1) & \textit{se } n \leq 1 \end{cases}$$

***ammette** come soluzione* $T(n) = O(\log n)$.

Dimostrazione. Assumiamo per semplicità che n sia una potenza di 2, e riscriviamo la relazione di ricorrenza in modo leggermente diverso:

$$T(n) = \begin{cases} a + T(n/2) & \text{se } n > 1 \\ b & \text{se } n \leq 1 \end{cases}$$

per due opportune costanti $a \geq 0, b \geq 0$. Notiamo che

$$T(n/2) \leq a + T(n/4).$$

e quindi sostituendo nella precedente si ottiene:

$$T(n) \leq a + T(n/2) \leq a + a + T(n/4)$$

Continuando con questo procedimento, otteniamo

$$\begin{array}{rcl} T(n) & \leq & a + T(n/2) \\ & \leq & 2 \cdot a + T(n/4) \\ & \leq & 3 \cdot a + T(n/8) \\ & \ldots & \ldots \\ & \leq & k \cdot a + T(n/2^k) \end{array} \qquad \begin{array}{c} \leq \\ \leq \\ \leq \\ \ldots \\ \end{array}$$

per ogni k, $1 \leq k \leq \log_2 n$. Per $k = \log_2 n$ otteniamo:

$$T(n) \leq k \cdot a + T(n/2^k) \leq a \log_2 n + T(1) \leq a \log_2 n + b = O(\log n)$$

□

Vedremo nel Capitolo 2 metodi generali e più formali per risolvere relazioni di ricorrenza. In questo libro, useremo i logaritmi in base 2, arrotondandoli ad interi, e quindi $\log_2 n$ è in pratica il numero di bit necessari per scrivere n in binario (oppure, il più piccolo valore di i tale che $n < 2^i$). Se n vale un miliardo, $\log_2 n$ sarà 30, e l'algoritmo `fibonacci6` sarà più veloce degli algoritmi `fibonacci3` e `fibonacci4` nello stesso modo in cui gli algoritmi `fibonacci3` e `fibonacci4` sono più veloci dell'algoritmo `fibonacci2`.

Prima di concludere questo capitolo, vorremmo fare alcune osservazioni. Per prima cosa, osserviamo che all'interno della notazione asintotica O abbiamo omesso la base dei logaritmi. Questo può essere spiegato se si considera che, date due costanti a e b, con a e b maggiori di zero ($a > 0$, $b > 0$), si effettua un cambiamento di base dei logaritmi secondo la nota formula:

$$\log_b n = \log_b a \cdot \log_a n$$

Tale uguaglianza implica che

$$O(\log_b n) = O(\log_b a \cdot \log_a n) = O(\log_a n)$$

All'interno della notazione asintotica O, quindi, la base di un logaritmo può essere trascurata, in quanto si può cambiare base a meno di moltiplicare per una opportuna costante.

	Tempo di esecuzione	Memoria
`fibonacci2`	$O(2^n)$	$O(n)$
`fibonacci3`	$O(n)$	$O(n)$
`fibonacci4`	$O(n)$	$O(1)$
`fibonacci5`	$O(n)$	$O(1)$
`fibonacci6`	$O(\log n)$	$O(1)$

Tabella 1.1 Tempi di esecuzione e memoria richiesta dagli algoritmi per il calcolo di F_n, n-esimo numero di Fibonacci, in funzione del parametro n.

La seconda osservazione, di natura più metodologica, è relativa al parametro che abbiamo utilizzato per analizzare i tempi di esecuzione degli algoritmi presentati finora. In particolare, in tutto il capitolo abbiamo espresso i tempi di esecuzione degli algoritmi per il calcolo di F_n, n-esimo numero di Fibonacci, in funzione del parametro n, come illustrato nella Tabella 1.1. Ma siamo proprio sicuri di aver considerato il parametro giusto? Dato un generico problema $\mathcal{P}$ (come ad esempio il calcolo di F_n), ed un algoritmo $\mathcal{A}$ per risolvere il problema $\mathcal{P}$, l'algoritmo $\mathcal{A}$ riceverà in ingresso un'istanza I del problema $\mathcal{P}$, che ad esempio nel nostro caso è l'intero n, e produrrà una soluzione al problema $\mathcal{P}$, che nel nostro caso è F_n. In questo scenario, sembra naturale esprimere il tempo di esecuzione dell'algoritmo $\mathcal{A}$ in funzione della dimensione dell'istanza di ingresso, ovvero come una funzione $f(|I|)$. Se quindi il problema $\mathcal{P}$ è il calcolo di F_n, così come considerato in questo capitolo, allora l'istanza di ingresso è n, e la dimensione dell'istanza di ingresso è $|I| = O(\log n)$, dato che sono sufficienti $\lceil \log_2 n \rceil$ bit per descrivere l'intero n. In questo scenario, il tempo di esecuzione degli algoritmi presentati in questo capitolo deve essere correttamente espresso in funzione della dimensione dell'istanza di ingresso $|I|$, come illustrato nella Tabella 1.2. Scopriamo quindi che l'algoritmo `fibonacci2` è in realtà doppiamente esponenziale nella dimensione dell'istanza di ingresso!

	Tempo di esecuzione	Memoria
`fibonacci2`	$O\left(2^{2^{\lvert I\rvert}}\right)$	$O\left(2^{\lvert I\rvert}\right)$
`fibonacci3`	$O\left(2^{\lvert I\rvert}\right)$	$O\left(2^{\lvert I\rvert}\right)$
`fibonacci4`	$O\left(2^{\lvert I\rvert}\right)$	$O(1)$
`fibonacci5`	$O\left(2^{\lvert I\rvert}\right)$	$O(1)$
`fibonacci6`	$O(\lvert I\rvert)$	$O(1)$

Tabella 1.2 Tempo di esecuzione e memoria richiesta dagli algoritmi per il calcolo di F_n, n-esimo numero di Fibonacci in funzione della dimensione dell'istanza di ingresso $|I|$, $|I| = O(\log n)$.

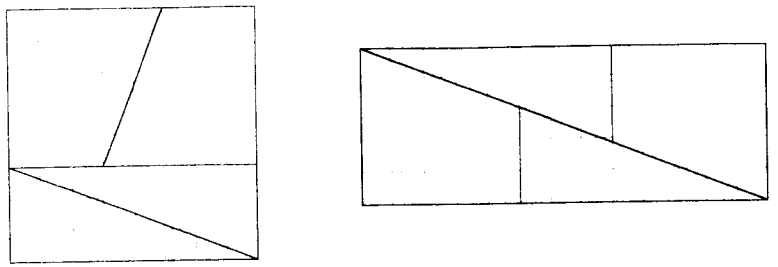

Figura 1.12 La successione di Fibonacci e un'illusione geometrica.

Anche l'ultima osservazione è di natura metodologica, e riguarda tutti gli algoritmi presentati nel corso di questo capitolo. Come sappiamo, il valore F_n cresce esponenzialmente all'aumentare di n e quindi, anche per valori di n non troppo grandi, F_n potrebbe non essere più rappresentabile in una sola parola di memoria del calcolatore, che ha una lunghezza limitata: ad esempio, già F_{48} non è rappresentabile usando parole di memoria con 32 bit. Operare con numeri così grandi implica quindi che, anche per poter eseguire una semplice operazione di addizione o moltiplicazione, la si debba scomporre in una sequenza di lunghezza non costante di passi su numeri più piccoli. L'analisi dovrebbe quindi tenere in considerazione questo aspetto, che abbiamo in prima approssimazione ignorato assumendo che tutte le operazioni aritmetiche elementari abbiano un costo $O(1)$. Vedremo come affrontare questo problema nel Capitolo 2.

1.8 Problemi

Problema 1.1 In una successione di Fibonacci, in cui ogni numero è ottenuto dalla somma dei due precedenti, i primi due numeri non devono essere necessariamente 1. Scegliete due numeri qualunque e generate una successione di Fibonacci a partire dai numeri prescelti: la somma dei primi dieci numeri della successione è sempre pari a undici volte il settimo numero! Riuscite a dimostrarlo per ogni scelta dei numeri di partenza?

Problema 1.2 Prendete un quadrato di 8×8 quadretti. Tagliatelo in quattro pezzi e riassemblatelo in un rettangolo come mostrato in Figura 1.12. Il quadrato originale aveva 64 quadretti, mentre il rettangolo prodotto ne ha 65!

(1) Come è possibile?
(2) È possibile estendere il ragionamento ad ogni quadrato di $F_n \times F_n$ quadretti?

Problema 1.3 Il Professor Tribonacci, dell'Università di Pizza, ha definito la sequenza dei numeri di Tribonacci:

$$X_k = \begin{cases} 0 & \text{se } k = 0 \\ 1 & \text{se } k = 1, 2 \\ X_k = X_{k-1} + X_{k-2} + X_{k-3} & \text{se } k \geq 3 \end{cases}$$

Il Prof. Tribonacci sostiene che l'algoritmo più efficiente per calcolare l'n-mo numero di Tribonacci X_n è dato dal seguente pseudocodice:

algoritmo `tribonacci`($intero\ n$) $\rightarrow$ $intero$
1. **if** ($n = 0$) **then return** 0
2. **else if** ($n \leq 2$) **then return** 1
3. **else return** `tribonacci`$(n-1)$ + `tribonacci`$(n-2)$ + `tribonacci`$(n-3)$

Sia $T(n)$ il tempo di esecuzione dell'algoritmo `tribonacci` su input n.

(a) Scrivere una relazione di ricorrenza per $T(n)$.
(b) Stimare $T(n)$ per n grande.
(c) Descrivere un algoritmo più efficiente per lo stesso problema, e dimostrare il suo tempo di esecuzione.

1.9 Sommario

In questo capitolo abbiamo introdotto, tramite un semplice esempio relativo al calcolo dei numeri di Fibonacci, l'importanza del progetto di algoritmi efficienti per le prestazioni dei sistemi software. Questo esempio, nonostante la sua semplicità, sembra contenere già *in nuce* molte delle problematiche che saranno centrali in questo libro.

Innanzitutto, abbiamo scoperto che progettare algoritmi efficienti può avere un effetto drammatico sull'incremento delle prestazioni. Ma *come misuriamo l'efficienza di un algoritmo?* Come abbiamo osservato, sembra naturale tentare di misurare le risorse richieste da un algoritmo durante la sua esecuzione: la quantità di tempo di calcolo (tempo di CPU) e di spazio (spazio di memoria). Per quanto riguarda il tempo di calcolo, sembra cruciale definire una metrica che sia indipendente dalle tecnologie e dalle piattaforme utilizzate, ad esempio misurando opportunamente il numero di passi richiesto da un algoritmo piuttosto che il suo tempo fisico di esecuzione in un particolare sistema di elaborazione.

Ma in funzione di cosa esprimiamo le risorse richieste da un algoritmo? Per misurare il tempo di esecuzione o l'occupazione di memoria di un algoritmo senza dover guardare i dettagli della particolare istanza del problema, sembrerebbe una buona idea esprimere tali grandezze in funzione della dimensione dell'istanza di ingresso stessa. Per riuscire a confrontare algoritmi diversi, senza arrivare allo stadio finale di implementazione del relativo codice, non sembra necessario disporre del numero esatto dei passi eseguiti dall'algoritmo: una semplice stima dell'ordine di grandezza può già darci un'idea di massima su quale algoritmo sarà più

Figura 1.13 La sorprendente presenza dei numeri di Fibonacci e della spirale aurea in molte specie vegetali e in comuni organismi marini.

veloce. Per questo motivo abbiamo introdotto informalmente la notazione asintotica O, che ci consente di ignorare, in prima approssimazione, fattori e dettagli poco rilevanti.

Infine, come abbiamo visto in questo capitolo introduttivo, sia la fase di progetto che quella di analisi di un semplice algoritmo evidenziano la necessità di ricorrere a strumenti matematici, come ad esempio la risoluzione di relazioni di ricorrenza. Tutti questi concetti, qui introdotti in maniera del tutto informale, saranno definiti più formalmente e con rigore matematico nei capitoli seguenti.

1.10 Note bibliografiche

Gli algoritmi hanno radici storiche profonde, e sono stati utilizzati sin dall'antichità per varie attività, come predire il futuro, prescrivere cure mediche, o preparare cibi. La parola "algoritmo" deriva direttamente da al-Khwārizmī, autore di uno dei più antichi trattati di algebra. Il suo nome completo, Muḥammad ibn Mūsā al-Khwārizmī significa letteralmente Muḥammad figlio di Mūsā da Khwarezm, una regione dell'Asia centrale a sud del Lago di Aral. Muḥammad al-Khwārizmī era un matematico vissuto nel IX secolo, ed il suo libro, "*al-Mukhtaṣar fī al-Jabr wa l-Muqābala*", ci ha regalato anche la parola "algebra" (da al-Jabr).

La semplicità della relazione di Fibonacci può forse spiegare perché i numeri di Fibonacci appaiono in vari fenomeni naturali. Essi sono infatti presenti in varia misura in botanica: non solo numerosi fiori hanno un numero di Fibonacci di petali (come il girasole in Figura 1.13), ma la successione di Fibonacci si ritrova anche nel rapporto fillotattico, da una parola greca che significa "disposizione delle foglie". Ad esempio, scegliete una foglia su uno stelo ed assegnatele il numero 0; contate poi quante foglie intercorrono fino ad incontrare una foglia perfettamente allineata. Se nessuna foglia è stata asportata, con ogni probabilità otterrete un numero di Fibonacci! Gli ordinamenti delle scaglie degli ananas e delle brattee sulle pigne esibiscono peculiarità simili.

I numeri di Fibonacci compaiono anche nel triangolo di Pascal, nella formula binomiale, nella sezione aurea usata in molte opere architettoniche ed artistiche: Seurat, Mondrian, Leonardo da Vinci, Dalì hanno tutti usato il rettangolo aureo in qualcuna delle loro opere! Come visto nei Problemi 1.1 e 1.2, la successione di Fibonacci è infine protagonista indiscussa di curiosi giochi matematici. In particolare, il Problema 1.2 è un paradosso geometrico basato sull'identità di Cassini [1]:

$$F_{n+1}F_{n-1} - (F_n)^2 = (-1)^n, \qquad \text{per } n > 0$$

e pare fosse uno dei puzzle favoriti di Lewis Carroll [2, 3, 4], l'autore di "*Alice nel Paese delle Meraviglie*".

Riferimenti bibliografici

[1] Jean-Dominique Cassini, "Une nouvelle progression de nombres", *Histoire de l'Académie Royale des Sciences* (1733), Paris, vol. 1, p. 201.

[2] Stuart Dodgson Collingwood, *The Lewis Carrol Picture Book*, T. Fisher Unwin, 1899.

[3] O. Schlömilch, "Ein geometrisches paradoxon", *Zeitschrift für Mathematik und Physik* 13 (1868), p. 162.

[4] Warren Weaver, "Lewis Carrol and a geometrical paradox", *American Mathematical Monthly* 45 (1938), p. 234–236.