

ALGORITMI FONDAMENTALI

Giuliano Saccetti

R. Geoff Dromey



GRUPPO
EDITORIALE
JACKSON
Via Rosellini, 12
20124 Milano



Impariamo di più
quando dobbiamo inventare
Piaget

© Copyright per l'edizione originale Prentice – Hall International
Titolo originale "How to solve it by computer"

© Copyright per l'edizione italiana
Gruppo Editoriale Jackson S.p.a. Milano

COLLANA A CURA DI: Emma Bennati

REVISIONE TECNICA: Mariangela Botti

COPERTINA: Emiliano Bernasconi

GRAFICA E IMPAGINAZIONE: Silvano Teruzzi

FOTOCOMPOSIZIONE: Rotolito - Cologno

Pur ribadendo la cura posta nel controllare i testi del presente libro e nel testarne i contenuti, né l'Editore, né l'Autore possono fornire qualsiasi garanzia implicita o esplicita in proposito. Libro e programmi sono venduti quindi "come sono" e chi li impiega si assume ogni possibile rischio derivante dal loro utilizzo.

In nessun caso né l'Editore né l'Autore saranno responsabili di qualsiasi danno sofferto dall'utilizzazione ed imputabile a difetti o imprecisioni riscontrati nei programmi o nel libro, anche se messi a conoscenza che i suddetti danni possono verificarsi.

Tutti i diritti sono riservati. Stampato in Italia. Nessuna parte di questo libro può essere riprodotta, memorizzata in sistemi d'archivio, o trasmessa in qualsiasi forma o mezzo, elettronico, meccanico, fotocopia, registrazione o altri senza la preventiva autorizzazione scritta dall'editore.

INDICE

PREFAZIONE	VII
Suggerimenti per gli insegnanti	V
RINGRAZIAMENTI	XV
CAPITOLO 1 INTRODUZIONE ALLA RISOLUZIONE DI PROBLEMI MEDIANTE COMPUTER	1
Introduzione	1
1.1 La fase di risoluzione	3
1.2 Progettazione Top-Down	7
1.3 Implementazione di algoritmi	14
1.4 Verifica dei programmi	20
1.5 L'efficienza degli algoritmi	31
1.6 L'analisi degli algoritmi	35
CAPITOLO 2 ALGORITMI DI BASE	43
Introduzione	43
Algoritmo 2.1 Scambio dei valori di due variabili	44
Algoritmo 2.2 Conteggio	48
Algoritmo 2.3 Sommatoria di un insieme di numeri	52
Algoritmo 2.4 Calcolo del fattoriale	58
Algoritmo 2.5 Calcolo della funzione seno	61
Algoritmo 2.6 Generazione di una successione di Fibonacci	66
Algoritmo 2.7 Inversione delle cifre di un intero	71
Algoritmo 2.8 Conversione di base	76
Algoritmo 2.9 Conversione da caratteri a numeri	82
CAPITOLO 3 METODI DI FATTORIZZAZIONE	87
Introduzione	87
Algoritmo 3.1 Ricerca della radice quadrata di un numero	88
Algoritmo 3.2 Il minimo divisore di un intero	94
Algoritmo 3.3 Il massimo comun divisore di due interi	100
Algoritmo 3.4 Generazione di numeri primi	107
Algoritmo 3.5 Calcolo dei fattori primi di un intero	118
Algoritmo 3.6 Generazione di numeri pseudo-casuali	123
Algoritmo 3.7 Elevamento di un numero a una potenza elevata	126
Algoritmo 3.8 Calcolo dell' n -esimo numero di Fibonacci	134
CAPITOLO 4 TECNICHE DI GESTIONE DEGLI ARRAY	141
Introduzione	141
Algoritmo 4.1 Inversione dell'ordine di un array	142
Algoritmo 4.2 Costruzione di histogrammi mediante array	146
Algoritmo 4.3 Ricerca del valore massimo di un insieme	150
Algoritmo 4.4 Eliminazione dei doppi in un array ordinato	154
Algoritmo 4.5 Partizione di un array	159
Algoritmo 4.6 Ricerca dell'elemento k -esimo minimo	169
Algoritmo 4.7 Sottosequenza monotona di massima lunghezza	178

CAPITOLO 5	FUSIONE, ORDINAMENTO E RICERCA	185
Introduzione	185	
Algoritmo 5.1	Fusione (merge) di due sequenze ordinate	186
Algoritmo 5.2	Ordinamento per selezione	196
Algoritmo 5.3	Ordinamento per interscambio	203
Algoritmo 5.4	Ordinamento per inserzione	208
Algoritmo 5.5	Ordinamento per diminuzione di incrementi	214
Algoritmo 5.6	Ordinamento per partizione	221
Algoritmo 5.7	Ricerca binaria	231
Algoritmo 5.8	Ricerca calcolata (hash)	241
CAPITOLO 6	TRATTAMENTO DI TESTI E RICERCA DI CONFIGURAZIONI	253
Introduzione	253	
Algoritmo 6.1	Determinazione della lunghezza di una linea di testo	254
Algoritmo 6.2	Allineamento dei margini di un testo	261
Algoritmo 6.3	Ricerca di parole chiave in un testo	272
Algoritmo 6.4	Editazione di una linea di testo	278
Algoritmo 6.5	Ricerca lineare di un elemento	287
Algoritmo 6.6	Ricerca sub-lineare di elementi	298
CAPITOLO 7	ALGORITMI PER STRUTTURE DINAMICHE DI DATI	309
Introduzione	309	
Algoritmo 7.1	Operazioni sulle pile (stack)	311
Algoritmo 7.2	Aggiunte e cancellazioni in una coda	319
Algoritmo 7.3	Ricerca in una lista	331
Algoritmo 7.4	Inserimento e cancellazione da una lista ordinata	337
Algoritmo 7.5	Ricerca in un albero binario	347
Algoritmo 7.6	Inserimento e cancellazione in un albero binario	354
CAPITOLO 8	ALGORITMI RICORSIVI	371
Introduzione	371	
Algoritmo 8.1	Visita di un albero binario	377
Algoritmo 8.2	Quicksort ricorsivo	387
Algoritmo 8.3	Problema delle Torri di Hanoi	395
Algoritmo 8.4	Generazioni di campioni	407
Algoritmo 8.5	Generazione di combinazioni	417
Algoritmo 8.6	Generazione di permutazioni	426

PREFAZIONE

L'idea di questo libro mi è venuta dal classico trattato di Polya, sulla risoluzione dei problemi (*problem-solving*) nella matematica e in generale. Come per la matematica, molti principianti di informatica trovano ostacolo non tanto nell'apprendere un linguaggio di programmazione, quanto nella loro scarsa preparazione a manipolare gli aspetti operativi della disciplina. Il sistema scolastico sembra purtroppo sforzarsi di insegnare alla gente a ricordare fatti e rispondere a domande, piuttosto che *risolvere* realmente i problemi.

In risposta a questa situazione, ho sentito la precisa esigenza di un libro, scritto nello spirito del lavoro di Polya, ma trasferito nell'ambito della scienza dell'elaborazione. Il lavoro di Polya è in buona parte attinente anche al nuovo contesto, ma nei problemi computazionali la ricerca in genere di soluzioni iterative o ricorsive aggiunge un'ulteriore dimensione al processo di soluzione.

Se riusciamo a sviluppare l'abilità di risolvere i problemi, accoppiandola ai criteri di progettazione dal generale al particolare (*top-down*), siamo ben avviati a diventare competenti nella progettazione di algoritmi e nell'implementazione di programmi. In questo libro si è particolarmente curata la presentazione di strategie adatte a "scoprire" algoritmi efficienti e ben strutturati: ovunque, si è attuato uno sforzo cosciente per trasferire un po' del sapore del dialogo (vuoi personale, vuoi tra studente e insegnante) che si instaura durante la soluzione di un problema. Questo stile di presentazione, unito ad una scelta accurata degli esempi, dovrebbe rendere il libro attraente per una vasta gamma di lettori.

Lo studente impegnato in un corso elementare di Pascal troverà nel testo un'utilissima guida per tenere distinta la fase di apprendimento — su come si sviluppa un algoritmo per il computer — da quella di implementazione dello stesso, in un linguaggio di programmazione quale il Pascal. Troppo spesso si confondono i mezzi con il fine. Un buon modo per adoperare il libro nello studio auto-guidato è quello di leggere quanto basta per partire sul problema; poi, una volta sviluppata la propria soluzione, la si confronterà con quella fornita e si rifletterà consapevolmente sulle strategie che hanno favorito o intralciato il raggiungimento dell'obiettivo. Sottolinea giustamente Yohe che "programmare [la soluzione di problemi col computer] è un'arte e, come in tutte le arti, ciascuno deve sviluppare lo stile che gli sembra più naturale e genuino" [*].

Anche gli insegnanti troveranno nello stile di presentazione una guida didattica assai utile, da cui trarre gli spunti di un dialogo equilibrato tra classe e docente. L'autore ha verificato che gli studenti seguono e apprendono, da lezioni presentate in questo modo.

Gli appassionati di home-computer, che volessero sviluppare una propria capacità di risolvere problemi e scrivere programmi senza accedere a corsi regolari, troveranno nella completezza di presentazione dei vari algoritmi un utile complemento a qualsiasi testo introduttivo di Pascal. Si spera che anche gli algoritmi più avanzati siano stati resi accessibili ai principianti più di quanto non accada di solito.

Il materiale presentato, benché in gran parte elementare, contiene esempi ed argomenti carichi di sufficiente interesse e stimoli per accendere, nello studente alle prime armi, quell'entusiasmo per l'elaborazione che spesso si vede negli studenti che iniziano a dominare il loro argomento. I lettori sono esortati a non accettare passivamente le soluzioni degli algoritmi qui proposte: ove possibile e conveniente, il lettore dovrebbe sforzarsi di ricercare un proprio algoritmo più semplice o più efficiente. Come sempre, le limitazioni di spazio hanno comportato che alcuni argomenti ed esempi anche importanti fossero omessi o ridotti di estensione; le discussioni ed i problemi aggiuntivi che accompagnano ciascun algoritmo intendono in qualche modo porre rimedio a ciò. L'insieme dei problemi è stato accuratamente progettato per verificare, consolidare ed accrescere nel lettore la comprensione delle strategie e dei concetti presentati; perciò si raccomanda vivamente al lettore di affrontarli in buon numero.

Ogni capitolo inizia con la discussione di esempi relativamente semplici e graduati, di norma collegati in modo molto diretto; verso la metà e la fine di ogni capitolo, gli esempi sono alquanto più complicati. Un modo adatto di operare con questo libro è di considerare, in una prima lettura, soltanto gli algoritmi fondamentali di ogni capitolo, ricavandone quell'insieme di conoscenze di base necessario per studiare dettagliatamente, in uno stadio successivo, alcuni degli algoritmi più avanzati.

I capitoli sono ordinati all'incirca secondo il grado crescente di difficoltà incontrato dalla media degli studenti. Il primo capitolo introduce gli aspetti centrali della risoluzione di problemi mediante computer e della progettazione di algoritmi; ad una prima lettura, la seconda metà del capitolo può essere semplicemente scorsa. Ove possibile, le idee sono rafforzate mediante esempi. La trattazione della risoluzione dei problemi cerca di affrontare il genere di cose che occorre fare quando ci si *impiana* su un problema; l'importante argomento della verifica dei programmi è presentato in modo tale che sia relativamente facile per il lettore comprenderlo ed applicarlo almeno agli algoritmi semplici. Alcuni esempi servono a trasmettere un po' del sapore dell'analisi probabilistica di un algoritmo.

Il secondo capitolo è dedicato a sviluppare la capacità di formulare soluzioni iterative dei problemi; un'abilità che siamo ben lungi dal possedere prima dell'incontro con i computer. Si presentano anche diversi problemi che permettono di comprendere in pratica come i computer rappresentano e manipolano i dati; si tratta anche il problema della conversione da una rappresentazione all'altra. Gli algoritmi di questo capitolo sono tutti trattati come se la loro scoperta fosse affrontata per la prima volta. Poiché il nostro scopo è quello di promuovere la capacità di risolvere problemi col computer, questo metodo di presentazione sembra appropriato sul piano pedagogico. Gli algo-

ritmi degli altri capitoli sono discussi con lo stesso stile. Nel terzo capitolo, consideriamo un numero di problemi che trovano la loro origine nella teoria dei numeri; questi algoritmi richiedono l'estensione delle tecniche iterative sviluppate nel primo capitolo. Per loro natura, molti di questi problemi possono essere risolti in modi diversi e possono differire molto quanto a onere di calcolo; l'esame dell'efficienza computazionale aggiunge una nuova dimensione al processo di soluzione dei problemi. Noi ci disponiamo a "scoprire" questi algoritmi come se fosse la prima volta. Gli algoritmi che fanno ricorso a tabelle (*array*) sono considerati dettagliatamente nel capitolo 4. Occorre sviluppare una certa abilità nell'uso degli array, non appena si è compresa l'iterazione e si è in grado di utilizzarla per lo sviluppo di algoritmi; gli array, in coppia con l'iterazione, ci forniscono strumenti assai potenti per eseguire calcoli su collezioni di dati che condividono attributi comuni. Le strategie per l'elaborazione efficiente di informazioni organizzate in tabelle conducono a interessanti risultati nella soluzione dei problemi; gli algoritmi discussi affrontano nelle intenzioni almeno i più importanti di tali risultati.

Una volta presi in considerazione array ed iterazioni, si dispone degli strumenti necessari per affrontare gli algoritmi di fusione (*merging*), ordinamento (*sorting*) e ricerca (*searching*); tali argomenti sono discussi nel capitolo 5. L'esigenza di organizzare e successivamente ricercare efficientemente elevati volumi d'informazione è basilare in informatica; vengono presi in considerazione alcuni dei più noti algoritmi di sorting e searching interno. Si tenta di fornire le basi da cui partire alla "scoperta" di questi algoritmi.

Il sesto capitolo, sull'elaborazione di testi e stringhe, rappresenta una digressione dal precedente filone "numerico": la natura e l'organizzazione delle informazioni a caratteri comporta nuove conseguenze nella soluzione dei problemi. Ancora una volta, la richiesta di efficienza porta a considerare alcuni interessanti algoritmi, assai diversi da quelli visti nei capitoli precedenti. Gli algoritmi di questo capitolo mettono in luce come sia necessario sviluppare l'abilità di riconoscere l'essenza di un problema senza farsi confondere o distrarre da dettagli estranei. La "scoperta" di alcuni di questi algoritmi è un interessante banco di prova della capacità acquisita nel risolvere problemi acquisita dopo lo studio dei primi cinque capitoli.

Il capitolo 7 è dedicato agli algoritmi fondamentali di trattamento e ricerca nelle principali strutture dinamiche di dati (ossia: pile, code, liste ed alberi binari). Gli argomenti di questo capitolo si presentano principalmente a livello di implementazione e con l'utilizzo di puntatori.

Nel capitolo conclusivo viene introdotto l'argomento della ricorsione. Una chiamata ricorsiva è presentata come un'estensione della chiamata di un sottoprogramma convenzionale.

La forma più semplice di ricorsione, quella lineare, è trattata solo brevemente poiché siamo del parere che si possano trovare quasi sempre schemi iterativi semplici che equivalgono alla ricorsione lineare. Inoltre, la ricorsione lineare non si presta né a mostrare qualcosa del potere della ricorsione né ad approfondire realmente la nostra comprensione del meccanismo. Lo stesso non si può dire di norma per

la ricorsione binaria, o più in generale non lineare. Gli algoritmi che appartengono a queste due classi sono discussi in dettaglio, ponendo particolare riguardo al riconoscimento delle loro analogie di base e all'approfondimento del meccanismo della ricorsione.

Suggerimenti per gli insegnanti

“...Ogni idea o problema o corpo di conoscenze può essere presentato in una maniera abbastanza semplice, così che qualunque studente possa comprenderlo in una forma riconoscibile.”

Bruner.

In confronto a molte altre attività intellettuali umane, l'uso dei computer è ancora ai primordi, nonostante il progresso avvenuto in breve lasso di tempo. Poiché l'esigenza di elaboratori e di gente con la capacità di usarli è così viva in un mondo che cambia rapidamente, non abbiamo avuto il tempo di sederci e riflettere sul modo migliore per diffondere su vasta scala il “concetto di elaborazione”. Col tempo, questa attività, andrà maturando ed evolverà in una disciplina ben conosciuta, con metodi chiari e definiti con cui proporsi ai principianti: ma questo in un futuro. Per il momento, poiché non siamo in tale felice situazione, è più prudente rivolgersi ad altre discipline mature, per trarne indicazioni su come andrebbe proposta l'elaborazione ai principianti. Se vogliamo avere il più elevato tasso di successo nell'introdurre gli studenti all'informatica (situazione che chiaramente al momento non si verifica) e vogliamo insegnare l'elaborazione su scala la più vasta possibile, la disciplina più ovvia cui far riferimento è *imparare a leggere e scrivere*.

Gli educatori ormai dominano a tal punto il procedimento per insegnare a leggere e scrivere alle persone che queste capacità possono essere trasmesse su larghissima scala, efficientemente, e con elevatissimo tasso di successo. Tralasciando i diversi metodi nell'ambito della disciplina che insegna la lettura e la scrittura, osserviamo che vi sono alcuni principi fondamentali riconosciuti da tutti i metodi, che si adattano perfettamente all'insegnamento dell'informatica.

Insegnando a leggere e a scrivere alla gente, una grossa fetta di tempo (addirittura anni) è dedicata alla lettura. Soltanto dopo tale preparazione si ritiene corretto cimentarsi nello scrivere storie o saggi, o libri, ecc. In realtà, ancor prima di provare ad imparare a leggere, trascorrono alcuni anni di preparazione ad ascoltare la lingua e a parlare. Chiaramente, nell'apprendere un linguaggio, lo scopo definitivo è quello di saper parlare e scrivere correntemente in quella lingua: analogamente in informatica, lo scopo è quello di saper programmare (ovvero progettare ed implementare algoritmi) in maniera efficiente. Nell'imparare a leggere e scrivere si è scoperto da tempo che la lettura è più semplice e che deve precedere la scrittura di un congruo lasso di tempo, per consentire l'assimilazione di un numero di modelli e costrutti lessicali sufficiente a rendere possibile la scrittura.

Nell'insegnare l'uso dei computer, sembra abbiano trascurato il passo che, nel processo di apprendimento a leggere e scrivere, corrisponde alla lettura. Per meglio dire, richiedere alle persone di progettare e scrivere programmi troppo presto nel corso della loro esperienza informatica è come pretendere che siano capaci di scrivere saggi di bravura *prima* di aver appreso a leggere o a scrivere brevi frasi: è aspettarsi troppo dalla maggior parte della gente. Questo probabilmente spiega anche come mai molte persone bravissime in altri campi non riescano a decollare in informatica.

Perciò nell'insegnare l'uso dei computer proponiamo di trarre esempio il più possibile dall'apprendimento della lettura e della scrittura; questo significa essere in grado di fornire allo studente una propria “esperienza di lettura” informatica prima di affrontare gli aspetti più impegnativi della risoluzione dei problemi mediante computer, che richiedono uno sforzo creativo ben maggiore, disciplina e acume tecnico.

A questo punto è importante osservare che l’“esperienza di lettura” non si può ottenere sedendosi con un libro di programmi e cercando di “leggerli”. Il difetto di tale approccio è che le istruzioni di un programma, scritte su un pezzo di carta, sono solo la rappresentazione statica di un algoritmo: come tali non comunicano efficacemente nulla dell'aspetto dinamico degli algoritmi e dei programmi.

Si può argomentare che è importante avere una conoscenza chiara e approfondita del carattere dinamico dei programmi prima di affrontare la progettazione di algoritmi e l'implementazione di programmi. Ciò che serve è un metodo pratico ed economico per fornire agli studenti la loro “esperienza di lettura” in informatica. Per guadagnarsi tale esperienza agli studenti occorre “vedere” l'esecuzione di programmi scritti in un linguaggio di alto livello. Tradizionalmente, tutto quello che uno studente vede è l'uscita prodotta sul terminale o sulla stampante ed i messaggi del programma quando da terminale richiede i dati d'ingresso. Questo livello di osservazione dell'esecuzione di un programma non può soddisfare i principianti poiché trasmette ben poco circa il flusso di controllo del programma o circa gli effetti di uno o più enunciati. Quel che occorre è una dimostrazione assai più esplicita di ciò che accade nel programma, portandolo a produrre i suoi risultati. Una maniera ben più trasparente per farlo è quella di sistemare dinanzi allo studente il testo del programma da eseguire; e quindi di seguirne passo passo l'esecuzione, aggiornando nel contempo dinamicamente sullo schermo i valori delle variabili, via via modificate in accordo con i passi del programma visualizzato. Questo modo dinamico di studiare gli algoritmi può aiutare molto lo studente ad acquisire il livello di comprensione necessario per progettarne e implementarne di propri.

I mezzi e gli strumenti software richiesti per illustrare adeguatamente in questo modo l'esecuzione di un programma sono abbastanza economici a procurarsi. Tutto ciò che occorre è un terminale con possibilità di indirizzamento del cursore ed un *piccolo set di strumenti software* (qualche centinaio di linee di codice) che possano essere inseriti nel programma in esame per renderne l'esecuzione “visibile” sullo schermo. Un tale software è disponibile presso l'autore: per

operare in modo visibile è richiesta solo la chiamata di una procedura per ogni istruzione del programma in esame; il software di visualizzazione è trasparente per l'utente. Strumenti appropriati consentono di vedere l'esecuzione del programma un passo per volta, controllando al tempo stesso l'evoluzione delle variabili, rappresentate dinamicamente sullo schermo. Lo studente può ad ogni passo far avanzare l'esecuzione all'istruzione successiva, premendo semplicemente un tasto sul terminale (p. es. RETURN). A titolo d'esempio in Fig. 1 è riprodotto il tracciato dello schermo per l'esecuzione "visibile" di una procedura di ordinamento in Pascal. Ad ogni passo, la prossima istruzione da eseguire è contrassegnata da una freccia, che può essere spostata lungo il testo della procedura rappresentato sullo schermo. Una descrizione più completa del software è fornita in altra sede [**].

La natura di causa ed effetto di quanto viene visualizzato durante l'esecuzione del programma in questo modo (modo passo-passo automatico) assolve diverse funzioni:

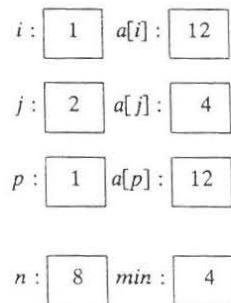
Fig. 1.
Tracciato dello schermo per l'esecuzione visibile di un programma.

```
procedure selectionsrt(a:elements; n:integer);
var i {index for sorted part}, j {index for unsorted part},
    p {position of minimum}, min {minimum in unsorted part}: integer;
```

```
begin
  for i := 1 to n-1 do
    begin
      min := a[i];
      p := i;
      for j := i+1 to n do
        if a[j] < min then
          begin
            min := a[j];
            p := j;
          end;
        a[p] := a[i];
        a[i] := min
      end
    end
  end
```

array: 12 4 56 67 9 23 45
i *j*

condition: *a[j] < min* : true



- (1) Come fatto della massima importanza, fornisce una buona comprensione della natura dinamica degli algoritmi, prerequisito essenziale se gli studenti devono successivamente progettarne e implementarne di propri.
- (2) Fornisce un'approfondita conoscenza delle leggi fondamentali di formazione degli algoritmi per computer (sequenza, selezione, iterazione e modularità).

- (3) È un utile veicolo per aiutare gli studenti ad apprendere diversi costrutti della programmazione. In effetti, osservando i costrutti utilizzati in differenti contesti operativi e collegandoli col cambiamento di valore di variabili e condizioni, lo studente ottiene le dimostrazioni concrete che lo portano a dominare l'utilizzo di tali costrutti.
- (4) Chiarisce in modo molto vivace il funzionamento di un particolare algoritmo. Per esempio, l'algoritmo di ordinamento di Fig. 1, qualora eseguito in maniera visibile, introduce il difficile concetto di come si possa avere un ciclo (*loop*) in esecuzione all'interno di un altro. Inoltre evidenzia la differenza tra variabili con indice e indici stessi.
- (5) Fornisce infine uno strumento ideale per insegnare agli studenti le tecniche di *debugging* (ricerca ed eliminazione degli errori). Ossia, si può implementare un programma con errori logici e invitare lo studente a individuarli muovendo dall'esame del programma eseguito in modo visibile.

L'esecuzione di un programma in maniera visibile che ha l'ulteriore vantaggio di essere perfettamente indicata per l'utilizzo di seminari, classi, o biblioteche. Molti terminali sono dotati di un'uscita video collegabile con un videoregistratore. Con tali mezzi è assai facile ed economico seguire e registrare l'esecuzione visibile di una serie di programmi destinati ad un impiego successivo su monitor, in classe o in biblioteca. Se ne ottiene un ausilio didattico ben superiore agli esempi scritti a mano sulla lavagna quando si parla in classe degli algoritmi. Abbiamo registrato ed impiegato in questo modo già molti programmi.

Si può portare ancora più avanti il modo visibile di esecuzione di un programma, rendendolo più *attivo* nei confronti dello studente. Ad ogni passo dell'esecuzione si può richiedere allo studente di fornire il valore opportuno della variabile o della condizione, ecc. Il funzionamento della cosa è meglio illustrato tornando alla Fig. 1. La freccia sta indicando che la prossima istruzione da eseguire è la numero 7 (cioè *p := j*); il software può muovere il cursore nella casella della *p* sullo schermo, indicando allo studente che deve fornire l'adatto valore di *p* (in questo caso, 2) prima che il programma proseguo. Se l'utente restituisce un valore errato, la parola ERRORE lampeggerà nella casella di *p*; seguirà poi il lampeggi del segno di INPUT, per chiedere all'utente di tentare nuovamente l'immissione di un valore adatto per *p*. Se l'utente sbaglia due volte, il software fa lampeggiare nella casella di *p* la parola VALORE; indi fornisce all'utente la risposta corretta, commuta dal modo interattivo al modo automatico passo-passo e si posiziona sull'istruzione successiva da eseguire.

L'impiego di un programma visibile in modo interattivo passo per passo può rafforzare in maniera molto diretta la comprensione dello studente su come opera realmente un programma e sul compito dei singoli enunciati. Contemporaneamente gli può fornire una reazione molto diretta e positiva quando commette qualche errore. Perciò quello che raccomandiamo è che gli studenti vengano sottoposti a un'abbon-

dante razione di "lettura-programmi", in entrambe la maniere descritte, come passo preparatorio all'esame degli aspetti della soluzione dei problemi nella progettazione di algoritmi. Il contatto con i programmi in esecuzione visibile sarà accompagnato dallo studio delle leggi fondamentali di composizione degli algoritmi (sequenza, selezione, interazione e modularità) e di come tali leggi formali si collegino al modo di operare dei programmi. Contemporaneamente si farà anche lo studio sintattico del linguaggio di programmazione in esame.

RINGRAZIAMENTI

L'ispirazione per l'approccio assunto in questo libro è scaturita dall'ammirazione per il classico lavoro di George Polya sulla soluzione dei problemi. Durante l'estate e l'autunno del 1980, ho avuto il piacere di spendere alcuni pomeriggi in amabili conversazioni con il Prof. Polya e signora.

L'influenza del lavoro di Polya e quella dei pionieri delle discipline informatiche, E.W. Dijkstra, R.W. Floyd, C.A.R. Hoare, D.E. Knuth, e N. Wirth, è chiaramente riconosciuta. C'è anche una forte influenza del lavoro di Jeff Rohl nell'ultimo capitolo sulla ricorsione.

Apprezzo sinceramente la guida e l'incoraggiamento fornитomi dal Prof. Hoare, Henry Hirschberg, Ron Decent, e dai miei revisori. Sono altresì grato all'Università di Wollongong e a quella di Standford per l'utilizzo dei loro mezzi.

Molte persone hanno generosamente fornito commenti e supporto durante la preparazione del manoscritto. Vorrei ringraziare particolarmente Tom Bailey, Miranda Baker, Harold Brown, Bruce Buchanan, Ann Cartwright, John Farmer, Tony Guttman, Joan Hutchinson, Leanne Koring, Donald Knuth, Rosalyn Maloney, Richard Miller, Ross Nealon, Jurg Nievergelt, Richard Patis, Ian Pirie, Juris Reinfelds, Tom Richards, Michael Shepanski, Stanley Smerin e Natesa Sridharan e i miei studenti di Wollongong.

Sono profondamente grato a mia moglie Aziza per la comprensione e l'incoraggiamento che ne ho ricevuto durante l'intero progetto. Infine, voglio esternare la mia più viva gratitudine a Bronwyn James per la devozione, l'instancabilità e l'abile aiuto nella preparazione e nella battitura del manoscritto.

[*] J. M. Yohe, "An overview of programming practices", Comp. Surv., 6, 221-46 (1974)

[**] R.G. Dromey, *Before Programming-On Teaching Introductory Computing*, Technical Report N°. 81/6, Dept. of Computing Science, University of Wollongong (1981).



CAPITOLO 1

INTRODUZIONE ALLA RISOLUZIONE DI PROBLEMI MEDIANTE COMPUTER

INTRODUZIONE

La risoluzione di problemi mediante computer può riassumersi in un aggettivo: *impegnativo*. È infatti un processo intricato che richiede molta ponderazione, pianificazione accurata, coerenza logica, perseveranza e attenzione al dettaglio. Al tempo stesso può essere un'esperienza nuova, eccitante e soddisfacente, con notevole spazio per la creatività e l'inventiva personali. Afrontando con questo spirito la soluzione di problemi col computer, le possibilità di successo sono notevolmente ampliate. Nella discussione di questo capitolo introduttivo, cercheremo di gettare le basi del nostro studio della risoluzione dei problemi mediante computer.

Programmi e algoritmi

La soluzione di un problema col computer comporta un insieme di istruzioni esplicite e non ambigue, espresse in un linguaggio di programmazione: tale insieme di istruzioni è detto *programma*.

Un programma può essere anche visto come un *algoritmo* espresso in un linguaggio di programmazione. Un algoritmo pertanto corrisponde alla soluzione di un problema, *indipendente* da qualsiasi linguaggio di programmazione.

Per ottenere la soluzione di un problema col computer, una volta che si disponga del programma occorre normalmente fornirgli i dati di ingresso (*input*). Il programma assume tali dati, li manipola secondo le proprie istruzioni e produce un *output* che rappresenta la soluzione calcolata del problema. La realizzazione dell'output del calcolatore è soltanto l'ultimo passo di una lunga catena di eventi che hanno preparato la soluzione del problema.

Scopo del nostro lavoro è studiare approfonditamente il processo di progettazione degli algoritmi, sottolineandone in particolare gli aspetti collegati alla risoluzione dei problemi. Per gli algoritmi esistono molte definizioni; quella che segue è idonea in ambito informatico. Un *algoritmo* consiste in un insieme finito di passi distinti e non ambigui

che, eseguito a partire da assegnate condizioni iniziali, produce l'output corrispondente e termina in un tempo finito.

Requisiti per la risoluzione dei problemi col computer

Nella vita di tutti i giorni talvolta ci accade di usare un algoritmo per risolvere un problema. Per esempio, la ricerca di un numero telefonico nell'elenco dei telefoni richiede l'impiego di un algoritmo. Operazioni di questo tipo vengono di norma eseguite automaticamente, senza la consapevolezza del complicato meccanismo implicito, necessario per eseguire efficacemente la ricerca. Pertanto restiamo in qualche modo sorpresi, sviluppando algoritmi per il computer, che la soluzione debba essere specificata con tanta precisione logica e tale dettaglio. Dopo aver studiato un esempio anche limitato di problemi di elaborazione, appare subito evidente come il *livello di comprensione* conscia, richiesto per progettare algoritmi efficienti per il computer, sia assai più profondo di quello che si incontra probabilmente in quasi tutte le altre situazioni problematiche.

Riflettiamo per un istante sul problema della ricerca nella guida telefonica. Molto spesso una rubrica telefonica contiene centinaia di migliaia di nomi e di numeri, tuttavia non ci costa molto trovare il numero telefonico che cerchiamo. Ci domandiamo come mai incontriamo così poca difficoltà con un problema all'apparenza di così grandi dimensioni. La risposta è semplice: noi traiamo vantaggio in modo del tutto naturale dall'ordinamento della rubrica, eliminando in fretta ampie sezioni della lista e andando a puntare sul nome e sul numero desiderati. Non ci verrà mai in mente di cercare il numero di Asdrubale Rossi partendo da pagina 1 ed esaminando uno per uno tutti i nomi, fino ad incontrare il Rossi desiderato e il suo numero. E nemmeno ricercheremo direttamente il nome dell'abbonato il cui numero è 2987533. Per eseguire questa ricerca, non c'è modo di trarre vantaggio dall'ordinamento della guida e pertanto non ci rimarebbe che la prospettiva di sviluppare la ricerca numero per numero a partire dalla pagina 1; se però disponessimo di un elenco di nomi e numeri telefonici ordinati per numero invece che per nome, il compito sarebbe semplicissimo. Questi esempi servono per mettere in luce l'influenza notevole che ha l'organizzazione dei dati sulla prestazione degli algoritmi: solo quando la struttura dei dati è simbioticamente collegata con l'algoritmo possiamo attenderci elevate prestazioni. Prima di affrontare questo ed altri aspetti della progettazione degli algoritmi occorre esaminare con un certo dettaglio l'argomento della risoluzione dei problemi.

1.1 LA FASE DI RISOLUZIONE

È ampiamente riconosciuto che la risoluzione dei problemi è un processo creativo che si sottrae in larga parte alla sistematizzazione e alla meccanizzazione. Ciò non suona troppo d'incoraggiamento a chi vorrebbe diventare un abile solutore; in compenso molti, negli anni di scuola, acquistano almeno una modesta abilità nel risolvere problemi, che ne siano consapevoli o meno.

Anche se uno non è naturalmente portato alla risoluzione dei problemi, vi sono alcuni passaggi che possono essere intrapresi per elevare il livello di abilità individuale. I suggerimenti che seguono non si devono intendere in nessun caso come una ricetta per la soluzione dei problemi. È un fatto che non esiste un metodo universale: strategie diverse si rivelano efficaci su persone diverse.

In questo contesto, allora, da dove potremo partire per dire qualcosa di utile circa la soluzione dei problemi col computer?

Partiremo dalla premessa che risolvere un problema è capirlo.

Fase di definizione del problema

Nella soluzione di qualsiasi problema è possibile riuscire soltanto dopo essersi sforzati di acquisire confidenza o capire il problema in questione. Non possiamo sperare di compiere significativi progressi nella soluzione di un problema finché non abbiamo la piena comprensione di che cosa stiamo cercando di risolvere; questa indagine preliminare può essere vista come la *fase di definizione del problema*. In altri termini, durante questa fase dobbiamo chiarire *che cosa va fatto* piuttosto che *come farlo*. Ovvero, dobbiamo cercare di estrarre dall'esposizione del problema (che è spesso del tutto imprecisa e fors'anche ambigua) un insieme di questioni definite precisamente. I solutori inesperti troppo spesso si slanciano sul come fare a risolvere il problema, per scoprire unicamente che hanno risolto il problema sbagliato, oppure che hanno risolto soltanto un caso particolare di quanto veniva richiesto. In breve, occorre porre molta attenzione nel definire esattamente che cosa deve essere fatto. Lo sviluppo degli algoritmi che trovano la radice quadrata (algoritmo 3.1) ed il massimo comun divisore (algoritmo 3.3) sono buoni esempi di quanto sia importante definire accuratamente il problema: partendo dalle definizioni, siamo condotti naturalmente all'architettura degli algoritmi per quei due problemi.

Come partire

Molte sono le strade per risolvere tantissimi problemi e molte sono anche le soluzioni degli stessi. Questa situazione non rende più facile il compito di risolvere i problemi: messi dinanzi a una molteplicità di linee di attacco possibili, è normalmente difficile riuscire a capire rapidamente quale percorso è probabilmente infruttuoso e quale può essere produttivo.

Forse la situazione più comune, tra le persone che iniziano ad occuparsi di soluzione di problemi col computer, è quella di non sapere da che parte cominciare, anche dopo la fase di definizione iniziale. Che possiamo fare in questi casi? Spesso si verifica un blocco poiché la persona viene coinvolta nei dettagli implementativi *prima* di aver compreso completamente o sviluppato una soluzione indipendente dall'implementazione. Il consiglio migliore in questi casi è di non perdersi troppo in dettagli, cosa che potrà avvenire più tardi, quando la complessità del problema nel suo insieme sarà stata messa sotto controllo. Di norma è dolorosamente vero il proverbio che dice "quanto prima inizi a scrivere il tuo programma tanto più tardi funzionerà".

L'impiego di esempi specifici

Una strategia utile quando ci si impianta è quella di cercare di partire sul problema procedendo con appigli in maniera *euristica* (cioè a spanne). Un approccio che spesso ci consente tale partenza è quello di individuare un esempio specifico del problema generale che vogliamo risolvere e di cercare di mettere in piedi il meccanismo che risolve tale problema particolare (p.es. se si vuole trovare il massimo di un insieme di numeri, si sceglie un insieme particolare di numeri e si appronta il meccanismo che ne trova il massimo; vedi ad esempio l'algoritmo 4.3). Di norma è assai più semplice sviluppare i dettagli della soluzione di un problema specifico, poiché la relazione tra il meccanismo e il problema particolare risulta definita più chiaramente. Inoltre, un problema specifico spesso obbliga a mettere a fuoco dettagli che non appaiono così evidenti quando il problema è considerato in astratto. In molti casi si possono impiegare utilmente diagrammi geometrici o schemi che rappresentino determinati aspetti del problema (vedi ad esempio l'algoritmo 3.3).

Questa maniera di puntare su un problema particolare può spesso fornire la base di partenza per la soluzione generale del problema. Non se ne deve tuttavia abusare; è molto facile cadere nella trappola di ritenere che la soluzione di un problema specifico o di una specifica classe di problemi sia anche la soluzione del problema generale. Spesso ciò accade ma dovremmo andare sempre molto cauti nel fare una simile assunzione.

In teoria, le specifiche del nostro particolare problema vanno esaminate molto accuratamente cercando di stabilire se l'algoritmo proposto può soddisfare o meno le richieste. Se l'insieme completo delle specifiche è difficile da formulare, talvolta una scelta ben articolata di casi di prova ci può fornire un sufficiente grado di sicurezza sulla generalità della soluzione. Tuttavia, soltanto una completa verifica di correttezza dell'algoritmo è soddisfacente fino in fondo; discuteremo più dettagliatamente questo argomento tra breve.

Similitudini tra problemi

Abbiamo appena visto che un modo di affrontare un problema è quello di considerare un esempio specifico. Un'altra cosa che dovremmo sempre cercare di fare è di mettere il più possibile in relazione l'esperienza passata col problema attuale. A tale riguardo è importante esaminare se esistono similitudini tra il problema attuale ed altri problemi che abbiamo risolto o abbiamo visto risolvere. Una volta che ci siamo fatti una piccola esperienza di soluzione di problemi col computer è improbabile che un problema nuovo risulti completamente avulso dagli altri che abbiamo visto. Pertanto un giusto atteggiamento è quello di cercar sempre di riconoscere le similitudini tra i problemi.

Quanto maggiore è l'esperienza, tanto maggiore sono gli strumenti e le tecniche che uno può mettere in campo per affrontare un dato problema. Non sempre però il contributo dell'esperienza è in favore della nostra abilità a risolvere problemi; infatti qualche volta l'esperienza ci blocca dallo scoprire una soluzione desiderabile o migliore. Un esempio classico di progresso bloccato dall'esperienza è la scoperta della relatività di Einstein. Già molto tempo prima che Einstein compisse la sua scoperta gli scienziati dell'epoca possedevano tutti gli elementi che avrebbero potuto condurli alla relatività, ma è quasi certo che la loro esperienza li abbia bloccati persino dal concepire una simile ipotesi: la teoria di Newton era esatta e tutto ciò che aveva senso era pertinente ad essa! A questo punto è forse meglio far solo un cauto affidamento sull'esperienza passata. Talvolta, ricercando una soluzione più adeguata per un problema, l'approfondimento eccessivo della soluzione esistente per un problema analogo ci obbliga lungo la stessa linea di pensiero (che può non essere la migliore) e ci costringe in un vicolo cieco. Ricercando una soluzione più adeguata per un problema, è di norma saggio - almeno in un primo momento - tentare di risolverlo *in modo indipendente*. Ci garantiamo così maggiori possibilità di non cadere nelle stesse trappole dei nostri predecessori.

Concludendo, ogni problema deve essere considerato di per sé.

Un'attitudine che è importante cercare di sviluppare nella soluzione dei problemi è la capacità di vedere uno stesso problema da angolazioni differenti. Uno deve essere capace di rovesciare metaforicamente il problema da sotto in su, da dentro in fuori, di lato, all'indietro, in avanti e così via. Una volta sviluppata questa abilità sarà possibile risolvere qualunque problema.

Procedimento a ritroso (dalla soluzione)

Ci sono ancora altre cose che possiamo tentare quando non sappiamo come partire su un problema. Per esempio, in certi casi possiamo assumere di disporre già della soluzione e quindi provare a procedere dal fondo verso le condizioni di partenza.

Anche una soluzione di primo tentativo può essere sufficiente a fornirci un appiglio per partire (vedi, p.es., il problema della radice quadrata, algoritmo 3.1). Quali che siano i tentativi condotti per risolvere il problema, dovremmo annotare per iscritto i diversi passi esplorativi effettuati; ciò può essere importante per consentirci di rendere sistematiche le indagini ed evitare ripetizione di sforzi. Un'altra pratica che ci aiuta a sviluppare la nostra abilità solutoria consiste, una volta risolto il problema, nel ripensare coscientemente alla strada percorsa per scoprire la soluzione; ciò può esserci di grande aiuto, ma nello sviluppare capacità di risoluzione dei problemi la cosa più importante è la pratica. Piaget lo riassumeva molto elegantemente nell'affermazione che "impariamo di più quando dobbiamo inventare".

Strategie generali di risoluzione

C'è una quantità di strategie di calcolo generali e potenti che nell'informatica vengono ripetutamente impiegate in vari modi.

Spesso è possibile risolvere un problema mediante una di tali strategie e conseguire vantaggi assai cospicui in efficienza computazionale.

Probabilmente tra queste regole la più diffusa e più spesso usata è la strategia del *divide et impera*. L'idea di fondo è di suddividere il problema originale in due o più sottoproblemi che si spera possano essere risolti più efficacemente con la stessa tecnica. Se è possibile procedere in questa suddivisione verso sottoproblemi sempre più piccoli, raggiungeremo finalmente lo stadio in cui i problemi parziali sono sufficientemente piccoli da poterli risolvere senza ulteriori divisioni. Questo modo di spezzare la soluzione di un problema ha trovato vasta applicazione specialmente negli algoritmi di ordinamento, selezione e ricerca. Vedremo più oltre nel capitolo 5, dove è analizzato l'algoritmo di ricerca binaria, che l'applicazione di questa strategia ad un insieme ordinato di dati si traduce in un algoritmo che, per rintracciare un dato elemento in una lista ordinata di N elementi, richiede solo $\log_2 N$ confronti invece di N . Quando lo stesso principio è utilizzato negli algoritmi di ordinamento, il numero di confronti può essere ridotto da circa N^2 a circa $N \log_2 N$, con un vantaggio sostanziale per N grande.

La stessa idea può essere applicata al confronto di *file* e in molte altre occasioni, producendo vantaggi essenziali per l'efficienza computazionale. Nella tecnica del *divide et impera* non è nulla necessario dividere sempre il problema esattamente in due; l'algoritmo utilizzato nel capitolo 4 per ricercare il K -esimo elemento più piccolo riduce ripetuta-

mente le dimensioni del problema: benché ad ogni passo non divida esattamente in due il problema, mediamente possiede prestazioni eccellenti.

Per certi problemi è possibile anche applicare la strategia del *divide et impera* praticamente a rovescio. La strategia del *raddoppio binario* che ne consegue può produrre lo stesso tipo di vantaggi sull'efficienza computazionale. Nel capitolo 3 considereremo come tale tecnica complementare si possa usare per rendere più efficiente l'innalzamento di un numero ad una potenza elevata ed il calcolo dell' N -esimo numero di Fibonacci. In questa strategia del raddoppio dobbiamo esprimere il termine successivo da calcolare N in ragione del termine corrente, che è normalmente funzione di $N/2$, onde evitare la necessità di calcolare termini intermedi.

Un'altra strategia di soluzione generale che considereremo brevemente è quella della *programmazione dinamica*. Questo metodo è usato più spesso quando dobbiamo costruire una soluzione mediante una sequenza di passi intermedi; il problema della sottosequenza monotonica del capitolo 4 impiega una variante del metodo della programmazione dinamica. Questo metodo si collega al presupposto che una buona soluzione per un problema complesso possa talvolta essere costruita dalle soluzioni buone o ottimali di problemi più piccoli. Questo tipo di strategia è indicata particolarmente per molti problemi di ottimizzazione che si incontrano sovente in ricerca operativa: le tecniche di *greedy search*, *backtracking* e di valutazione *branch and bound* sono tutte variazioni dell'idea base della programmazione dinamica. Esse mirano tutte a guidare il calcolo in maniera tale da spendere il minimo sforzo nell'esplorazione di soluzioni individuate come sub-ottimali. Ci sono altre strategie generali di calcolo che potremmo considerare, ma poiché sono di norma associate ad algoritmi più avanzati non procederemo oltre in questa direzione. (')

1.2 PROGETTAZIONE TOP-DOWN

Lo scopo principale della soluzione di problemi con il computer è un algoritmo che sia possibile implementare come programma di elaborazione corretto ed efficiente. Nella discussione riguardante la progettazione degli algoritmi ci siamo preoccupati soprattutto degli aspetti del tutto generali della soluzione dei problemi. Ora è necessario considerare gli aspetti della soluzione dei problemi e della progettazione di algoritmi che sono più legati all'implementazione degli algoritmi stessi.

(') H.F. Ledgard, Programming Proverbs, Hayden, Rochelle Park, N.J., 1975.

Una volta definito il problema da risolvere e concepita almeno una vaga idea di come risolverlo, possiamo cominciare a mettere in atto tecniche potenti per progettare algoritmi. La chiave dell'abilità di progettare con successo algoritmi risiede nella capacità di gestire l'intrinseca complessità di molti problemi che richiedono una soluzione col computer. Chi risolve problemi è in grado di puntualizzare, e comprendere in una sola volta, una quantità assai limitata di logica o di istruzioni. La tecnica di progettazione degli algoritmi che tenta di compensare questa limitazione umana è nota come *progettazione top-down o affinamento passo-passo*.

La progettazione top-down è una strategia che possiamo applicare per tradurre la soluzione di un problema da una vaga traccia a un algoritmo definito precisamente e implementato in un programma.

La progettazione top-down ci fornisce un modo per maneggiare l'intrinseca complessità logica e il dettaglio sovente presente negli algoritmi di calcolo. Ci consente di costruire la nostra soluzione del problema per passi; in tal modo, i dettagli specifici e complessi dell'implementazione si incontrano solo dopo aver posto basi sufficienti per la struttura nel suo insieme e stabilito relazioni tra le varie parti del problema.

Frazionamento in sottoproblemi

Prima di poter applicare lo sviluppo top-down a un problema dobbiamo gettare le basi che ci forniscano almeno una traccia grossolana di risoluzione. Talvolta questo potrà richiedere un'indagine sul problema lunga e creativa, mentre altre volte la descrizione stessa del problema può fornire il necessario punto di partenza per la progettazione top-down. La traccia generale può consistere in una singola istruzione o in un insieme di istruzioni. La progettazione top-down suggerisce di prendere una per volta le istruzioni generali di cui disponiamo e di suddividerle in un insieme di sottofunzioni definite con maggiore precisione. Tali sottofunzioni dovrebbero descrivere con maggiore accuratezza come vada raggiunto lo scopo finale. Ad ogni suddivisione di una funzione in sottofunzioni è essenziale che sia definita con precisione la maniera con cui le sottofunzioni richiedono di interagire tra loro; solo in questo modo è possibile preservare la struttura globale della soluzione del problema. Conservare la struttura globale nella soluzione di un problema è importante sia per rendere comprensibile l'algoritmo, sia per consentire la prova di correttezza della soluzione. Il processo di suddivisione ripetuta di una funzione in sottofunzioni, e poi di queste in altre ancora più piccole, deve continuare finché non raggiungiamo sottofunzioni che possono essere implementate da istruzioni di programma. Per molti algoritmi dobbiamo scendere solo di due o tre livelli, benché ovviamente ciò non sia vero in grandi progetti software. Quanto più vasto e complesso è il problema, tanto più richiederà di essere spezzettato per divenire trattabile. La Fig. 1.1. mostra schematicamente la suddivisione di un problema.

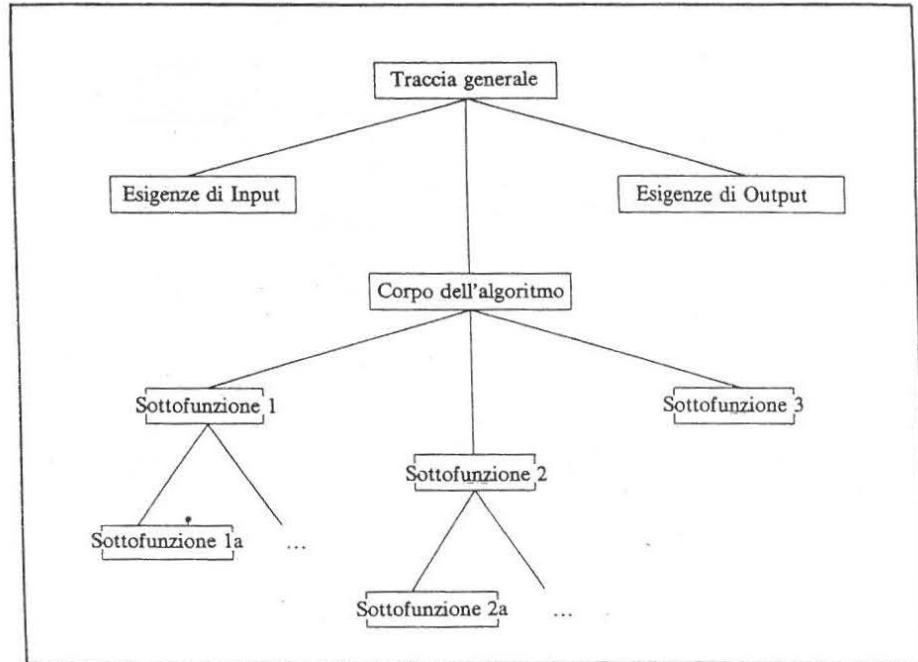


Fig. 1.1.
Suddivisione
schematica di un
problema in
sottofunzioni, da
impiegare nella
progettazione
top-down.

Il processo di frazionamento della soluzione di un problema in sottofunzioni nella maniera descritta si traduce in un insieme implementabile di funzioni che si sposa naturalmente con linguaggi strutturati a blocchi quali Pascal e Algol. Ci potrà essere quindi un'interfaccia piana e naturale tra l'algoritmo raffinato per passi e l'implementazione finale del programma: una situazione altamente desiderabile per mantenere il lavoro dell'implementazione il più semplice possibile.

Scelta di una struttura dati adeguata

Una delle decisioni più importanti da prendere nella formulazione della soluzione di problemi col computer è la scelta di una struttura di dati appropriata. Tutti i programmi operano su dati e pertanto il modo di organizzazione i dati può avere effetti profondi su ogni aspetto della soluzione finale. In particolare, una scelta inadeguata delle strutture dati spesso conduce a implementazioni grossolane, inefficienti e difficili. D'altra parte, una scelta appropriata comporta generalmente un'implementazione semplice, trasparente ed efficiente.

Non c'è una regola rigida e rapida che dica in quale momento dello sviluppo di un algoritmo occorra assumere decisioni circa le strutture di dati associate. In alcuni problemi può essere necessario considerare la struttura dei dati fin dall'inizio delle esplorazioni della soluzione e

prima dello sviluppo top-down, mentre in altri problemi la si può posporre finché non ci si è inoltrati nei dettagli dell'implementazione. In altri casi ancora, essa può essere raffinata durante lo sviluppo dell'algoritmo.

La chiave per risolvere efficientemente molti problemi si riconduce invero a scelte appropriate sulle strutture di dati associate. Strutture dati ed algoritmi sono di norma intimamente legati; un piccolo cambiamento nell'organizzazione dei dati può avere un'influenza significativa sull'algoritmo richiesto per risolvere il problema. Non è facile formulare una qualsiasi regola generale che indichi per ogni classe di problemi la scelta più idonea della struttura dei dati. Sfortunatamente da questo punto di vista ogni problema deve essere considerato a sé stante.

Tuttavia il genere di questioni che dobbiamo avere presenti nell'imbastire le strutture dei dati sono del tipo:

1. Come si possono organizzare i risultati intermedi per consentire un accesso veloce all'informazione che riduca la quantità di calcoli richiesta successivamente?
2. La struttura dati può essere facilmente esplorata?
3. La struttura dati può essere facilmente aggiornata?
4. La struttura dati offre un modo per recuperare uno stadio precedente del calcolo?
5. La struttura dati comporta un uso eccessivo della memoria?
6. È possibile impostare una qualche struttura dati ad un problema che non è ben chiaro all'inizio?
7. Il problema può essere formulato secondo una delle strutture dati comuni (p.es. array, insiemi, code, pile, alberi, grafi, liste)?

Queste considerazioni possono sembrare generiche ma rendono l'idea del tipo di questioni che occorre porsi affrontando lo sviluppo di un algoritmo.

Costruzione dei loop

Passando dalle affermazioni generali sull'implementazione alle sottofunzioni che possono essere realizzate sotto forma di calcoli, quasi invariabilmente ci si riconduce ad una serie di costrutti iterativi, o *loop* (cicli), e di strutture che sono eseguite sotto condizione. Queste strutture, insieme con le istruzioni di input/output, le espressioni calcolabili e le assegnazioni, costituiscono il cuore dell'implementazione di un programma.

Allorché una sottofunzione è stata ridotta a qualcosa di realizzabile sotto forma di costrutto iterativo, si può rendere più semplice il compito di implementare il ciclo se si è al corrente della struttura essenziale di tutti i loop. Per costruire qualsiasi loop occorre tener conto di tre cose: le condizioni iniziali che devono valere *prima* che il ciclo cominci; la *relazione invariante* che deve valere dopo ogni iterazione; e le con-

dizioni per il *termine* del processo iterativo. Costruendo i loop, molti trovano difficoltà nell'assegnare correttamente le condizioni iniziali e nell'imporre al ciclo di compiersi il numero esatto di volte, e non una di più o una di meno. Per la maggior parte dei problemi esiste un procedimento elementare che può essere applicato per evitare tali errori.

Stabilire le condizioni iniziali

Per stabilire le condizioni iniziali di un loop, una strategia di solito efficace prevede di assegnare alle variabili del ciclo i valori che esse dovrebbero assumere per risolvere il problema *minimo* associato ai loop. Tipicamente il numero i di iterazioni da compiere in un loop cade nell'intervallo $0 \leq i \leq n$. Il problema minimo di norma corrisponde al caso in cui i è uguale a 0 o è uguale a 1. Gli algoritmi 2.2. e 4.3 illustrano entrambe queste situazioni. Per fissare le idee supponiamo che si voglia sommare l'insieme dei numeri di un array (v. cap. 4), utilizzando un costrutto iterativo. Le variabili del loop sono: i , indice del loop e dell'array, ed s , variabile ove si accumulano i valori degli elementi dell'array. Il problema minimo in questo caso corrisponde alla somma di 0 numeri. La somma di zero numeri è zero e pertanto i valori iniziali per i ed s devono essere:

$$\left. \begin{array}{l} i := 0 \\ s := 0 \end{array} \right\} \text{soluzione per } n = 0$$

Individuazione del costrutto iterativo

Una volta che abbiamo le condizioni per risolvere il problema minimo, il passo seguente è tentare di estenderlo al problema più piccolo immediatamente successivo: in questo caso $i = 1$.

Ossia vogliamo costruire sulla soluzione del problema per $i = 0$ la soluzione per $i = 1$.

La soluzione per $n = 1$ è:

$$\left. \begin{array}{l} i := 1 \\ s := a[1] \end{array} \right\} \text{soluzione per } n = 1$$

Questa soluzione può essere ottenuta da quella per $n = 0$ utilizzando i valori di i ed s validi per $n = 0$ e le due espressioni:

$$\left. \begin{array}{l} i := i + 1 \\ s := s + a[i] \end{array} \right\} \text{soluzione generalizzata per } n > 0$$

sione che condiziona il loop divenga obbligatoriamente falsa. Non usando questo metodo di terminazione, in questa situazione la nostra sola alternativa sarebbe stata un loop implementato con un test doppio, per esempio:

```
i := 2;  
while (a[i-1]<a[i]) and (i<n) do i := i+1
```

Abbiamo così concluso un esame dei modi più comuni per terminare i loop.

1.3 IMPLEMENTAZIONE DI ALGORITMI

L'implementazione di un algoritmo che è stato correttamente progettato secondo la metodologia top-down dovrebbe essere un processo quasi meccanico. Vi sono, tuttavia, alcuni punti che andrebbero ricordati.

Se un algoritmo è stato correttamente progettato il percorso dell'esecuzione dovrebbe seguire una linea retta dall'alto verso il basso; è importante che l'implementazione del programma corrisponda a questa regola alto-basso. I programmi (ed i sottoprogrammi) implementati in questo modo sono normalmente molto più facili da comprendere e mettere a punto. Essi sono anche normalmente molto più facili da modificare all'occorrenza, poiché le relazioni tra le varie parti del programma sono più evidenti.

Impiego delle procedure per enfatizzare la modularità

Per aiutare sia lo sviluppo dell'implementazione sia la leggibilità del programma principale, è di solito vantaggioso dividere il programma in moduli secondo le linee che derivano naturalmente dalla progettazione top-down. Questa pratica ci consente di implementare un insieme di procedure indipendenti che eseguono compiti specifici e ben definiti. Per esempio, se come parte di un algoritmo è richiesto il riordino di un array, si userà una procedura indipendente specifica per l'ordinamento.

Una cosa da tenere presente nell'applicare la modularizzazione è di non spingere troppo oltre il processo, al punto che l'implementazione

torni ad essere di difficile lettura a causa della frammentarietà. Quando è necessario implementare progetti software piuttosto ampi, una strategia indicata è quella di completare innanzi tutto la progettazione generale in un quadro top-down; il meccanismo del programma principale può essere poi implementato con chiamate alle diverse procedure che saranno richieste nell'implementazione finale. Nella prima fase implementativa, prima di aver realizzato alcuna procedura, possiamo inserire nello scheletro delle procedure un comando di scrittura che evidenzia semplicemente il nome della procedura al momento della chiamata; per esempio:

```
procedure sort;  
begin  
  writeln("chiamata procedura sort")  
end
```

Questo modo di procedere ci consente di provare il meccanismo del programma principale in uno stadio iniziale e di implementare e collaudare le procedure una per una. Quando una procedura è stata implementata la sostituiamo semplicemente alla sua procedura "teniposto".

Scelta dei nomi delle variabili

Un altro dettaglio implementativo, che può rendere più significativi e più facili da comprendere i programmi, è la scelta di nomi appropriati per variabili costanti. Se per esempio dobbiamo eseguire manipolazioni sui giorni della settimana è meglio scegliere come nome della variabile *giorno*, piuttosto che una semplice *a* o qualche altro nome. Questo accorgimento tende a rendere i programmi molto più auto-dокументati.

Inoltre, ogni variabile dovrebbe avere *un solo* ruolo in un dato programma. Può essere anche assai utile una chiara definizione di tutte le variabili costanti all'inizio di ogni procedura.

Documentazione dei programmi

Un'altra utile pratica di documentazione, che può essere in particolare utilizzata in Pascal, è quella di associare un commento breve ma preciso ad ogni istruzione *begin*. Ciò è appropriato poiché le istruzioni *begin* di norma segnalano che sta per cominciare qualche parte modu-

lare del calcolo. Parte integrante della documentazione di un programma sono le informazioni che il programma presenta all'utente in fase di esecuzione. Una buona pratica di programmazione consiste nello scrivere sempre programmi che possano essere eseguiti ed impiegati da altri utenti, all'oscuro del funzionamento e delle esigenze di input del programma. Questo significa che il programma deve specificare con esattezza nel corso dell'esecuzione le risposte (e il relativo formato) che si aspetta dall'utente. Una cura particolare andrebbe posta nell'evitare ambiguità in tali specificazioni: dovrebbero essere concise ma precise accuratamente quanto si richiede. Il programma dovrebbe anche "intercettare" le risposte inesatte alle proprie domande e informarne l'utente in maniera appropriata.

Come mettere a punto i programmi

Implementando un algoritmo è sempre necessario eseguire una serie di controlli, per garantire che il programma si stia comportando correttamente in conformità con le sue specifiche. Anche in programmi brevi è plausibile che ci siano errori di logica, non rilevabili in fase di compilazione.

Per rendere in qualche modo più semplice l'individuazione degli errori di logica è una buona idea inserire nel programma un insieme di istruzioni che stampino informazioni in punti strategici del calcolo. Tali istruzioni, che stampano informazioni aggiuntive rispetto all'output desiderato, possono essere rese eseguibili sotto condizione: così facendo diviene non più necessaria la rimozione di dette istruzioni quando siamo soddisfatti del programma. Il modo più semplice per implementare questo strumento di *debugging* (messa a punto) consiste nel disporre di una variabile booleana (per esempio *debug*) cui attribuire il valore *vero* quando è richiesta la stampa estesa di controllo. Ciascun output di controllo può essere pertanto racchiuso tra parentesi come segue:

```
if debug then
begin
writeln(...)
:
end
```

In effetti il compito di trovare errori logici nei programmi può essere un'esperienza frustrante che mette a dura prova. Non ci sono metodi infallibili per il debugging, ma esistono alcuni passi che si possono intraprendere per alleviarne l'onore. Probabilmente la soluzione migliore è quella di provare sempre il programma a mano *prima* di mandarlo in esecuzione; se condotto sistematicamente ed accuratamente, ciò può evidenziare la maggior parte degli errori. Un modo per

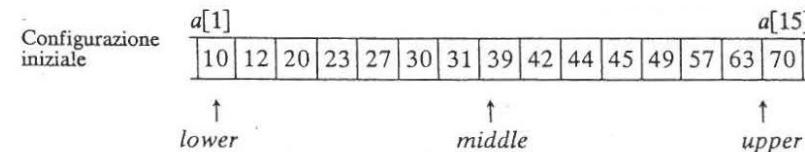
far ciò è analizzare uno per volta ciascun modulo o programma isolato, in corrispondenza di condizioni tipiche di input. Il modo più semplice per farlo è allestire una tabella bidimensionale contenente i passi eseguiti e tutte le variabili interessate nella sezione di programma in esame. Eseguiremo quindi una per una le istruzioni del programma, aggiornando la tabella delle variabili non appena una di esse viene modificata. Se il processo che stiamo riproducendo è un loop, di norma basterà controllare le prime due iterazioni e le ultime due prima della conclusione.

Come esempio, si consideri la tabella da allestire per la procedura di ricerca binaria (algoritmo 5.7).

I passi essenziali della procedura sono:

```
lower := 1; upper := n;
while lower < upper do
begin
  middle := (lower+upper) div 2;
  if x > a[middle] then
    lower := middle + 1
  else
    upper := middle
  end;
found := (x = a[lower])
```

Per un valore da cercare x e un array $a[1..n]$ con $x = 44$ e $n = 15$ si può avere



La tabella associata all'esecuzione è allora riprodotta in Tab.1.1.

Tab. 1.1. Tabella di esecuzione passo passo per la ricerca binaria.

Iterazione no.	lower	middle	upper	lower < upper	$a[middle]$	$x > a[middle]$
Inizialmente	1	—	15	true	—	—
1	9	8	15	true	39	true
2	9	12	12	true	49	false
3	9	10	10	true	44	false
4	10	9	10	false	42	true

NOTA: I valori delle variabili associate ad ogni iterazione valgono *dopo* che l'iterazione è stata conclusa

Se arriviamo a un punto in cui il nostro programma funziona ma produce risultati errati (per esempio una routine di ordinamento che mette in fila la maggior parte degli elementi, ma non tutti), l'idea migliore è usare dapprima una traccia di debugging per stampare informazioni strategiche. Il passo successivo è seguire a mano il programma un passo per volta, confrontando la stampa di controllo prodotta dal computer. Quando si intraprende una procedura di questo tipo, si deve far attenzione che il cammino di esecuzione venga percorso completamente; di solito è fatica sprecata assumere che qualcosa funzioni e iniziare lo studio sistematico dell'algoritmo partendo a metà lungo il percorso di esecuzione. Una buona regola da seguire nel debugging è di non dar nulla per scontato.

Collaudo dei programmi

Cercando di analizzare se un programma è in grado o meno di manipolare tutte le variazioni corrispondenti del problema, dobbiamo fare ogni sforzo per essere sicuri che esso possa affrontare i casi limite ed insoliti. Tra l'altro dovremo verificare se il programma risolve il problema più piccolo possibile, se tratta il caso di dati con valori tutti uguali, e così via. Casi insoliti di questo tipo sono di norma quelli che fanno vacillare un programma.

Come esempio, si considerino i controlli richiesti per l'algoritmo di ricerca binaria (algoritmo 5.7); in Tab. 1.2 sono riportati i dati ed i controlli opportuni.

Tab. 1.2. Dati per collaudare l'algoritmo di ricerca binaria.

prova	valori x da cercare	dati campione
1. L'algoritmo tratta la ricerca in un array di un solo elemento?	0, 1, 2	$a[1] = 1$ $n = 1$
2. L'algoritmo tratta il caso in cui tutti gli elementi dell'array sono uguali?	0, 1, 2	$a[1] = 1$ $a[2] = 1 \dots a[n] = 1$
3. L'algoritmo tratta il caso in cui il valore cercato uguaglia il <i>primo</i> elemento dell'array?	1	$a[1] = 1$ $a[2] = 2 \dots a[n] = n$
4. L'algoritmo tratta il caso in cui il valore cercato uguaglia l' <i>ultimo</i> elemento dell'array?	n	$a[1] = 1$ $a[2] = 2 \dots a[n] = n$

5. L'algoritmo tratta il caso in cui il valore cercato è <i>inferiore</i> al <i>primo</i> elemento dell'array?	0	$a[1] = 1$ $a[2] = 2 \dots a[n] = n$
6. L'algoritmo tratta il caso in cui il valore cercato è <i>maggiore</i> dell' <i>ultimo</i> elemento dell'array?	$n + 1$	$a[1] = 1$ $a[2] = 2 \dots a[n] = n$
7. L'algoritmo tratta il caso in cui il valore cercato è in posizione <i>pari</i> nell'array?	2	$a[1] = 1$ $a[2] = 2 \dots a[n] = n$
8. L'algoritmo tratta il caso in cui il valore cercato è in posizione <i>dispari</i> nell'array?	3	$a[1] = 1$ $a[2] = 2 \dots a[n] = n$
9. L'algoritmo tratta il caso in cui il valore cercato è assente, ma compreso nell'intervallo di valori dell'array?	5	$a[1] = 2$ $a[2] = 4 \dots a[n] = 2n$

Spesso non è possibile o necessario scrivere programmi che gestiscono tutte le combinazioni di input che possono essere associate a un dato problema. Appena possibile, i programmi dovrebbero essere accompagnati da asserzioni di input ed output come descritto nel paragrafo sulla verifica dei programmi.

Benché non sia sempre pratico implementare programmi che possono trattare tutte le possibili condizioni di input, ci si dovrebbe sempre sforzare di introdurre nel programma meccanismi che gli consentano di rispondere all'utente con garbo e chiarezza quando riceve input di cui non è previsto il trattamento.

Quest'ultima affermazione non va tuttavia intesa nel senso che si debbano progettare solo algoritmi che risolvono problemi specifici. Un tale approccio è ben lungi dai nostri intendimenti; quasi senza eccezioni, progetteremo algoritmi che siano del tutto generali, abbracciando così un'intera classe di problemi piuttosto che soltanto un caso specifico. L'altro approccio, di scrivere programmi per casi specifici, comporta, di norma, dispensio di tempo ed energie. È assai meglio eseguire un lavoro completo sin dall'inizio e produrre un programma che risolva un ampio spettro di problemi. Questo ci porta a ricordare una cattiva pratica spesso adottata dai programmati principianti (e anche da altri): quella di usare costanti laddove andrebbero adoperate variabili. Per esempio, non si dovrebbero usare istruzioni del tipo:

```
while  $i < 100$  do
```

Il 100 andrebbe sempre sostituito con una variabile, ossia:

```
while  $i < n$  do
```

Una buona regola da seguire è che le costanti numeriche andrebbero usate nei programmi solo per cose come il numero dei mesi dell'anno, e

simili. Il Pascal fornisce il modo di dichiarare le costanti.

Le considerazioni fatte circa l'implementazione degli algoritmi ci conducono ad altri argomenti importantissimi che riguardano appunto ciò che intendiamo per una buona soluzione di un problema e, inoltre, la questione di che cosa effettivamente costituisca un programma corretto.

1.4 VERIFICA DEI PROGRAMMI

Il costo di sviluppo del software è diventato la spesa preponderante nell'applicazione dei computer. L'esperienza operativa sui sistemi computerizzati ha indotto a osservare che in genere oltre la metà di tutti gli sforzi di programmazione e delle risorse è spesa nel correggere errori e introdurre modifiche nei programmi. Via via che vengono sviluppati programmi più vasti e complessi, entrambi questi compiti divengono più difficili e dispendiosi in termini di tempo. In alcune applicazioni specializzate, militari, spaziali, mediche, la correttezza dei programmi può essere questione di vita o di morte. Questo suggerisce due cose. In primo luogo, risparmi di tempo notevoli nella modifica dei programmi sarebbero possibili ponendo maggior cura nella creazione di un codice scritto con chiarezza all'atto dello sviluppo del programma. Abbiamo già visto che la progettazione top-down può essere un ausilio molto potente nella scrittura di programmi leggibili e facili da comprendere sia a livello superficiale, sia a livello dettagliato di implementazione.

Il secondo problema, di essere capaci di sviluppare un codice tanto corretto quanto chiaro, richiede anch'esso l'applicazione di una procedura sistematica. Provare la correttezza di un programma pur semplice finisce per essere un compito non facile. Non basta esaminare il comportamento di un programma sotto l'influenza di svariate situazioni di input per provarne la correttezza. Questo approccio per dimostrare la correttezza di un programma può, in qualche caso, evidenziare errori ma non può garantire l'assenza. È la debolezza di questo approccio che rende necessario il ricorso a metodi di verifica dei programmi basati su solidi principi matematici.

La *verifica dei programmi* consiste nell'applicazione di tecniche matematiche di prova per stabilire che i risultati ottenuti dall'esecuzione di un programma con input qualsivoglia concordano con gli output formalmente specificati.

Benché solo adesso iniziamo a considerare questo aspetto della progettazione di algoritmi, non è detto che questo processo debba essere compiuto dopo lo sviluppo completo dell'algoritmo. In effetti, una strategia piuttosto valida consiste nello svolgere la verifica in maniera top-down parallelamente allo sviluppo top-down dell'algoritmo. Ovvero, si inizia col provare la correttezza della struttura di larga massima dell'algoritmo, a un livello astratto o superficiale; quindi,

appena le funzioni astratte sono sostituite da meccanismi più specifici, è necessario garantire che questi raffinamenti non alterino la correttezza del livello più astratto. Il procedimento è ripetuto finché non si giunge agli specifici passi del programma.

Modello dell'esecuzione di un programma

Per conseguire lo scopo della verifica del programma, dobbiamo comprendere appieno che cosa accade quando un programma è eseguito sotto l'influenza di assegnate condizioni di input. Ciò che importa da questo punto di vista è il cammino che l'esecuzione percorre per una data situazione di input. Un programma può avere svariati percorsi di esecuzione che conducono ad una conclusione soddisfacente; per un assegnato insieme di condizioni d'ingresso verrà seguito *uno soltanto* di tali cammini (benché ovviamente alcuni cammini possano parzialmente coincidere per input differenti). Pertanto l'implementazione scritta di un algoritmo definisce un intero set di percorsi di esecuzione. Il procedere del calcolo da specifiche condizioni d'ingresso fino alla terminazione può essere visto come una successione di transizioni da uno *stato di elaborazione* ad un altro. Ogni stato, incluso quello iniziale, è definito dai valori di *tutte* le variabili nel corrispondente istante temporale. Una *transizione di stato* (ed un progresso verso la conclusione) si ottiene modificando il valore di una variabile e trasferendo quindi il controllo all'istruzione successiva lungo il percorso di esecuzione corrente. Oltre alle istruzioni che cambiano lo stato di elaborazione, ve ne sono altre che compiono soltanto confronti sullo stato attuale; tali confronti sono impiegati per effettuare modifiche nel flusso sequenziale dell'esecuzione. Questo modello per l'esecuzione di un programma ci fornisce le basi su cui costruire le prove di correttezza degli algoritmi.

Asserzioni di input ed output

Il primissimo passo da attuare per verificare la correttezza di un programma è produrre una dichiarazione formale delle specifiche riguardanti le variabili adoperate. La dichiarazione formale ha due parti: un'asserzione di input ed un'asserzione di output, che possono essere espresse in notazione logica come predicati che descrivono lo stato delle variabili del programma in esecuzione. L'*asserzione di input* specificherà eventuali vincoli imposti ai valori delle variabili d'ingresso utilizzate dal programma (per esempio, la variabile d'ingresso d può giocare il ruolo di divisore nel programma: chiaramente d non può assumere il valore 0. L'asserzione di input è pertanto $d \neq 0$). Quando non vi sono restrizioni ai valori delle variabili d'ingresso, all'asserzione di input è attribuito il valore logico *vero*. L'*asserzione di output* deve

specificare simbolicamente i risultati che ci si attende vengano prodotti dal programma, per dati d'ingresso che soddisfino l'asserzione di input (per esempio, se un programma è progettato per calcolare il quoziente q ed il resto r risultanti dalla divisione di x per y , l'asserzione di output può essere scritta come:

$$(x = q*y + r) \wedge (r < y)$$

dove il simbolo “ \wedge ” rappresenta il connettore logico “and”).

Implicazioni ed esecuzione simbolica

Il problema di verificare effettivamente un programma si può formulare come un insieme di implicazioni che devono essere dimostrate logicamente vere. Tali implicazioni hanno la forma generale:

$$P \supset Q$$

che si legge “ P implica Q ”. P è chiamata *premessa* e Q *conseguenza*. La tavola di verità che definisce l'implicazione è riprodotta in Tab. 1.3.

Tab. 1.3. Tavola di verità che definisce l'implicazione.

P	Q	$P \supset Q$
true	true	true
true	false	false
false	true	true
false	false	true

Per dimostrare che le dette implicazioni o proposizioni sono logicamente vere, occorre considerare gli effetti dell'esecuzione del programma per dati d'ingresso arbitrari che soddisfino l'asserzione di input. Un modo relativamente elementare per farlo è utilizzare la tecnica dell'*esecuzione simbolica*. Con tale tecnica, tutti i dati d'ingresso sono sostituiti da valori simbolici e tutte le operazioni aritmetiche tra numeri sono tradotte in manipolazioni algebriche di espressioni simboliche. A titolo d'esempio, si consideri il seguente segmento di programma contenente esplicitamente le asserzioni di input ed output.

```

A  readln(x,y);
{assert: true}
x := x-y;
y := x+y;
x := y-x
B  {assert x = y0 \wedge y = x0}
```

dove $x0$ ed $y0$ rappresentano i valori iniziali di x ed y rispettivamente.

Sia l'esecuzione normale che quella simbolica, riprodotte in Tab. 1.4, mostrano che i valori di x ed y vengono scambiati.

Tab. 1.4. Esecuzione normale e simbolica del meccanismo di scambio.

Passo	Esecuzione normale	Esecuzione simbolica
1	valori di input $x = 3$ $y = 1$ $x := x - y \Rightarrow x = 3 - 1 = 2$	valori di input $x = \alpha$ $y = \beta$ $x := x - y \Rightarrow x = \alpha - \beta$
2	$y := x + y \Rightarrow y = 2 + 1 = 3$	$y := x + y \Rightarrow y = (\alpha - \beta) + \beta = \alpha$
3	$x := y - x \Rightarrow x = 3 - 2 = 1$	$x := y - x \Rightarrow x = ((\alpha - \beta) + \beta) - (\alpha - \beta) = \beta$

L'esecuzione simbolica ci consente di trasformare la procedura di verifica nella dimostrazione che l'asserzione di input, con valori simbolici sostituiti a tutte le variabili di ingresso, implica l'asserzione di output con i valori simbolici finali sostituiti a tutte le variabili. Una proposizione espressa in questo modo è conosciuta come *condizione di verifica* (CV) a cavallo del segmento di programma compreso tra l'asserzione di input e l'asserzione di output.

Di norma per attuare la procedura di verifica è necessario stabilire tra le asserzioni di input e di output una serie di condizioni di verifica intermedie. Portato al limite, ciò significa eseguire la procedura di verifica su ciascuna istruzione. Tuttavia, ai fini pratici è di norma sufficiente considerare le condizioni di verifica per blocchi di istruzioni, designati come segmenti a sequenza lineare, segmenti con diramazioni, segmenti ciclici. Adotteremo la convenzione di indicare con $CV(A - B)$ la condizione di verifica a cavallo del segmento di programma tra A e B . Ora ci accingiamo a considerare separatamente le verifiche dei tipi di segmento fondamentali.

Verifica dei segmenti di programma a sequenza lineare

Il modo migliore per illustrare la procedura di verifica è mediante un esempio. Il già menzionato meccanismo di scambio servirà come esempio di un segmento lineare di programma.

La condizione di verifica per questo segmento di programma è:

$$CV(A - B): \text{true} \supset \{x = y0 \wedge y = x0\}$$

Dopo la sostituzione di tutte le variabili con i valori iniziali e finali, otteniamo:

$$CV(A - B): \text{true} \supset ((\alpha - \beta) + \beta) - (\alpha - \beta) = \beta \wedge (\alpha - \beta) + \beta = \alpha$$

La parte conclusiva della condizione di verifica può essere semplificata fino a dare $\beta = \beta$ e $\alpha = \alpha$ che è evidentemente vero, col che l'implicazione è vera.

Verifica dei segmenti di programma con diramazioni

Per trattare segmenti di programma che contengono diramazioni occorre stabilire e dimostrare le condizioni di verifica separatamente per ogni ramo. A titolo d'esempio, si consideri il seguente segmento di programma che garantisce che x risulti minore o uguale ad y .

```

readln(x,y);
A {assert PA; true}
  if x>y then
    begin
      t := x;
      x := y;
      y := t
    end
  B {assert PB: ((x<=y) ∧ (x = x0 ∨ y = y0)) ∨ (x = y0 ∨ y = x0)}

```

In generale, le proposizioni che devono essere dimostrate per il costrutto base **if** sono:

$$\begin{aligned} P_A \wedge C_A \supset P_B \\ P_A \wedge \neg C_A \supset P_B \end{aligned}$$

dove C_A è la condizione di diramazione.

Le due condizioni di verifica richieste in questo caso sono riprodotte di seguito, essendo α e β i valori iniziali rispettivamente di x ed y .

$$CV(A - (t) - B): \text{true} \wedge \alpha > \beta \supset ((\alpha \leq \beta) \wedge (\beta = \alpha \wedge \alpha = \beta)) \vee (\beta = \beta \wedge \alpha = \alpha)$$

Poiché $\alpha > \beta$ è vero e la seconda parte della conclusione (cioè $\alpha = \alpha \wedge \beta = \beta$) è vera, la condizione di verifica per il cammino (t), *vero*, è vera.

La condizione di verifica per il cammino (f), *falso*, è:

$$CV(A - (f) - B): \text{true} \wedge \neg(\alpha > \beta) \supset ((\alpha \leq \beta) \wedge (\alpha = \alpha \wedge \beta = \beta)) \vee (\alpha = \beta \wedge \beta = \alpha)$$

Poiché $\neg(\alpha > \beta) \supset (\alpha < \beta)$ e la conclusione ($\alpha = \alpha \wedge \beta = \beta$) è vera, la condizione di verifica per il cammino *falso* è vera. Ne segue che il segmento di programma racchiuso tra A e B è corretto. Analogamente possono essere trattate le istruzioni di tipo **case**.

Verifica dei segmenti di programmi con loop

Tentando di verificare segmenti ciclici direttamente mediante l'esecuzione simbolica, si incontrano problemi poiché il numero di interazioni è di norma arbitrario. Per superare questo problema, si deve impiegare una forma speciale di asserzione detta *invariante ciclico*. Un invariante ciclico (o di loop) sarà una proprietà (predicato) che identifica il ruolo computazionale progressivo del loop ma al tempo stesso rimane vera prima e dopo ciascun passaggio attraverso il ciclo, indipendentemente dal numero di volte che il loop è eseguito. Una volta stabilito l'invariante di loop, vi sono alcuni passi da intraprendere per verificare il segmento ciclico. Per comprendere tale procedura di verifica, utilizzeremo come modello la seguente struttura di programma con loop singolo.

```

A |asserzione di input PA| { ... segmento di programma lineare ... }
B |invariante ciclico IB| { ... segmento di programma ciclico ... }
C |asserzione di output PC|

```

- Il primo passo da seguire è dimostrare che l'invariante di loop è vero inizialmente, prima che inizi il ciclo. Ciò si può fare allestendo la condizione di verifica $CV(A - B)$ per il segmento di programma da A e B. Ossia, l'asserzione di input, insieme con tutti i cambiamenti prodotti sulle variabili nel segmento $A - B$, deve implicare che l'invariante di loop è vero. Si può utilizzare l'esecuzione simbolica per portare a termine questo passo di verifica. In altri termini, dobbiamo dimostrare che $P_A \supset I_B$.
- La seconda parte della verifica di un loop richiede la dimostrazione che l'invariante ciclico è ancora vero *dopo* l'esecuzione del segmento di programma compreso nel loop. Per far ciò possiamo stabilire una condizione di verifica $CV(B - B)$. Occorre dimostrare che l'invariante ciclico, insieme con la condizione C_B di esecuzione del loop, implica la verità dell'invariante di loop calcolato per i valori finali delle variabili, ossia:

$$I_B \wedge C_B \supset I_B$$

- Anche in questo passo si può utilizzare l'esecuzione simbolica.
- Come passo finale della verifica di un segmento ciclico, occorre dimostrare che l'invariante, insieme con la negazione della con-

dizione di esecuzione del ciclo, implica l'asserzione valida all'uscita del loop. La condizione di verifica in questo caso per la nostra struttura base sarà $CV(B - C)$ e la corrispondente proposizione:

$$I_B \wedge \sim C_B \supset P_C$$

Una comprensione migliore dell'applicazione della tecnica di verifica dei loop si può raggiungere studiando un esempio specifico. Si consideri il seguente segmento di programma, utilizzato per il calcolo del fattoriale.

```

A  {assert P_A: n>=0}
    i := 0;
    fact := 1;
B  {invariant I_B: fact = i! & i ≤ n}
    while i < n do
        begin
            i := i + 1;
            fact := i * fact
        end
C  {assert P_C: fact = n!}

```

Siano γ il valore d'ingresso per α e β rispettivamente i valori correnti per i e $fact$. Con l'esecuzione simbolica possiamo dimostrare che le condizioni associate di verifica da provare sono:

$$1. \quad CV(A - B) : P_A \supset I_B \\ : \gamma \geq 0 \supset 1 = 0! \wedge 0 \leq \gamma$$

Ora, per definizione $1 = 0!$ mentre $\gamma \geq 0 \supset 0 \leq \gamma$ è banalmente vero; pertanto è vera la condizione di verifica $CV(A - B)$.

$$2. \quad CV(B - B) : I_B \wedge C_B \supset I_B \\ : \underbrace{\beta = \alpha! \wedge \alpha \leq \gamma \wedge \alpha < \gamma \supset (\alpha+1)\beta = (\alpha+1)!} \wedge (\alpha+1) \leq \gamma \\ I_B \qquad \qquad \qquad C_B$$

Poiché $\beta = \alpha!$
 risulta per definizione $(\alpha+1)\beta = (\alpha+1)\alpha! = (\alpha+1)! \equiv (\alpha+1)\alpha!$ by
 e perciò è vero $\beta = \alpha! \supset (\alpha+1)\beta = (\alpha+1)!$
 Inoltre $\alpha \leq \gamma \wedge \alpha < \gamma \supset \alpha < \gamma$
 e quindi $(\alpha+1) \leq \gamma$ per α e γ interi.
 Ne segue che è vera la condizione di verifica $CV(B - B)$.

$$3. \quad CV(B - C) : I_B \wedge \sim C_B \supset P_C \\ : \beta = \alpha! \wedge \alpha \leq \gamma \wedge \sim(\alpha < \gamma) \supset \beta = \gamma!$$

Poiché $\sim(\alpha < \gamma) \supset \alpha \geq \gamma$
 allora $\alpha \leq \gamma \wedge \alpha \geq \gamma \supset \alpha = \gamma$

e quindi essendo $\alpha = \gamma$ ne segue che è vera $\beta = \alpha! \supset \beta = \gamma!$
 ed è vera infine la condizione di verifica $CV(B - C)$.

Poiché le condizioni di verifica per tutti e tre i segmenti di programma sono corrette, il segmento completo si dice *parzialmente corretto*.

Quella che il metodo scritto ci fornisce è solo una prova di *correttezza parziale*, con l'implicazione che se l'algoritmo ha termine il risultato prodotto sarà corretto. Per stabilire la *correttezza totale* di programmi che contengono loop, ci rimane perciò il compito distinto e necessario di dimostrarne la terminazione.

Prima di considerare dettagliatamente un metodo di prova della terminazione, stabiliremo le condizioni di verifica per un programma esemplificativo che utilizza un array.

Verifica dei segmenti di programma che usano array

L'idea dell'esecuzione simbolica, sviluppata nei paragrafi precedenti, si può estendere ad alcuni dei più semplici esempi di impiego di array, sebbene ora divenga necessario tener conto di valori simbolici per *tutti* gli elementi dell'array. Come esempio di verifica di un segmento di programma che contiene un array, stabiliremo le condizioni di verifica per un programma che ricerca la posizione dell'elemento minimo dell'array.

Il programma correddato dalle asserzioni può assumere il seguente aspetto:

```

A  {assert P_A: n ≥ 1}
    i := 1;
    p := 1;
B  {invariant I_B: (1 ≤ i ≤ n) ∧ (1 ≤ p ≤ i) ∧ (a[p] ≤ a[1], a[2], ..., a[i])}
    while i < n do
        begin
            i := i + 1;
            if a[i] < a[p] then p := i
        end
C  {assert P_C: (1 ≤ p ≤ n) ∧ (a[p] ≤ a[1], a[2], ..., a[n])}

```

ove abbiamo usato per brevità la notazione:

$$a[p] \leq a[1], a[2], \dots, a[n] \equiv a[p] \leq a[1] \wedge a[p] \leq a[2] \wedge \dots \wedge a[p] \leq a[n]$$

Assumendo rispettivamente per $a[1], a[2], \dots, a[n]$ i valori iniziali $\alpha_1, \alpha_2, \dots, \alpha_n$ e per n il valore iniziale δ , possiamo utilizzare l'esecuzione simbolica per provare le condizioni di verifica:

$$CV(A - B): \delta \geq 1 \supset (1 \leq 1 \leq \delta) \wedge (1 \leq 1 \leq 1) \wedge \alpha_1 \leq \alpha_1$$

Assegnando rispettivamente i valori iniziali β e γ ad i e p , le condizioni di verifica per i due rami del loop sono:

$$CV(B - (l) - B): (1 \leq \beta \leq \delta) \wedge (1 \leq \gamma \leq \beta) \wedge (\alpha_\gamma \leq \alpha_1, \alpha_2, \dots, \alpha_\beta) \wedge \beta < \delta \wedge \alpha_{\beta+1} < \alpha_\gamma \\ \supset (1 \leq \beta + 1 \leq \delta) \wedge (1 \leq \beta + 1 \leq \beta + 1) \wedge \alpha_{\beta+1} \leq \alpha_1, \alpha_2, \dots, \alpha_{\beta+1}$$

$$CV(B - (f) - B): (1 \leq \beta \leq \delta) \wedge (1 \leq \gamma \leq \beta) \wedge (\alpha_\gamma \leq \alpha_1, \alpha_2, \dots, \alpha_\beta) \\ \wedge \beta < \delta \wedge \sim(\alpha_{\beta+1} < \alpha_\gamma)$$

$$\supset (1 \leq \beta + 1 \leq \delta) \wedge (1 \leq \gamma \leq \beta + 1) \wedge (\alpha_\gamma \leq \alpha_1, \alpha_2, \dots, \alpha_{\beta+1})$$

$$CV(B - C): (1 \leq \beta \leq \delta) \wedge (1 \leq \gamma \leq \beta) \wedge (\alpha_\gamma \leq \alpha_1, \alpha_2, \dots, \alpha_\beta) \wedge \sim(\beta < \delta) \\ \supset (1 \leq \gamma \leq \delta) \wedge (\alpha_\gamma \leq \alpha_1, \alpha_2, \dots, \alpha_\delta)$$

L'esempio precedente costituisce solo un "assaggio" di come possano essere trattati i programmi con array. Torniamo adesso a considerare il problema della prova della terminazione dei programmi.

Prova di terminazione

Per dimostrare che un programma ha termine, occorre provare che esso raggiunge l'obiettivo stabilito in un numero finito di passi. Ciò equivale a dimostrare che *qualsiasi* loop del programma termina in un numero finito di passi. In molti casi, la prova della terminazione discende direttamente dalle proprietà dei vari costruttori iterativi. Si consideri, per esempio, il seguente **for-loop**:

```
for i := 1 to n do
begin
:
end
```

Quando n è positivo e finito, la terminazione è garantita in quanto, ad ogni iterazione, il numero di passi che mancano al soddisfacimento della condizione di termine si riduce almeno di uno. Tale diminuzione può essere compiuta solo un numero finito di volte e pertanto il loop deve terminare.

Vi sono ovviamente cicli per cui la prova di terminazione è assai più sottile ed elusiva; in questi casi si presentano di solito due alternative. Quando non c'è una sola variabile che evolve monotonicamente verso la condizione di terminazione, spesso scopriamo che una combinazione

aritmetica di due (o più) variabili ha questo comportamento ad ogni iterazione.

Se la caratteristica di terminazione del programma non è di questo tipo, di solito resta da dimostrare che esiste qualche proprietà dei dati (eventualmente una *sentinella*) che assicura la terminazione.

Più formalmente, il problema di dimostrare la terminazione di un ciclo può essere affrontato associando ad ogni loop un'altra espressione, oltre all'invariante ciclico. Tale espressione, ϵ , dovrà essere una funzione delle variabili utilizzate nel loop; sarà *sempre* non negativa e occorre dimostrare che diminuisce di valore ad ogni iterazione. Se tali criteri sono soddisfatti, il loop deve aver termine. La prova di terminazione può pertanto ridursi a stabilire la verità delle seguenti *condizioni di terminazione* (*CT*).

1. Tornando a far riferimento alla struttura di loop generalizzata, dobbiamo dimostrare che la verità dell'invariante ciclico I_B , insieme con la condizione del loop C_B , implica che il valore dell'espressione ϵ è positivo.
Cioè:

$$CT1(B): I_B \wedge C_B \supset \epsilon > 0$$

La condizione $\epsilon \geq 0$ diventa un invariante del loop. In taluni casi, l'invariante utilizzato per stabilire la correttezza parziale non è sufficiente per l'utilizzo nella prova di terminazione. Questa difficoltà può essere superata aggiungendo all'invariante condizioni supplementari (derivate dalle asserzioni precedenti).

2. La seconda proposizione da dimostrare è che l'invariante di loop I_B , insieme con la condizione del loop C_B , implica che il valore ϵ_0 dell'espressione *prima* dell'esecuzione è strettamente maggiore del suo valore ϵ dopo l'esecuzione del loop; cioè per il loop B :

$$CT2(B - B): I_B \wedge C_B \supset (\epsilon_0 > \epsilon) \wedge (\epsilon \geq 0)$$

Il valore finale di ϵ si può ottenere dall'esecuzione simbolica delle istruzioni del loop.

Una volta dimostrato che queste due proposizioni sono vere, possiamo concludere immediatamente che il processo di ripetizione è finito poiché ϵ può essere diminuita solo un numero finito di volte mantenendosi positiva come richiesto da $CT1(B)$. Le considerazioni per la terminazione dei loop **repeat..until** seguono una linea di ragionamento analoga.

Ora indichiamo a grandi linee la prova di terminazione per il seguente programma quoziante/resto.

```

begin
A {assert  $P_A$ :  $(x \geq 0) \wedge (y > 0)$ }
     $r := x$ ;  $q := 0$ ;
B {invariant  $I_B$ :  $r \geq 0 \wedge x = y * q + r$ }
    while  $y \leq r$  do
        begin
             $r := r - y$ ;
             $q := q + 1$ 
        end
C {assert  $P_C$ :  $(x = y * q + r) \wedge (0 \leq r < y)$ }

```

Per dimostrare la terminazione di questo segmento di programma useremo:

$$\epsilon = r + y$$

Per stabilire la prima condizione di terminazione, dovremo aggiungere all'invariante I_B la condizione addizionale $y > 0$ da P_A . Abbiamo così:

$$CTI(B): (r \geq 0) \wedge (x = y * q + r) \wedge (y > 0) \wedge (y \leq r) \supset (r + y > 0)$$

Ora:

$$(r \geq 0) \wedge (y > 0) \supset (r + y > 0)$$

da cui si dimostra che $CTI(B)$ è vera.

Assegnando rispettivamente ad x , y e q i valori simbolici α , β , γ , δ all'inizio del loop, otteniamo per $CT2(B - B)$ l'espressione:

$$CT2(B - B): (\gamma \geq 0) \wedge (\alpha = \beta * \delta + \gamma) \wedge (\beta > 0) \wedge (\beta \leq \gamma) \supset \gamma + \beta > (\gamma - \beta) + \beta$$

ove:

$$\epsilon_0 = \gamma + \beta \quad \text{e} \quad \epsilon = (\gamma - \beta) + \beta$$

che si può facilmente dimostrare vera, così che la prova di terminazione è conclusa.

Una volta stabilite sia la correttezza parziale sia la terminazione, la verifica è completa. Intraprendere la verifica dettagliata dei programmi richiede un grado di notevole maturità e di esperienza matematica; la lunghezza e il dettaglio di tali dimostrazioni sono di solito assai maggiori nella mole del testo di programma originario, perciò, nelle discussioni che seguono, cercheremo di annotare i programmi solo con le asserzioni che interessano.

1.5 L'EFFICIENZA DEGLI ALGORITMI

Le considerazioni di efficienza per gli algoritmi sono intimamente legate alla progettazione, all'implementazione e all'analisi di questi. Ciascun algoritmo deve far uso di alcune risorse del computer per completare il suo compito; le risorse più importanti in relazione all'efficienza sono il *tempo di unità centrale di elaborazione* (tempo di CPU) e la *memoria centrale*. A causa del costo delle risorse di calcolo, è sempre desiderabile progettare algoritmi che facciano un uso economico del tempo di CPU e della memoria. È questa un'affermazione facile da fare, ma spesso difficile da mettere in atto per le cattive abitudini di progetto, per l'intrinseca complessità del problema, o per entrambi questi motivi. Come in molti altri aspetti della progettazione degli algoritmi, non esiste una ricetta per produrre algoritmi efficienti; al di là di qualche regola comune, ciascun problema ha caratteristiche proprie che richiedono risposte specifiche per risolverlo efficientemente. Sulla traccia di quest'ultima affermazione cercheremo di fornire qualche indicazione che possa essere talvolta d'aiuto nella costruzione di algoritmi efficienti.

Ridondanza nei calcoli

Molte delle defezioni che si insinuano nell'implementazione degli algoritmi sono dovute all'esecuzione di calcoli inutili o all'impiego di memoria non necessaria. Gli effetti di calcoli ridondanti sono ancor più gravi quando questi sono inclusi in un ciclo da ripetersi più volte. L'errore più comune nell'impiego dei loop è il calcolo ripetuto di parte di un'espressione che rimane invariata durante tutta la fase di esecuzione del loop. Questo fatto è illustrato nell'esempio seguente:

```

x := 0;
for i := 1 to n do
begin
    x := x + 0.01;
    y := (a*a*a+c)*x*x+b*b*x;
    writeln ('x = ', x, 'y = ', y)
end

```

Questo loop esegue il *doppio* delle moltiplicazioni necessarie per portare a termine il calcolo. Le moltiplicazioni e le addizioni inutili possono essere rimosse, calcolando le due nuove costanti $a3c$ e $b2$ prima di eseguire il loop:

```

a3c := a*a*a+c;
b2 := b*b;
x := 0;
for i := 1 to n do
begin
  x := x+0.01;
  y := a3c*x*x+b2*x;
  writeln ('x = ',x,'y = ',y)
end

```

Il risparmio in questo caso non è particolarmente significativo, ma vi sono molte altre situazioni in cui può esserlo assai di più. È sempre molto importante cercare di eliminare le ridondanze nei loop interni, poiché possono rivelarsi molto costose.

Riferimento agli elementi di un array

Calcoli ridondanti possono facilmente insinuarsi anche nell'elaborazione degli array, se non si esercita la dovuta attenzione. Si considerino, per esempio, le due versioni di un algoritmo che cerca l'elemento massimo e la sua posizione in un array.

Versione (1)

```

p := 1;
for i := 2 to n do
  if a[i]>a[p] then p := i;
max := a[p]

```

Versione (2)

```

p := 1;
max := a[1];
for i := 2 to n do
  if a[i]>max then
  begin
    max := a[i];
    p := i
  end

```

L'implementazione nella versione (2) sarà di norma preferita in quanto il test di condizione (cioè $a[i] > max$), che è l'istruzione dominante, risulta più efficiente in esecuzione rispetto al test corrispondente nella versione (1). È più efficiente poiché l'uso della variabile *max*

richiede una sola istruzione di riferimento alla memoria, mentre la variabile *a[p]* richiederebbe due riferimenti alla memoria e un'operazione di addizione per individuare il valore corretto da usare nel confronto. Inoltre nella versione (2), l'introduzione della variabile *max* rende più evidente qual è il compito da assolvere.

Inefficienza causata da terminazione ritardata

Un altro punto in cui le inefficienze fanno breccia nell'implementazione è dove si eseguono molti più tests di quanti sarebbero richiesti per risolvere il problema in esame. Questo tipo di inefficienza è meglio illustrato con un esempio. Supponiamo di dover ricercare linearmente un nome particolare in una lista di nomi ordinata alfabeticamente. In questo caso un'implementazione inefficiente sarebbe quella in cui venissero esaminati *tutti* i nomi della lista, anche quando si fosse raggiunto il punto oltre il quale il nome cercato non può più apparire (per esempio supponiamo di cercare il nome ROSSI: non appena incontrato un nome che in ordine alfabetico viene dopo ROSSI, p.es. RUSSO, non occorrerà procedere ulteriormente).

L'implementazione inefficiente potrebbe avere la forma:

1. While nome cercato < > nome corrente and not end-of-file do
 - (a) recupera il prossimo nome dalla lista.

Un'implementazione più efficiente sarebbe:

1. while nome cercato > nome corrente and not end-of-file do
 - (a) recupera il prossimo nome dalla lista.
2. esamina se il nome corrente è quello cercato.

Lo stesso tipo di inefficienza può essere introdotto nell'algoritmo di *bubblesort* (algoritmo 5.3) se si è posta scarsa cura nell'implementazione. Ciò può capitare se il loop interno che pilota il meccanismo di scambio percorre sempre l'intera lunghezza dell'array. Per esempio:

```

for i := 1 to n-1
  for j := 1 to n-1
    if a[j]>a[j+1] then "scambia a[j] con a[j+1]"

```

Con questo meccanismo di ordinamento, dopo l'*i*-esima iterazione, gli ultimi *i* valori dell'array saranno già ordinati. Pertanto, per ogni *i*, il loop interno non dovrà andare oltre $n - i$. La struttura del loop sarà perciò:

```

for i := 1 to n-1
  for j := 1 to n-i
    if a[j]>a[j+1] then "scambia a[j] con a[j+1]"

```

La lezione che se ne ricava è che dovremmo sempre cercare di trarre vantaggio dall'ordine già presente nella struttura di un problema.

Rilievo anticipato delle condizioni richieste per l'output

Il *bubblesort* ci fornisce l'esempio di un altro tipo di inefficienza che interessa la terminazione. A causa della natura dei dati in ingresso, succede talvolta che l'algoritmo raggiunga la condizione di output desiderata *prima* che siano soddisfatte le condizioni generali per la terminazione. Per esempio, un *bubblesort* potrebbe essere impiegato per ordinare un insieme di dati già quasi ordinato; quando ciò si verifica, è molto probabile che l'algoritmo disponga dei dati ordinati ben prima che siano soddisfatte le condizioni di terminazione dei loop. È quindi desiderabile che il sort abbia termine non appena si scopre che l'array è già ordinato. Per ottenere questo non dobbiamo far altro che controllare se si sono verificati scambi nel passo corrente del loop interno. Se non ci sono stati scambi al passo corrente, i dati devono essere ordinati e pertanto si può attuare una terminazione anticipata (un esame dell'algoritmo 5.3 mostra come si sia implementato ciò per il *bubblesort*). In generale, per rilevare le condizioni di una terminazione anticipata, dovremo includere ulteriori passi e confronti; tuttavia è buona cosa introdurli, se si possono mantenere poco costosi (come nel *bubblesort*). In altri termini, quando è possibile la terminazione anticipata, dobbiamo sempre spendere test e forse anche memoria per ottenerla.

Compromesso tra memoria ed efficienza

Spesso per migliorare le prestazioni di un algoritmo si baratta la memoria a vantaggio dell'efficienza. Ciò che normalmente accade in questi casi è che si precalcolano o si salvano alcuni risultati intermedi, evitando così di compiere più avanti una quantità di confronti e di calcoli non necessari. (Si veda per esempio il problema della sequenza monotona più lunga, algoritmo 4.7).

Una strategia spesso utilizzata per cercare di accelerare un algoritmo è di implementarlo usando il numero minimo di loop, il che se di norma è possibile, inevitabilmente rende i programmi più difficili da leggere e da mettere a punto. Perciò di solito è meglio attenersi alla regola di *un loop per ogni compito* così come ogni variabile ha una sola funzione. Quando si chiede una soluzione più efficiente per un problema è assai meglio cercare di migliorare l'algoritmo piuttosto che ricorrere a "trucchi di programmazione" che tendono a renderne oscuro il funzionamento. L'implementazione nitida di un algoritmo è preferibile all'implementazione "truccata" di un algoritmo non altrettanto buono. Ora non ci resta che tentare di misurare l'efficienza degli algoritmi.

1.6 L'ANALISI DEGLI ALGORITMI

Ci sono di norma *molte* strade per risolvere un dato problema. In informatica, come in quasi tutti i campi dell'attività umana, ci occupiamo di norma di soluzioni "buone" dei problemi. Questo solleva alcune questioni come, per l'appunto, che cosa debba intendersi per soluzione "buona" di un problema. Nel progetto di algoritmi, "buono" va inteso in senso qualitativo e quantitativo; vi sono sovente alcuni aspetti estetici e personali in ciò ma, a livello più pratico, siamo di norma interessati a soluzioni che siano *economiche nell'impiego delle risorse umane e di calcolo*. Tra le altre, di norma i buoni algoritmi possiedono le seguenti qualità e possibilità:

1. Sono soluzioni semplici ma potenti e generali.
2. Possono essere facilmente compresi da altri, ossia l'implementazione è chiara e concisa, senza "trucchi".
3. Possono essere facilmente modificati in caso di necessità.
4. Sono corretti per situazioni chiaramente definite.
5. Possono essere compresi a diversi livelli.
6. Sono economici quanto ad impiego di tempo di calcolo, di memoria e periferiche.
7. Sono documentati a sufficienza per essere impiegati da utenti che non hanno una conoscenza dettagliata del loro funzionamento.
8. Non sono obbligati a funzionare su un particolare computer.
9. Sono utilizzabili come sottoprocedure per problemi diversi.
10. La soluzione è gradevole e soddisfacente per il suo progettista: un prodotto che egli si sente fiero d'aver creato.

Questi aspetti qualitativi di un buon algoritmo sono importanti, ma è altrettanto importante stabilire alcune misure quantitative che completino la valutazione della "bontà" di un algoritmo. Le misurazioni quantitative sono utili per darci un mezzo per predire direttamente le prestazioni di due o più algoritmi destinati a risolvere lo stesso problema. Ciò può essere importante poiché l'impiego di un algoritmo più efficiente significa un risparmio di risorse computazionali, che si traduce in un risparmio di tempo e denaro.

Complessità computazionale

Per effettuare una misura quantitativa delle prestazioni di un algoritmo occorre perfezionare un modello di calcolo che ne riproduca il comportamento sotto definite condizioni di input. Tale modello deve catturare l'essenza del calcolo ma al tempo stesso deve essere indipendente da ogni linguaggio di programmazione.

Pertanto ci serve caratterizzare la prestazione di un algoritmo in funzione della dimensione (di solito n) del problema da risolvere. Ovviamente, per risolvere in una stessa classe problemi più ampi sono necessarie risorse di calcolo maggiori. L'importante domanda che dobbiamo porci è: come varia il costo della soluzione del problema al variare di n ? La prima risposta a questa domanda potrebbe essere che con n cresce nella stessa misura il costo (p. es. un problema con $n = 200$ impiega il doppio di un problema con $n = 100$). Mentre questa dipendenza *lineare* da n è vera per alcuni semplici algoritmi, in generale il comportamento con n segue altri andamenti completamente differenti.

All'estremo inferiore della scala abbiamo algoritmi con dipendenza da n logaritmica (o ancora migliore), mentre al limite superiore della scala abbiamo algoritmi con dipendenza da n esponenziale. All'aumentare di n , la differenza relativa del costo di calcolo tra questi due estremi è enorme. La Tab. 1.5 illustra il costo comparativo per diversi valori di n .

Tab. 1.5. Costo di calcolo in funzione della dimensione del problema, per diversi valori di complessità computazionale.

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
1	2	2	4	8	4
3.322	10	33.22	10^2	10^3	$>10^3$
6.644	10^2	664.4	10^4	10^6	$>>10^{25}$
9.966	10^3	9966.0	10^6	10^9	$>>10^{250}$
13.287	10^4	132 877	10^8	10^{12}	$>>10^{2500}$

Appare ovvio da tale tabella che con un algoritmo che manifesta un comportamento esponenziale si possono risolvere solo problemi *molto piccoli*. Anche ammettendo che il computer esegua circa un milione di operazioni al secondo, un algoritmo esponenziale con $n = 100$ richiederebbe per terminare un tempo immensamente più lungo dell'età della terra. All'altro estremo, per un algoritmo con dipendenza da n logaritmica, un problema con $n = 10^4$ richiedrebbe solo 13 passi, per un totale di 13 microsecondi di tempo di calcolo. Questi esempi mettono in evidenza quanto sia importante conoscere come si comportano gli algoritmi in funzione delle dimensioni del problema.

Per decidere come caratterizzare il comportamento di un algoritmo in funzione della dimensione n del problema, dobbiamo studiare molto attentamente il meccanismo e decidere esattamente quale ne sia l'aspetto essenziale. Può essere il numero di volte per cui è calcolata una particolare espressione aritmetica (o altro), oppure il numero di confronti o di scambi che devono essere attuati al crescere di n . Per esempio, molti algoritmi di ordinamento sono caratterizzati da confronti, scambi e spostamenti di dati; di norma prevale il numero dei confronti, così che nel modello computazionale degli algoritmi di ordinamento utilizzeremo i confronti.

Ordine di complessità

Per rappresentare le funzioni che delimitano il tempo di esecuzione di un algoritmo si è sviluppata una notazione standard; si tratta di una notazione d'ordine, nota comunemente come *notazione O*. Un algoritmo in cui il meccanismo dominante è eseguito cn^2 volte, con c costante ed n dimensione del problema, si dice che ha complessità di *ordine* n^2 e si indica $O(n^2)$.

Formalmente, una funzione $g(n)$ è $O(f(n))$ a patto che esista una costante c per cui valga la relazione:

$$g(n) \leq c f(n)$$

per qualsiasi valore di n finito e positivo. Con tali convenzioni abbiamo gli strumenti per caratterizzare la complessità asintotica di un algoritmo e determinare quindi la dimensione del problema che esso è in grado di risolvere utilizzando un computer sequenziale convenzionale. Possiamo scrivere la relazione di cui sopra nella forma:

$$\lim_{n \rightarrow \infty} \frac{g(n)}{f(n)} = c \quad \text{in cui } c \text{ è non nullo.}$$

Come esempio, supponiamo di avere un algoritmo che per completare la sua funzione richiede $(3n^2 + 6n + 3)$ confronti. In accordo con le convenzioni appena delineate abbiamo:

$$g(n) = 3n^2 + 6n + 3 \quad \text{e} \quad \lim_{n \rightarrow \infty} \frac{3n^2 + 6n + 3}{n^2} = 3$$

Ne segue che questo particolare algoritmo ha complessità asintotica $O(n^2)$. Impiegando questa metodologia per decidere della superiorità di un algoritmo rispetto a un altro, dobbiamo porre molta attenzione alle costanti di proporzionalità; può succedere che un algoritmo con una complessità asintotica più elevata abbia una costante di proporzionalità piccolissima e perciò, per un certo campo di valori di n , fornisca una prestazione migliore a confronto di un algoritmo con complessità inferiore ma costante di proporzionalità più elevata.

Comportamento nel caso medio e nel caso peggiore

Nell'analizzare un dato algoritmo si considerano normalmente due misure di prestazione: il comportamento nel *caso peggiore* e nel *caso medio*. Entrambe le misure possono riferirsi sia alla complessità spaziale sia a quella temporale. La *complessità del caso peggiore* per una data dimensione n di problema corrisponde alla massima complessità incontrata in *tutti* i problemi di dimensione n . La determinazione della complessità del caso peggiore è relativamente semplice per molti al-

ritmi, sceglio un insieme di condizioni di ingresso che obblighino l'algoritmo a procedere ad ogni passo verso il suo scopo finale con la minima velocità.

In molte applicazioni pratiche è assai più importante avere una misura della *complessità probabile* di un dato algoritmo piuttosto che il comportamento del caso peggiore. La complessità probabile fornisce una misura del comportamento dell'algoritmo *mediata* su tutti i problemi di dimensione n . Nel confrontare due algoritmi che risolvono un dato problema, generalmente preferiremo l'algoritmo che ha la complessità probabile più bassa. Sfortunatamente, la messa a punto di un modello di calcolo che caratterizzi il comportamento medio spesso implica analisi combinatorie alquanto complesse e sofisticate.

Analisi probabilistica del caso medio

Come esempio semplice, supponiamo di voler caratterizzare il comportamento di un algoritmo che ricerca linearmente un determinato valore x in una lista ordinata di elementi.

1	2	...	n
---	---	-----	-----

Nel *caso peggiore* l'algoritmo dovrà esaminare *tutti* gli n valori della lista prima di terminare.

La situazione di *caso medio* è alquanto diversa. Per un'analisi probabilistica del caso medio si assume generalmente che tutti i possibili punti di terminazione siano *ugualmente probabili*, ossia la probabilità che x venga trovato in posizione 1 è $1/n$, in posizione 2 è $1/n$, e così via. Il costo medio di ricerca è pertanto la somma di tutti i possibili costi di ricerca, ciascuno moltiplicato per la rispettiva probabilità. Per esempio, per $n = 5$ avremo:

$$\text{costo medio di ricerca} = 1/5 (1 + 2 + 3 + 4 + 5) = 3$$

Ricordiamo che $1 + 2 + 3 + \dots + n = n(n+1)/2$ (formula di Gauss), nel caso generale avremo:

$$\text{costo medio di ricerca} = \frac{1}{n} \left(\frac{n(n+1)}{2} \right) = \frac{n+1}{2}$$

Come secondo esempio, lievemente più complesso, consideriamo l'analisi del caso medio per la procedura di ricerca binaria descritta nell'algoritmo 5.7. Ciò che vogliamo stabilire in questa analisi è il *numero medio di iterazioni* del loop di ricerca necessario perché l'algoritmo termini *con successo*. Questa analisi si riferirà solo alla *prima*

implementazione proposta per la ricerca binaria nell'algoritmo 5.7, che termina non appena si trova il valore cercato; l'albero binario di ricerca corrispondente a un array di dimensione 15 è riprodotto in Fig. 1.2.

Con riferimento all'albero, vediamo che un elemento può essere trovato con un confronto, due elementi con 2 confronti, 4 elementi con 3 confronti, e così via. Nel caso generale, 2^i elementi richiedono $i + 1$ confronti. Ora assumendo che tutti gli elementi dell'array abbiano la stessa probabilità di essere trovati (ossia $1/n$), il costo medio di ricerca è ancora la somma di tutti i possibili costi di ricerca, ciascuno moltiplicato per la rispettiva probabilità. Ossia, [*]

$$\text{costo medio di ricerca} = \frac{1}{n} \sum_{i=0}^{\lfloor \log_2 n \rfloor} (i+1) \times 2^i = \frac{1}{n} \left\{ \sum_{i=0}^{\lfloor \log_2 n \rfloor} i \times 2^i + n \right\}$$

Questa formula è esatta solo per n pari a una potenza di due diminuita di uno. Occorre un po' di analisi matematica per calcolare la sommatoria:

$$s = \sum_{i=0}^{\lfloor \log_2 n \rfloor} i \times 2^i$$

Possiamo riconoscere che questa somma è una progressione geometrica e sappiamo che in generale:

$$1 + x + x^2 + \dots + x^k = \frac{x^{k+1} - 1}{x - 1}$$

Possiamo anche osservare che:

$$i \times 2^i = \frac{d}{dx} (2x^i) \Big|_{x=2}$$

Ne segue che per calcolare la nostra sommatoria possiamo eseguire la derivata di $(x^{k+1} - 1)/(x - 1)$ moltiplicata per 2 e calcolata per $x = 2$ quando $k = \lfloor \log_2 n \rfloor$. Così facendo abbiamo:

$$s = \sum_{i=0}^{\lfloor \log_2 n \rfloor} i \times 2^i = (n+1) (\lfloor \log_2 n \rfloor - 1) + 2$$

[*] I simboli $\lfloor \cdot \rfloor$ sono usati per indicare "il massimo intero non maggiore di", mentre Σ è il simbolo matematico di sommatoria.

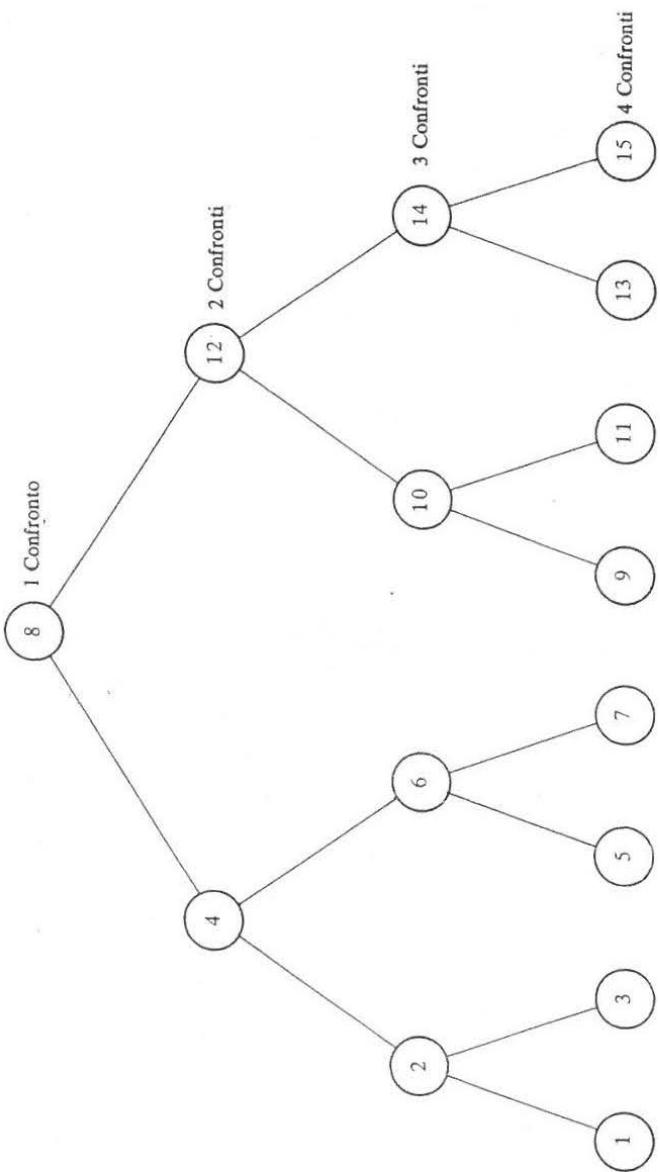


Fig. 1.2. Albero binario di decisione per un insieme di 15 elementi

Infine, dopo sostituzione della sommatoria nell'espressione del costo medio di ricerca, otteniamo:

$$\text{costo medio di ricerca} = \frac{(n+1) \lfloor \log_2 n \rfloor + 1}{n} \approx \lfloor \log_2 n \rfloor \text{ per } n \text{ grande.}$$

Questo esempio mostra che anche per un caso relativamente semplice occorre un certo livello di sofisticazione matematica nel condurre l'analisi. Abbiamo finalmente alcuni degli strumenti necessari per affrontare lo studio della progettazione di algoritmi.

ALGORITMI DI BASE

INTRODUZIONE

Il computer é ormai prossimo a raggiungere lo stato di macchina universale, per la sua abilità unica di eseguire un insieme di compiti quasi infinito. Esso combina la versatilità con una straordinaria velocità. Quello che sorprende in questo contesto é che soltanto un piccolissimo numero di istruzioni elementari é utilizzato per eseguire tutti i calcoli. Al livello più fondamentale, un computer presenta istruzioni per memorizzare l'informazione, per eseguire calcoli aritmetici e per confrontare le informazioni; vi sono anche istruzioni per controllare il modo con cui l'elaborazione viene eseguita. Queste istruzioni fondamentali sono impiegate in definitiva per costruire le istruzioni più potenti e complesse che troviamo come primitive nei linguaggi di programmazione ad alto livello, come il Pascal. A livello del linguaggio di programmazione succede la stessa cosa: ancora una volta, si usano pochi meccanismi elaborativi per costruire procedure di calcolo molto più vaste. In questo capitolo prenderemo in esame tali meccanismi fondamentali, insieme con considerazioni sul modo di rappresentare l'informazione nei computer.

Tra questi meccanismi fondamentali, uno dei più largamente usati é lo scambio dei valori associati a due variabili; in particolare questa operazione é ampiamente adoperata in algoritmi di ordinamento. Anche l'idea del conteggio é parte essenziale di molte procedure di calcolo più elaborate; un'estensione dell'idea del conteggio sta alla base del meccanismo molto diffuso per la sommatoria dei dati di un insieme.

Un numero apparentemente illimitato di problemi può essere espresso in un modo che consenta di risolverli ripetendo più e più volte un meccanismo di base. Per esempio, un insieme di n numeri può essere sommato semplicemente aggiungendo i numeri successivi dell'insieme ad un accumulatore (v. algoritmo 2.3).

Il meccanismo di base che é utilizzato nel procedimento iterativo o ripetitivo deve provocare modifiche nelle variabili e/o nelle informazioni considerate dal meccanismo stesso. Solo in questo modo si può procedere verso la terminazione del procedimento iterativo, che é sempre concluso dal verificarsi di una condizione.

Implementando le soluzioni dei problemi mediante computer, probabilmente l'abilità più importante che dobbiamo sviluppare é quella di esprimere i problemi in una maniera che consenta di formularne una soluzione iterativa. Sfortunatamente, prima dell'impatto iniziale con la

soluzione dei problemi via computer, la maggior parte di noi ha avuto un'esperienza assai limitata (per lo meno a livello consciente) nel formulare soluzioni iterative. Spesso e volentieri, nelle nostre prime esperienze di programmazione abbiamo affrontato semplici problemi che possiamo risolvere molto facilmente (per esempio trovare il numero massimo di un insieme), ma per i quali abbiamo grande difficoltà a formulare una soluzione iterativa. Ciò accade poiché di molti problemi sappiamo dare la soluzione, senza formulare esplicitamente o comprendere i metodi usati per ottenerla.

Occupandoci di computer non possiamo più permetterci questo lusso. Una volta che abbiamo afferrato l'idea del "pensare iterativamente (e anche ricorsivamente)", ci siamo messi in condizione di poter formulare soluzioni elaborate anche per problemi veramente complessi.

Un altro caposaldo che dobbiamo stabilire presto nella pratica elaborativa è la comprensione di come i computer rappresentino e manipolino l'informazione. In particolare, dobbiamo apprezzare le differenze tra la rappresentazione a numeri e quella a caratteri, e la maniera di manipolare e convertire le diverse rappresentazioni.

Solo dopo che abbiamo digerito l'essenza di questi concetti fondamentali, siamo in una posizione da cui affrontare i più complicati aspetti della soluzione dei problemi mediante computer.

ALGORITMO 2.1 SCAMBIO DEI VALORI DI DUE VARIABILI

Problema

Assegnate due variabili, a e b , scambiare i loro valori.

Sviluppo dell'algoritmo

Il problema di scambiare tra loro i valori associati a due variabili coinvolge un meccanismo fondamentale che si presenta in molti algoritmi di ordinamento e di manipolazione dei dati. Per definire il problema più chiaramente esamineremo un esempio specifico.

Supponiamo che alle variabili a e b siano assegnati valori come indicato di seguito. Vale a dire:

Configurazione di partenza

a	b
721	463

Ciò significa che la locazione di memoria o variabile a contiene il valore 721 e la locazione di memoria o variabile b contiene il valore 463. Il nostro scopo è di sostituire il contenuto di a con 463 e il contenuto di b con 721. In altri termini, vogliamo trovarci alla fine con la configurazione seguente:

Configurazione finale

a	b
463	721

Per modificare il valore di una variabile possiamo utilizzare l'operatore di assegnazione. Poiché vogliamo che a assuma il valore attualmente posseduto da b e b il valore di a , potremmo forse eseguire lo scambio con le assegnazioni seguenti:

$$a := b; \quad (1)$$

$$b := a \quad (2)$$

dove " $:=$ " è l'operatore di assegnazione. In (1) " $:=$ " fa sì che il valore memorizzato nella cella di memoria b venga *copiato* nella cella di memoria a .

Ripercorriamo questi due passi per assicurarci che producano l'effetto desiderato.

Partiamo con la configurazione:

a	b
721	463

quindi, dopo l'assegnazione $a := b$, abbiamo:

a	b
463	463

L'assegnazione (1) ha *modificato* il valore di a ma ha lasciato inalterato il valore di b . Controllando la nostra configurazione di destinazione vediamo che a ha assunto il valore 463 come richiesto. Niente male! Dobbiamo controllare anche b . Quando il passo (2) di assegnazione, cioè $b := a$, viene eseguito *dopo* l'esecuzione del passo (1) otteniamo:

a	b
463	463

Nell'esecuzione del passo (2) *a* non è cambiato mentre *b* assume il valore che appartiene correttamente ad *a*. La configurazione cui siamo pervenuti non rappresenta la soluzione che cercavamo.

Il guaio è che nell'eseguire l'assegnazione:

$a := b$

abbiamo perso il valore che inizialmente apparteneva ad *a* (ossia è andato perduto 721), valore che volevamo fosse assunto alla fine da *b*. Il nostro problema perciò va definito più accuratamente così:

nuovo valore di *a* := vecchio valore di *b*;
nuovo valore di *b* := vecchio valore di *a*

Ciò che abbiamo ottenuto con la presente formulazione è di eseguire l'assegnazione:

nuovo valore di *b* := nuovo valore di *a*

In altre parole, quando eseguiamo il passo (2) non utilizziamo il valore di *a* che corrisponderebbe al funzionamento corretto, perché *a* è già stato *cambiato*.

Per risolvere questo problema dello scambio dobbiamo trovare il modo di non distruggere il vecchio valore di *a* quando eseguiamo l'assegnazione:

$a := b$

Un modo per farlo è introdurre una variabile temporanea *t* e copiare in tale variabile il valore di *a* prima di eseguire il passo (1). I passi per far ciò sono:

$t := a;$

$a := b$

A valle di questi due passi abbiamo:

<i>a</i>	<i>t</i>	<i>b</i>
463	721	463

Siamo messi meglio dell'ultima volta poiché ora abbiamo il vecchio valore di *a* memorizzato in *t*; è il valore che ci occorre per attribuirlo a *b*. Possiamo perciò eseguire l'assegnazione:

$b := t$

Dopo l'esecuzione di questo passo abbiamo:

<i>a</i>	<i>t</i>	<i>b</i>
463	721	721

Controllando di nuovo la configurazione obiettivo, vediamo che ora *a* e *b* sono stati scambiati come richiesto.

Adesso possiamo indicare le linee della procedura di scambio.

Descrizione dell'algoritmo

1. Salvare in *t* il valore originale di *a*.
2. Assegnare ad *a* il valore originale di *b*.
3. Assegnare a *b* il valore originale di *a* salvato in *t*.

Il meccanismo di scambio come strumento di programmazione è implementato nel modo più utile sotto forma di procedura che accetta due variabili e ne restituisce i valori scambiati.

Implementazione in Pascal

```
procedure exchange (var a,b: integer);
var t: integer;
begin {save the original value of a then exchange a and b}
  {assert: a=a0 ∧ b=b0}
  t := a;
  a := b;
  b := t;
  {assert: a=b0 ∧ b=a0}
end
```

Note di progetto

1. L'impiego di una variabile temporanea intermedia consente allo scambio delle due variabili di avvenire correttamente.
2. Questo esempio evidenzia come, in ogni fase del calcolo, una variabile assuma sempre il valore imposto dall'assegnazione più recente fatta su quella variabile.
3. Ripercorrere il meccanismo con un esempio particolare può essere una maniera utile per rilevare difetti di progettazione.
4. Un'applicazione dello scambio più comune coinvolge due elementi di un array (p. es. $a[i]$ e $a[j]$). I passi per tale scambio sono:

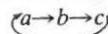
$t := a[i];$
 $a[i] := a[j];$
 $a[j] := t$

Applicazioni

Algoritmi di ordinamento.

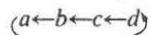
Problemi supplementari

- 2.1.1 Sono dati due bicchieri contrassegnati A e B. Il bicchiere A è pieno di succo di lampone, il bicchiere B di limonata. Suggerire un metodo per scambiare i contenuti dei bicchieri A e B.
2.1.2 Progettare un algoritmo che esegua lo scambio seguente:



Le frecce indicano che b deve assumere il valore di a , c il valore di b , e così via.

- 2.1.3 Progettare un algoritmo che esegua lo scambio seguente:



- 2.1.4 Date due variabili intere a e b , scambiarne i valori senza utilizzare una terza variabile temporanea.

- 2.1.5 Che succede quando gli argomenti forniti alla procedura *exchange* sono i e $a[i]$?

ALGORITMO 2.2 CONTEGGIO

Problema

Dato l'insieme dei risultati d'esame (nell'intervallo da 0 a 100) di n studenti, contare il numero di studenti che hanno superato la prova. L'esame s'intende superato con un voto maggiore o uguale a 50.

Sviluppo dell'algoritmo

I meccanismi di conteggio sono usati molto spesso negli algoritmi per computer. Solitamente occorre contare il numero degli elementi di un insieme che posseggono una particolare proprietà o soddisfano una o più determinate condizioni. Questa classe di problemi è rappresentata tipicamente dal problema dei "voti d'esame".

Come punto di partenza per lo sviluppo di un algoritmo che risolva questo problema col computer, possiamo considerare come risolveremmo un esempio particolare a mano.

Supponiamo d'avere l'insieme di voti:

55, 42, 77, 63, 29, 57, 89

Per contare i promossi in questo lotto, possiamo partire da sinistra, esaminare il primo voto (ossia 55), vedere che è ≥ 50 , e ricordarci che finora è passato uno studente. Si esamina quindi il secondo voto e non si esegue alcuna modifica al contatore; quando arriviamo al terzo voto scopriamo che è ≥ 50 e perciò aumentiamo di uno il contatore precedente. Il processo continua in modo simile finché non si sono esaminati tutti i voti.

Più in dettaglio abbiamo:

Voti	Evoluzione dei contatori dei promossi
55	cont. precedente = 0 cont. corrente = 1
42	cont. precedente = 1 cont. corrente = 1
77	cont. precedente = 1 cont. corrente = 2
63	cont. precedente = 2 cont. corrente = 3
29	cont. precedente = 3 cont. corrente = 3
57	cont. precedente = 3 cont. corrente = 4
89	cont. precedente = 4 cont. corrente = 5
Numero di studenti promossi = 5	

Dopo che ogni voto è stato elaborato, il contatore corrente riproduce il numero di studenti promossi incontrati sino a questo punto nella lista dei voti.

Ora ci dobbiamo domandare: come possiamo eseguire il conteggio? Dall'esempio fatto vediamo che ogni volta che dobbiamo incrementare il contatore lo facciamo sul valore precedente. Ossia:

$$\text{contatore corrente} = \text{contatore precedente} + 1$$

Quando, per esempio, arriviamo al voto 57, abbiamo:

$$\text{contatore precedente} = 3$$

Il contatore corrente pertanto diventa 4. Quando passiamo al voto successivo (cioè 89) il contatore corrente 4 assume il ruolo di contatore precedente. Ciò significa che ogni volta che viene generato un nuovo contatore corrente, esso deve poi assumere il ruolo di contatore precedente prima di poter considerare il voto successivo. I due passaggi di questo procedimento possono essere rappresentati da:

$$\text{contatore-corrente} := \text{contatore-precedente} + 1 \quad (1)$$

$$\text{contatore-precedente} := \text{contatore-corrente} \quad (2)$$

Questi due passi possono essere applicati ripetutamente fino a ottenere il conteggio richiesto. Insieme col test di condizione e l'input del voto successivo, eseguiremo il passo (1), seguito dal passo (2), seguito dal passo (1), seguito dal passo (2), e così via.

Per via del modo in cui nel passo (1) viene usato *contatore-precedente*, possiamo sostituire nel passo (1) l'espressione di *contatore-precedente* del passo (2) ottenendo l'espressione più semplice:

Il *contatore-corrente* a secondo membro dell'espressione assume il ruolo di *contatore-precedente*. Poiché questo enunciato indica un'assegnazione e non un'uguaglianza (che sarebbe impossibile) è un'istruzione valida per il computer. Questa descrive il fatto che il *nuovo valore di contatore-corrente* si ottiene aggiungendo 1 al vecchio valore di *contatore-corrente*.

Analizzando il meccanismo in questo modo, appare chiaro che l'esistenza della variabile *contatore-precedente* è del tutto non necessaria. Ne risulta un meccanismo di conteggio più semplice. Pertanto i passi essenziali del nostro algoritmo di conteggio dei promossi possono essere così riassunti: fintanto che si sono esaminati meno di n voti

- (a) leggere il prossimo voto
 - (b) se il voto corrente soddisfa il requisito della promozione aggiungere uno al contatore.

Prima che venga esaminato un qualsiasi voto, il contatore deve avere valore zero. Per completare l'algoritmo occorre includere l'introduzione dei voti e la stampa del numero dei promossi. In dettaglio l'algoritmo è descritto qui di seguito.

Descrizione dell'algoritmo

- Richiedere e leggere il numero dei voti da elaborare.
 - Inizializzare con zero il contatore.
 - Finché ci sono ancora voti da elaborare, ripetere quanto segue:
 - leggere il prossimo voto,
 - se è da promozione (cioè ≥ 50), aggiungere uno al contatore.
 - Stampare il numero totale di promossi.

Implementazione in Pascal

```

program passcount (input, output);
const passmark = 50;
var count {contains number of passes on termination},
    i {current number of marks processed},
    m {current mark},
    n {total number of marks to be processed}: integer;
begin {count the number of passes (>=50) in a set of marks}
  writeln ('enter a number of marks n on a separate line followed by the

```

```

marks');
readln (n);
{assert:  $n \geq 0$ }
count := 0;
i := 0;
{invariant: count = number of marks in the first i read that
are  $\geq$  passmark  $\wedge$   $i \leq n$ }
while  $i < n$  do
  begin {read next mark, test it for pass and update count if necessary}
    i := i + 1;
    read (m);
    if eoln (input) then readln;
    if  $m \geq$  passmark then count := count + 1
  end;
{assert: count = number of passes in the set of n marks read}
writeln ('number of passes = ', count)
end.

```

Note di progetto

1. Inizialmente, e ad ogni passaggio nel corso del loop, la variabile *count* rappresenta il numero di promossi incontrati fino a quel momento. Al termine (quando $i = n$) *count* rappresenta il numero totale di promossi. Poiché i è incrementato di 1 ad ogni iterazione, la condizione $i < n$ verrà alla fine violata e il loop avrà termine.
 2. È stata fatta qualche modifica per migliorare la soluzione originale del problema. La soluzione più semplice e più efficiente di un problema è di norma quella più versatile.
 3. Si è inserito un test di *end-of-line* per trattare più linee di dati.

Applicazioni

Tutti i tipi di conteggio.

Problemi supplementari

- 2.2.1 Modificare l'algoritmo precedente in modo che i voti siano letti fino all'incontro di un *end-of-file*. Per questo insieme di voti determinare il numero totale dei voti, il numero dei promossi e la percentuale di questi sul totale.
 - 2.2.2 Progettare un algoritmo che legge una lista di numeri e conta il numero dei termini negativi e di quelli positivi o nulli.
 - 2.2.3 Il problema è quello già descritto. Tuttavia, assumiamo che il programma parta con la variabile *count* inizializzata al valore *n*

del numero totale di studenti. Al termine, il contatore rappresenta ancora il numero di studenti che hanno superato l'esame. Se ci sono più promossi che bocciati, perché questa implementazione sarebbe migliore di quella originale?

ALGORITMO 2.3 SOMMATORIA DI UN INSIEME DI NUMERI

Problema

Dato un insieme di n numeri progettare un algoritmo che li sommi e restituisca la somma risultante. Si assuma n maggiore o uguale a zero.

Sviluppo dell'algoritmo

La somma di un insieme di n numeri è tra le cose più probabili che accada di fare con un computer. Quando affrontiamo questo problema in assenza del computer, scriviamo semplicemente i numeri l'uno sotto l'altro e sommiamo partendo dalla colonna di destra. Per esempio, ecco l'addizione di 421, 583 e 714.

$$\begin{array}{r} 421 \\ 583 \\ 714 \\ \hline \dots 8 \end{array}$$

Nel progettare un algoritmo che compia lo stesso lavoro dobbiamo usare un approccio alquanto diverso. Il computer ha un proprio dispositivo interno che riceve due numeri da sommare, esegue l'addizione e restituisce la somma dei due numeri (v. Fig. 2.1).

Nel progettare un algoritmo che sommi un insieme di numeri, il *meccanismo* per l'operazione di *addizione* costituisce la questione principale. Ci concentreremo per intanto su questo aspetto del problema, prima di pensare al progetto globale.

Il modo più semplice per insegnare all'unità aritmetica del computer la somma di un insieme di numeri è scrivere un'espressione che specifichi l'addizione da compiere. Per i tre numeri di prima potremo scrivere:

$$s := 421 + 583 + 714 \quad (1)$$

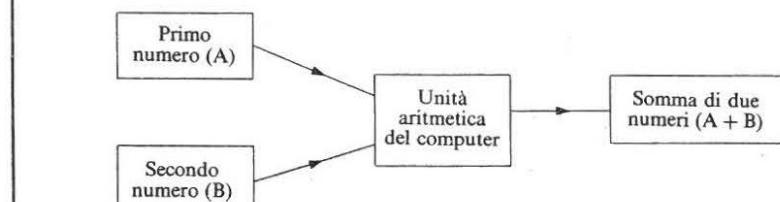


Fig. 2.1.
*Mecanismo
schematico
dell'operazione di
addizione mediante
computer.*

L'operatore di assegnazione fa sì che il risultato del calcolo a secondo membro dell'enunciato (1) venga posto nella cella di memoria assegnata alla variabile s .

L'espressione (1) sommerà come richiesto tre numeri *specifici*. Sfortunatamente non è in grado di far altro: volendo sommare altri tre numeri, avremo bisogno di una nuova istruzione di programma.

Sembra perciò ragionevole sostituire con variabili tutte le costanti dell'espressione (1), ottenendo così:

$$s := a + b + c \quad (2)$$

L'espressione (2) somma tre numeri *qualsiasi* purché essi siano stati assegnati preventivamente come valori di a , b e c rispettivamente. L'espressione (2) come base di un programma per sommare dei numeri è più generale e più utile dell'espressione (1). Ha ancora un serio difetto: può sommare solo insieme di tre numeri.

Progettando algoritmi e implementando programmi, un obiettivo fondamentale è quello di rendere i programmi generali al punto da trattare con successo un'ampia varietà di condizioni d'ingresso. Più specificamente, vogliamo un programma che sommi n numeri *qualsiasi*, con n che può assumere un ampio spettro di valori.

L'approccio richiesto per formulare un algoritmo che sommi n numeri in un computer è diverso da quello con cui affronteremmo convenzionalmente lo stesso problema. Convenzionalmente potremmo scrivere l'equazione generale:

$$s = (a_1 + a_2 + a_3 + \dots + a_n) \quad (3)$$

o, in maniera equivalente:

$$s = \sum_{i=1}^n a_i \quad (4)$$

Potremmo scrivere anche per il computer un'istruzione in qualche modo simile all'equazione (3) per sommare gli n numeri, ma ciò si rivelerebbe inattuabile poiché vogliamo essere in grado di modificare n (potremmo voler usare il programma per insiemi numerici di diversa estensione). Dobbiamo perciò cercare un meccanismo migliore, che tenga maggiormente conto di come il computer è naturalmente progettato a compiere le cose. Alcuni comportamenti del computer sono attinenti al nostro problema attuale: primo, i computer sono particolarmente adatti a compiti ripetitivi; secondo, il dispositivo di addizione dei computer è progettato in modo che possa sommare solo due numeri per volta. Ci chiediamo perciò come formulare un algoritmo per la somma di n numeri che faccia il miglior uso di questi fatti.

Per tener conto del fatto che il computer somma due numeri per volta, partiamo sommando i primi due numeri a_1 e a_2 . Cioè:

$$s := a_1 + a_2 \quad (1)$$

Possiamo quindi procedere aggiungendo a_3 a s calcolata nel passo (1)

$$s := s + a_3 \quad (2)$$

(cfr. l'istruzione di conteggio dell'algoritmo 2.2).

In modo analogo:

$$\left. \begin{array}{l} s := s + a_4 \\ s := s + a_5 \\ \vdots \\ s := s + a_n \end{array} \right\} \quad (3, \dots, n-1)$$

Dal passo (2) in poi stiamo ripetendo più e più volte lo stesso procedimento: la sola differenza è che i valori di a e di s cambiano ad ogni passo. Per il generico passo i -esimo abbiamo:

$$s := s + a_{i+1} \quad (i)$$

Questo passo generico può essere immesso in un loop per generare iterativamente la somma di n numeri.

L'algoritmo che vogliamo sviluppare per la somma di n numeri dovrebbe funzionare correttamente per tutti i valori di n maggiori o uguali a zero (ossia, $n \geq 0$). Pertanto deve operare correttamente per la somma di zero ($n = 0$) e la somma di uno ($n = 1$) elementi. Ciò significa che il passo (1) che abbiamo adoperato è inadatto. Tuttavia, se nel generico passo (i) sostituiamo $i + 1$ con i e imponiamo $i = 1$ otteniamo:

$$s := s + a_1 \quad (1')$$

Il passo (1') è corretto purché prima di questo passo si ponga $s = 0$. Ne consegue che tutte le somme per $n \geq 1$ possono essere prodotte iterativamente.

Il caso in cui $n = 0$ è speciale e non può essere generato iterativamente; la somma di zero numeri è zero e quindi possiamo produrla direttamente con l'assegnazione:

$$s := 0$$

Il nucleo dell'algoritmo per sommare n numeri comprende pertanto un passo speciale seguito dall'insieme di n passi iterativi. Ovvero:

1. Calcolare la prima somma ($s = 0$) come caso speciale.
2. Formare ciascuna delle rimanenti n somme partendo dalla precedente, con un processo iterativo.
3. Scrivere la somma degli n numeri.

Le uniche altre considerazioni comprendono l'introduzione di n , numero di elementi da sommare, e la lettura successiva dei numeri ad ogni passo dell'iterazione. L'algoritmo completo può essere ora descritto.

Descrizione dell'algoritmo

1. Richiedere e leggere il numero degli elementi da sommare.
2. Inizializzare la somma per zero numeri.
3. Finché si sono sommati meno di n numeri ripetere quanto segue:
 - (a) leggere il prossimo numero;
 - (b) calcolare la somma corrente aggiungendo il numero letto alla somma più recente.
4. Stampare la somma degli n numeri.

Implementazione in Pascal

```
program sum (input, output);
var i {summing loop index},
n {number of numbers to be summed}: integer;
a {current number to be summed},
s {sum of n numbers on termination}: real;
begin {computes sum of n real numbers for n >= 0}
```

```

writeln ('input n on a separate line, followed by the numbers to be
summed');
readln (n);
{assert: n>=0}
i := 0;
s := 0.0;
{invariant: s = sum of first i numbers read ∧ i <= n}
while i < n do
begin {calculate successive partial sums}
  i := i+1;
  read (a);
  if eoln (input) then readln;
  s := s+a;
end;
{assert: s = sum of n numbers read}
writeln ('sum of n = ', n, ' numbers = ', s)
end.

```

Note di progetto

- Per sommare n numeri occorre eseguire $(n-1)$ addizioni. Il presente algoritmo ha invece usato n addizioni per consentire un'implementazione più semplice e pulita.
- Inizialmente, e ad ogni passaggio nel corso del loop, la somma s rappresenta la somma dei primi i numeri letti. Al termine (quando $i = n$) s rappresenta la somma di n numeri. Poiché i è incrementato di 1 ad ogni iterazione, la condizione $i < n$ verrà alla fine violata e il loop avrà termine.
- Lo schema utilizzato non tiene conto dell'accuratezza del risultato né della dimensione finita dei numeri che possono essere rappresentati precisamente nel computer. Un algoritmo che minimizza gli errori della sommatoria lo fa commando, ad ogni passo, i due numeri più piccoli che rimangono.
- Si osservi che solo algoritmi di uso generale andrebbero implementati come programmi. Un programma parametrizzato per risolvere un solo problema di solito è fatica sprecata.
- La soluzione ovvia o diretta del problema è notevolmente diversa dalla soluzione per il computer. L'esigenza della flessibilità impone questa differenza alla soluzione per il computer.
- L'esame del problema al suo limite inferiore (ossia $n = 0$) conduce a un meccanismo che può essere esteso a valori di n più grandi mediante semplice ripetizione. Questo è un accorgimento assai diffuso nella progettazione di algoritmi.
- Un programma che legge e somma n numeri non è un gran che come strumento di programmazione. Un'implementazione molto più pratica è una funzione che restituisce la somma di un array di n numeri. Ossia:

```

function asum(var a:nelements;n:integer):real;
var sum {the partial sum},
  i:integer;
begin {compute the sum of n real array elements (n≥0)}
  sum := 0.0;
  for i := 1 to n do
    sum := sum+a[i];
  asum := sum
end

```

Applicazioni

Calcolo di medie, varianze e minimi quadrati.

Problemi supplementari

- Progettare un algoritmo per calcolare la media di n numeri.
- Riprogettare l'algoritmo 2.3 in modo che per sommare n numeri richieda solo $n - 1$ addizioni.
- Progettare un algoritmo per calcolare la somma dei quadrati di n numeri. Ossia:

$$s = \sum_{i=1}^n (a_i)^2$$

- La media armonica definita come:

$$H = \frac{n}{\sum_{i=1}^n (1/a_i)}$$

è spesso usata come indice di tendenza centrale.
Sviluppare un algoritmo per il calcolo della media armonica di n valori dati.

- Sviluppare un algoritmo per calcolare la somma dei primi n termini ($n \geq 0$) delle seguenti serie:

- (a) $s = 1 + 2 + 3 + \dots$
- (b) $s = 1 + 3 + 5 + \dots$
- (c) $s = 2 + 4 + 6 + \dots$
- (d) $s = 1 + 1/2 + 1/3 + \dots$

2.3.6 Generare i primi n termini della successione:

$$1 \ 2 \ 4 \ 8 \ 16 \ 32 \dots$$

senza adoperare la moltiplicazione.

2.3.7 Sviluppare un algoritmo che stampa n valori della successione:

$$1 - 1 \quad 1 - 1 \quad 1 - 1 \dots$$

2.3.8 Sviluppare un algoritmo per calcolare la somma dei primi n termini ($n \geq 1$) della serie:

$$s = 1 - 3 + 5 - 7 + 9 - \dots$$

ALGORITMO 2.4 CALCOLO DEL FATTORIALE

Problema

Dato il numero n , calcolare n fattoriale (scritto come $n!$) con $n \geq 0$.

Sviluppo dell'algoritmo

Possiamo iniziare lo sviluppo di questo algoritmo esaminando la definizione di $n!$. Sappiamo che:

$$n! = 1 * 2 * 3 * \dots * (n-1) * n \quad \text{per } n \geq 1$$

e per definizione:

$$0! = 1$$

Formulando il nostro schema per questo problema, dobbiamo ricordare che l'unità aritmetica del computer può moltiplicare soltanto *due* numeri per volta.

Applicando la definizione di fattoriale abbiamo:

$$\begin{aligned} 0! &= 1 \\ 1! &= 1 * 1 \\ 2! &= 1 * 2 \\ 3! &= 1 * 2 * 3 \\ 4! &= 1 * 2 * 3 * 4 \\ \dots & \end{aligned}$$

Vediamo che $4!$ contiene *tutti* i fattori di $3!$; la sola differenza è l'inclusione del numero 4. Possiamo generalizzare questa osservazione dicendo che $n!$ può sempre ottenersi da $(n-1)!$ semplicemente moltiplicando questo per n (per $n \geq 1$). Cioè:

$$n! = n * (n-1)! \quad \text{per } n \geq 1$$

Con questa definizione possiamo scrivere i primi fattoriali come:

$$\begin{aligned} 1! &= 1 * 0! \\ 2! &= 2 * 1! \\ 3! &= 3 * 2! \\ 4! &= 4 * 3! \\ \dots & \end{aligned}$$

Se partiamo con $p = 0! = 1$ possiamo riscrivere i primi passi del calcolo di $n!$ come:

$$\begin{array}{lll} p := 1 & (1) & = 0! \\ p := p * 1 & & = 1! \\ p := p * 2 & & = 2! \\ p := p * 3 & (2 \dots n+1) & = 3! \\ p := p * 4 & & = 4! \\ \dots & & \dots \end{array}$$

Dal passo (2) in poi ripetiamo più e più volte lo stesso procedimento. Per il generico passo $(i+1)$ -esimo abbiamo:

$$p := p * i \quad (i+1)$$

Questo passo generico può essere inserito in un loop per produrre iterativamente $n!$. Ciò ci consente di approfittare del fatto che l'unità aritmetica del computer può moltiplicare solo due numeri per volta.

Sotto molti aspetti questo problema è simile a quello della somma di un insieme di n numeri (algoritmo 2.3). Nel problema della sommatoria eseguiamo un insieme di addizioni, mentre in questo problema dobbiamo generare un insieme di prodotti. Dal generico passo $(i+1)$ -esimo segue che tutti i fattoriali per $n \geq 1$ possono essere generati per via iterativa. La situazione $n=0$ è un caso speciale di cui si può tener conto direttamente con l'assegnazione:

$$p := 1 \quad (\text{per definizione di } 0!)$$

La parte centrale dell'algoritmo che calcola $n!$ comprende perciò un passo speciale iniziale seguito da n passi iterativi.

1. Trattare $0!$ come caso speciale ($p := 1$).
2. Calcolare ciascuno degli n prodotti p rimanenti a partire dal precedente, con un procedimento iterativo.
3. Scrivere il valore di n fattoriale.

Descrizione dell'algoritmo

1. Stabilire n (con $n \geq 0$) di cui si vuole il fattoriale.
2. Fissare il prodotto p a $0!$ (caso speciale). Azzerare anche il contatore dei prodotti.
3. Finché sono stati calcolati meno di n prodotti, ripetere quanto segue:
 - (a) incrementare il contatore dei prodotti,
 - (b) calcolare l' i -esimo prodotto p moltiplicando i per il prodotto più recente.
4. Restituire il risultato $n!$.

Questo algoritmo è implementato più convenientemente come funzione che accetta in ingresso un numero n e restituisce in uscita il valore $n!$. Nell'implementazione in Pascal p è stato sostituito dalla variabile *factor*.

Implementazione in Pascal

```
function nfactorial (n:integer):integer;
var i {loop index representing ith factorial}:integer;
    factor {i!}:integer;

begin {computes and returns n! for n >= 0}
  {assert:n >= 0}
  factor := 1;
  {invariant : factor = i! after the ith iteration ∧ i <= n}
  for i := to n do
    factor := i * factor;
  nfactorial := factor
  {assert: nfactorial = n!}
end
```

Note di progetto

1. L'algoritmo impiega n moltiplicazioni per calcolare $n!$ In effetti esiste un algoritmo più efficiente (v. nota 5) che calcola $n!$ essenzialmente in $\log n$ passi.
2. Dopo l' i -esimo passaggio attraverso il loop, il valore di *factor* è $i!$. Tale condizione vale per qualsiasi i . Al termine (quando $i = n$) *factor* corrisponde a $n!$. Poiché i è incrementato di 1 ad ogni iterazione, la condizione $i = n$ verrà alla fine raggiunta e il loop avrà termine. Tuttavia non viene fatta alcuna considerazione circa la dimensione finita dei numeri che possono essere rappresentati nel computer.
3. Un'accurata definizione del problema e l'esame di qualche esempio specifico sono fondamentali per lo sviluppo dell'algoritmo. Il problema più semplice conduce a una generalizzazione che costituisce la base dell'algoritmo.
4. L'idea di accumulare prodotti è molto simile a quella di accumulare somme (v. algoritmo 2.3).

- 5 Un meccanismo più efficiente si basa sul fatto che si può esprimere $n!$ in funzione di $(n/2)!$. In linea di principio, il meccanismo è analogo a quello usato per calcolare l' n -esimo numero di Fibonacci (v. A. Shamir, Factoring numbers in $O(\log n)$ arithmetic steps', *Inf. Proc. Letts.*, 8, 28-31, 1979). Vedi anche algoritmo 3.8.

Applicazioni

Calcoli probabilistici, statistici e matematici.

Problemi supplementari

- 2.4.1 Per un dato n , progettare un algoritmo che calcola $1/n!$.
- 2.4.2 Per dati x e n , progettare un algoritmo che calcola $x^n/n!$.
- 2.4.3 Progettare un algoritmo che determina se un numero n è un fattoriale o no.
- 2.4.4 Progettare un algoritmo che, assegnato un certo n intero, determina il massimo fattoriale presente come divisore di n .
- 2.4.5 Progettare un algoritmo che simuli la moltiplicazione mediante l'addizione. Il programma dovrà accettare in ingresso due interi (positivi, negativi o nulli).
- 2.4.6 In algebra il coefficiente binomiale " C_r ", ossia il coefficiente dell' r -esima potenza di x nello sviluppo di $(1+x)^n$ è dato da:

$$C_r = \frac{n!}{r!(n-r)!}$$

Progettare un algoritmo che calcoli tutti i coefficienti di x per un dato valore di n .

ALGORITMO 2.5 CALCOLO DELLA FUNZIONE SENO

Problema

Progettare un algoritmo per calcolare la funzione $\sin(x)$, definita dallo sviluppo in serie:

$$\sin(x) = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Sviluppo dell'algoritmo

Questo problema impiega alcune delle tecniche che abbiamo visto in precedenti algoritmi. Analizzando l'espressione di $\sin(x)$ vediamo che sono richiesti potenze e fattoriali secondo la successione:

$$1, 3, 5, 7, \dots$$

Possiamo generare facilmente tale successione partendo da 1 e sommando ripetutamente 2. Un altro problema è il calcolo del generico termine $x^i/i!$ che può essere espresso come

$$\frac{x^i}{i} = \frac{x}{1} * \frac{x}{2} * \frac{x}{3} * \dots * \frac{x}{i} \quad \text{per } i \geq 1$$

La funzione atta a calcolarlo comprenderà i passi seguenti:

```
fp := 1;
j := 0;
while j < i do
begin
  j := j + 1;
  fp := fp * x / j
end
```

L'algoritmo può essere completato implementando le addizioni e sottrazioni e le condizioni appropriate di terminazione. Con questo approccio, si calcola *ciascun* termine $x^i/i!$ procedendo progressivamente a partire dal valore più piccolo di i . Per calcolare $n!$ abbiamo usato l'efficace approccio di calcolare ciascun termine partendo dal suo predecessore; ci domandiamo: è possibile fare altrettanto col problema presente? Per esplorare tale possibilità possiamo esaminare il sovrapporsi di specifici termini della serie. Per esempio:

$$\frac{x^3}{3!} - \frac{x^2 * x}{3 * 2 * 1} = \frac{x^2}{3 * 2} \frac{x^1}{1!} \quad i = 3$$

$$\frac{x^5}{5!} = \frac{x^4 * x^3 * x^2 * x}{5 * 4 * 3 * 2 * 1} = \frac{x^2}{5 * 4} \frac{x^3}{3!} \quad i = 5$$

$$\frac{x^7}{7!} = \frac{x^6 * x^5 * x^4 * x^3 * x^2}{7 * 6 * 5 * 4 * 3 * 2 * 1} = \frac{x^2}{7 * 6} \frac{x^5}{5!} \quad i = 7$$

Ognuno dei termini $x^2/(3*2)$, $x^2/(5*4)$, $x^2/(7*6)$, ... può essere rappresentato dal termine generico:

$$\frac{x^2}{i * (i - 1)} \quad \text{per } i = 3, 5, 7, \dots$$

Pertanto per generare i termini consecutivi della serie seno potremo usare:

$$\text{termine corrente } i\text{-esimo} = \frac{x^2}{i(i-1)} * (\text{termine precedente})$$

Per far sì che i termini cambino segno alternativamente possiamo usare l'accorgimento adoperato nel problema 2.3.8. L'esecuzione ripetuta di

$$\text{sign} := -\text{sign}$$

produrrà termini positivi e negativi alternati. Si può incorporare questo meccanismo direttamente nell'espressione dei termini; pertanto le condizioni iniziali sono:

$$\left. \begin{array}{l} t \sin = x; \\ \text{term} := x; \\ i := 1 \end{array} \right\}$$

Il termine i -esimo e la sommatoria che possono essere generati iterativamente dai rispettivi predecessori sono:

$$\begin{aligned} i &:= i + 2; \\ \text{term} &:= -\text{term} * x * x / (i * (i - 1)); \\ t \sin &= t \sin + \text{term} \end{aligned}$$

Ora abbiamo un efficace meccanismo iterativo per produrre i termini successivi della funzione seno. Mancano solo le considerazioni su come terminare l'algoritmo.

Evidentemente possiamo calcolare $\sin(x)$ solo per un numero finito di termini; è importante a questo punto considerare la precisione richiesta per $\sin(x)$. Poiché x è limitato all'intervallo $-1 \leq x \leq 1$, osserviamo che il contributo dei termini più elevati diventa rapidamente piccolissimo. Per esempio il quarto termine (ossia $x^7/7!$) fornisce un contributo inferiore a 0.0002. In situazioni di questo tipo una maniera utile di impostare la terminazione è fissare un livello d'errore ammissibile (p. es. $1 \cdot 10^{-6}$) e generare termini in successione fino a che il contributo del termine corrente non diventi *inferiore* all'errore ammesso. Un'analisi dettagliata degli errori conferma che questa è una condizione di terminazione accettabile; poiché i termini cambiano segno alternativamente, per il test di terminazione dovremo usare il valore assoluto nel confronto con l'errore.

La strategia globale per l'algoritmo che calcola la funzione seno può riassumersi come segue.

Descrizione dell'algoritmo

1. Predisporre le condizioni iniziali per il primo termine, che non può essere calcolato iterativamente.
2. Finché il valore assoluto del termine corrente si mantiene superiore all'errore ammissibile, eseguire quanto segue:
 - (a) identificare il termine i -esimo corrente,
 - (b) generare il termine corrente dal suo predecessore,
 - (c) sommare il termine corrente, con il segno appropriato, all'accumulatore che approssima il valore della funzione seno.

Poiché l'espressione del seno interessa il calcolo di un valore isolato è meglio implementata come funzione.

Implementazione in Pascal

```

function sin (x: real): real;
const error = 1.0e-6;
var i {variable to generate sequence 1 3 5 7}: integer;
    x2 {x squared},
    term {current sum of terms - eventually approximates sin}: real;

begin {function returns sin (x) with an accuracy of =<error}
  {assert:-1.0=<x=<1.0}
  term := x;
  tsin := x;
  i := 1;
  x2 := x * x;
  {invariant: after the jth iteration, i=2j+1 ∧ term = (-1)↑j * (x↑i) / i!
   ∧ tsin is sum of first (j+1) terms}
  while abs (term)>error do
    begin {generate and accumulate successive terms of sine
           expression}
      i := i + 2;
      term := -term * x2 / (i * (i-1));
      tsin := tsin + term
    end;
  sin := tsin
  {assert: sin=sine (x) ∧ abs (term)=<error}
end

```

Note di progetto

1. Il numero di iterazioni cicliche di questo algoritmo è funzione delle esigenze di precisione. Per un errore inferiore a 10^{-6} occorre calcolare solo 5 termini; questa precisione è sufficiente nella maggior parte delle applicazioni pratiche. Il costo per produrre i termini successivi della serie è costante; ciò è più favorevole del progetto originale, ove il costo di generazione dei termini succes-

sivi era proporzionale alla posizione del termine nella serie (ossia, termini più lontani richiedevano calcoli più complessi).

2. Prima e durante il processo iterativo, al passo j -esimo (j non è definito esplicitamente) $term$ rappresenta il termine j -esimo dello sviluppo in serie del seno e $tsin$ continua la sommatoria con i segni appropriati dei primi j termini dello sviluppo in serie. I limiti invarianti per le variabili $term$ e $tsin$ sono validi per ogni $j \geq 1$. Possiamo esser certi della terminazione dell'algoritmo poiché per l'intervallo di x ammesso (cioè $-1 \leq x \leq 1$) il valore assoluto di $term$ è una funzione *strettamente decrescente*. Pertanto alla fine diverrà inferiore al livello d'errore stabilito. In questa discussione non abbiamo considerato il contributo degli errori di arrotondamento.
3. Il numero delle moltiplicazioni è ridotto precalcolando il termine x^2 .
4. Questo algoritmo include lo stesso meccanismo di base che è applicato negli algoritmi del fattoriale e della sommatoria. La sola differenza riguarda la scelta delle condizioni iniziali, dei termini iterativi e delle condizioni di terminazione.
5. Quando è applicabile, la generazione di ciascun termine a partire dal suo predecessore conduce all'implementazione più semplice ed efficiente. Perciò è di norma conveniente compiere qualche sforzo in più per vedere se i termini possono essere generati dai loro predecessori. A questo scopo sono idonei esempi specifici.
6. Si osservi in modo particolare come si è ottenuto l'effetto di alternanza dei segni ed anche come si è imposta la terminazione.

Applicazioni

Calcoli matematici e statistici.

Problemi supplementari

- 2.5.1 Progettare un algoritmo per trovare la somma dei primi n termini della serie:

$$f_s = 0! + 1! + 2! + 3! + \dots + n! \quad (n \geq 0)$$

- 2.5.2 La costante matematica e è definita dall'espressione:

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \dots$$

Escogitare un algoritmo che calcoli e con n termini.

- 2.5.3 Progettare un algoritmo per calcolare la funzione $\cos(x)$, definita dallo sviluppo in serie:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \dots$$

L'errore ammissibile per il calcolo è 10^{-6} .

ALGORITMO 2.6 GENERAZIONE DI UNA SUCCESSIONE DI FIBONACCI

Problema

Generare e stampare i primi n termini della successione di Fibonacci, con $n \geq 1$. I primi termini sono:

0, 1, 1, 2, 3, 5, 8, 13, ...

Ciascun termine dopo i primi due è ottenuto sommando i suoi due predecessori più vicini.

Sviluppo dell'algoritmo

Dalla definizione abbiamo:

termine nuovo = termine precedente + termine prima del precedente

L'ultima frase della formulazione del problema suggerisce la possibilità di usare la definizione per generare iterativamente (tranne i primi due) i termini successivi.

Definiamo:

- a* il termine prima del precedente
- b* il termine precedente
- c* il termine nuovo

Pertanto per partire avremo:

- a* := 0 primo numero di Fibonacci
- b* := 1 secondo numero di Fibonacci
- c* := *a* + *b* terzo numero di Fibonacci (per definizione).

Quando il termine *c* è stato generato abbiamo il terzo numero di Fibonacci. Per generare il quarto numero, o i successivi, occorre applicare di nuovo la stessa definizione; ma prima di poterlo fare occorre ancora qualche aggiustamento. Il quarto numero di Fibonacci deriva dalla somma del secondo e del terzo: con riferimento alla definizione, il secondo numero di Fibonacci gioca il ruolo di *termine prima del precedente* ed il terzo numero il ruolo di *termine precedente*. Pertanto, prima di eseguire il prossimo calcolo (ossia il quarto) dobbiamo assicurarsi che:

- (a) il termine nuovo (ossia il terzo) assuma il ruolo di termine precedente,
- (b) quello che è correntemente il termine precedente assuma il ruolo di termine prima del precedente.

Così:

<i>a</i> := 0;	[1]	termine prima del precedente
<i>b</i> := 1;	[2]	termine precedente
<i>c</i> := <i>a</i> + <i>b</i> ;	[3]	termine nuovo
<i>a</i> := <i>b</i> ;	[4]	il termine prima del precedente diventa termine precedente
<i>b</i> := <i>c</i>	[5]	il termine precedente diventa termine nuovo.

Dopo aver completato il passo [5] siamo nella condizione di poter generare il prossimo numero di Fibonacci usando la definizione. Un modo per farlo è tornare con l'esecuzione al passo [3]. L'esame ulteriore dei passi da [3] a [5] mostra che essi possono essere collocati in un ciclo per generare iterativamente i numeri di Fibonacci per $n > 2$. Il meccanismo che potremmo usare è pertanto:

```

a := 0;
b := 1;
i := 2;
while i < n do
begin
  i := i + 1;
  c := a + b;
  a := b;
  b := c
end

```

Prima di concludere questa discussione ci domandiamo se sia possibile qualche miglioramento al nostro algoritmo. Col meccanismo attuale, dopo il calcolo di un nuovo termine *c*, il termine prima del precedente, *a*, perde di significato per il calcolo del prossimo numero di

Fibonacci. Per restituirgli validità, abbiamo fatto l'assegnazione:

$$a := b$$

Sappiamo che *tutte* le volte per generare il numero di Fibonacci successivo servono solo due numeri; tuttavia, nel nostro calcolo, abbiamo introdotto una terza variabile, *c*. Perciò quello che tenteremo di fare è mantenere sempre significative le due variabili *a* e *b*. Poiché il primo numero di Fibonacci diventa irrilevante non appena si calcola il terzo, possiamo partire con:

$$\begin{aligned} a &:= 0; & [1] \\ b &:= 1; & [2] \\ a &:= a + b & [3] \end{aligned}$$

(così *a* conserva validità per il prossimo numero di Fibonacci)

Se iteriamo sul passo [3] per generare i successivi numeri di Fibonacci, finiamo nei pasticci poiché *b* non è mai modificato. Tuttavia, dopo il passo [3] sappiamo che il prossimo numero di Fibonacci (il quarto) si può ottenere mediante

$$next := a + b \quad [4] \quad (\text{quarto numero di Fibonacci})$$

Ci domandiamo dov'è l'errore del nostro ragionamento. Eseguendo il passo [4], il vecchio valore di *b*, definito nel passo [2], perde importanza. Per conservare validità a *b* nel passo [4] dobbiamo eseguire l'assegnazione:

$$b := a + b$$

I primi quattro passi diventano allora:

$$\begin{aligned} a &:= 0; & [1] \\ b &:= 1; & [2] \\ a &:= a + b; & [3] \\ b &:= a + b & [4] \end{aligned}$$

Dopo il passo [4] ci accorgiamo che i valori di *a* e *b* sono quelli corretti per il calcolo del quinto numero di Fibonacci; a questo punto diventa irrilevante il vecchio valore di *a*. Procedendo per qualche passo ancora, otteniamo la conferma che possiamo iterare sui passi [3] e [4] per generare correttamente la successione richiesta. Poiché l'algoritmo genera a coppie i numeri di Fibonacci, occorre far attenzione a scrivere esattamente *n* numeri; il modo migliore per farlo è mantenere la stampa un passo indietro rispetto alla fase di generazione.

Adesso si può dare una completa descrizione dell'algoritmo.

Descrizione dell'algoritmo

1. Richiedere e leggere *n*, numero dei numeri di Fibonacci da calcolare.
2. Assegnare i primi due numeri di Fibonacci *a* e *b*.
3. Inizializzare il contatore dei numeri generati.
4. Finché sono stati prodotti meno di *n* numeri di Fibonacci eseguire quanto segue:
 - (a) scrivere i due ultimi numeri di Fibonacci;
 - (b) generare il prossimo numero di Fibonacci, conservando *a* significativo;
 - (c) generare il prossimo numero di Fibonacci a partire dalla coppia più recente, conservando *b* significativo per il prossimo calcolo;
 - (d) aggiornare il contatore *i* dei numeri di Fibonacci prodotti.
5. Se *n* è pari scrivere gli ultimi due numeri di Fibonacci, altrimenti scrivere soltanto il penultimo.

Implementazione in Pascal

```
program Fibonacci (input, output);
var a {Fibonacci number variable},
    b {Fibonacci number variable},
    i {number of Fibonacci numbers generated},
    n {number of Fibonacci numbers to be generated}: integer;
begin {generate each Fibonacci number from the sum of its two
predecessors}
  a := 0;
  b := 1;
  i := 2;
  writeln ('enter n the number of Fibonacci numbers to be
generated');
  readln (n);
  {assert: n > 0}
  {invariant: after jth iteration i = 2j + 2 ^ first i Fib. numbers have
been generated ^ a = (i-1)th Fib. no. ^ b = ith Fib. no.
  ^ i = < n + 1}
  while i < n do
    begin
      writeln (a, b);
      a := a + b;
      b := a + b;
      i := i + 2
    end;
  if i = n then writeln (a, b) else writeln (a)
  {assert: first n Fibonacci numbers generated and written out}
end.
```

Note di progetto

1. Per generare n numeri di Fibonacci sono richiesti essenzialmente n passi. L'algoritmo funziona correttamente per tutti i valori di $n \geq 1$.
2. Durante il calcolo le variabili a e b contengono sempre i due numeri di Fibonacci generati più di recente. Perciò, ogni volta che viene eseguita un'addizione per generare il numero di Fibonacci successivo, sono sempre soddisfatte le regole che definiscono i numeri di Fibonacci. Poiché i viene incrementato ad ogni iterazione, la condizione $i < n$ verrà alla fine violata e l'algoritmo avrà termine.
3. Il secondo algoritmo è più efficiente del primo poiché esegue solo un'assegnazione per ogni numero di Fibonacci generato. Il primo algoritmo compie tre assegnazioni per ogni numero di Fibonacci calcolato.
4. Con il presente algoritmo i numeri di Fibonacci vengono generati a coppie; quando n è dispari viene prodotto un numero in più di quanto richiesto, che tuttavia non verrà stampato. Questo è un piccolo sacrificio in cambio degli altri vantaggi.
5. Nell'algoritmo (3.8) verrà discusso un algoritmo più avanzato che può calcolare l' n -esimo numero di Fibonacci in $\log n$ passi.

Applicazioni

La successione di Fibonacci ha applicazioni pratiche in botanica, teoria delle reti elettriche, ordinamenti e ricerche.

Problemi supplementari

- 2.6.1 Implementare l'algoritmo di Fibonacci come funzione che riceve in ingresso due numeri consecutivi di Fibonacci e restituisce in output il successivo numero di Fibonacci.
- 2.6.2 I primi numeri della successione di Lucas, che è una variazione della successione di Fibonacci, sono:

1 3 4 7 11 18 29 ...

Progettare un algoritmo che generi la successione di Lucas.

- 2.6.3 Siano $a = 0$, $b = 1$ e $c = 1$ i primi tre numeri di una data successione. Tutti gli altri numeri della sequenza sono generati a partire dalla somma dei loro tre predecessori più recenti. Progettare un algoritmo per generare questa successione.
- 2.6.4 Dati due numeri d ed e , possibili di essere elementi contigui della successione di Fibonacci, progettare un algoritmo che confermi o smentisca tale congettura.

- 2.6.5 La successione ascendente di tutte le frazioni ridotte ai minimi termini comprese tra 0 e 1 aventi denominatori $\leq n$ è detta serie di Farey di ordine n . La serie di Farey di ordine 5 è:

$$\frac{0}{1}, \frac{1}{5}, \frac{1}{4}, \frac{1}{3}, \frac{2}{5}, \frac{1}{2}, \frac{3}{5}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \frac{1}{1}$$

Indicando tale serie con:

$$\frac{x_0}{y_0}, \frac{x_1}{y_1}, \frac{x_2}{y_2}, \dots$$

si può dimostrare che:

$$x_0 = 0 \quad y_0 = 1 \quad x_1 = 1 \quad y_1 = n$$

in generale per $k \geq 1$:

ove $[q]$ indica il massimo intero minore o uguale a q . Progettare un algoritmo che generi la serie di Farey per un assegnato valore di n . (Si veda D.E. Knuth, *The Art of Computer Programming*, Vol. 1, *Fundamental Algorithms*, Addison-Wesley, Reading, Mass., p. 157, 1969).

- 2.6.6 Generare la successione in cui ciascun membro è la somma di fattoriali adiacenti, cioè:

$$\begin{aligned}f_3 &= 1! + 0! \\f_4 &= 2! + 1! \\f_5 &= 3! + 2!\end{aligned}$$

Si ricordi che per definizione $0! = 1$.

ALGORITMO 2.7 INVERSIONE DELLE CIFRE DI UN INTERO

Problema

Progettare un algoritmo che accetta un intero positivo e ne inverte l'ordine delle cifre.

Sviluppo dell'algoritmo

Il rovesciamento delle cifre è una tecnica talvolta usata in informatica per eliminare distorsioni in un insieme di numeri. È importante in alcuni algoritmi di reperimento veloce di informazioni. Un esempio specifico definisce chiaramente la relazione richiesta tra l'input e l'output. Per esempio:

Input: 27953
Output: 35972

Sebbene a questo punto non possiamo conoscere esattamente che cosa fare per ottenere questo rovesciamento, un fatto è certo: ci serve poter accedere individualmente alle cifre del numero in esame. Per il momento ci concentreremo su questo aspetto della procedura. Il numero 27953 è in effetti:

$$2 \cdot 10^4 + 7 \cdot 10^3 + 9 \cdot 10^2 + 5 \cdot 10 + 3$$

Per accedere alle singole cifre risulterà probabilmente più semplice partire ad un estremo del numero e procedere verso l'altro estremo. La domanda è: da che estremo partire? Poiché non è facile, se non a occhio, dire quante cifre ci sono nel numero in input sarà meglio cercare di stabilire l'identità della cifra meno significativa (cioè la prima a destra). Per far ciò occorre "tagliar via" la cifra meno significativa dal numero; in altri termini vogliamo ritrovarci con 2795 ed il numero 3 eliminato ed identificato.

Possiamo ottenere il numero 2795 mediante la divisione intera del numero originale per 10, cioè:

$$27953 \text{ div } 10 \rightarrow 2795$$

Questo elimina il 3 ma non ci consente di recuperarlo. Tuttavia, 3 è il resto della divisione di 27953 per 10; per ottenere tale resto possiamo usare la funzione **mod**. Ossia:

$$27953 \text{ mod } 10 \rightarrow 3$$

Pertanto se applichiamo i due passi:

$$\begin{aligned} r &:= n \text{ mod } 10 & [1] \Rightarrow (r = 3) \\ n &:= n \text{ div } 10 & [2] \Rightarrow (n = 2795) \end{aligned}$$

otteniamo la cifra 3 ed il nuovo numero 2795. Applicando gli stessi due passi al nuovo valore di n possiamo ottenere la cifra 5. Abbiamo allora un meccanismo per accedere iterativamente alle singole cifre del numero di input.

Il nostro successivo compito è la disposizione delle cifre in ordine inverso. Applicando alle prime due cifre la procedura di estrazione, ricaviamo prima 3 e poi 5, che nell'output definitivo appaiono come:

3 seguito da 5 (ovvero 35)

Se il numero originale fosse stato 53 ne avremmo ottenuto il rovescio estraendo per primo il 3, moltiplicandolo per 10, e sommando quindi il 5 per ottenere 35. Ossia:

$$3 \cdot 10 + 5 \rightarrow 35$$

Le ultime tre cifre del numero di input sono 953; nel numero "rovesciato" esse appaiono come 359. Perciò nel momento in cui abbiamo 35 ed estraiamo il 9 possiamo ottenere la sequenza 359 moltiplicando per 10 il 35 e aggiungendo 9. Ossia:

$$35 \cdot 10 + 9 \rightarrow 359$$

Analogamente:

$$359 \cdot 10 + 7 \rightarrow 3597$$

e

$$3597 \cdot 10 + 2 \rightarrow 35972$$

L'ultimo numero ottenuto col processo di moltiplicazione ed addizione è l'intero "a cifre invertite" che stavamo cercando. Esaminando da vicino l'estrazione delle cifre ed il processo di inversione è evidente che entrambi coinvolgono un insieme di passi che possono essere svolti iterativamente.

Dobbiamo ora trovare un meccanismo per costruire cifra per cifra il numero "rovesciato". Supponiamo che sia **reverse** la variabile da impiegare per costruire l'intero invertito; ad ogni passo si utilizzano il suo valore **precedente** e la cifra più recente estratta.

Riscrivendo in funzione della variabile **reverse** il processo di moltiplicazione e addizione che abbiamo appena descritto otteniamo:

Iterazione	Valore di reverse
[1] reverse := reverse *10 + 3	3
[2] reverse := reverse *10 + 5	35
[3] reverse := reverse *10 + 9	359
.	.

Pertanto per costruire l'intero rovesciato possiamo utilizzare il costrutto:

$$\begin{aligned} \text{reverse} := & (\text{valore precedente di reverse}) * 10 + \\ & + (\text{cifra destra estratta più di recente}) \end{aligned}$$

La variabile **reverse** può essere impiegata in **entrambi** i membri di questa espressione e deve essere inizialmente azzerata affinché il suo

valore sia corretto dopo la prima iterazione (cioè $dreverse = 3$). Questo passo di inizializzazione per $dreverse$ serve a garantire che l'algoritmo funzioni correttamente anche quando il numero da rovesciare è zero.

Quel che non abbiamo ancora stabilito sono le condizioni sotto cui il processo iterativo ha termine. La condizione di terminazione deve essere in qualche modo collegata al numero di cifre dell'intero in input; infatti la terminazione deve verificarsi non appena tutte le cifre sono state estratte ed elaborate. Ad ogni iterazione il numero di cifre dell'intero da rovesciare diminuisce di uno, dando luogo alla sequenza mostrata in Tab. 2.1. Le successive divisioni intere per 10 del numero da rovesciare producono la sequenza 27953, 2795, 279, ...

Nell'esempio, quando la divisione intera è applicata per la quinta volta si ottiene 0 poiché 2 è inferiore a 10. Poiché a questo punto del calcolo il numero "rovesciato" è stato completamente costruito, possiamo utilizzare il risultato zero per terminare il processo iterativo.

Tab. 2.1. Passi di rovesciamento delle cifre.

Numero da invertire	Numero invertito in costruzione	Passo
27953	3	[1]
2795	35	[2]
279	359	[3]
27	3597	[4]
2	35972	[5]

I passi centrali nel nostro algoritmo di inversione delle cifre sono:

1. Finché ci sono ancora cifre nel numero da invertire eseguire quanto segue:
 - (a) estrarre la cifra più a destra dal numero da invertire e aggiungere tale cifra all'estremo destro della rappresentazione corrente del numero rovesciato;
 - (b) eliminare la cifra più a destra nel numero da invertire.

Quando si includano considerazioni di input ed output e dettagli sull'inizializzazione e la terminazione arriviamo alla seguente descrizione dell'algoritmo.

Descrizione dell'algoritmo

1. Stabilire n , l'intero positivo da rovesciare.
2. Predisporre le condizioni iniziali per l'intero rovesciato $dreverse$.
3. Finché l'intero da invertire è maggiore di zero eseguire quanto segue:
 - (a) utilizzare la funzione resto per estrarre la cifra più a destra nel numero da invertire;

- (b) incrementare di un fattore 10 la precedente rappresentazione $dreverse$ del numero rovesciato e sommargli la cifra estratta più di recente per ottenere il valore corrente di $dreverse$;
- (c) utilizzare la divisione intera per 10 per eliminare la cifra più a destra nel numero che deve essere invertito

Questo algoritmo si può implementare più convenientemente sotto forma di funzione, che accetta in input il numero da rovesciare e restituisce in output il numero con le cifre invertite.

Implementazione in Pascal

```
function dreverse (n: integer): integer;
var reverse: integer;

begin {reverse the order of the digits of a positive integer}
  {assert: n >= 0 ∧ n contains k digits a(1), a(2), a(3), ..., a(k)}
  reverse := 0;
  {invariant: after jth iteration, n = a(1), a(2), a(3), ... a(k-j) ∧
   reverse = a(k), a(k-1), ..., a(k-j+1)}
  while n > 0 do
    begin
      reverse := reverse * 10 + n mod 10;
      n := n div 10
    end;
  {assert: reverse = a(k), a(k-1), ..., a(1)}
  dreverse := reverse
end
```

Note di progetto

1. Il numero di passi per invertire l'ordine delle cifre in un intero è direttamente proporzionale al numero di cifre dell'intero.
2. Dopo l'-esimo passaggio attraverso il loop (i non è definita esplicitamente nell'algoritmo) la variabile $dreverse$ contiene le i cifre più a sinistra del numero rovesciato. Tale condizione rimane invariante per ogni i . Inoltre, dopo i iterazioni la variabile n è ridotta di i cifre. Al termine, quando n è stato ridotto a zero cifre la variabile $dreverse$ conterrà lo stesso numero di cifre dell'intero in ingresso.
L'algoritmo terminerà poiché la variabile n è diminuita di una cifra ad ogni iterazione. L'algoritmo funziona correttamente per tutti i valori $n \geq 0$.
3. In questo progetto vediamo una volta di più come la soluzione completa di un problema sia costruita iterativamente a partire da una serie di soluzioni parziali. Questa è la struttura fondamentale di molti algoritmi.

4. Nel progettare questo algoritmo abbiamo proceduto implicitamente a ritrso dalla soluzione al punto di partenza. Quest'idea di procedere partendo dal fondo è un concetto molto potente ed importante nell'informatica, che utilizzeremo più concretamente nel progetto di algoritmi più avanzati.
5. Un esempio specifico è servito per guidarci alla costruzione del metodo di rappresentazione inversa.

Applicazioni

Hashing e reperimento di informazioni, applicazioni di *data base*.

Problemi supplementari

- 2.7.1. Progettare un algoritmo che conti il numero di cifre di un intero.
- 2.7.2. Progettare un algoritmo per sommare le cifre di un intero.
- 2.7.3. Progettare un algoritmo che legga un insieme di n cifre isolate e le converta in un numero intero. Per esempio, l'algoritmo convertirà l'insieme {2, 7, 4, 9, 3} nell'intero 27493.

ALGORITMO 2.8 CONVERSIONE DI BASE

Problema

Convertire un intero decimale nella sua corrispondente rappresentazione ottale.

Sviluppo dell'algoritmo

Spesso in informatica è necessario convertire un numero decimale nei sistemi di numerazione binario, ottale o esadecimale. Per progettare un algoritmo che esegua tali conversioni occorre comprendere chiaramente che cosa implica il cambiamento di base.

Poiché probabilmente all'inizio non abbiamo idee precise sul meccanismo della conversione di base, cominceremo ad inquadrare l'argomento. Possiamo partire cercando di stabilire esattamente che cosa siano un numero decimale e un numero ottale; a tale scopo possiamo esaminare alcuni esempi specifici.

Il numero 275 decimale (cioè in base 10), in virtù della sua stessa rappresentazione, consta di:

5 unità	$5 \cdot 1$
7 decine	$7 \cdot 10$
2 centinaia	$2 \cdot 100$
	275

Il sistema decimale usa per rappresentare i numeri le dieci cifre 0, 1, 2, 3, ... 9; la *posizione* corrente di ciascuna cifra ne determina il valore.

Una convenzione analoga vale per il sistema di numerazione ottale. Questo usa per rappresentare i numeri solo le otto cifre 0, 1, 2, 3, ... 7. Nel sistema ottale la posizione di ciascuna cifra ne determina il valore in un modo simile (ma diverso) a quello del sistema decimale. Facendo qualche passo avanti a scopo dimostrativo, si può vedere che la rappresentazione ottale del numero decimale 275 è 423. Il numero ottale consiste di:

3 unità	$3 \cdot 1$
2 volte 8	$2 \cdot 8$
4 volte 64	$4 \cdot 64$
	275 decimale

Come ulteriore esempio, al contrario del numero 275 decimale (scritto 275_{10}), il numero 275 ottale (scritto 275_8) consiste di:

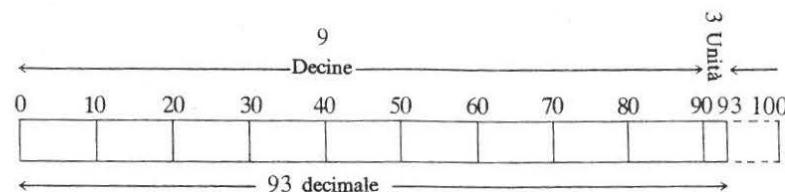
5 unità	$5 \cdot 1$
7 volte 8	$7 \cdot 8$
2 volte 64	$2 \cdot 64$
	189 decimale

Possiamo vedere che 275_8 è molto più piccolo di 275_{10} .

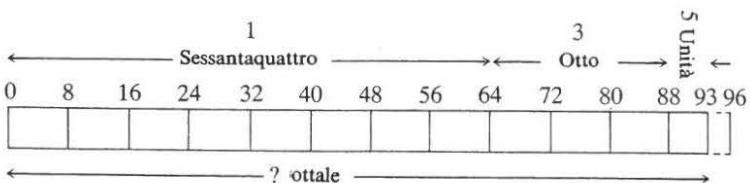
Conclusa questa fase d'inquadramento possiamo adesso tornare al nostro problema di conversione di base. Abbiamo imparato che nel cambiare base a un numero non ne modifichiamo per niente il valore, ma ne cambiamo solo il modo di rappresentazione.

Considereremo, per partire, la conversione di un particolare numero decimale nella sua rappresentazione ottale; speriamo che ciò evidenzi buona parte del meccanismo dell'algoritmo che andiamo cercando.

Supponiamo di voler convertire il numero 93 decimale nella sua rappresentazione ottale. Sappiamo che 93 si compone di 9 decine e 3 unità. In forma grafica abbiamo:



Il numero ottale corrispondente dovrà dividersi in unità, gruppi di otto, di sessantaquattro, e così via, invece di unità, decine, centinaia e via dicendo. Utilizzando un diagramma per la rappresentazione ottale abbiamo:



Dal diagramma possiamo vedere che dividendo 93 (decimale) in blocchi da otto restano alla fine ancora cinque unità. La nostra rappresentazione ottale avrà pertanto la forma ...5. Dividendo 93_{10} in blocchi da otto scopriamo anche che ce ne stanno 11 (infatti $11 \cdot 8 = 88$). A questo punto potremmo esser tentati di scrivere 115 quale rappresentazione ottale di 93_{10} . Tuttavia c'è un problema, in quanto 11 non è una cifra ottale valida (le sole cifre ottali che possiamo usare sono 0, 1, 2, 3, ... 7); ci chiediamo come fare ad uscirne. Con sole otto cifre il massimo numero di gruppi da otto che possiamo rappresentare è 7; ricordando l'esempio di prima sappiamo che nella rappresentazione possono essere usati gruppi di sessantaquattro. Infatti 8 blocchi da otto possono (e devono) essere rappresentati come 1 blocco da sessantaquattro; ne segue che 11 gruppi da otto possono essere rappresentati come un blocco da sessantaquattro e 3 da otto. Applicando la convenzione posizionale che abbiamo già incontrato, possiamo concludere che la rappresentazione ottale di 93 decimale è 135.

Il numero ottale 135 consta di:

$$\begin{array}{ll}
 \begin{array}{l} 5 \text{ unità} \\ 3 \text{ volte } 8 \\ 1 \text{ volta } 64 \end{array} &
 \begin{array}{l} 5 \cdot 1 \\ 3 \cdot 8 \\ 1 \cdot 64 \\ \hline 93 \text{ decimale} \end{array}
 \end{array}$$

Utilizzando una rappresentazione grafica siamo pervenuti alla rappresentazione ottale di 93_{10} . Quel che dobbiamo fare adesso è tradurre in algoritmo quanto indicato dal diagramma. La divisione intera di 93 per 8 ci dirà quanti gruppi da otto sono contenuti in 93_{10} , ossia:

$$93 \text{ div } 8 \Rightarrow 11 \text{ gruppi da 8}$$

Si deve anche trovare il numero di gruppi da sessantaquattro contenuti in 93_{10} ; questo implica di spezzare gli 11 gruppi da otto in 1 gruppo da sessantaquattro e 3 da otto. Per essere sistematici in questo lavoro dovremo cercare per prime le unità, quindi i gruppi da 8, poi quelli da 64, quelli da 512, e così via. Abbiamo:

$$\begin{aligned}
 93 \text{ unità} &\Rightarrow 11 \text{ gruppi da 8 e 5 unità} \\
 11 \text{ gruppi da 8} &\Rightarrow 1 \text{ gruppo da 64 e 3 da 8} \\
 1 \text{ gruppo da 64} &\Rightarrow 0 \text{ gruppi da 512 e 1 da 64}
 \end{aligned}$$

Adesso possiamo identificare più specificamente il meccanismo di conversione di base:

$$\begin{aligned}
 93 : 8 &= 11 \text{ resto } 5 \\
 11 : 8 &= 1 \text{ resto } 3 \\
 1 : 8 &= 0 \text{ resto } 1
 \end{aligned}$$

Per convertire un numero decimale nella sua rappresentazione ottale partiamo dividendolo per 8. Il resto di questa operazione è la cifra meno significativa (cioè 5) della rappresentazione ottale. Il quoziente (cioè 11) viene preso e diviso per 8. Il resto di questa operazione è la penultima cifra significativa della rappresentazione ottale. Il procedimento - di dividere il quoziente ed assumere il resto come nuova cifra significativa della rappresentazione ottale - continua finché non si incontra un quoziente nullo. Questa situazione indica che non ci sono ulteriori cifre nella rappresentazione ottale (nell'esempio precedente il quoziente zero dopo la terza divisione per 8 indica che non ci sono gruppi da 512 nel numero originale 93, e la cosa è ovvia).

Abbiamo finalmente gli elementi essenziali per implementare l'algoritmo di conversione di base. Esso implica un procedimento iterativo in cui le cifre ottali sono successivamente ricavate come resti di una serie di quozienti, ciascuno dei quali a sua volta discende dal suo predecessore mediante divisione per 8.

Più in dettaglio, partendo con $q := 93$ otteniamo:

$$\begin{aligned}
 q := q \text{ div } 8 &\Rightarrow q := 11 \text{ con resto } 5 & [1] \\
 q := q \text{ div } 8 &\Rightarrow q := 1 \text{ con resto } 3 & [2] \\
 q := q \text{ div } 8 &\Rightarrow q := 0 \text{ con resto } 1 & [3]
 \end{aligned}$$

Per ottenere sia il quoziente q sia il resto r ad ogni passo, dovremo impiegare le due funzioni: divisione intera e modulo (resto); al generico passo i -esimo avremo:

$$\begin{aligned}
 r &:= q \text{ mod } 8 \Rightarrow \text{resto} \\
 q &:= q \text{ div } 8 \Rightarrow \text{quoziente ridotto}
 \end{aligned}$$

Come abbiamo già riconosciuto, il procedimento iterativo deve essere concluso non appena si incontra un quoziente nullo. Il valore di partenza del quoziente q sarà il valore iniziale decimale del numero da convertire in ottale. I passi essenziali dell'algoritmo sono:

1. Inizializzare il quoziente q col numero decimale da convertire.
2. Ricavare iterativamente le cifre ottali mediante una serie di calcoli di resto e di quoziente sui valori successivi del quoziente. Concludere il processo quando si incontra un quoziente nullo.

L'algoritmo dovrà essere implementato in modo da trattare anche conversioni diverse dalla decimale in ottale (per esempio, da decimale in binario, da decimale in ternario). Per consentire ciò, i dati in ingresso all'algoritmo saranno la nuova base richiesta ed il numero decimale da convertire. Per comodità di uscita le cifre della nuova rappresentazione verranno memorizzate in un array e stampate solo dopo che il loop si è concluso.

Includendo queste migliorie otteniamo l'algoritmo dettagliato che segue.

Descrizione dell'algoritmo

1. Stabilire la nuova base *newbase* ed inizializzare il quoziente *q* col numero decimale da convertire.
2. Porre a zero il numero *ndigit* delle cifre del numero convertito.
3. Ripetere i passi seguenti:
 - (a) calcolare la prossima cifra più significativa (p. es. ottale) a partire dal quoziente corrente *q*, come resto *r* della divisione per *newbase*;
 - (b) convertire *r* nel corrispondente codice ASCII [*];
 - (c) incrementare il contatore *ndigit* e memorizzare *r* nell'array di output *newrep*;
 - (d) calcolare il successivo quoziente *q* dal precedente utilizzando la divisione intera per *newbase* finché il quoziente non è nullo.

Viene usato un array di *caratteri* per consentire l'impiego di nuove basi di numerazione anche oltre il 10.

Implementazione in Pascal

```
procedure basechange(n,newbase: integer);‡
var i {index for new digit output array},
    ascii {ascii value for current digits},†
    ndigit {current counter of new digits computed},
    q {current quotient},
    r {current digit for newbase representation},
    zero {ascii value of zero character}: integer;
    newrep: array[1..100] of char {output array};

begin {changes base of integer from decimal to any base <=36}
  {assert: n>0  $\wedge$  2 <= newbase <= 36}
  zero := ord('0');
  q := n;
  ndigit := 0;
```

[*] V. algoritmo 2.9 per la spiegazione del codice ASCII.

* La procedura chiamante è responsabile del rispetto dei limiti ammessi per *n* e *newbase*.

invariant: after the *ndigit* iteration, $q = n \text{ div } (\text{newbase}^{\lceil \frac{n}{\text{newbase}} \rceil})$
 $\wedge \text{newrep}[1..\text{ndigit}]$ contains the *ndigit* least significant digits
 of *n* in *newbase* in reverse order $\wedge r$ contains *ndigit* least
 significant digit}

```
repeat
  r := q mod newbase;
  ndigit := ndigit + 1;
  ascii := zero + r;
  if ascii > ord ('9') then ascii := ascii + 7;
  newrep[ndigit] := chr (ascii);
  q := q div newbase
until q = 0;
{assert: newrep[1..ndigit] contains the ndigit representation of n in the
base newbase in reverse order}
writeln ('Base ', newbase, ' representation of ', n, ' is ');
for i := ndigit downto 1 do
  write (newrep[i]);
writeln
end
```

Note di progetto

1. Il numero di passi in questo algoritmo è funzione di *newbase* e dell'entità del numero *n*. Più precisamente, il numero di passi è proporzionale ad *x*, dove *x* è l'intero più piccolo che soddisfa la condizione $n < (\text{newbase})^x$.
2. Lungo il processo iterativo, dopo l'iterazione *ndigit*-esima sono state calcolate le *ndigit* cifre più a destra della rappresentazione in base *newbase*. Tale condizione rimane invariante durante tutto il processo iterativo: al termine saranno state calcolate *ndigit* cifre. Dopo ogni iterazione *r* contiene la cifra in posizione *ndigit* contando da destra nella nuova rappresentazione. Poiché *newbase* è un intero maggiore di uno e su *q* si esegue una divisione intera ad ogni iterazione, *q* diminuirà fino a che *q* = 0. Pertanto è garantita la terminazione. L'algoritmo funziona correttamente per tutti gli interi non negativi rappresentabili in memoria; se l'output contiene un numero di cifre superiore a quello stampabile su una sola linea, si verificherà un errore.
3. Nel progettare questo algoritmo è essenziale una chiara comprensione della natura dei dati con cui abbiamo a che fare. Esempi specifici ci hanno condotto al costrutto iterativo richiesto ed alla condizione di terminazione adatta.
4. Una volta di più in questo algoritmo osserviamo il modo di procedere generale che consiste nello stabilire le condizioni iniziali e costruire la soluzione iterativamente da una serie di predecessori.
5. L'algoritmo non tratta valori di *newbase* superiori a 36 (ossia le 10 cifre decimali più i 26 caratteri alfabetici) per mancanza di caratteri convenienti a rappresentare le cifre.

6. L'algoritmo assume che i caratteri alfabetici seguano la cifra 9 nel set di caratteri a sette posizioni di distanza, come è in ASCII (American Standard Code for Information Interchange).
7. L'algoritmo potrebbe essere reso più generale ammettendo in ingresso numeri in una base qualsiasi fino a 36.

Applicazioni

Interpretazione di dati e istruzioni memorizzate nel computer.

Problemi supplementari

- 2.8.1. Modificare l'algoritmo per includere la generalizzazione suggerita alla nota (7).
- 2.8.2. Progettare un algoritmo che converta i numeri da binario in ottale.
- 2.8.3. Progettare un algoritmo che converta i numeri da binario in decimale.
- 2.8.4. Progettare un algoritmo che accetti in ingresso un numero decimale e lo converta nella sua rappresentazione BCD (binary-coded decimal). In BCD ogni cifra è rappresentata da un codice binario di 4 bit.
- 2.8.5. Progettare un algoritmo che accetti in ingresso un numero decimale frazionario e lo converta nel corrispondente numero binario, con un livello assegnato di precisione (p. es. $0.625_{10} = 0.101_2 = 1*2^{-1} + 0*2^{-2} + 1*2^{-3}$).

ALGORITMO 2.9 CONVERSIONE DA CARATTERI A NUMERI

Problema

Data la rappresentazione a caratteri di un intero, convertirlo nella sua forma decimale convenzionale.

Sviluppo dell'algoritmo

Prima di affrontare la soluzione di questo problema di conversione, dobbiamo comprendere appieno la differenza tra la rappresentazione di un intero a caratteri e la sua rappresentazione convenzionale come numero.

Il numero di caratteri distinti necessario per rappresentare le informazioni di un testo è relativamente piccolo; i caratteri alfabetici maiuscoli e minuscoli, le dieci cifre decimali ed un po' di caratteri di interpunkzione e controllo danno in totale solo un centinaio di segni diversi. Al contrario, l'estensione richiesta per i numeri interi (e quelli reali) può arrivare ai milioni, ai miliardi e anche oltre. Per avere una rappresentazione *univoca* di una così ampia gamma di numeri occorre una quantità considerevole di "spazio" ove memorizzare ciascun numero; i numeri sono rappresentati nel computer in notazione binaria utilizzando appena le cifre 0 e 1. Come esempio, confrontiamo le rappresentazioni binaria e decimale del numero 221.

$$\text{notazione decimale: } = 2*10^2 + 2*10^1 + 1*10^0 = 221$$

$$\begin{aligned}\text{notazione binaria: } &= 1*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 0*2^1 \\ &+ 1*2^0 = 11011101\end{aligned}$$

Confrontando le due diverse rappresentazioni, osserviamo che per rappresentare 221 bastano solo 3 cifre decimali contro le 8 binarie. Per rappresentare un miliardo, occorrono 10 cifre decimali e circa 32 cifre binarie o bit. L'unità fondamentale di memorizzazione per i numeri, o parola del computer (*word*), è normalmente di 16, 32, 36, 60 o 64 bit. Poiché per distinguere il centinaio circa di caratteri che sembra ci servano occorrono solo 7 o 8 bit (8 bit si dicono un *byte*; $2^8 = 256$), è assai dispendioso in termini di memoria del computer immagazzinare solo un carattere per ogni unità di memorizzazione numerica; la soluzione consiste pertanto nell'impaccare più caratteri in ogni parola del computer. Per esempio, nei computer a 32 bit scopriamo che normalmente vengono memorizzati quattro caratteri da 8 bit per ogni parola.

Per rendere possibile l'impaccamento è stato necessario attribuire codici fissi di 8 bit a qualcosa come un centinaio di caratteri. Uno dei più usati di questi sistemi di codifica è il codice ASCII (American Standard Code for Information Interchange); la Tab. 2.2 mostra alcuni esempi di tale sistema.

Tab. 2.2. Codici ASCII dei caratteri e decimali corrispondenti.

Carattere	Codice di 8 bit	Valore decimale
0	00110000	48
1	00110001	49
2	00110010	50
3	00110011	51
⋮	⋮	⋮
A	01000001	65
B	01000010	66
C	01000011	67
⋮	⋮	⋮
a	01100001	97
b	01100010	98
⋮	⋮	⋮

Si osservi che anche alle cifre decimali 0, 1, 2, 3,... 9 sono assegnati codici di 8 bit; per questo motivo, in molte applicazioni è conveniente rappresentare i numeri come sequenze di caratteri. Per esempio, possiamo rappresentare come stringa di caratteri una data:

23 Aprile 1984

Con rappresentazioni come questa, capita di dover eseguire operazioni aritmetiche convenzionali su numeri rappresentati da sequenze di caratteri da 8 bit. Nell'esempio precedente, 23 non ha il valore di 2 decine e 3 unità, ma in due byte consecutivi sono memorizzati i valori decimali 50 e 51. Non possiamo eseguire direttamente operazioni aritmetiche sui codici della rappresentazione a caratteri in quanto la rappresentazione binaria corrispondente non coincide con la notazione posizionale standard usata per rappresentare i numeri nel computer. In un caso come questo la nostra sola alternativa è convertire il numero dalla rappresentazione a caratteri alla rappresentazione interna convenzionale. Siamo così ricondotti al primitivo problema di conversione da caratteri a numeri.

Il nostro compito è quindi assumere in ingresso la rappresentazione a caratteri di un numero, con una certa quantità di cifre, e convertirlo nella notazione decimale convenzionale. Per far ciò è necessario conoscere la particolare codifica adottata. Fortunatamente, molti linguaggi di programmazione tra cui Pascal forniscono alcune funzioni aggiuntive per rendere abbastanza semplici compiti come questo. Supponiamo per esempio di voler convertire la sequenza di caratteri 1984 nel numero decimale 1984; la rappresentazione da cui partiamo è:

1	9	8	4	caratteri
↓	↓	↓	↓	
49	57	56	52	valori ASCII

Per eseguire la conversione, il 49 dovrà essere tradotto in 1000, il 57 in 900, il 56 in 80 e il 52 in 4. Per ottenere la cifra decimale corrispondente, dobbiamo sottrarre 48 (valore ASCII del carattere 0) dalla rappresentazione ASCII di quella cifra. Il Pascal rende questo lavoro più semplice fornendo una funzione chiamata *ord* che accetta come argomento un carattere di 8 bit e ne restituisce il valore decimale corrispondente (ASCII nel nostro caso); per esempio l'istruzione:

x := ord('9')

assegna ad *x* il valore decimale 57. Nell'esempio, per ottenere le cifre decimali dovremo sottrarre *ord('0')* dai valori ASCII di ciascun carattere della sequenza.

Giunti a questo punto, il passo successivo è utilizzare le cifre per costruire il numero decimale corrispondente. La sequenza di caratteri

dovrà essere elaborata un byte alla volta per ottenere le cifre decimali; ciò che dobbiamo capire adesso è come ciascuna cifra debba essere convertita, in modo da dare il contributo appropriato al numero decimale che stiamo cercando. Se cominciamo ad elaborare i caratteri da sinistra, incontrando l'“1” non è immediatamente evidente per quale potenza di dieci debba essere moltiplicato dopo la conversione. Tuttavia, non appena incontriamo un secondo carattere nella successione, sappiamo che il moltiplicatore sarà almeno 10. E, una volta incontrato un terzo carattere, sappiamo che la seconda cifra dovrà essere moltiplicata per 10 e la prima per 100. Ne deriva l'idea che possiamo procedere essenzialmente con lo stesso meccanismo utilizzato nell'algoritmo 2.7 per assemblare l'intero richiesto. Per il nostro esempio:

$$\begin{array}{ll}
 \begin{array}{l} 1 \rightarrow 1 \\ 9 \rightarrow 1*10 + 9 \\ 8 \rightarrow 19*10 + 8 \\ 4 \rightarrow 198*10 + 4 \end{array} & \begin{array}{l} \rightarrow 1 \\ \rightarrow 19 \\ \rightarrow 198 \\ \rightarrow 1984 \end{array}
 \end{array}$$

Non ci resta che definire i dettagli del meccanismo di implementazione di questo processo. Il meccanismo di “slittamento a sinistra” si ottiene ad ogni passo moltiplicando per 10 il valore precedente e aggiungendovi la cifra decimale corrente. Nella nostra implementazione, supporremo predefinita la lunghezza della stringa di caratteri e già eseguito il controllo che questa rappresenti un numero intero positivo.

Descrizione dell'algoritmo

1. Definire la stringa di caratteri da convertire in forma decimale e la sua lunghezza *n*.
2. Inizializzare con zero il risultato.
3. Porre nella variabile *base* 0 il valore ASCII di “0”.
4. Finché non sono stati esaminati *n* caratteri ripetere quanto segue:
 - (a) convertire il prossimo carattere nella cifra decimale corrispondente,
 - (b) spostare a sinistra di una posizione il valore decimale corrente e sommargli la cifra del carattere corrente.
5. Restituire l'intero decimale che corrisponde alla rappresentazione a caratteri in ingresso.

Implementazione in Pascal

```
function chrtodec (string: nelements; n: integer): integer;
var i {index for count of characters converted},
  dec {used to build converted decimal integer},
  base0 {ascii or ordinal value of character 0 }: integer;
```

```

begin {converts character string integer representation to decimal}
  assert:  $n \geq 0 \wedge \text{string}[1..n]$  represents a non-negative number
  dec := 0;
  base0 := ord ('0');
  invariant: after the  $i$ th iteration, dec contains the  $i$  leftmost digits
    of the string in integer form  $\wedge i \leq n$ 
  for  $i := 1$  to  $n$  do
    dec := dec * 10 + ord ( $\text{string}[i]$ ) - base0;
  assert: dec contains the integer representation of the  $n$  digits in string
  chrtodec := dec
end

```

Note di progetto

- Il numero di passi per eseguire la conversione in decimale di un intero rappresentato a caratteri è direttamente proporzionale al numero n di caratteri della rappresentazione di partenza.
- All' i -esimo passaggio attraverso il loop, saranno stati convertiti in cifre i primi i caratteri della stringa rappresentativa. Pertanto dopo i iterazioni, la variabile dec conterrà le i cifre più a sinistra della rappresentazione decimale, con la cifra più a sinistra moltiplicata per 10^{i-1} , la cifra vicina per 10^{i-2} e quella più a destra moltiplicata per 10^0 . La terminazione del loop è assicurata dopo n iterazioni. Questo algoritmo non tratta numeri reali, né ci garantisce dalle situazioni in cui la stringa da convertire provoca overflow sugli interi.

Applicazioni

Applicazioni gestionali, elaborazione di nastri.

Problemi supplementari

- Progettare un algoritmo che operi conversioni in decimale anche in presenza di una virgola decimale nella stringa di caratteri in ingresso.
- Progettare un algoritmo per convertire la rappresentazione decimale di un numero nella corrispondente rappresentazione a caratteri.
- Sapendo che tutti i codici ASCII sono inferiori a 128, progettare un algoritmo che legga un insieme di caratteri e decida se può rappresentare o meno un numero decimale.

CAPITOLO 3

METODI DI FATTORIZZAZIONE

INTRODUZIONE

Nella lunga storia della matematica, già dal tempo degli antichi Greci, la teoria dei numeri in generale, ed in particolare i metodi di fattorizzazione, sono stati studiati estesamente. Con lo sviluppo dei computer la conoscenza di queste tecniche è cresciuta d'importanza, sia per ragioni eminentemente pratiche, sia per l'interesse teorico. Molti degli algoritmi sviluppati assai prima dell'avvento del computer oggi hanno una parte rilevante nelle sue applicazioni.

I numeri primi, ad esempio, sono utilizzati in svariate applicazioni. Metodi sviluppati di recente per crittografare un testo e renderlo incomprensibile a tutti tranne che all'autore, si basano sul fatto che il prodotto di due numeri primi molto grandi (ciascuno con più di 100 cifre) è in effetti impossibile da fattorizzare in un tempo accettabile con i metodi di fattorizzazione conosciuti. I metodi di fattorizzazione più noti si basano sull'uso dei massimi comuni divisori e tecniche collegate. I numeri primi giocano anche un ruolo nei metodi di accesso rapido alle informazioni che impiegano algoritmi di *hashing* (se ne parla nel capitolo 5); in certi schemi di hashing risulta appropriato scegliere dimensioni delle tabelle che siano numeri primi. Negli algoritmi crittografici, come in generale lavorando con grandi numeri primi, c'è la necessità di innalzare numeri a potenze elevate con metodi efficienti; in questo capitolo considereremo una tecnica per questo scopo.

I numeri casuali sono impiegati diffusamente in molti studi di simulazione. Il metodo per la generazione di numeri casuali che considereremo si è sviluppato da considerazioni teoriche della teoria della congruenza.

ALGORITMO 3.1

RICERCA DELLA RADICE QUADRATA DI UN NUMERO

Problema

Dato un numero m trovare un algoritmo che ne calcoli la radice quadrata.

Sviluppo dell'algoritmo

Affrontando dal principio il problema di progettare un algoritmo di calcolo per le radici quadrate, potremmo essere in dubbio su come partire. In tal caso dobbiamo essere ben certi di che cosa si intenda per "radice quadrata di un numero". Facendo qualche esempio specifico, sappiamo che la radice quadrata di 4 è 2, quella di 9 è 3, quella di 16 è 4 e così via. In altri termini:

$$\begin{aligned} 2 \cdot 2 &= 4 \\ 3 \cdot 3 &= 9 \\ 4 \cdot 4 &= 16 \\ \vdots & \end{aligned}$$

Da questi esempi possiamo concludere che in generale la radice quadrata n di un numero m deve soddisfare l'equazione

$$n \cdot n = m \quad (1)$$

Se siamo incerti sul da farsi, possiamo attribuire un valore di primo tentativo alla radice quadrata e verificare con l'equazione (1) se il nostro tentativo è stato corretto. Supponiamo, per esempio, di non conoscere la radice quadrata di 36. Potremmo supporre che 9 sia un valore corretto; utilizzando l'equazione (1) troviamo che $9 \cdot 9 = 81$, maggiore di 36. Il tentativo con 9 è troppo alto, potremmo tentare successivamente con 8; $8 \cdot 8 = 64$, che è ancora maggiore di 36, ma meglio approssimato del tentativo iniziale.

L'indagine sin qui fatta ci suggerisce l'adozione del seguente approccio sistematico per risolvere il problema:

1. Scegliere un numero n minore del numero m di cui vogliamo la radice quadrata.
2. Elevare n al quadrato e se il risultato è maggiore di m diminuire di 1 n ripetendo il passo 2, altrimenti andare al passo 3.

3. Quando il quadrato del valore di tentativo è minore di m possiamo cominciare ad incrementare n di 0.1 finché non calcoliamo nuovamente un quadrato che superi m . A questo punto, cominciamo a diminuire di 0,01 i nostri tentativi; e così via, finché non otteniamo la radice quadrata con la precisione desiderata.

Sotto forma di diagramma, possiamo vedere in Fig. 3.1 come questo algoritmo approssimi la soluzione richiesta.

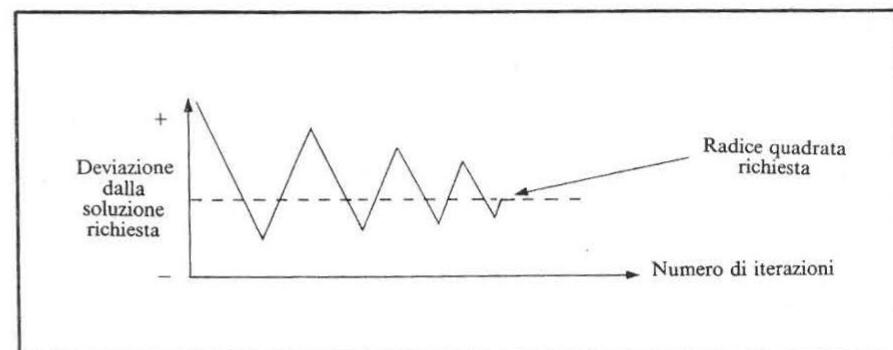


Fig. 3.1.
Processo di convergenza alla radice quadrata.

Analizzando con attenzione l'algoritmo, osserviamo che il numero di iterazioni richieste dipende in maniera critica dalla bontà della scelta iniziale (per esempio, se m è 10000 ed il valore iniziale supposto n è 500, occorrono più di 400 iterazioni prima di cominciare a convergere rapidamente sulla radice quadrata). Questa osservazione fa sorgere la domanda: possiamo ricavare un metodo più rapido per avvicinarcisi alla radice quadrata, che non sia così fortemente dipendente dalla scelta iniziale?

Per cercare di evolvere verso un algoritmo migliore, torniamo al problema di trovare la radice quadrata di 36. Assumendo come tentativo iniziale 9, abbiamo scoperto che:

$$9^2 = 81 \text{ che è maggiore di } 36.$$

Sappiamo dall'equazione (1) che, se il numero 9 fosse la vera radice quadrata, esso sarebbe contenuto in 36 dando 9 come quoziente. Invece, dividendo 36 per 9 otteniamo 4; se avessimo scelto inizialmente 4 come candidato per la nostra radice quadrata, avremmo ottenuto:

$$4^2 = 16 \text{ che è minore di } 36.$$

Quanto sopra ci mostra che, scegliendo un valore di tentativo troppo grande, possiamo ricavarne facilmente un altro che è troppo piccolo. Quanto più discosto è il tentativo per eccesso, tanto più lontano sarà anche il corrispondente tentativo per difetto. In altri termini, il 9 e il 4 tendono ad annullarsi l'un l'altro scostandosi in versi opposti dalla radice n .

Quadrato	Radice quadrata
81	9
36	??
16	4

La radice quadrata di 36 si troverà in una zona compresa tra 9, che è troppo grande, e 4, che è troppo piccolo. Facendo la media tra 9 e 4:

$$\frac{(9+4)}{2} = 6.5$$

ci procuriamo una stima "a mezza strada" tra 9 e 4. Di nuovo, tale scelta potrà essere maggiore, uguale o minore di 36.

Troviamo che $6.5^2 = 42.25$ superiore a 36; dalla divisione:

$$\frac{36}{6.5} = 5.53$$

otteniamo ancora un valore complementare (5.53) approssimato per difetto.

Quadrato	Radice quadrata
81	9
42.25	maggior di 36 6.5
36	??
30.5809	minore di 36 5.53
16	4

Abbiamo adesso due stime della radice quadrata, una per parte, più prossime alla soluzione che non le prime due. Possiamo proseguire alla ricerca di una migliore approssimazione, mediando i due valori di tentativo più recenti:

$$\frac{(6.5 + 5.53)}{2} = 6.015$$

ove $6.015^2 = 36.6025$, che è di poco maggiore del quadrato che stiamo cercando. Sebbene non l'abbiamo dimostrato, possiamo sospettare che la strategia di mediare le stime complementari di una radice quadrata converga assai rapidamente al risultato richiesto, anche con stime iniziali sfavorevoli. A questo punto dovremmo compiere alcune analisi matematiche per confermare la validità di questa ipotesi (vedi Note di progetto); tuttavia daremo per scontato che si tratti di una strategia valida e procederemo allo sviluppo dell'algoritmo.

Il primo impegno è ora quello di chiarire la regola di formazione della media che intendiamo adoperare per generare approssimazioni

sempre migliori della radice quadrata richiesta. Per far ciò torniamo al "problema della radice quadrata di 36". Come valore iniziale di primo tentativo $g1$ abbiamo scelto 9; quindi abbiamo eseguito la media di questo valore con il suo complementare ($36/9 = 4$). Nel caso generale, il valore complementare è dato da:

$$\text{valore complementare} := \frac{m}{g1}$$

Nel passo successivo abbiamo ottenuto una stima migliore $g2$ della radice quadrata, facendo la media tra $g1$ ed il suo valore complementare (ossia $g2 = (9 + 36/9)/2 = 6.5$). Possiamo pertanto scrivere l'espressione di $g2$ nel caso generale come:

$$g2 := ((g1 + (m/g1))/2$$

Siamo ora in grado di usare tale espressione come base per il nostro algoritmo di ricerca della radice quadrata. Nell'esempio per la ricerca della radice quadrata di 36, abbiamo iniziato con $g1 = 9$ ed abbiamo stabilito che $g2 = 6.5$. Quindi abbiamo ripetuto il processo di media. Tuttavia, al fine di ottenere un'approssimazione ancora migliore, adesso $g2$ (cioè 6.5) assume il ruolo che prima apparteneva a $g1$. Possiamo realizzare questo scambio ripetitivo di ruoli implementando il seguente loop:

$$\begin{array}{l} g2 := ((g1 + (m/g1))/2 \\ g1 := g2 \end{array}$$

Tale loop produrrà approssimazioni $g2$ sempre migliori per la radice quadrata di m .

Una questione ancora aperta riguarda quando terminare il processo iterativo. Sembra non abbiamo modo di sapere in anticipo quante iterazioni occorrono in generale per calcolare un valore accettabile per una certa radice quadrata; ci occorre perciò qualche altro criterio per arrestare il procedimento iterativo. Sappiamo che con successive iterazioni l'algoritmo produce approssimazioni sempre più vicine alla radice quadrata. Per esempio, per il problema della radice quadrata di 36, abbiamo la successione:

$$9 \rightarrow 6.5 \rightarrow 6.015 \rightarrow \dots$$

All'aumentare delle iterazioni possiamo attenderci che le differenze tra le radici stimate in successive iterazioni divengano progressivamente più piccole. Possiamo quindi terminare l'algoritmo quando la differenza tra $g1$ e $g2$ diventa inferiore ad un certo errore prefissato (0.0001 potrebbe essere un errore accettabile). Non possiamo sapere in anticipo se $g2$ sarà progressivamente maggiore o minore di $g1$; per sicurezza dovremo adottare come criterio di terminazione il *valore assoluto della differenza* tra $g1$ e $g2$.

La sola questione ancora aperta è come scegliere il valore di tentativo iniziale. Le considerazioni fatte sinora ci dicono che in un certo senso

non importa quale sia la scelta iniziale, poiché il meccanismo di equilibrimento dell'algoritmo garantisce una sufficiente rapidità di convergenza verso la soluzione. Potremmo pertanto assumere inizialmente il numero m stesso o forse il valore $m/2$ (vedi Note di progettazione). Possiamo ora descrivere in dettaglio l'algoritmo della radice quadrata.

Descrizione dell'algoritmo

1. Fissare il numero m di cui si richiede la radice quadrata e l'errore e (condizione di terminazione).
2. Porre a $m/2$ il primo tentativo g_2 ;
3. Ciclicamente
 - (a) far assumere a g_1 il ruolo di g_2 ;
 - (b) generare con la regola di formazione della media una stima migliore g_2 della radice quadrata, finché il valore assoluto della differenza fra g_1 e g_2 non sia minore dell'errore e .
4. Restituire il valore stimato g_2 della radice quadrata.

Implementazione in Pascal

```
function sqroot(m,error: real): real;
var g1 {previous estimate of square root},
    g2 {current estimate of square root}: real;

begin {estimates square root of number m}
  assert: m>0  $\wedge$  g1=m/2
  g2 := m/2;
  {invariant: |g2 * g2 - m| <= |g1 * g1 - m|  $\wedge$  g1 > 0  $\wedge$  g2 > 0}
  repeat
    g1 := g2;
    g2 := (g1 + m/g1)/2
  until abs(g1-g2)<error;
  assert: |g2 * g2 - m| <= |g1 * g1 - m|  $\wedge$  |g1 - g2| < error
  sqroot := g2
end
```

Note di progetto

1. In generale, non è facile dimostrare quante iterazioni occorrono per trovare la radice quadrata con la precisione voluta. Tuttavia possiamo dimostrare in maniera abbastanza semplice che il metodo converge rapidamente. Al passo n -esimo abbiamo:

$$g_n = s - e$$

ove s è la radice quadrata richiesta di m ed e è l'errore corrispondente. Sostituendo nella formula della media, otteniamo:

$$g_{n+1} = \left((s-e) + \frac{m}{(s-e)} \right) / 2$$

e poiché $m = s^2$ per e piccolo otteniamo:

$$g_{n+1} = s - e + \frac{e^2}{2s}$$

e quindi:

$$g_{n+1} = g_n + \frac{e^2}{2s}$$

Il termine quadratico conferma che il metodo converge rapidamente al risultato desiderato. Anche l'errore relativo $e_n = |1 - (g_n/\sqrt{m})|$ diminuisce rapidamente poiché $g_{n+1} = e^{2n}/2$

2. Dall'esame della regola di formazione della media, deduciamo che al limite di $g_1 = \sqrt{m}$ la formula diviene:

$$g_2 = \left(g_1 + \frac{m}{g_1} \right) / 2 = (g_1 + g_1)/2 = g_1$$

Pertanto il metodo presenta teoricamente una convergenza al limite richiesto. Per stabilire che l'algoritmo ha termine, occorre dimostrare che il valore assoluto della differenza tra due successive iterazioni è strettamente decrescente; ciò discende direttamente dal fatto che l'algoritmo ha convergenza quadratica.

3. In questo progetto, applichiamo un meccanismo di retroazione (*feedback*). Ossia, imponiamo correzioni alla stima corrente legate a quanto la soluzione precedente si discosta dal risultato desiderato. Quanto più drastica è la deviazione, tanto più drastica è la correzione; questo è un principio molto importante.
4. Un esempio specifico ci ha fornito gli appigli necessari per imbastire un modello generale.
5. La formula che abbiamo dedotto per calcolare la radice quadrata si può ricavare in altra maniera dalla formula di Newton.

$$x_{n+1} = x_n + \frac{f(x_n)}{f'(x_n)}$$

6. Per calcolare la radice n -esima (ovvero la x per cui $x^n = m$) possiamo usare la formula:

$$x_{i+1} = \left((n-1)x_i + \frac{m}{x_i(n-1)} \right) / n$$

7. È possibile allestire un'espressione polinomiale per ottenere valori di partenza ottimi per g1 (vedi E.G. Maursund, "Optimal starting values for Newton-Raphson calculation of \sqrt{x} ", *Communications ACM*, 10, 430-2 (1967)). La funzione radice quadrata in un sistema di calcolo può essere utilizzata migliaia di volte al giorno. In tal caso è conveniente spendere conoscenze matematiche per ricercare un algoritmo migliore; per queste routine usate di frequente, i costi di ricerca aggiuntivi ripagano più volte.

Problemi supplementari

- 3.1.1 Implementare l'algoritmo di calcolo della radice quadrata proposto all'inizio.
 3.1.2 La media geometrica è impiegata per esprimere una tendenza centrale. È definita come:

$$\text{M.G.} = \sqrt[n]{x_1 * x_2 * x_3 * \dots * x_n}$$

Sviluppare un algoritmo che, letti n numeri, ne faccia la media geometrica.

- 3.1.3 Progettare un algoritmo che determini l'intero il cui quadrato è il più prossimo ma non superi un intero assegnato.
 3.1.4 Progettare ed implementare un algoritmo che calcoli iterativamente il reciproco di un numero.

ALGORITMO 3.2 IL MINIMO DIVISORE DI UN INTERO

Problema

Assegnato un intero n , trovare un algoritmo che determini il suo divisore esatto più piccolo, diverso da uno.

Sviluppo dell'algoritmo

A prima vista, questo problema appare alquanto banale. Possiamo prendere l'insieme dei numeri $2, 3, 4, \dots, n$ e dividere di volta in volta n per ciascuno di tali numeri; appena incontriamo un numero che è divisore esatto di n , l'algoritmo può concludersi con successo. Tutto ciò è molto semplice; resta la domanda: è possibile progettare un algoritmo più efficiente?

Per partire nella nostra ricerca, esaminiamo l'insieme completo dei divisori di un certo numero. Assunto come esempio il numero 36, scopriamo che l'insieme completo dei suoi divisori è:

[2, 3, 4, 6, 9, 12, 18]

Sappiamo che un *divisore esatto* di un numero lo divide senza dar luogo a resti. Per il divisore 4 di 36 abbiamo:

36									
	4	4	4	4	4	4	4	4	4
	1	2	3	4	5	6	7	8	9

Ossia, in 36 sono contenuti 9 quattro; ne discende che anche il numero (più grande) 9 è un divisore esatto di 36:

$$\begin{array}{r} 36 \\ 4 \end{array} \rightarrow 9$$

$$\begin{array}{r} 36 \\ 9 \end{array} \rightarrow 4$$

e

$$\begin{array}{r} 36 \\ (9*4) \end{array} \rightarrow 1$$

Analogamente, scegliendo il divisore 3, scopriamo l'esistenza di un numero maggiore, 12, che è pure divisore di 36. Da questa analisi possiamo trarre la conclusione che i divisori esatti di un numero sono in numero pari.

Chiaramente, nel nostro esempio, non dovremo considerare né 9 né 12 come candidati ad essere il divisore minimo poiché entrambi sono in corrispondenza con un altro divisore più piccolo. Nell'insieme completo dei divisori di 36, osserviamo che:

fattore più piccolo		fattore più grande
2	è associato a	18
3	è associato a	12
4	è associato a	9
6	è associato a	6

Possiamo vedere che il divisore minimo (2) è associato al massimo (18), il più piccolo successivo (3) è associato al più grande immediatamente inferiore (12) e così via. Seguendo fino in fondo questo ragionamento

namento, concludiamo che l'algoritmo può terminare sicuramente quando si abbia la coppia di fattori comprendente:

- (a) il più grande dei fattori più piccoli, s ,
- (b) il più piccolo dei fattori più grandi, b .

In altre parole, vogliamo due valori s e b (con $s < b$) tali che:

$$s \cdot b = n$$

La situazione limite si verifica quando $s = b$, ovvero:

$$s \cdot s = n$$

Ne viene che non è necessario ricercare divisori di n oltre il valore della sua radice quadrata.

Questa considerazione è particolarmente significativa se c'è il rischio di avere a che fare con numeri primi molto grandi (un numero primo è un intero divisibile esattamente solo per 1 e per se stesso).

Questo limite della radice quadrata ci consente di arrestare la ricerca del divisore esatto assai prima di quanto non ci sarebbe altrimenti consentito se il numero in esame fosse un numero primo (p.es. per il numero primo 127 l'algoritmo può concludersi dopo sole 10 iterazioni).

Prima di ritenerci completamente soddisfatti del nuovo approccio, ci domandiamo se non si possa attuare ancora qualche miglioramento. L'insieme dei divisori che dovremo d'ora innanzi considerare è quello di tutti gli interi da 2 a n . Per il numero primo 127, i divisori da considerare sarebbero:

2, 3, 4, 5, 6, 7, 8, 9, 10, 11

Sappiamo che *tutti* i numeri pari sono divisibili per 2; ne viene che, se il numero in esame non è divisibile per 2, a maggior ragione non lo sarà per 4, 6, 8, 10, ... E perciò se il numero in esame è dispari, come candidati al minimo divisore dovremo considerare solo numeri dispari (3, 5, 7, 9, ...).

In effetti, l'esame di tutti i numeri dispari è inutile, ma non tratteremo qui quest'ultima miglioria poiché coinvolge il più complesso problema di generare numeri primi, che considereremo più avanti.

Possiamo ora riassumere la strategia globale dell'algoritmo:

1. Se il numero n è pari, il divisore minimo è 2 altrimenti
 - (a) si calcola la radice quadrata, r , di n ;
 - (b) finché non si trova un divisore esatto minore della radice quadrata di n ;
 - (b.1) Si prova la divisione con il prossimo numero della successione 3, 5, 7, ...

Resta soltanto da eseguire l'implementazione dei dettagli. I divisori da provare possono essere prodotti assegnando inizialmente 3 a d ed utilizzando:

$$d := d + 2$$

Per stabilire se un numero è un divisore esatto di un altro oppure no, possiamo verificare se c'è un resto dopo la divisione. Per questo possiamo usare la funzione **mod**. Se $n \bmod d = 0$, d è un divisore esatto di n .

Per la terminazione, possono essere valide direttamente due condizioni: l'algoritmo ha termine quando sono verificate una o entrambe le condizioni $n \bmod d = 0$ e $d \geq r$. Pertanto, in chiusura occorrerà un controllo per verificare se n ha effettivamente un divisore oppure no. Possiamo ora procedere alla descrizione dettagliata dell'algoritmo.

Descrizione dell'algoritmo

1. Stabilire l'intero n di cui è richiesto il minimo divisore.
2. Se n è pari restituire 2 come minimo divisore altrimenti
 - (a) calcolare la radice quadrata r di n ;
 - (b) inizializzare il divisore d con 3;
 - (c) finché il divisore non è esatto e non si è raggiunto il limite r
 - (c.1) produrre il prossimo dispari d ;
 - (d) se il valore corrente d è divisore esatto restituirlo come risultato altrimenti porre il risultato a 1.

Implementazione in Pascal

```
function sdivisor (n: integer): integer;
var d {current divisor and member of odd sequence},
    r {integer less than or equal to square root of n}: integer;
begin {finds the smallest exact divisor of an integer n, returns 1 if n
prime}
{assert: n > 0}
if not odd(n) then
  sdivisor := 2
else
  begin {terminate search for smallest divisor at sqrt (n)}
    r := trunc(sqrt(n));
    d := 3;
    {invariant: d = <r+1 and no odd integer in [3..d-2] exactly
    divides n}
  end;
end;
```

```

while ( $n \bmod d < 0$ ) and ( $d < r$ ) do  $d := d + 2$ ;
{assert:  $d$  is smallest exact divisor of  $n \wedge d = < r \vee (d = < r + 1) \wedge n$ 
is prime}
if  $n \bmod d = 0$  then
     $sdivisor := d$ 
else
     $sdivisor := 1$ 
end
end

```

Note di progetto

- Per determinare il divisore minimo di un numero n , l'algoritmo impiega al massimo un numero di iterazioni non maggiore di $\lceil \sqrt{n} \rceil / 2$. Se n è pari, non vengono eseguite iterazioni.
- Dopo l'-esima iterazione del loop di **while**, sono stati provati come divisori esatti di n i primi i membri della successione 3, 5, 7, 9,... Pertanto dopo l'-esima iterazione viene stabilito che l'-esimo membro della successione 3, 5, 7,... è il divisore minimo di n oppure che tale divisore non è compreso nei primi i membri di detta sequenza. La terminazione dell'algoritmo è garantita poiché ad ogni iterazione d viene incrementato di 2 e alla fine verrà soddisfatta la condizione $d \geq r$.
- Osserviamo che, come spesso avviene, la soluzione più ovvia di un problema non è la migliore. Dovremmo sempre ricordarlo quando progettiamo algoritmi.
- L'esame di un caso particolare ci dice molto sul progetto, anche se dobbiamo essere sempre attenti a non incappare in qualche caso speciale.
- Dovremmo sempre guardarci dal compiere passi inutili nei nostri algoritmi (p.es. considerare possibili divisori pari quando il numero n in ingresso è dispari non è buona cosa). Anche il nostro algoritmo nella forma in cui si presenta considera candidati non necessari (come 9, 15,...); teoricamente, dovremmo considerare candidati solo i numeri primi, ma questo è un problema già di per sé difficile (v. algoritmo 3.4).
- È possibile terminare il **while**-loop con un solo test utilizzando istruzioni aggiuntive di assegnazione. Ciò si traduce di solito in un'implementazione più efficiente; nel nostro caso il guadagno, se esiste, è marginale. Si osservi che non occorre assegnare ad una variabile $n \bmod d$; non useremo tuttavia ($d \geq \text{sqrt}(n)$) come condizione di terminazione poiché il ricalcolo di $\text{sqrt}(n)$ ad ogni iterazione è costoso e non necessario.
- Dijkstra ha fornito un metodo interessante ed efficiente per trovare il fattore primo più piccolo di un numero grande (vedi E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., pag. 143, 1976).

Applicazioni

Problemi di allocazione.

Problemi supplementari

- Modificare l'algoritmo in modo da non dover calcolare esplicitamente la radice quadrata di n .
- Progettare un algoritmo che produca l'elenco di tutti i divisori esatti di un intero n positivo assegnato.
- Progettare ed implementare un algoritmo che determini il minimo intero positivo, avente n o più divisori.
- Si può migliorare l'efficienza del nostro algoritmo di ricerca del divisore minimo generando una sequenza di d che escluda i multipli di 3 oltre ai multipli di 2. Implementare l'algoritmo che include tale miglioria.
- Tra gli interi dell'intervallo da 1 a 100 determinare quello che ha più divisori.
- Per trovare il massimo fattore f (non maggiore di n) di un numero dispari si può utilizzare un algoritmo dovuto a Fermat. Fermat afferma che per n valgono le seguenti relazioni:

$$\begin{aligned} n &= f * g \text{ con } f \leq g \\ n &= x^2 + t^2 \text{ con } 0 \leq t < x \leq n \\ \text{ove } x &= \lfloor (f+g)/2 \rfloor, y = \lfloor (g-f)/2 \rfloor \end{aligned}$$

L'algoritmo può essere implementato introducendo due variabili ausiliarie x' ed y' tali che:

$$\begin{aligned} x' &= 2x + 1 \\ y' &= 2y + 1 \end{aligned}$$

I valori dispari che queste due variabili possono assumere sono:

$$\begin{aligned} x' &: 2\lfloor \sqrt{n} \rfloor + 1, 2\lfloor \sqrt{n} \rfloor + 3, 2\lfloor \sqrt{n} \rfloor + 5, \dots \\ y' &: 1, 3, 5, 7, \dots \end{aligned}$$

Osservando che dalla somma della successione dei numeri dispari si ottiene una successione di quadrati, la differenza (errore) e , posta inizialmente pari a $\lfloor \sqrt{n} \rfloor^2 - n$ viene ridotta se positiva sottraendo valori successivi di y' finché non cambia segno o si annulla; se negativa, viene ridotta sommando i valori x' . Il processo ha fine quando $e = 0$. A questo punto il fattore f si ricava da:

$$f = \lfloor (x' - y')/2 \rfloor$$

Implementare l'algoritmo di Fermat.

ALGORITMO 3.3

IL MASSIMO COMUN DIVISORE DI DUE INTERI

Problema

Dati due interi positivi non nulli, n ed m , progettare un algoritmo che ne calcoli il massimo comun divisore (MCD).

Sviluppo dell'algoritmo

Al primo approccio con questo problema, ci accorgiamo che in certo qual modo è differente da altri che abbiamo incontrato. L'aspetto difficile della questione riguarda le relazioni tra i divisori di due numeri. Perciò il primo passo da compiere sembra quello di spezzare il problema e ricercare indipendentemente i divisori dei due numeri n ed m . Con in mano le due liste di questi divisori, scopriamo immediatamente che dobbiamo individuare l'elemento massimo comune ai due elenchi; questo elemento sarà il divisore *comune* massimo per i due interi n ed m .

Ci aspettiamo che questo algoritmo sia piuttosto dispendioso in termini di tempo per valori elevati di n ed m , in conseguenza del compito noioso di ricerca di tutti i fattori dei due interi.

Fu il filosofo greco Euclide che per primo, più di 2000 anni fa, divulgò una via migliore per la soluzione di questo problema. Il suo algoritmo, da allora noto come algoritmo di Euclide, fu probabilmente inventato da un predecessore di nome Eudoro. Anche gli antichi Cinesi conoscevano questo algoritmo.

Per avviarcici a nostra volta alla ricerca di un algoritmo migliore, una buona partenza consiste nell'esaminare attentamente che cosa si intende per massimo comun divisore di due interi. Il MCD di due interi è l'intero più grande che li divida entrambi esattamente, senza dar luogo a resti. Arriviamo al divisore comune di due interi partendo dalla considerazione di un divisore esatto di *un singolo* intero. Un divisore esatto di un intero è un intero più piccolo che lo divide in un insieme di parti uguali. Per esempio, il divisore 5 divide 30 in 6 parti uguali. In forma di diagramma, questo fatto si può rappresentare come segue:

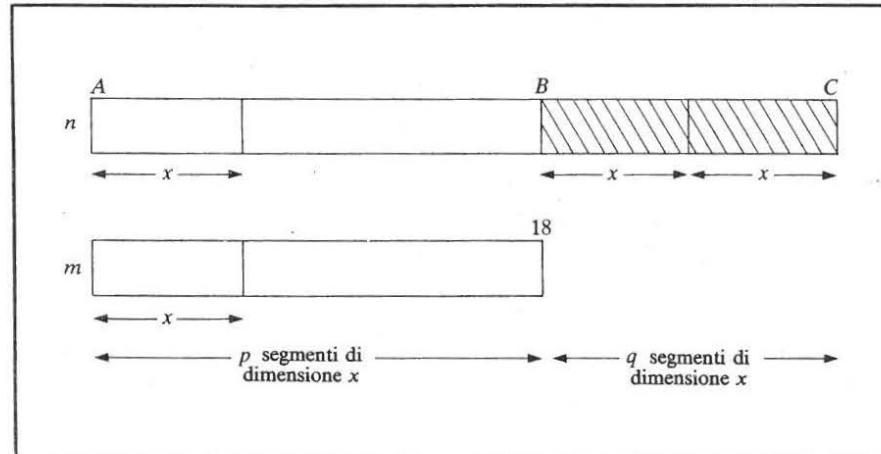
0	5	10	15	20	25	30
P_1	P_2	P_3	P_4	P_5	P_6	

$$P_1 = P_2 = P_3 = P_4 = P_5 = P_6$$

Fig. 3.2.
Rappresentazione
schemaica del
problema del MCD.

Estendiamo ora questa idea e la sua rappresentazione a due interi, 30 e 18, di cui cerchiamo il MCD x . Analizziamo la Fig. 3.2, vediamo che in presenza di un divisore comune ambedue i numeri n e m possono essere pensati suddivisi in parti di lunghezza x . Allineando a sinistra i segmenti rappresentativi di n e m , il tratto AB di n corrisponde all'intera lunghezza di m ; tuttavia il numero n supera m del tratto BC: che dire a proposito di questo segmento?

Se x è un divisore esatto sia di n che di m ed è contenuto esattamente in AB, anche BC deve essere diviso esattamente da c .



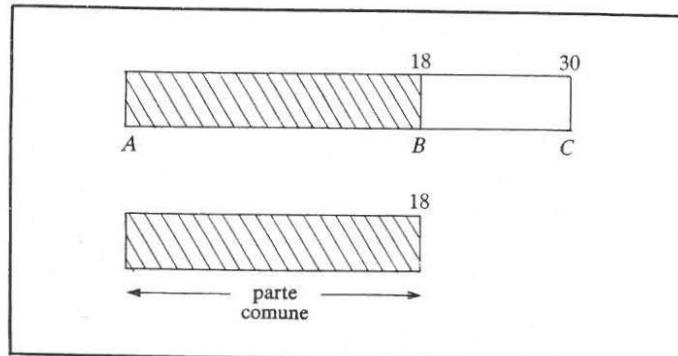
Dopo aver fatto questa osservazione, il problema successivo è trovare il modo di individuare il più grande dei divisori comuni condivisi da n e m . Considerando dapprima il problema più semplice, sappiamo che il massimo divisore di un numero singolo è il numero stesso (p. es. il massimo divisore di 30 è 30).

Nel nostro esempio abbiamo:

- (a) il massimo divisore di 30 è 30;
- (b) il massimo divisore di 18 è 18.

Ma il problema è trovare il massimo divisore *comune* a entrambi i numeri; chiaramente nessun numero maggiore di 18 può essere un candidato per il MCD, poiché non è contenuto esattamente in 18. Possiamo generalizzare questa affermazione dicendo che il MCD di due numeri non può essere maggiore del più piccolo dei due. La domanda successiva che ci dovremmo porre è: il MCD di due numeri può essere *uguale* al più piccolo dei due? (nel caso di 18 e 30 ciò non si verifica, ma considerando 12 e 36 scopriremmo che il MCD è 12). Possiamo così concludere che il più piccolo di due numeri n e m deve essere il limite superiore per il loro MCD. Dobbiamo ora decidere come continuare quando il più piccolo di due numeri n e m non è MCD.

Fig. 3.3.
Rappresentazione
schemaica del
problema del MCD
con la parte comune
in evidenza.



Per provare a rispondere a questa domanda torniamo al problema specifico di ricercare il MCD di 18 e 30. Abbiamo la situazione illustrata in Fig. 3.3. Sappiamo che il segmento BC deve essere diviso esattamente dal MCD; e poiché 18 non è MCD, il numero x che stiamo cercando dovrà essere inferiore a 18, nonché divisore esatto del segmento BC. Il massimo numero che divide BC è 12, ossia BC stesso; se deve essere il MCD di 18 e 30 deve essere anche un divisore esatto di entrambi questi numeri. Con questo passaggio, abbiamo ridotto il problema originale al sottoproblema di trovare il MCD di 12 e 18, come mostrato in Fig. 3.4.

Ripetendo per il problema ridotto le considerazioni precedenti, scopriamo che poiché 12 non è un divisore di 18 dobbiamo ricondurci ad un problema di MCD ancor più ridotto; abbiamo così la situazione di Fig. 3.5. In quest'ultimo caso il più piccolo tra 6 e 12 (ossia 6) è un divisore esatto di 12. Al raggiungimento di questa condizione, è risolto il problema corrente di MCD, nonché il problema originario.

Fig. 3.4.
Sottoproblema di
MCD da risolvere.

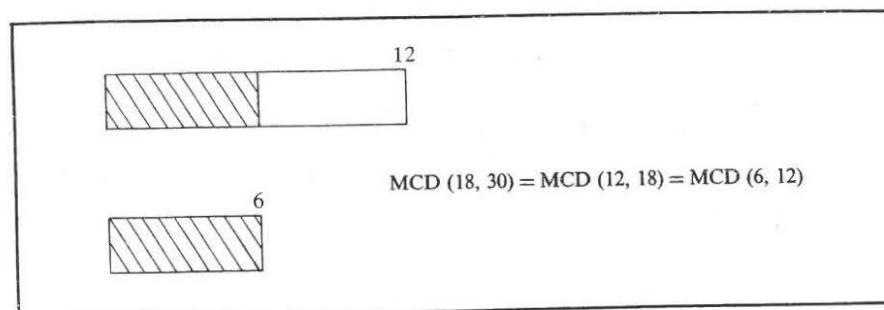
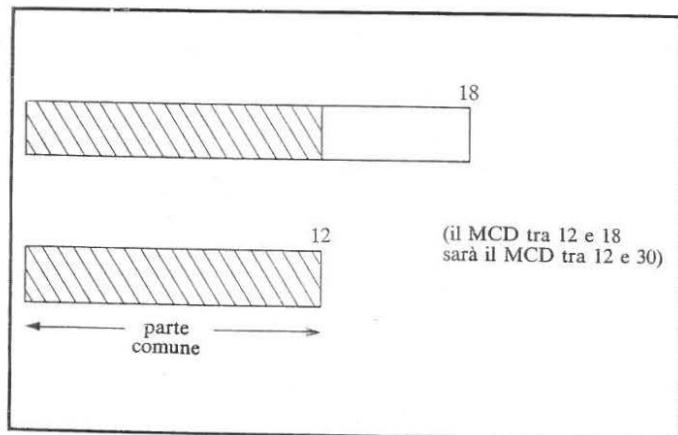


Fig. 3.5.
Problema di MCD
ancora più ridotto da
risolvere.

1. Dividere il più grande dei due numeri per il più piccolo.
2. Se il più piccolo è un divisore esatto del più grande il numero piccolo è il MCD altrimenti eliminare dal numero più grande la parte in comune col numero più piccolo e ripetere l'intera procedura con la nuova coppia di numeri.

Dobbiamo ora definire i dettagli per l'implementazione e la terminazione dell'algoritmo del MCD. Per prima cosa esaminiamo come stabilire se il numero più piccolo è un divisore esatto del più grande. Questa divisibilità si può individuare dall'assenza di resto nella divisione fra interi. La funzione **mod** ci consente di calcolare il resto risultante da una divisione fra interi. Possiamo usare:

$$r := n \bmod m$$

purché si sia precedentemente garantito che $n >= m$. Se r è nullo, m è il MCD. Se r non è nullo, risulta coincidente con la parte "non comune" tra n e m (p. es. $30 \bmod 18 = 12$). Si dà quindi il caso fortunato che la funzione **mod** ci fornisca proprio la parte di n necessaria per risolvere il nuovo problema ridotto di MCD. Inoltre, per definizione r sarà minore di m . Dobbiamo adesso imbastire il nostro costrutto iterativo con l'impiego della funzione **mod**. Per cercare di formulare tale costrutto, torniamo al problema del MCD (18,30).

Per il nostro esempio specifico, abbiamo:

$$\begin{aligned} r &:= 30 \bmod 18 = 12 && \text{passo (1)} \\ r &:= 18 \bmod 12 = 6 && \text{passo (2)} \\ r &:= 12 \bmod 6 = 0 && \text{passo (3)} \end{aligned}$$

"MCD"

L'esempio suggerisce che, ad ogni riduzione di dimensione del problema, l'intero più piccolo assume il ruolo del più grande ed il resto assume il ruolo dell'intero più piccolo.

La riduzione di dimensione del problema e lo scambio dei ruoli avvengono ripetutamente; l'esempio pertanto suggerisce che il meccanismo di calcolo del MCD possa attuarsi iterativamente con un ciclo della forma:

finché non si trova il MCD

- calcolare il resto della divisione dell'intero maggiore per il minore;
- assegnare all'intero minore il ruolo dell'intero maggiore;
- assegnare al resto il ruolo dell'intero minore.

Ora dobbiamo decidere in dettaglio come debba concludersi il loop. La discussione precedente ha stabilito che il divisore corrente è il MCD quando divide esattamente l'intero che costituisce il membro maggiore (o uguale) della coppia. Ne viene che possiamo usare questa condizione per terminare il ciclo:

finché il resto è non nullo

"continuare la ricerca del MCD".

Esaminando con un po' più di attenzione il meccanismo che abbiamo costruito, vediamo che ci sono due aspetti non ancora considerati. Innanzitutto, quando viene analizzata per la prima volta la condizione di terminazione del loop, non abbiamo ancora calcolato nessun resto. Un modo per ovviare a ciò è calcolare il resto per la prima coppia di numeri, prima di entrare nel ciclo. Ossia:

- Calcolare il resto per la prima coppia di interi;
- finché il resto è non nullo "continuare la ricerca del MCD".

Vediamo che con questo meccanismo, poiché il resto deve essere calcolato almeno una volta, abbiamo una situazione in cui la terminazione verrà esaminata dopo almeno un passaggio attraverso il loop. Ciò suggerisce per questo caso un meccanismo più semplice:

ciclicamente
"cercare il MCD"
fino a trovare resto zero.

L'altra considerazione da compiere è che, consegnando al loop la coppia di interi originale, non sappiamo quale sia il maggiore. Un modo per ovviare al problema è scambiare i due interi prima di iniziare il ciclo, se i loro ruoli non sono stati assegnati correttamente. Tale soluzione si presenta poco elegante, per cui vediamo che cosa accade se non eseguiamo lo scambio. Nel nostro esempio precedente, se iniziamo il loop con:

$$r := 18 \bmod 30$$

il resto diventa 18, il numero maggiore diventa 30 e il più piccolo 18. È accaduto che i ruoli dei due interi, il maggiore e il minore, si sono scambiati pervenendo allo scambio progettato in origine. Ne segue che non occorre preoccuparsi di decidere quale sia il maggiore dei due numeri di ingresso. Una considerazione finale: quale sarà la variabile contenente il MCD al termine del meccanismo? Rifacendoci alla descrizione precedente del meccanismo vediamo che nell'iterazione finale prima della conclusione, viene assegnato il ruolo di intero maggiore al numero più piccolo, che è il MCD. Dopo questa osservazione possiamo ora descrivere in dettaglio l'algoritmo.

Descrizione dell'algoritmo

- Fissare i due interi positivi e non nulli (maggiore e minore) di cui si cerca il MCD.
- Ciclicamente
 - calcolare il resto della divisione dell'intero maggiore per il minore;
 - assegnare all'intero minore il ruolo dell'intero maggiore;
 - assegnare al resto il ruolo dell'intero minore finché si ottiene resto zero.
- Restituire il MCD della coppia originale di interi.

Implementazione in Pascal

```
function gcd(n,m: integer): integer;
var r {remainder after integer division of n by m}: integer;

begin {computes the greatest common divisor for two positive
non-zero integers}
{assert: n>0^m>0}
repeat
  {compute next gcd candidate and associated remainder}
  r := n mod m;
  n := m;
  m := r;
until r=0;
{assert: n=gcd of original pair n, and m}
gcd := n
end
```

Note di progetto

- Il numero di iterazioni richieste dall'algoritmo del MCD dipende fortemente dai dati in ingresso e dal fatto che i due interi abbiano o meno un divisore comune maggiore di 1.

Una tipica situazione di “caso peggiore” si verifica quando la coppia originale è costituita da due numeri di Fibonacci adiacenti. In tal caso i resti successivi seguono a ritroso la successione di Fibonacci fino allo zero. Il numero di iterazioni è pertanto limitato superiormente dal numero di termini della successione di Fibonacci estesa a comprendere la coppia di interi originale. Se N è il più piccolo numero di Fibonacci che supera sia n che m , si può dimostrare che il numero di iterazioni è:

$$\lceil \log_{\Phi} (\sqrt{5N}) \rceil - 2 \text{ con } \Phi = 0.5(1 + \sqrt{5}).$$

2. È difficile dimostrare la correttezza dell’algoritmo che abbiamo implementato, benché concettualmente nitido, poiché esso non ha una conveniente relazione invariante che valga per tutto il calcolo. Operando i cambiamenti necessari per fare assumere al loop la forma:

```
repeat
    n := m;
    m := r;
    r := n mod m
until r := 0
```

Otteniamo un algoritmo la cui correttezza può essere più facilmente dimostrata. Partendo con, $m := a$ ed $r := b$ (dove a e b sono la coppia originale di numeri di cui è richiesto il MCD) abbiamo la seguente relazione invariante:

$$\text{MCD}(n, m) = \text{MCD}(a, b) \wedge r \geq 0$$

Al termine, avremo:

$$m = \text{MCD}(n, m) = \text{MCD}(a, b) \wedge r = 0$$

Possiamo concludere che l’algoritmo ha termine poiché $n \bmod m$ è sempre inferiore ad m e quindi r è strettamente decrescente, per cui alla fine sarà verificata la condizione $r = 0$.

3. Osserviamo che la soluzione del problema del MCD è ottenuta spezzando il problema iniziale in sottoproblemi che possono essere risolti dallo stesso meccanismo. Vedremo più avanti che questa è una tecnica assai potente che di solito suggerisce l’impiego della ricorsione.
4. Le definizioni giocano un ruolo importante nella soluzione di questo problema.
5. Diagrammi semplici sono spesso utili per scoprire gli algoritmi, come si è visto in questo caso.

Applicazioni

Riduzione di una frazione ai minimi termini.

Problemi supplementari

- 3.3.1 Implementare un algoritmo per il MCD che utilizzi un **while**-loop anziché un **repeat**-loop.
- 3.3.2 Progettare un algoritmo per il MCD che non utilizzi né la divisione né la funzione **mod**.
- 3.3.3 Progettare un algoritmo che determini il MCD di n interi positivi non nulli.
- 3.3.4 Se i due interi di cui si cerca il MCD contengono multipli di due, un modo migliore di procedere è quello di eliminare preventivamente da entrambi i multipli comuni di due, tenendone conto nel risultato. Quando un intero contiene multipli di due è meglio eliminare tali contributi prima di procedere col meccanismo standard del MCD. Introdurre queste idee in un algoritmo di MCD più efficiente.
- 3.3.5 Progettare un algoritmo che determini tutti i divisori primi comuni a due numeri. (*Suggerimento*: può essere utile l’algoritmo 3.5).
- 3.3.6 È noto che numeri di Fibonacci adiacenti non condividono divisori maggiori di 1 (sono cioè primi fra loro). Progettare un algoritmo che verifichi questa osservazione per i primi n numeri inerti.
- 3.3.7 Progettare un algoritmo che calcoli il minimo comune multiplo (mcm) di due interi positivi non nulli n e p . Il mcm è definito come il più piccolo intero m che sia diviso esattamente da n e da p .
- 3.3.8 Progettare un algoritmo che determini il minimo comune divisor diverso da 1 di due numeri interi positivi non nulli.
- 3.3.9 Date due frazioni a/b e c/d , progettare un algoritmo che ne calcoli la somma.

ALGORITMO 3.4 GENERAZIONE DI NUMERI PRIMI

Problema

Progettare un algoritmo che individui tutti i numeri primi tra i primi n interi positivi.

Sviluppo dell'algoritmo

La generazione efficiente di numeri primi è un problema tuttora aperto. Considereremo qui il problema più ristretto di generare tutti i numeri primi contenuti nei primi n interi. Un numero primo è un intero positivo che è divisibile esattamente solo per 1 e per se stesso. Ecco l'inizio della sequenza dei numeri primi:

2 3 5 7 11 13 17 19 23 29 31 37 ...

Tutti i numeri primi, ad eccezione di 2, sono dispari.

Come punto di partenza per lo sviluppo del nostro generatore di numeri primi, analizziamo come si possa stabilire se un certo numero è primo o no. Nel far ciò avremo presente un esempio particolare: il numero 13. La definizione di numero primo ci fornisce l'avvio richiesto; sappiamo che se il numero in esame è primo non avrà altri divisori che se stesso e l'unità. Questo ci suggerisce di verificare se 13 è primo dividendolo di volta in volta per 2, 3, 4, 5...12. Se uno di tali numeri divide 13 senza resto sapremo che 13 non può essere primo.

Pertanto la verifica di primalità del numero 13 richiede undici chiamate della funzione *mod*. È facile vedere che all'aumentare di n il costo per effettuare queste divisioni e i confronti diventa proibitivo. Dobbiamo quindi cercare una strada per migliorare l'efficienza del nostro algoritmo. Una piccola riflessione ci mostra l'esistenza di molte possibilità aperte. Innanzitutto, possiamo cercare di ridurre al minimo i numeri da analizzare; in secondo luogo possiamo cercare di migliorare l'efficienza dell'analisi di primalità per un certo numero.

Seguendo il primo suggerimento, sappiamo che ad eccezione del 2 non dobbiamo esaminare *nessuno* dei numeri pari. Partendo con x uguale ad 1 ed usando l'incremento:

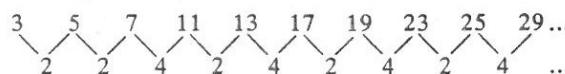
$$x := x + 2$$

otteniamo la sequenza di numeri:

3, 5, 7, 9, 11, 13, 15, 17, ...

Per n grande ci restano ancora troppi numeri da considerare.

Finora abbiamo eliminato i numeri divisibili per 2: possiamo estendere la stessa tecnica ad eliminare i numeri divisibili per 3, 5, e così via? Per esplorare questa idea cominciamo a scrivere la successione dei numeri dispari da cui sono stati eliminati i multipli di 3. Abbiamo:



Dopo il 5, abbiamo una successione alternata di differenze 2 e 4, che dovremmo essere in grado di produrre. Senza indulgere su questo, è facile vedere che, per un valore iniziale di $dx = 4$, il seguente costrutto

ha il comportamento desiderato:

$$dx := \text{abs}(dx - 6)$$

Questo accorgimento ci permette di eliminare i due terzi dei candidati nella ricerca dei numeri primi.

Ci domandiamo adesso se si possano eliminare i multipli di 5 in maniera analoga. La risposta è affermativa ma con qualche leggera complicazione in più. Questa linea d'approccio non sembra dopo tutto così fruttuosa. Tuttavia ne possiamo dedurre che una maniera per generare numeri primi è semplicemente quella di elencare tutti i numeri interi e cancellare via via dall'elenco i multipli di 2, 3, 5, 7, 11, ecc.

$$(a) 2 3 \cancel{5} \cancel{7} \cancel{9} \cancel{10} \cancel{11} \cancel{12} \cancel{13} \cancel{14} \cancel{15} \cancel{16} \cancel{17} \cancel{18} \cancel{19}$$

cancellazione dei multipli di 2

$$(b) 2 3 5 7 \cancel{9} 11 13 \cancel{15} 17 19$$

cancellazione dei multipli di 3

$$(c) 2 3 5 7 11 13 17 19$$

Al passo (c) in questa occasione siamo rimasti con tutti i numeri primi inferiori a 20. Se fossimo partiti con una lista molto più ampia ed avessimo successivamente cancellato i multipli di 2, 3, 5, 7, 11... i numeri rimasti sarebbero stati solo numeri primi. Questa idea per produrre numeri primi risale a un antico matematico greco, Eratostene, ed è nota comunemente come "crivello di Eratostene".

Quando è necessario generare molti numeri primi c'è un problema nell'implementare direttamente questo metodo: può essere richiesta una quantità di memoria proporzionale all'estensione degli interi fino ad n .

Un'ulteriore indagine sul meccanismo della "cancellazione" mostra che, per n grande, molti elementi vengono cancellati più volte (p. es. 15 è cancellato due volte perché multiplo di 3 e anche di 5).

Queste osservazioni ci suggeriscono che è meglio esaminare se esiste qualche via per abbattere la quantità di memoria e di confronti necessari per individuare un set di numeri primi.

Abbiamo visto in precedenza che per determinare che 13 è numero primo esso andrebbe diviso per l'insieme dei numeri 2, 3, 4,..., 11, 12. Analizzando attentamente questo meccanismo e ricordando come si concludeva la ricerca del minimo divisore di un numero (algoritmo 3.2), vediamo che non è necessario esaminare divisori oltre $\lfloor \sqrt{13} \rfloor$. Se un numero x ha un divisore esatto, esisterà certamente un suo fattore minore o uguale a \sqrt{x} può essere un'operazione abbastanza efficiente (p. es. se x è all'incirca 1000, sono richieste per giudicarne la primalità solo le divisioni per i numeri primi fino al 31: 10 divisioni in tutto). La divisione per numeri composti non è mai richiesta, poiché se un numero composto è un divisore esatto, lo è a maggior ragione un suo fattore primo più piccolo.

A questo punto possiamo proporre una struttura di base per il nostro algoritmo:

```

while x < n do
begin
    (a) produrre il prossimo termine x col costrutto
        dx := abs(dx - 6),
    (b) esaminare se x è primo usando tutti i numeri pri-
        mi  $\leq \sqrt{x}$ 
    (c) se si trova un numero primo minore di  $\sqrt{x}$  lo si
        memorizza per l'esame successivo con valori mag-
        giori di x.
end

```

Per verificare la primalità di tutti gli interi fino ad n occorre ricordare tutti i primi fino a \sqrt{n} .

Ogni volta che si assume un nuovo x da esaminare occorre garantire l'esistenza del set appropriato di divisori primi da provare.

Facendo qualche esempio otteniamo:

intervallo per x	divisori primi richiesti
$2 \leq x < 9$	2
$9 \leq x < 25$	2, 3
$25 \leq x < 49$	2, 3, 5
$49 \leq x < 121$	2, 3, 5, 7
\vdots	\vdots

Il nostro metodo di generazione dei valori x esclude tutti i multipli di 2 e di 3. Pertanto per valori di x inferiori a 25, il nostro metodo produrrà solo numeri primi. Appena x raggiunge 25 occorrerà il divisore 5; raggiungendo 49 dovrà essere incluso il divisore 7 e così via.

Partendo con:

$p[1] := 2; p[2] := 3; p[3] := 5;$

e

$plimsq := 25; limit := 3$

possiamo includere la seguente istruzione condizionale *prima* di esaminare la primalità di ciascun x :

```

if x >= plimsq then
begin
    limit := limit + 1;
    plimsq := sqr(p[limit])
end

```

Poiché la differenza tra i quadrati di due numeri primi adiacenti è sempre maggiore di 4, massimo incremento dato a x , basta incrementare *limit* di 1 soltanto.

Potrebbe presentarsi a questo punto il rischio di non avere numeri primi da confrontare quando x diviene molto grande. Per evitarlo occorre garantire che il programma termini se si esauriscono i divisori per il test di primalità; per rendere le cose più facili, utilizzeremo un risultato della teoria dei numeri che dice:

$$p[i] < p[i-1]^2$$

Questa condizione è sufficiente per garantire che non potremo mai restare a corto di divisori.

Una volta stabilito l'insieme appropriato di divisori richiesto per esaminare un dato x , il passo successivo è eseguirne l'esame. Per far ciò possiamo usare un ciclo che provi su x tutti i divisori primi con indice inferiore a *limit*.

Una breve riflessione mostra che esistono due condizioni per cui il loop può terminare:

1. si è trovato un divisore esatto di x : x non è primo;
2. esprimersi come $x = pkq$, con $k \geq 1$ e q numero primo 1.

Utilizzando la funzione **mod** per verificare l'esattezza della divisione ed il resto *rem* per assegnare la condizione logica *prime*, otteniamo:

```

j := 3; prime := true;
while prime and (j < limit) do
begin
    rem := x mod p[j];
    prime := rem <> 0;
    j := j + 1
end

```

Possiamo prevedere che questo loop sarà usato intensamente per valori grandi di n e pertanto nell'implementazione pratica sarà meglio ridurre i due test del ciclo ad uno solo eliminando il confronto $j < limit$. Ciò si può fare sostituendo temporaneamente all'elemento $p[limit]$ una "sentinella" uguale ad x ed eseguendo su j un test al di fuori del loop. Lo lasciamo come esercizio per il lettore.

Adesso occorre soltanto verificare se il loop di riconoscimento ha stabilito che x è un numero primo. In caso affermativo, se x è inferiore a \sqrt{n} viene memorizzato, altrimenti può essere direttamente stampato.

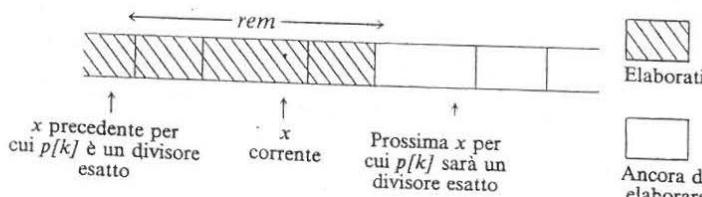
La parte centrale dell'algoritmo, in forma più dettagliata, è quindi (dopo idonea inizializzazione di *limit*, *dx* e *plimsq*):

```

while x < n do
begin
  dx := abs(dx - 6);
  x := x + dx;
  if limit ≤ i then
    if x ≥ plimsq then
      begin {include next prime as divisor}
        limit := limit + 1;
        if limit ≤ i then
          plimsq := sqr(p[limit])
      end;
  j := 3;
  prime := true;
  while prime and (j < limit) do
    begin {test next number x for primality}
      rem := x mod p[j];
      prime := rem <> 0;
      j := j + 1
    end
  if prime then "write out x and save if necessary"
end

```

Abbiamo ora le basi di un algoritmo funzionante per la ricerca di numeri primi. Per n grande, nel loop interno si spenderà una quantità di tempo ad eseguire costose divisioni (nella funzione `mod`). Pertanto ci chiediamo: esiste la possibilità di ridurre queste divisioni di controllo? Ogni divisione ci fornisce zero o un intero non nullo. Un resto non nullo ci dice che il particolare numero primo usato non è un divisore di x . Ma il suo valore ci segnala anche il prossimo valore nel campo di variabilità di x per il quale divisore corrente $p[k]$ sarà un divisore esatto. Per esempio:



Possiamo utilizzare questa informazione per “cancellare” in anticipo i prossimi valori di x divisibili per $p[k]$. Per far ciò occorre disporre di un ulteriore array $out[1..\sqrt{n}]$ in cui memorizzare i valori cancellati in anticipo. La cancellazione si ottiene con:

```

nxtout := p[k] - rem;
out[nxtout] := false

```

Quindi esamineremo l’array out prima di verificare la primalità di un certo numero; se è già stato cancellato non occorre eseguire alcun test. Analizzando e verificando questa idea si scopre che il numero complessivo di test da eseguire si riduce di un fattore 4 o 5.

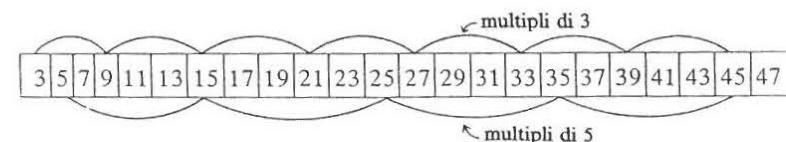
Questo sembra un miglioramento utile. Sfortunatamente tutte le volte che si incontra un numero primo x occorre provare per esso tutti i divisori primi minori di \sqrt{x} . Per n grande, stabilire i primi con questo metodo risulta costoso sul piano computazionale.

Perciò è forse meglio tornare al crivello di Eratostene e vedere se se ne può ridurre in qualche modo il grande costo di memoria. Il vantaggio del metodo del crivello è che non richiede alcuna costosa divisione per stabilire i numeri primi.

Per generare tutti primi fino ad n dobbiamo cancellare i multipli inferiori ad n di tutti i numeri primi minori o uguali a \sqrt{n} . Vogliamo far ciò impiegando assai meno di n locazioni di memoria. Come fare? L’algoritmo del crivello nella sua formulazione iniziale richiede la memorizzazione dei multipli dei numeri primi 5, 7, 11,... \sqrt{n} in una memoria di dimensione n . Per esempio, per $n=5$ abbiamo i multipli:

5, 10, 15, 20, 25, 30, 35,...

Osserviamo in questo insieme che un multiplo su due è pari e pertanto non può essere candidato ad essere numero primo. Ogni nuovo membro della successione “multipli di 5” si ottiene sommando il numero primo iniziale ai suoi multipli. Utilizzando n locazioni di memoria, per $n < 49$ occorre cancellare soltanto i multipli di 3 e di 5. In questo caso abbiamo:



I numeri rimasti sono numeri primi. Col metodo del crivello, occorre preventivamente cancellare tutti i multipli, eseguendo poi una passata di tutto l’array per prelevare i numeri non cancellati.

È evidente che una volta cancellato il 9, lo spazio occupato da 3, 5, 7, 9 non serve più poiché a questo punto conosciamo tutti i numeri primi minori di 9. Analoga argomentazione vale quando scopriamo che il 15 è stato cancellato. Questo ci suggerisce che, se riusciamo in qualche modo ad eseguire la cancellazione in maniera localizzata, non saremo costretti a pagare un elevato costo di memoria. Sappiamo che per $n < 49$ dobbiamo considerare solo multipli di 3 e di 5 (assumendo che i multipli di 2 siano già automaticamente tralasciati). Occorrerà in tal caso lo spazio per *due sole* variabili, volendo generare in successione ordinata tutti i multipli di 3 e di 5, vale a dire $3*3, 3*5, 3*7, 5*5, 3*9, \dots$

Un modo per farlo è disporre di una variabile x (come nel metodo precedente) che assuma tutti i possibili valori dispari. Ogni volta che x è incrementata occorrerà incrementare i multipli di 3 e/o di 5 a patto che si mantengano *inferiori* ad x . Il passo successivo è quello di verificare se i multipli correnti di 3 e 5 coincidono con x . Se nessuno dei due è uguale ad x , x deve essere un numero primo. Nel caso generale questo confronto deve essere eseguito con i multipli di tutti i numeri primi minori o uguali a $\lfloor \sqrt{x} \rfloor$.

Se $linus$ corrisponde all'indice del massimo numero primo minore o uguale a $\lfloor \sqrt{x} \rfloor$, il test di primalità per il valore corrente di x sarà:

```
while prime and (j<limit) do
begin
  while multiple[j]<x do multiple[j] := multiple[j]+p[j]*2;
  prime := x <> multiple[j];
  j := j+1
end
```

ove l'array *multiple* contiene ad ogni passo i multipli di tutti i primi $\leq \lfloor \sqrt{x} \rfloor$. Si usa $2*p[k]$ invece di $p[k]$ per evitare di generare i multipli pari come candidati. Questo metodo meno costoso di verifica può sostituire il metodo di verifica mediante divisione impiegato nell'algoritmo precedente. Grazie al modo d'impiego dell'array *multiple*, alla lista dei multipli viene aggiunto un nuovo primo soltanto quando viene raggiunto il suo quadrato. Esso viene incluso con il confronto.

```
if limit<=i then
  if x>=plimsq then
    begin
      multiple[limit] := plimsq;
      limit := limit+1;
      if limit<=i then
        plimsq := sqr(p[limit])
    end
```

Con queste modifiche all'algoritmo precedente, abbiamo adesso un metodo di produzione dei numeri primi che evita la verifica mediante divisione ma ottiene gli stessi risultati. Il prezzo da pagare è la memoria addizionale per immagazzinare i multipli dei numeri primi minori di \sqrt{n} . L'algoritmo è esposto in dettaglio nel seguito.

Descrizione dell'algoritmo

1. Inizializzare e stampare i primi tre numeri primi. Inizializzare il quadrato del terzo numero primo.
2. Assegnare il valore iniziale 5 ad x

3. Finché x è minore di n eseguire quanto segue:
 - (a) attribuire ad x il prossimo valore escludendo i multipli di due e di 3;
 - (b) se non si è superata la fine della lista dei multipli.
 - (b.1) se $x \geq \sqrt{n}$ al quadrato del numero primo più grande
 - (1.a) includere tale quadrato nella lista dei multipli,
 - (1.b) aggiornare il quadrato col valore del quadrato del numero primo successivo $> \sqrt{x}$;
 - (c) finché non si è stabilito se x è primo
 - (c.1) finché il multiplo corrente è minore di x , incrementarlo di un valore pari al doppio del numero primo corrente,
 - (c.2) eseguire la verifica di primalità confrontando x con il multiplo corrente;
 - (d) se x è primo
 - (d.1) stampare x e se è minore di \sqrt{n} memorizzarlo.

Implementazione in Pascal

```
procedure primes (n: integer);
const np = 100;
var multiple: array[1..np] of integer; {multiples of primes}
  p: array[1..np] of integer; {primes up to sqrt (n)}
  i {index for primes saved},
  j {index of primes and multiple array},
  limit {upper index for primes less than sqrt(x)},
  plimsq {square of largest prime included so far},
  rootn {truncated sqrt(n)},
  dx {increment either 2 or 4 to avoid multiples of 3},
  x {current candidate for prime test}: integer;
  prime: boolean;
begin
  {assert: n>1}
  p[1] := 2; p[2] := 3; p[3] := 5; i := 3;
  if n<5 then for j := 1 to (n+1) div 2 do writeln (p[j])
  else
    begin
      for j := 1 to 3 do writeln (p[j]);
      x := 5; plimsq := 25; limit := 3; dx := 2;
      rootn := trunc(sqrt(n));
      {invariant: after current iteration all primes in [2..x] have been
      written out & x=<n+2}
      while x < n do
        begin {test x for primality}
          x := x + dx;
          dx := abs(dx - 6);
          if limit<=i then
            if x>=plimsq then
              begin
```

```

multiple[limit] := plimsq;
limit := limit + 1;
if limit <= i then
    plimsq := sqr(p[limit])
end;
prime := true;
j := 3;
{invariant: after jth iteration x not divisible by primes
p[1..j-1] \& j = < limit \& x < p[limit] * p[limit]}
while prime and (j < limit) do
    begin {test x by comparing with all multiples of primes}
        while multiple[j] < x do
            multiple[j] := multiple[j] + p[j] * 2;
        prime := x <> multiple[j];
        j := j + 1
    end;
    {assert: (j=limit \& x is prime) \vee (j <= limit \& x not prime)}
    if prime then
        begin
            writeln(x);
            if x <= rootn then
                begin
                    i := i + 1;
                    p[i] := x
                end
        end
    end
end;
{assert: all primes in range [2..n+2] have been written out}

```

Note di progetto

- Nei primi n interi positivi vi sono circa $n/\log n$ numeri primi. L'analisi dell'algoritmo finale è piuttosto complessa e pertanto non verrà sviluppata in questa sede; il numero di addizioni richieste per determinare i numeri primi entro i primi n interi positivi è essenzialmente $(n^{3/2}/\log^3 n)$.
- Dopo il passaggio corrente attraverso il **while-loop** più esterno, del numero x si è giudicata la primalità, e tutti gli elementi dell'array di posto inferiore a j (con l'eccezione delle posizioni 1 e 2) sono maggiori o uguali ad x ; inoltre x è minore di $plimsq$. Dopo il passaggio corrente, la variabile *prime* risulterà vera se x è primo, falsa in caso contrario. Nel **while-loop** interno di verifica della primalità, dopo il passo j -esimo, i primi $(j - 1)$ elementi dell'array *multiple* saranno $\geq x$ e la variabile *prime* sarà vera se nessuno dei primi $j - 1$ numeri primi è multiplo o divisore esatto di x . Il **while-loop** più interno mantiene la relazione invariante $multiple[j] \geq x$ dopo il passaggio $(j - 1)$ -esimo attraverso il loop di analisi di primalità.

Il loop più esterno ha termine poiché la differenza tra x ed n

diminuisce di almeno 2 ad ogni passaggio attraverso il ciclo stesso.

Il loop di verifica di primalità ha termine poiché la differenza tra j e *limit* diminuisce di 1 ad ogni passaggio per il ciclo. Il **while-loop** più interno che interessa i multipli ha termine poiché *multiple[j]* è aumentato dell'intero positivo $2*p[j]$ ad ogni iterazione e la condizione di terminazione è prima o poi raggiunta. Poiché tutti i loop hanno termine l'intero processo avrà termine. L'algoritmo funziona correttamente per valori di $n > 1$; non si è tenuto conto tuttavia del fatto che n possa eccedere la dimensione della parola (del computer) o che vengano superati i limiti per l'array.

- In questo problema abbiamo visto come un vantaggio nell'efficienza di calcolo sia ottenuto mediante l'aggiunta di memoria. In genere, le divisioni sono costose e pertanto nelle applicazioni indirizzate al calcolo si dovrebbe tentare di limitarle.
- La maggior parte del tempo di calcolo in questo problema si spende nel loop di verifica di primalità; pertanto andrebbe fatto ogni sforzo per mantenere tale loop il più semplice ed efficiente possibile. In pratica sarebbe meglio usare in tale loop un solo test di condizione (vedi discussione).

Problemi supplementari

- Il loop di verifica di primalità può essere semplificato e reso più veloce, a prezzo di un po' di memoria in più. Questo implica lo spostamento all'esterno del loop della fase di aggiornamento dell'array *multiple*. Si cerchi di modificare l'algoritmo per introdurre questa miglioria.
- È possibile implementare un algoritmo di "setacciamento", che cancelli ciascun numero non primo una volta soltanto anziché molte; l'algoritmo si fonda sull'idea che ogni x non primo può esprimersi come $x = p^k q$, con $k \geq 1$ e q numero primo $\geq p$. Si provi ad implementare questo algoritmo (vedi D. Gries e J. Misra, "A linear sieve algorithm for finding prime numbers", Comm. ACM21, 999-1003 (1978)).
- Un'altra interessante successione di numeri si ottiene a partire dalla lista completa degli interi 1, 2, 3, ..., N ; da tale lista si ricava una nuova lista, eliminando un numero *ogni due*; da questa si elimina un numero *ogni tre* e si ottiene ancora un'altra lista; da questa si elimina un numero *ogni quattro*, e il procedimento continua. I numeri rimasti al termine del processo si chiamano *numeri fortunati*; i primi sette di essi sono 1, 3, 7, 9, 13, 15, 21. Progettare un algoritmo che scriva i numeri fortunati compresi tra i primi n interi.
- I numeri primi più grandi che si conoscano sono i numeri primi di Mersenne. Essi hanno la forma $2^p - 1$ con p numero primo. Per verificare se un numero di questa forma è primo si può impiegare un test, detto di Lucas, formulato come segue. Se $p > 2$, il numero $2^p - 1$ è primo solo se $l_{p-2} = 0$, essendo la successione degli l generata da:

$$l_0 = 4, l_{i+1} = (l_i^2 - 2) \pmod{2^p - 1}.$$

Progettare un algoritmo che produca i numeri primi di Mersenne compresi tra i primi n interi.

ALGORITMO 3.5 CALCOLO DEI FATTORI PRIMI DI UN INTERO

Problema

Ogni intero può essere espresso come il prodotto di numeri primi. Progettare un algoritmo che calcoli tutti i fattori primi di un intero n .

Sviluppo dell'algoritmo

L'analisi dell'enunciato del problema ci suggerisce che:

$$n = f_1 * f_2 * f_3 * \dots * f_k \quad \text{con} \quad n > 1 \quad \text{e} \quad f_1 \leq f_2 \leq \dots \leq f_k$$

Gli elementi f_1, f_2, \dots, f_k sono tutti numeri primi. Applicando la definizione ad alcuni esempi specifici abbiamo:

$$\begin{aligned} 8 &= 2 * 2 * 2 \\ 12 &= 2 * 2 * 3 \\ 18 &= 2 * 3 * 3 \\ 20 &= 2 * 2 * 5 \\ 60 &= 2 * 2 * 3 * 5 \end{aligned}$$

Un approccio immediato alla soluzione di questo problema di fattorizzazione consiste nel partire col divisore 2, continuando a ridurre n di un fattore 2 finché questo non è più un divisore esatto. Si passa quindi al divisore 3, ripetendo il procedimento di riduzione, e così via, finché n non si è ridotto ad 1.

Analizziamo il caso di $n = 60$.

Contrassegnando con un asterisco i tentativi di divisione falliti, abbiamo:

2	2	2	3	3	4	5
60	30	15*	15	5*	5*	5

Possiamo trarre diverse considerazioni da questo esempio.

Innanzitutto, 2 è il solo numero pari che occorre provare. In effetti, ponendo maggior attenzione alla definizione originale, osserviamo che come candidati alla divisione andrebbero presi *soltanto numeri primi*; il nostro approccio attuale mette in campo una quantità di divisori inutili. Pertanto possiamo cominciare a vedere che la generazione di un insieme di numeri primi sarà parte integrale del nostro algoritmo; dai lavori precedenti sui numeri primi ed i divisori minimi (algoritmi 3.2 e 3.4) sappiamo che tutti i fattori primi di n dovranno essere minori o uguali a \sqrt{n} . Questo ci suggerisce che forse dovremmo produrre una lista dei numeri primi fino a \sqrt{n} prima di addentrarci nel procedimento di ricerca dei fattori primi di n . Un'analisi più approfondita mostra che c'è un difetto in questa strategia, dal momento che i fattori primi possono comparire ripetutamente nella fattorizzazione che stiamo cercando; ne consegue che precalcolando tutti i primi fino a \sqrt{n} possiamo finire col calcolare molti più divisori di quanti non siano richiesti dal problema attuale. (Come esempio limite, se n fosse 1024, calcoleremmo i numeri primi fino a 32, mentre in effetti il massimo divisore primo di 1024 è soltanto 2).

Una strategia migliore e più economica è pertanto quella di calcolare i divisori primi man mano che questi sono necessari; per far ciò possiamo includere una versione modificata del crivello di Eratostene, precedentemente sviluppato. Come nel nostro algoritmo precedente, non appena scopriamo che n è primo possiamo concludere. A questo punto riassumiamo i progressi fatti; la descrizione ad alto livello della parte centrale dell'algoritmo è:

```
while n non è ancora primo do
    begin
        (a) se nxtprime è divisore di n, salvare nxtprime come fattore e dividere n per nxtprime
            altrimenti ricercare il numero primo successivo,
        (b) provare nxtprime come divisore di n.
    end
```

Dobbiamo ora pensare ad implementare il test di "non primalità" per il loop esterno. La tecnica impiegata in precedenza era la divisione intera ed il confronto del resto con zero. Quest'idea può essere applicata ancora: sappiamo anche che il procedimento può arrestarsi non appena il divisore primo che stiamo usando diventa maggiore di \sqrt{n} .

All'inizio, quando i divisori primi che impieghiamo sono molto più piccoli di \sqrt{n} , sappiamo che il test deve continuare. Nel far ciò vogliamo evitare di calcolare ripetutamente la radice quadrata di n ; ogni volta che eseguiamo la divisione:

$$n \text{ div } \text{nxtprime} \quad (\text{p. es. } 60 \text{ div } 2)$$

sappiamo che il processo deve continuare a meno che il quoziente q di tale divisione non risulti inferiore a nxtprime .

A quel punto avremo:

$$(\text{nxtprime})^2 > n$$

che indicherà che n è primo. Pertanto le condizioni per cui non si sa ancora se n è primo sono:

- (a) divisione esatta (ossia $r := n \text{ div } \text{nxtprime} = 0$),
- (b) quoziente maggiore del divisore (ossia $q := n \text{ mod } \text{nxtprime} > \text{nxtprime}$).

Il verificarsi di una delle due condizioni precedenti è sufficiente per richiedere che il test prosegua.

Adesso dobbiamo analizzare come si concluderà l'algoritmo. Se-
guendo il processo di fattorizzazione per un certo numero di esempi,
scopriamo che l'algoritmo può terminare in due modi. Un modo è quando n viene alla fine ridotto ad 1; ciò si verifica quando il fattore primo maggiore è presente più di una volta (p. es. nel caso di 18, fattorizzato come $2 \cdot 3 \cdot 3$). L'altra situazione possibile è quando si termi-
na con un fattore primo che capita una volta soltanto (p. es. 70 è fattorizzato come $2 \cdot 5 \cdot 7$); in questo caso abbiamo una condizione di terminazione in cui $n > 1$. Perciò, a conclusione del loop, dobbiamo determinare quale delle condizioni di terminazione si verifica e aggiu-
stare di conseguenza i fattori.

Le ultime considerazioni riguardano le condizioni iniziali e la gene-
razione dinamica dei numeri primi, via via che sono richiesti. Poiché
abbiamo appena considerato il problema della generazione dei numeri
primi, supporremo la disponibilità di una funzione che, ricevuto come
argomento un certo numero primo, restituisce il successivo. A tale
scopo si può facilmente adattare il crivello di Eratostene; nel nostro
esempio supporremo di disporre della procedura *erathostenes* che resti-
tuisce *nxtprime*. Possiamo enunciare ora nel dettaglio l'algoritmo di
fattorizzazione.

Descrizione dell'algoritmo

1. Porre n uguale al numero di cui si cercano i fattori primi.
2. Calcolare il resto r ed il quoziente q per il primo divisore *nxtprime* = 2.
3. Finché non si è deciso che n è primo
 - (a) se *nxtprime* è divisore esatto di n
 - (a.1) salvare *nxtprime* come fattore f ,
 - (a.2) dividere n per *nxtprime*,
 - altrimenti
 - (a.'1) ricavare il prossimo primo dal crivello di Eratostene.
- (b) calcolare il quoziente q e il resto r per i valori correnti di n e *nxtprime*.
4. Se n è maggiore di 1 aggiungere n alla lista dei fattori primi f .
5. Restituire i fattori primi f del numero n originale.

Implementazione in Pascal

```
procedure primefactors (var f: nelements; var i: integer; n: integer);
var
  q {quotient of n div nxtprime},
  r {remainder of n div nxtprime},
  nxtprime {next prime divisor to be tested}: integer;
  d: array[1..100] of integer; {multiples array for sieve}
begin {computes the prime factors f of n by division by successive
primes}
  {assert: n > 1}
  nxtprime := 2;
  q := n div nxtprime;
  r := n mod nxtprime;
  i := 0;
  {invariant: after current iteration f [1..i] will contain all prime factors
  (including repeats) < nxtprime
  ^one or more contributions of nxtprime if it is a factor}
  while (r=0) or (q>nxtprime) do
    begin {record factor if exact divisor or get next prime}
      if r=0 then
        begin {exact divisor so save prime factor and reduce n}
          j := i+1;
          f[i] := nxtprime;
          n := q
        end
      else erathostenes(d,nxtprime); {get next prime}
      q := n div nxtprime;
      r := n mod nxtprime
    end;
  end;
  if n>1 then
    begin {n is a prime factor}
      i := i+1;
      f[i] := n
    end
  {assert: f[1..i] will contain all prime factors (including repeats) of
original n}
end
```

Note di progetto

1. Il costo di calcolo di questo algoritmo può essere diviso in due parti: la parte per generare i divisori primi (che abbiamo già considerato in precedenza) e quella per calcolare i fattori primi. Nel caso peggiore (quando n è primo), per calcolare i fattori di n occorrerà un numero di passi dell'ordine di \sqrt{n} ; non è altrettanto facile caratterizzare il comportamento medio.
2. La condizione invariante per il loop di fattorizzazione è che, dopo un'iterazione col divisore *nxtprime*, si sono trovati tutti i fattori minori di *nxtprime*.

Contemporaneamente, n sarà stato ridotto almeno del prodotto di tutti i suoi fattori primi minori di $nxtprime$ (multipli compresi). Al termine deve valere la condizione $n \text{ div } nxtprime} \leqslant nxtprime$. L'algoritmo deve alla fine concludersi poiché, ad ogni iterazione, o viene diminuito n o viene incrementato $nxtprime$; entrambi questi cambiamenti hanno come effetto la riduzione del rapporto di n su $nxtprime$, per cui prima o poi non sussiste più la condizione $n > nxtprime$.

3. L'algoritmo che abbiamo qui sviluppato, benché corretto, è adatto solo per numeri di 6 cifre o meno. Oltre questo limite, comincia ad essere costoso e sono pertanto richiesti algoritmi di fattorizzazione più sofisticati. Un ultimo commento: un metodo pratico (che sia cioè economico in termini di tempo e di memoria) per la fattorizzazione di numeri molto grandi è ancora da sviluppare; sul permanere di questa situazione fanno affidamento i moderni algoritmi di codifica crittografica.
4. Se questo algoritmo deve essere impiegato ripetutamente, una strategia migliore potrebbe essere quella di precalcolare una volta per tutte una tabella di numeri primi, anziché generarli via via che sono richiesti.
5. Knuth fornisce un metodo pratico per fattorizzare numeri grandi (*The Art of Computer Programming*, vol. 2, pagg. 347-9).

Applicazioni

Fattorizzazione di numeri fino a sei cifre.

Problemi supplementari

- 3.5.1 Implementare una versione dell'algoritmo di fattorizzazione che incorpori una procedura del crivello di Eratostene.
- 3.5.2 Implementare un algoritmo di fattorizzazione che elimini dai divisori soltanto i multipli di 2, 3 e 5; confrontarlo con il precedente dal punto di vista delle divisioni fatte.
- 3.5.3 I numeri *amichevoli* sono coppie di numeri i cui divisori (comprendendo 1, ma escludendo il numero stesso) danno per somma l'altro numero. Progettare ed implementare un algoritmo che, dati due numeri, verifica se sono numeri amichevoli.
- 3.5.4 Un numero perfetto è un numero uguale alla somma dei suoi divisori (escluso il numero stesso). Progettare ed implementare un algoritmo che stampi tutti i numeri perfetti tra 1 e 500.

ALGORITMO 3.6 GENERAZIONE DI NUMERI PSEUDO-CASUALI

Problema

Utilizzare il metodo della congruenza lineare per produrre un insieme uniforme di numeri pseudo-casuali.

Sviluppo dell'algoritmo

I generatori di numeri casuali sono spesso impiegati in informatica per verificare ed analizzare, tra gli altri, il comportamento degli algoritmi. Una successione di numeri casuali dovrebbe mostrare il comportamento seguente:

1. La sequenza dovrebbe apparire come se ciascun numero si presentasse per caso.
2. Ciascun numero dovrebbe avere una probabilità definita di cadere in un determinato intervallo.

Nella scienza degli elaboratori, per generare numeri casuali si simuloano di solito i processi aleatori, producendo deterministicamente una successione di numeri che all'apparenza manifestano un comportamento casuale. Tali sequenze sono prevedibili in anticipo ed è per questo che sono usualmente indicate come successioni *pseudo-casuali*. Vi sono molti metodi per generare numeri pseudo-casuali; il più largamente usato di tali algoritmi è forse il *metodo della congruenza lineare*.

Quando tale metodo è opportunamente parametrizzato, genererà successioni pseudo-casuali che, per gli scopi pratici, soddisfano i criteri statistici richiesti dalle variabili casuali uniformemente distribuite. In una distribuzione uniforme ogni numero è ugualmente probabile.

L'implementazione del metodo della congruenza lineare è assai semplice. I membri della successione lineare congruente $\{x\}$ sono ottenuti in sequenza con l'espressione:

$$x_{n+1} = (ax_n + b) \bmod m \quad \text{per } n \geq 0$$

ove i parametri a , b , m e x_0 devono essere accuratamente scelti in anticipo, secondo determinati criteri. I parametri a , b e m sono conosciuti rispettivamente come il moltiplicatore, l'incremento ed il modulo. Knuth (*The Art of Computer Programming*, vol. 2, pagg. 9-157) fornisce una base teorica eccellente per la scelta di questi parametri; i risultati si possono riassumere come segue:

Tutti i parametri dovranno essere interi maggiori o uguali a zero, con m maggiore di x_0 , a e b .

Parametro x_0

Il parametro x_0 può essere scelto arbitrariamente nell'intervallo $0 \leq x_0 < m$.

Parametro m

Il valore di m dovrà essere maggiore o uguale alla lunghezza della sequenza casuale richiesta. Inoltre dovrà essere possibile eseguire l'operazione $(a*x + b) \bmod m$ senza arrotondamenti.

Parametro a

La scelta di a dipende da quella di m . Se m è una potenza di 2, a deve soddisfare la condizione:

$$a \bmod 8 = 5$$

Se m è una potenza di 10, a andrà scelto in modo che:

$$a \bmod 200 = 21.$$

Altri requisiti di a sono che risulti maggiore di \sqrt{m} e minore di $m - \sqrt{m}$, che $(a - 1)$ sia multiplo di tutti i numeri primi divisori di m e, se m è multiplo di 4, che lo sia anche $(a - 1)$. Queste condizioni, insieme con quella che b ed m risultino primi fra loro, sono tutte richieste per garantire che la successione abbia periodo m .

Parametro b

La costante b dovrà essere dispari e non multipla di 5.

Quando a , b e m sono scelti in accordo con le condizioni sopradette, prima che la sequenza cominci a ripetersi, si potrà produrre una successione di m numeri pseudo-casuali nell'intervallo $0..(m-1)$. Un'implementazione in Pascal del metodo della congruenza lineare è fornita nel seguito, con un valore di m pari a 4096.

Implementazione in Pascal

```
procedure random (var x: integer);
var
  a {multiplier},
  b {increment},
  m {modulus}: integer;

begin {generates pseudo-random numbers x by the linear
congruent method}
  m := 4096;
  {assert: 0=<x=<m-1}
  b := 853;
```

```
a := 109;
x := (a * x + b) mod m
{assert: 0=<x=<m-1}
end
```

Note di progetto

1. Il metodo della congruenza lineare è un metodo semplice, efficiente e pratico per generare numeri pseudo-casuali.
2. Una distribuzione uniforme di numeri pseudo-casuali può essere adoperata come base per generare altre distribuzioni, come quella normale e quella esponenziale.
3. Le basi teoriche per la scelta dei parametri coinvolgono un'analisi molto sofisticata.

Applicazioni

Analisi di algoritmi, problemi di simulazione e giochi.

Problemi supplementari

- 3.6.1 Verificare che l'algoritmo si ripete dopo m numeri casuali; calcolare la media e la varianza dell'insieme degli m numeri pseudo-casuali.
- 3.6.2 Controllare l'uniformità della distribuzione prodotta dal metodo della congruenza lineare per $m = 4096$, raggruppando i numeri prodotti a blocchi di 64 nell'intervallo 0..4095 (ossia, il primo blocco 0..63, ecc.). Tracciare l'istogramma risultante.
- 3.6.3 Un insieme $|r|$ di numeri casuali può essere utilizzato per produrre l'insieme $|x|$ di numeri a distribuzione esponenziale mediante la formula:

$$x_i = -\frac{1}{\lambda} \log_e (1-r_i)$$

ove λ è il parametro della distribuzione esponenziale. Implementare l'algoritmo corrispondente.

- 3.6.4 Per generare numeri casuali aventi distribuzione normale nell'intervallo 0..1 si può usare il metodo polare. Esso richiede innanzitutto la generazione di due numeri uniformemente casuali r_1 e r_2 ; quindi se l'espressione d seguente è ≥ 1 , n_1 e n_2 (v. formule) risultano essere numeri casuali con distribuzione normale:

$$d = (2r_1 - 1)^2 + (2r_2 - 1)^2$$

$$n_1 = (2r_1 - 1) \left(\frac{-2 \log_e(d)}{d} \right)^{\frac{1}{2}}$$

$$n_2 = (2r_2 - 1) \left(\frac{-2 \log_e(d)}{d} \right)^{\frac{1}{2}}$$

Utilizzare tali espressioni per produrre numeri casuali con distribuzione normale.

- 3.6.5 In talune applicazioni, si richiede un generatore casuale "migliore" del semplice metodo della congruenza lineare. A tale scopo, si debbono produrre due set *indipendenti* $[r_i]$ e $[s_i]$ di numeri casuali, a partire da valori a e b diversi, ma con *identico* valore di m . Un'area di memoria ausiliaria $[t_i]$ di 100 elementi è quindi riempita con i primi 100 valori della successione casuale $[r_i]$. Dopo questo passo di inizializzazione, possiamo generare successivamente coppie r_i e s_i di numeri casuali. Per produrre l' i -esimo numero casuale "migliorato", adoperiamo s_i ed il modulo m per calcolare un indice j nella tabella $t[1...100]$:

$$j = \lfloor 100s_i/m \rfloor$$

L' i -esimo numero casuale "migliorato" si trova in $t[j]$; una volta adoperato, il valore di $t[j]$ è sostituito con l' r_i corrente. Implementare questo generatore "migliorato" di numeri casuali. (Vedi M.O.Mac-Laren e G. Marsaglia (1965), "Uniform random number generators". *JACM*, 83-9)

ALGORITMO 3.7 ELEVAMENTO DI UN NUMERO A UNA POTENZA ELEVATA

Problema

Dato un intero x , calcolare x^n con n intero positivo molto maggiore di 1.

Sviluppo dell'algoritmo

Calcolare l'espressione:

$$p = x^n$$

con x e n assegnati è compito elementare. Un metodo semplice di calcolo consiste nel moltiplicare per x un prodotto cumulativo p , per n iterazioni; ad esempio:

```
p := 1
for i := 1 to n do p := p*x
```

In molti casi, questo modo di calcolo può ritenersi soddisfacente. Esistono tuttavia situazioni (p. es. in alcuni algoritmi crittografici di recente pubblicazione) ove è necessario perseguire una maggiore efficienza nel calcolo delle potenze di un intero. Dedichiamoci allora al tentativo di scoprire un metodo più efficiente per il calcolo delle potenze. Non è chiaro da dove prender le mosse per progettare un algoritmo più efficiente che calcoli x^n . In tal caso è forse meglio studiare come viene risolto un esempio specifico mediante il primo algoritmo, per vedere se ciò ci può aiutare a partire. Consideriamo il calcolo di x^{10} . Nel nostro approccio passo passo abbiamo:

$$\begin{aligned} p_1 &= x^1 = x \\ p_2 &= x^2 = x*x \\ p_3 &= x^3 = x^2*x \\ p_4 &= x^4 = x^3*x \end{aligned}$$

$$p_{10} = x^{10} = x^5*x^5$$

Analizzando questi passi, vediamo che ad ogni passo x è elevato di un grado. Che cosa significa in realtà cercare di calcolare x^{12} in modo più efficiente? Significa dover eseguire *meno* passi per portare a compimento l'operazione. Ci domandiamo come ciò si possa fare. Tornando all'esempio, vediamo che potremmo produrre x^4 moltiplicando semplicemente x^2 per se stesso, ossia:

$$x^4 = x^2*x^2$$

Ciò richiede *una sola* moltiplicazione al posto delle due impiegate in precedenza; in termini di somma di potenze abbiamo $2 + 2 = 4$, da confrontare con $2 + 1 + 1 = 4$. Quest'ultimo modo di calcolare x^4 può fornirci lo spunto che cercavamo per indagare ulteriormente nella stessa direzione; quest'esempio ci dice che risolveremo in maniera più efficiente il nostro problema iniziale quando avremo determinato l'insieme di numeri che sommati danno 10 nel minor numero di passi. (Ricordiamo che in questo caso la moltiplicazione si traduce nella *somma* delle potenze).

Alcuni esempi di numeri che sommati danno 10 sono:

$$\begin{aligned} 8 + 2 &= 10 \\ 7 + 3 &= 10 \\ 6 + 4 &= 10 \\ 5 + 5 &= 10 \\ 4 + 4 + 2 &= 10 \end{aligned}$$

Osserviamo subito che, qualunque sceglieremo tra queste possibilità, ci troviamo ad affrontare il problema di calcolare potenze più piccole, da risolvere efficientemente. Ora, poiché sommare tre numeri è meno efficiente che sommarne due, è plausibile che la soluzione "migliore" non possa venire dalla somma di insiemi come $4 + 4 + 2$. Questo ci conduce a domandarci quale coppia di numeri, la cui somma è 10, comporti il calcolo delle potenze inferiori *più piccole*.

Supponiamo di aver scelto 8 e 2 come coppia di potenze per produrre x^{10} (ossia $x^{10} = x^8 * x^2$). Facendo questa scelta, per compiere l'operazione dovremmo calcolare x^8 che è già prossimo a x^{10} . Ci dobbiamo chiedere allora quale sia la coppia di numeri *più piccoli* che sommati danno 10: la risposta è una coppia di 5, ovvero:

$$5 + 5 = 10$$

Tutte le altre combinazioni, p. es. $6 + 4$, ecc., ci lasciano da risolvere un problema con potenze inferiori più grandi (in questo caso x^6). Nel caso in esame, una volta prodotto x^5 si passa *direttamente* a x^{10} moltiplicando semplicemente x^5 per se stesso.

È ora la volta di scoprire come calcolare efficientemente x^5 . Possiamo tentare nuovamente l'approccio del dimezzamento delle potenze; ma sfortunatamente la metà di 5 è 2.5 che non è una potenza intera. In questo caso, l'unica scelta conveniente è produrre x^5 quadrando x^2 in modo da ottenere x^4 e moltiplicando il risultato per x . Ovvero:

$$x^5 = x^2 * x^2 * x$$

I passi per calcolare x^{10} sono perciò

$$\begin{aligned} x^2 &= x * x \\ x^4 &= x^2 * x^2 \\ x^5 &= x^4 * x \\ x^{10} &= x^5 * x^5 \end{aligned}$$

Con questo schema, per calcolare x^{10} abbiamo utilizzato *solo quattro* moltiplicazioni invece di nove; di questo passo, è ragionevole attenersi che il guadagno sia molto maggiore per le potenze più elevate.

Siamo ancora ben lungi dall'aver determinato l'implementazione dell'algoritmo, per cui considereremo un altro esempio per vedere quali generalizzazioni si possano fare. Analizziamo il caso del calcolo di x^{23} . Per calcolare x^{10} eravamo partiti dalla potenza finale procedendo a *ritroso*; possiamo cercare di applicare la stessa idea a questo secondo esempio.

$$\begin{aligned} x^{23} &= x^{22} * x \\ x^{22} &= x^{11} * x^{11} \\ x^{11} &= x^{10} * x \\ x^{10} &= x^5 * x^5 \\ x^5 &= x^4 * x \\ x^4 &= x^2 * x^2 \\ x^2 &= x * x \end{aligned}$$

È possibile vedere che sono bastate 7 moltiplicazioni per elevare un numero alla 23-esima potenza. Inoltre, dopo l'esame dei due esempi, si può osservare l'evidenziarsi di uno schema preciso.

Ad ogni passo del processo di generazione delle potenze, vale una delle due condizioni:

- (a) Se la potenza è *dispari*, essa deve essere stata prodotta dalla potenza immediatamente inferiore (p. es. $x^{23} = x^{22} * x$).
- (b) Se la potenza è *pari*, essa si può ottenere dalla potenza di ordine *metà* (p. es. $x^{22} = x^{11} * x^{11}$).

Queste due affermazioni inquadrono l'essenza dell'algoritmo. Ciò significa che l'algoritmo dovrà avere due parti:

- una parte che determini la strategia delle moltiplicazioni;
- una seconda parte che esegua effettivamente il calcolo della potenza.

Per delineare la sequenza delle moltiplicazioni, possiamo cominciare dalla potenza richiesta e decidere se è pari o dispari. Il passo successivo è quello di eseguire la divisione intera per 2 della potenza corrente e di ripetere la determinazione pari/dispari. Possiamo vedere dagli esempi che questa procedura va ripetuta finché non si raggiunge 2.

L'informazione sulla parità può essere registrata in un array, memorizzando 1 per le potenze dispari e 0 per le pari.

Per il secondo esempio abbiamo:

$$\begin{array}{ll} x^{23} & d[1] = 1 \\ x^{11} & d[2] = 1 \\ x^5 & d[3] = 1 \\ x^2 & d[4] = 0 \\ x^1 & d[5] = 1 \end{array}$$

Per eseguire la seconda parte dell'algoritmo (ossia il calcolo della potenza), occorre procedere *in avanti* anziché all'indietro; ciò significa che dovremo partire dall'elemento più elevato del vettore d ($d[4]$ nell'esempio) e risalire fino a $d[1]$.

Partendo con il prodotto p uguale ad 1, possiamo procedere al calcolo della potenza secondo la regola seguente:

se l'elemento corrente del vettore d è zero

- (a) eleviamo al quadrato il prodotto corrente p ,
- altrimenti

(a') quadriamo il prodotto corrente p e moltiplichiamo per x , così da ottenere una potenza dispari.

Per il calcolo di x^{23} i passi sono:

$$\begin{array}{lll} d[5] = 1 \Rightarrow p := p * p * x & (1 * 1 * x) & = x \\ d[4] = 0 \Rightarrow p := p * p & (x * x) & = x^2 \end{array}$$

$$\begin{array}{lll}
 d[3] = 1 \Rightarrow p := p * p * x & (x^2 * x^2 * x) & = x^5 \\
 d[2] = 1 \Rightarrow p := p * p * x & (x^5 * x^5 * x) & = x^{11} \\
 d[1] = 1 \Rightarrow p := p * p * x & (x^{11} * x^{11} * x) & = x^{23}
 \end{array}$$

Dal nostro precedente lavoro sulla conversione di base (cap. 2), possiamo riconoscere che la ripetuta divisione per 2 di n ne produce la rappresentanza binaria. Poiché il calcolo efficiente della potenza è basato sulla rappresentanza binaria dell'esponente, ci domandiamo se esso non si possa fare direttamente anziché dopo aver ricavato la rappresentazione binaria. Potendo procedere in questa maniera non sarebbe necessario memorizzare per l'impiego successivo la strategia di moltiplicazione.

Per cercare di scoprire se è praticabile un approccio diretto possiamo tornare all'esempio precedente e alla sua rappresentazione binaria:

$$23_{10} = 10111_2$$

Lo schema di calcolo originale costruiva la potenza facendo scorrere verso sinistra i bit più significativi, ossia avevamo:

$$\begin{array}{ll}
 1 & x \\
 10 & x^2 \\
 101 & x^5 \\
 1011 & x^{11} \\
 10111 & x^{23}
 \end{array}$$

Con questo schema il contributo dei bit più significativi è tenuto in conto *per primo*; al contrario, la derivazione della rappresentazione binaria dell'esponente ci fornisce le cifre a partire dalla *meno* significativa. Ci domandiamo: il calcolo della potenza può avvenire in questo ordine? Con riferimento alla rappresentazione binaria di 23 vediamo che si può scrivere:

$$x^{23} = x^1 * x^2 * x^4 * x^{16}$$

ove

$$1 * 2^0 + 1 * 2^1 + 1 * 2^2 + 0 * 2^3 + 1 * 2^4 = 23$$

il che suggerisce che il calcolo della potenza può essere accoppiato *direttamente* alla derivazione della rappresentazione binaria dell'esponente n .

Per il nostro esempio occorrerà saper produrre consecutivamente le potenze:

$$x^1, x^2, x^4 \text{ e } x^{16}$$

via via che le cifre binarie vengono definite.

Ciascuna di queste potenze può ottenersi moltiplicando la precedente per se stessa: per esempio:

$$x^4 = x^2 * x^2$$

Si osservi che x^8 manca dalla lista ma va ugualmente calcolata, quando si trova il bit 0, se nel passo successivo si deve ricavare x^{16} . Studiando ancora una volta il nostro algoritmo nel suo insieme vediamo che il calcolo della potenza implica i passi seguenti:

1. La generazione in sequenza dei numeri della successione di potenze $x^1, x^2, x^4, x^8, x^{16}, \dots$
2. L'inclusione della potenza corrente nel prodotto cumulativo se il bit corrispondente è 1.

Tab. 3.1. Calcolo della potenza con la strategia del raddoppio per x^{23} .

Cifra binaria	Successione di potenze	Prodotto cumulativo
1	x	x
1	x^2	$x^2 * x = x^3$
1	x^4	$x^4 * x^3 = x^7$
0	x^8	$\underline{\quad}$
1	x^{16}	$x^{16} * x^7 = x^{23}$

La tabella 3.1 riassume il calcolo della potenza x^{23} secondo la nostra nuova proposta.

Per verificare se la prossima cifra significativa binaria è 1, si può usare il test $n \bmod 2 = 1$. Designando con *product* il *prodotto cumulativo* e con *psequence* i successivi termini della sequenza di potenze, possiamo utilizzare la seguente formula standard per aggiungere nuovi termini al prodotto cumulativo:

$$\text{product} := \text{product} * \text{psequence}$$

La variabile *product* deve essere inizialmente posta ad 1 per comprendere anche il caso in cui n è zero. La variabile della successione di potenze *psequence* occorre sia inizializzata ad x , numero da elevare alla potenza n . Pertanto i passi di inizializzazione sono:

$$\begin{aligned}
 \text{product} &:= 1; \\
 \text{psequence} &:= x
 \end{aligned}$$

Nelle iterazioni seguenti il valore di *psequence* dovrà essere *raddoppiato*. Ciò si ottiene eseguendo:

$$\text{psequence} := \text{psequence} * \text{psequence}$$

L'altro passo che si deve realizzare ad ogni iterazione è la divisione di n per 2. Ovvero:

$$n := n \bmod 2$$

L'algoritmo si concluderà quando n è ridotto a zero.

Disponiamo adesso di tutti i dettagli per descrivere completamente l'algoritmo per il calcolo delle potenze.

Descrizione dell'algoritmo

1. Stabilire n , l'esponente intero della potenza, ed x , l'intero da elevare alla n .
2. Inizializzare per il caso di potenza zero le variabili della successione di potenze e del prodotto.
3. Finché n è maggiore di zero eseguire quanto segue:
 - (a) se il prossimo bit più significativo di n è 1
 - (a.1) moltiplicare il prodotto cumulativo per il valore corrente della successione di potenze;
 - (b) ridurre di un fattore 2 l'esponente n mediante divisione intera;
 - (c) ottenere il prossimo termine della successione di potenze, moltiplicando per se stesso il valore corrente.
4. Restituire x elevato alla n .

Implementazione in Pascal

```
function power(x, n: integer): integer;
var product {current accumulated product, eventually contains result},
    psequence {current power sequence value}: integer;

begin {computes x raised to the power n using doubling strategy}
  {assert: x>0 ∧ n >= 0 ∧ n0 = n}
  product := 1;
  psequence := x;
  {invariant: product * (psequence) ↑n = x↑n0 ∧ n >= 0}
  while n > 0 do
    begin {incorporate power for next most significant binary digit if
          not zero}
      if (n mod 2)=1 then
        product := product * psequence;
      n := n div 2;
      psequence := psequence * psequence
    end;
    {assert: product = x↑n0}
  power := product
end
```

Note di progetto

1. Per elevare x alla n occorre un numero di moltiplicazioni dell'ordine di $\lceil \log_2 n \rceil$. Questo metodo di valutazione delle potenze, benché non ideale per tutti i valori di n , è efficiente per valori di n elevati.
2. Il while-loop genera la rappresentazione binaria dell'intero n per tutti i valori $n > 0$. Dopo l' i -esima iterazione, sono stati prodotti i

primi i bit a destra della rappresentazione binaria di n , ed x è stato elevato alla potenza corrispondente al valore di tali bit. Al tempo stesso n è stato ridotto di 2^i .

L'algoritmo ha termine poiché ad ogni iterazione n si riduce e pertanto risulterà infine falsa la condizione $n > 0$.

- L'algoritmo funziona correttamente per tutti i valori $n \geq 0$. Per valori elevati di n , il valore di x^n supera rapidamente i limiti di rappresentazione degli interi nella maggior parte dei computer; in questo algoritmo non si è attuato nessun accorgimento per proteggere da questo rischio: evidentemente in questi casi è necessaria una rappresentazione estesa, che faccia ricorso ad array o all'aritmetica modulare. Tuttavia, da un punto di vista funzionale, si può sempre utilizzare il metodo di calcolo delle potenze di cui sopra.
3. Il metodo del raddoppio nella versione iniziale non è elegante quanto l'algoritmo finale, poiché richiede la memorizzazione di un array separato.
 4. Un esempio specifico è stato di grande aiuto per individuare il meccanismo dell'elevamento a potenza.
 5. Vediamo che in questo algoritmo si applica la strategia del divide-et-impera. Il problema originale si risolve attraverso la soluzione di un problema che ha ampiezza metà. Quest'idea viene applicata ricorsivamente; la strategia del divide-et-impera assai spesso ci conduce ad algoritmi efficienti ed equilibrati.

Applicazioni

Crittografia (codifica segreta dell'informazione) ed analisi di non-primalità dei numeri.

Problemi supplementari

- 3.7.1 Progettare un algoritmo per il calcolo delle potenze costruito su una strategia in base 3 anziché sul metodo attuale in base 2. Confrontare i risultati per i due metodi.
- 3.7.2 Progettare un algoritmo per il calcolo delle potenze in precisione estesa, per valori di x^n che superino i limiti di rappresentazione intera del computer.
- 3.7.3 È talvolta importante stabilire che un numero non è primo. Per far ciò possiamo utilizzare un risultato dovuto a Fermat. È possibile dimostrare che per tutti i numeri primi ad eccezione di 2 vale la regola seguente:

$$2^{p-1} \bmod p = 1$$

Questo test può essere eseguito in un numero di passi dell'ordine di $\log_2(p)$. Progettare un algoritmo per verificare la non primalità di un numero. Per semplicità, scegliere un numero primo tale che 2^{p-1} non superi il limite di rappresentazione della parola intera del computer.

ALGORITMO 3.8 CALCOLO DELL'N-ESIMO NUMERO DI FIBONACCI

Problema

Dato un numero n , produrre il termine n -esimo della successione di Fibonacci.

Sviluppo dell'algoritmo

Ricordando il precedente algoritmo 2.6, sappiamo che l' n -esimo termine f^n della successione di Fibonacci è definito ricorsivamente come segue:

$$\begin{aligned}f_1 &= 0 \\f_2 &= 1 \\f_n &= f_{n-1} + f_{n-2} \quad \text{per } n > 2\end{aligned}$$

Abbiamo visto in precedenza come tale definizione, unita ad un costrutto iterativo, potesse produrre i numeri di Fibonacci. Qui ci interessa indagare se esista un modo più efficiente per produrre l' n -esimo numero di Fibonacci, che non sia quello di generare la successione completa degli n numeri di Fibonacci.

Talvolta accade che, essendo richiesto un elemento di un insieme, questo si possa ottenere in maniera più efficiente che non calcolando tutti gli $(n - 1)$ predecessori. Per esempio, nell'algoritmo 3.7 abbiamo visto come tale principio si possa applicare assai efficacemente: siamo stati capaci di produrre l' n -esima potenza di un numero senza generarne gli $(n - 1)$ predecessori, ma producendone solo una quantità dell'ordine di $\log n$. Qui ci interessa capire se sia applicabile al problema attuale la strategia del divide-et-impera. Nel problema del calcolo delle potenze, il nostro compito era semplice dal momento che il legame tra la potenza i -esima e la $2i$ -esima era elementare e facilmente definito (p. es. $x^6 = x^3 * x^3$).

Nel problema attuale non siamo certi che esista una simile "relazione di raddoppio". Per esplorare tale possibilità scriviamo i primi termini della successione dei numeri di Fibonacci.

numero di Fibonacci	0	1	1	2	3	5	8	13	21	34
indice	1	2	3	4	5	6	7	8	9	10

Per fare un esempio vediamo se l'ottavo numero di Fibonacci può essere collegato col quarto. Abbiamo:

$$\begin{aligned}f_4 &= 2 \\f_8 &= 13\end{aligned}$$

Il legame tra i due è:

$$f_8 = 6 * f_4 + 1$$

Provando questa relazione tra f_{10} e f_5 , scopriamo che non è generalizzabile: possiamo sospettare che non esista un legame di raddoppio tra due soli numeri di Fibonacci. Il tentativo successivo è la ricerca di una qualche relazione tra due numeri di Fibonacci che generi il numero di Fibonacci "raddoppiato": questo ci sembra avere maggiori possibilità di successo, poiché, nello schema originale per produrre i numeri di Fibonacci, si impiegano due numeri per generare il successivo.

Cerchiamo pertanto di scrivere f_8 in funzione soltanto di f_4 e f_5 ; per farlo dobbiamo utilizzare le definizioni originali della successione, ovvero:

$$\begin{aligned}f_4 &= 2 \\f_5 &= 3 \\f_6 &= f_5 + f_4 \\f_+ &= f_6 + f_5 = (f_5 + f_4) + f_5 \quad (\text{dopo aver sostituito } f_6) \\f_8 &= f_+ + f_6 = [(f_5 + f_4) + f_5] + f_5 + f_4\end{aligned}$$

raccogliendo i termini comuni otteniamo:

$$f_8 = 3f_5 + 2f_4 = 3 * 3 + 2 * 2 = 13$$

Poiché $f_5 = 3$ e $f_4 = 2$, l'ultima espressione ci suggerisce:

$$f_8 = f_5^2 + f_4^2$$

Provando con f_{10} , f_5 e f_6 per vedere se la formula si applica in generale, otteniamo:

$$f_{10} = f_6^2 + f_5^2 = 5^2 + 3^2 = 34$$

Senza addentrarci in una dimostrazione dettagliata, siamo ragionevolmente convinti che in generale:

$$f_{2n} = f_{n+1}^2 + f_n^2$$

Questa espressione è confermata da ulteriori controlli nella successione.

Siamo quasi arrivati alla fine? Riandando all'algoritmo per il calcolo delle potenze, ricordiamo che, per ottenere la soluzione finale ottima, dovevamo talvolta moltiplicare solo per x , tenendo conto in tal modo delle potenze dispari. Ci possiamo aspettare che un'idea analoga si

applichi al problema attuale: per estendere il procedimento, dobbiamo saper maneggiare il numero di Fibonacci f_{2n+1} al pari del numero f_{2n} . Per produrre il numero di Fibonacci f_{2n+1} ci occorrono f_{2n} e f_{2n-1} :

$$f_{2n+1} = f_{2n} + f_{2n-1}$$

Il problema è che non disponiamo di f_{2n-1} e pertanto non abbiamo compiuto veri progressi. Perciò cerchiamo come si possa produrre f_9 a partire da f_5 e f_4 . Applicando lo stesso metodo di sostituzione che ci ha permesso di scrivere:

$$f_{2n} = f_{n+1} \quad (\text{per } n \geq 1)$$

Per il successivo raddoppio, f_{2n} e f_{2n+1} assumono i ruoli assegnati inizialmente a f_n e f_{n+1} rispettivamente. Abbiamo ora le basi di un metodo per generare l' n -esimo numero di Fibonacci in maniera simile all'innalzamento di x all' n -esima potenza nell'algoritmo 3.7.

In questo caso, l'algoritmo dovrà dividersi in due parti:

- una parte che determini la strategia di raddoppio, generando la rappresentazione binaria di n ;
- una seconda parte che calcoli l' n -esimo numero di Fibonacci in accordo con la strategia di raddoppio.

Dobbiamo adesso risolvere alcuni esempi per vedere esattamente come possiamo produrre la rappresentazione binaria di n con il metodo usato nell'algoritmo 3.7.

Consideriamo l'esempio $n = 10$ (tabella 3.2).

Tab. 3.2. Numeri di Fibonacci per $n = 10$.

Coppia di numeri di Fibonacci richiesta	Numero di Fibonacci	Rappresentazione binaria
(f_{10}, f_{11})	f_{10}	$d[1] = 0$
(f_5, f_6)	f_5	$d[2] = 1$
(f_2, f_3)	f_2	$d[3] = 0$
(f_1, f_2)	f_1	$d[4] = 1$

Si osservi che il "raddoppio" di (f_i, f_j) ci fornisce soltanto (f_i, f_j) ; tuttavia, per produrre la coppia "raddoppiata" (f_{10}, f_{11}) occorre la coppia (f_5, f_6) . Per fortuna, per ricavare f_5 possiamo utilizzare semplicemente la definizione standard (ovvero $f_5 = f_3 + f_4$).

Consideriamo un altro esempio con $n = 13$ (tabella 3.3).

Tab. 3.3. Numeri di Fibonacci per $n = 13$.

Coppia di numeri di Fibonacci richiesta	Numero di Fibonacci	Rappresentazione binaria
(f_{13}, f_{14})	f_{13}	$d[1] = 1$
(f_6, f_7)	f_6	$d[2] = 0$
(f_3, f_4)	f_3	$d[3] = 1$
(f_1, f_2)	f_1	$d[4] = 1$

Dallo studio attento dei due esempi, appare chiaro come impiegare la rappresentazione binaria per produrre il numero di Fibonacci richiesto.

Partiremo sempre dalla coppia (f_1, f_2) e per produrre il numero di Fibonacci richiesto dovremo sempre eseguire un numero di raddoppi pari ad $(i - 1)$ se i sono le cifre della rappresentazione binaria di n (p. es. la rappresentazione binaria di $n = 13$ è 1011 e di conseguenza sono richieste $4 - 1 = 3$ fasi di raddoppio). La cifra binaria di cui si tiene sempre conto con le condizioni iniziali è la cifra più significativa ($d[4]$ nell'esempio $n = 13$) quando n è stato ridotto ad 1; possiamo eliminare tale cifra arrestando la generazione dei bit un passo prima.

Considerando l'esempio $n = 13$, dobbiamo partire applicando la regola del raddoppio alla coppia (f_1, f_2) per ottenere così la coppia (f_2, f_3) . Prima di poter proseguire, ci serve la coppia (f_3, f_4) che si può ricavare direttamente dalla (f_1, f_2) con la formula standard $f_{k+1} = f_k + f_{k-1}$. In generale, tutte le volte che la cifra binaria corrispondente è dispari ($d[3]$ nel nostro caso), dobbiamo eseguire il procedimento di raddoppio e quindi estendere di uno la successione prima di raddoppiare di nuovo. Indicando con fn e $fnp1$ i numeri di Fibonacci f_n e f_{n+1} e con $f2n$ e $f2np1$ rispettivamente f_{2n} e f_{2n+1} e con $f2n$ e $f2np1$ rispettivamente f_{2n} e f_{2n+1} abbiamo i passi iniziali di raddoppio:

$$\begin{aligned} f2n &:= fn * fn + fnp1 * fnp1; \\ f2np1 &:= 2 * fn * fnp1 + fnp1 * fnp1 \end{aligned}$$

e poi le seguenti riassegnazioni, in preparazione del passo di raddoppio successivo:

$$\begin{aligned} fn &:= f2np1; \\ fnp1 &:= f2np1 + f2n \quad (\text{estensione di 1}) \end{aligned}$$

A questo punto, alle variabili fn e $fnp1$ sono attribuiti i valori corretti perché possa aver luogo la successiva fase di raddoppio.

Tornando all'esempio, una volta ottenuta la coppia (f_1, f_2) , otteniamo la coppia successiva (f_3, f_4) per semplice raddoppio. Lo zero contenuto in $d[2]$ ci dice che possiamo utilizzare *direttamente* la coppia (f_3, f_4) per il prossimo passo di raddoppio, senza dover estendere di uno la sequenza.

Utilizzando ancora le variabili fn , $fnp1$, $f2n$, $f2np1$, in questo caso abbiamo gli stessi passi iniziali di prima; ossia:

$$\begin{aligned} f2n &:= fn * fn + fnp1 * fnp1; \\ f2np1 &:= 2 * fn * fnp1 + fnp1 * fnp1 \end{aligned}$$

e poi le seguenti riassegnazioni, in preparazione del passo di raddoppio successivo:

$$\begin{aligned} fn &:= f2n; \\ fnp1 &:= f2np1 \end{aligned}$$

A questo punto, alle variabili fn e $fnpl$ sono attribuiti i valori corretti perché possa aver luogo la successiva fase di raddoppio. Nell'implementazione effettiva, poiché il quadrato di $fnpl$ si presenta due volte, è più efficiente precalcolarlo. Il passo di raddoppio è comune ad entrambi i casi $d[k] = 0$ e $d[k] = 1$; pertanto andrà collocato *prima* dei due casi possibili di riasssegnazione. Le riasssegnazioni in preparazione dei successivi raddoppi potranno allora essere condizionate dal valore della cifra binaria corrente. Osservati questi fatti, possiamo adesso fornire la descrizione dettagliata dell'algoritmo; il metodo per determinare la rappresentazione binaria di n è esattamente lo stesso impiegato nell'algoritmo 3.7.

Descrizione dell'algoritmo

1. Stabilire il numero n , che individua il numero di Fibonacci da calcolare.
2. Derivare la rappresentazione binaria di n mediante ripetute divisioni per 2 e memorizzarla nell'array $d[1..i - 1]$.
3. Inizializzare i primi due elementi della sequenza di raddoppio.
4. Procedendo a ritroso nella rappresentazione binaria di n , dalla $(i - 1)$ -esima cifra più significativa alla prima, eseguire quanto segue:
 - (a) impiegare la coppia corrente di numeri di Fibonacci (f_n, f_{n+1}) per produrre la coppia (f_{2n}, f_{2n+1})
 - (b) se la cifra binaria corrente $d[k]$ è zero eseguire il riasssegnamento di f_n e f_{n+1} altrimenti estendere di un numero la successione e quindi eseguire il riasssegnamento di f_n e f_{n+1}
5. Restituire l' n -esimo numero di Fibonacci f_n .

Implementazione in Pascal

```
function nfib (n: integer): integer;
var d: array [1..100] of integer; {array containing binary digits}
  fn {the nth fibonacci number – on termination contains final result},
  f2n {the 2nth fibonacci number},
  fnpl {the (n + 1)th fibonacci number},
  f2npl {the (2n + 1)th fibonacci number},
  i {binary digit count less 1 of n and index for binary array},
  k {index for array of binary digits},
  sqfnpl {square of the (n + 1)th fibonacci number}: integer;

begin {generate the nth fibonacci number by repeated doubling}
  {assert: n > 0 ∧ n has < 100 digits in its binary representation}
  i := 0;
  {invariant: after the ith iteration the i least significant bits of binary
  representation of original n stored in d[i..1]}
```

```
while n > 1 do
  begin {generate binary digits for n without the most significant
  digit}
    i := i + 1;
    if odd(n) then d[i] := 1 else d[i] := 0;
    n := n div 2
  end;
  fn := 0;
  fnpl := 1;
  {invariant: after current iteration fn = fibonacci number
  corresponding to i - k + 1 leftmost bits of binary representation
  of n}
  for k := i downto 1 do
    begin {generate the 2n and (2n + 1)th fibonacci numbers from
    the nth and (n + 1)th}
      sqfnpl := fnpl * fnpl;
      f2n := fn * fn + sqfnpl;
      f2npl := 2 * fn * fnpl + sqfnpl;
      if d[k] = 0 then
        begin {reassign nth and (n + 1)th ready for next doubling
        phase}
          fn := f2n;
          fnpl := f2npl
        end
      else
        begin {extend sequence by one and reassign nth and
        (n + 1)th}
          fn := f2npl;
          fnpl := f2npl + f2n
        end
      end;
      {assert: fn = the nth fibonacci number}
      nfib := fn
    end
```

Note di progetto

1. Per un dato valore di n , l'algoritmo richiede un numero di passi dell'ordine di $\log_2 n$ per calcolare l' n -esimo numero di Fibonacci.
2. Il while-loop genera la rappresentazione binaria dell'intero n per tutti i valori $n > 1$. Dopo l' i -esima iterazione, sono stati prodotti i primi i bit a destra della rappresentazione binaria di n ; al termine, quando $n = 1$ sono stati prodotti tutti i bit della rappresentazione di n tranne il primo a sinistra.
Dopo la j -esima iterazione del loop for (j non è definito esplicitamente), è stato prodotto un numero di Fibonacci fn che corrisponde ai primi $(j + 1)$ bit a sinistra di n . Al termine, quando $j = i$ e $k = 1$ la variabile fn corrisponde all' $n - 2$ -esimo numero di Fibonacci.

Il **while**-loop termina poiché n segue una successione strettamente decrescente fino ad 1, a causa delle ripetute divisioni per 2. Per definizione, anche il **for**-loop ha termine.

L'algoritmo funziona correttamente per tutti i valori $n \geq 1$. Per valori elevati di n , il valore di f_n supera rapidamente i limiti di rappresentazione degli interi nella maggior parte dei computer; in questo algoritmo non si è attuato nessun accorgimento per proteggere da questo rischio.

3. In questo algoritmo, siamo stati in grado di trasferire conoscenze di progetto acquisite nell'algoritmo precedente.
4. L'algoritmo in sé non è particolarmente utile. Tuttavia la tecnica che esso illustra è molto importante.
5. In questo problema siamo stati capaci di procedere avendo un obiettivo (ossia la maniera con cui volevamo tentare di calcolare l' n -esimo numero di Fibonacci), e di raggiungere lo scopo utilizzando un esempio specifico per inventare la formula necessaria.
6. Si osservi che per risparmiare sulle moltiplicazioni è preferibile precalcolare f_{n+1} .

Problemi supplementari

- 3.8.1 Sviluppare una implementazione ricorsiva per il calcolo dell' n -esimo numero di Fibonacci, che includa le idee precedenti. Confrontare le prestazioni del metodo ricorsivo e di quello iterativo.
- 3.8.2 Quale successione di coppie di numeri di Fibonacci sarebbe necessaria per calcolare il 23-esimo numero di Fibonacci, utilizzando il presente algoritmo?
- 3.8.3 È possibile moltiplicare due numeri x ed y con procedimento iterativo, dividendo y (con divisione intera) quando è pari e diminuendolo di 1 quando è dispari.
Quando y è dispari, si accumula il valore corrente di x ; quando è pari, x si raddoppia. Implementare questo algoritmo di moltiplicazione. (*Nota:* Le operazioni di raddoppio e di dimezzamento corrispondono alle operazioni di "shift" [scorrimento] che sono molto efficienti sulla maggior parte dei computer).
- 3.8.4 È possibile calcolare $n!$ in un numero di passi $\mathcal{O}(\log n)$. Cercare di sviluppare tale algoritmo per il calcolo di $n!$ (vedi: A. Shamir (1979), "Factoring numbers in $\mathcal{O}(\log n)$ arithmetic steps", *Inf. Proc. Letts.*, 8, 28-31).

CAPITOLO 4

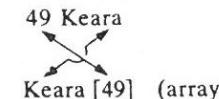
TECNICHE DI GESTIONE DEGLI ARRAY

INTRODUZIONE

L'array è un potente strumento ampiamente usato in programmazione. Gli array servono ad immagazzinare ed organizzare i dati nella memoria di un calcolatore: la loro potenza deriva soprattutto dal fatto che ci forniscono un modo molto semplice ed efficace di eseguire e fare riferimento a computazioni su collezioni di dati che condividono attributi comuni.

Chiunque inizia a lavorare su un calcolatore impiega molto tempo e necessita di un certo numero di esempi per poter apprezzare completamente i diversi modi in cui possono essere usati gli array in programmazione. Un'analogia che spesso aiuta a chiarire il concetto è quella di pensare che un array sia simile ad una strada con una fila di case. L'attributo che tutte le case hanno in comune è il nome della via.

Ogni casa della via ha un unico indirizzo che la distingue da tutte le altre case della stessa via. L'indirizzo è formato da due parti, il nome della via ed il numero. Il nome della via corrisponde al nome dell'array ed il numero corrisponde al suffisso.



Ogni singola casa, col proprio indirizzo, corrisponde all'unità di allocazione in memoria o "parola del computer" (o byte) di ogni indirizzo dell'array. Le persone ed il mobilio ecc. sono contenute in case, mentre noi immagazziniamo numeri o caratteri attraverso rappresentazioni binarie di lunghezza fissa in locazioni dell'array. È una copia del numero o dei caratteri immagazzinati in una locazione dell'array che noi recuperiamo quando ci riferiamo ad una particolare locazione dell'array specificando il suo nome ed il suffisso.

Il trattamento degli array è semplificato dall'uso di variabili per specificare i suffissi. Trattando un particolare array non importa quale nome si dia al suffisso — è solo il valore del suffisso che determina a quale locazione dell'array ci si riferisce (ad esempio, se le variabili i e j hanno entrambe il valore 49, $a[i]$ ed $a[j]$ si riferiscono entrambe alla stessa locazione dell'array, cioè ad $a[49]$).

Il concetto di array unidimensionale che abbiamo considerato si estende in maniera molto semplice agli array multidimensionali.

Gli array sono parte integrante di molti algoritmi per il computer. Essi semplificano l'implementazione di algoritmi che devono rappresentare le stesse computazioni su insiemi di dati. Inoltre, l'impiego di array, spesso porta ad implementazioni di algoritmi che sono più efficienti di come sarebbero altrimenti.

Il contenuto di una locazione dell'array può essere cambiato ed usato in espressioni condizionali come una singola variabile. Anche il suffisso può essere il risultato di una computazione così come una singola variabile (ad esempio $a[2*i + 1]$ ed $a[j - 1]$ sono validi riferimenti a locazioni di un array purché il suffisso risultante sia all'interno dei limiti dell'array).

È possibile cambiare il contenuto di una locazione di un array principalmente mediante computazione diretta ed assegnamento, scambio dei contenuti di due celle dell'array e conteggio. Negli algoritmi che seguono esamineremo una varietà di tecniche e di applicazioni di array. Nei capitoli seguenti vedremo inoltre come vengono usati gli array per simulare pile e code ed altre importanti strutture dati.

Strutture dati non lineari come alberi e grafi possono essere rappresentate usando array mono e bidimensionali.

In applicazioni di programmazione più avanzate gli array possono essere usati per costruire e simulare automi a stati finiti.

ALGORITMO 4.1 INVERSIONE DELL'ORDINE DI UN ARRAY

Problema

- Disporre gli elementi di un array in modo che essi risultino in ordine inverso.

Sviluppo dell'algoritmo

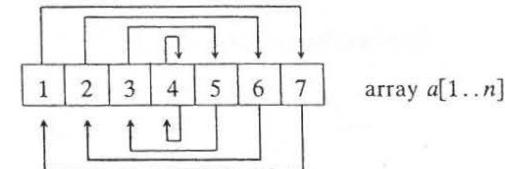
Il problema di invertire l'ordine di un array di numeri sembrerebbe estremamente semplice. Anche se ciò è sostanzialmente vero, è tuttavia necessaria un'attenta riflessione per implementare l'algoritmo.

Possiamo iniziare lo studio dell'algoritmo esaminando attentamente gli elementi di un array prima e dopo la loro inversione; ad esempio:

1	2	3	4	5	6	7
prima dell'inversione						

7	6	5	4	3	2	1
dopo l'inversione						

Ciò che appare dal diagramma è che il primo elemento va a finire nell'ultima posizione, il secondo elemento nella penultima posizione e così via. Continuando in questo modo si arriva all'insieme di scambi:



In termini di suffissi gli scambi sono:

- passo [1] $a[1] \leftarrow a[7]$
- passo [2] $a[2] \leftarrow a[6]$
- passo [3] $a[3] \leftarrow a[5]$
- passo [4] $a[4] \leftarrow a[4]$ qui non ci sono scambi

Esaminando gli effetti di questi scambi vediamo che dopo il passo [3] l'array è completamente invertito. Vediamo che ad ogni passo i suffissi a sinistra aumentano di uno, mentre contemporaneamente quelli a destra diminuiscono di uno.

Nell'implementazione del nostro algoritmo occorrono un paio di suffissi per gli incrementi e i decrementi. Come suffisso incrementale può essere usata la variabile i che viene aumentata di un'unità ad ogni passo.

Per il suffisso decrescente possiamo usare $[n - i]$ che diminuisce di una unità ogni volta che i si incrementa di un'unità. Sorge però il problema che quando $i = 1$, $[n - i]$ è uguale a 1 anziché ad n come è richiesto per il nostro scambio. Ciò può essere corretto aggiungendo 1. Il suffisso $[n - i + 1]$ può essere allora usato come suffisso decrescente. Con tali suffissi abbiamo:

$$\begin{array}{lll} i & n - i + 1 \\ 1 & 7 - 1 + 1 = 7 \\ 2 & 7 - 2 + 1 = 6 \\ 3 & 7 - 3 + 1 = 5 \\ 4 & 7 - 4 + 1 = 4 \end{array}$$

Ogni scambio (cfr. algoritmo 2.1) può essere portato a termine mediante un meccanismo del tipo:

$$\begin{aligned} t &:= a[1]; \\ a[i] &:= a[n - i + 1]; \\ a[n - i + 1] &:= t \end{aligned}$$

L'ultimo aspetto dell'algoritmo da considerare è il campo di valori che può assumere i per un dato n . Studiando il nostro array originario è chiaro che sono necessari solo 3 scambi per permutare array di 6 o 7 elementi. Considerazioni fatte su alcuni esempi ci portano alla generalizzazione che il numero degli scambi r per permutare l'ordine di un array è sempre il più piccolo intero minore o uguale alla metà di n .

Descrizione dell'algoritmo

1. Definisci l'array $a[1..n]$ di n elementi da invertire.
2. Calcola il numero r di scambi necessari per invertire l'array.
3. Finché ci sono ancora coppie di elementi dell'array da scambiare,
 - (a) scambia l' i -esimo elemento con l' $[n - i + 1]$ -esimo.
4. Ritorna l'array invertito.

L'algoritmo può essere opportunamente implementato come procedura avente in ingresso l'array da invertire e restituisca in uscita l'array invertito.

Implementazione in Pascal

```

procedure reverse (var a: nelements; n: integer);
var i {increasing index for array},
    r {number of exchanges required}: integer;
    t {temporary variable needed for exchange}: real;

begin {reverses an array of n elements}
  {assert: n>0 ∧ a[1]=a1, a[2]=a2,...,a[n]=a(n)}
  r := n div 2;
  {invariant: 1 = < i = < [n/2] ∧ a[1]=a(n), a[2]=a(n-1),...,a[i]=a(n-i+1),
  a[i+1]=a(i+1), a[n-i]=a(n-i), a[n-i+1]=a(i), ..., a[n]=a1}
  for i := 1 to r do
    begin {exchange next pair}
      t := a[i];
      a[i] := a[n-i+1];
      a[n-i+1] := t
    end;
  {assert: a[1]=a(n), a[2]=a(n-1),...,a[n-1]=a2, a[n]=a1}
end;

```

Note di progetto

1. Per permutare un array di n elementi occorrono $n/2$ scambi.
2. Ci sono $r = n/2$ coppie di elementi in un array di n elementi. Per permutare un array di n elementi devono essere scambiate r coppie di elementi. Dopo l' i -esima iterazione (per $1 \leq i \leq r$) devono essere

scambiate tra loro le prime i coppie di elementi. Tale relazione resta invariata. L' i -esima coppia consiste dell' i -esimo elemento e dell' $(n - i + 1)$ -esimo. L'algoritmo termina perché ad ogni iterazione i si incrementa di un'unità, cosicché alla fine saranno state scambiate r coppie di elementi.

3. In pratica, ogni volta che bisogna accedere agli elementi dell'array in ordine inverso, non è necessario permutare realmente il loro ordine; ciò può essere fatto usando semplicemente una variabile decrescente come $(n - i + 1)$. In alternativa con un'appropriata inizializzazione si può usare $j := j - 1$.
4. Un esempio aiuta a stabilire quante coppie di scambi sono in generale necessarie per array di dimensioni pari e dispari.
5. Per scambiare una coppia di elementi di un array viene impiegata la stessa tecnica usata per scambiare una coppia di variabili singole.
6. Esiste un semplice algoritmo di permutazione di array che parte con due indici, $i = 0$ e $j = n + 1$. Ad ogni iterazione i è incrementato e j viene decrementato per $i < j$.

Applicazioni

Trattamento di vettori e matrici.

Problemi supplementari

- 4.1.1 Cosa succede se il processo di scambio continua per n passi anziché per $n/2$?
- 4.1.2 Implementare l'algoritmo di inversione dell'array suggerito nella nota 6.
- 4.1.3 Progettare un algoritmo che porti il k -esimo elemento di un array in posizione 1, il $(k + 1)$ -esimo in posizione 2 ecc. L'elemento inizialmente in posizione 1 viene portato nella posizione $(n - k + 1)$ e così via.
- 4.1.4 Progettare un algoritmo che permuti gli elementi di un array in modo che quelli posti originariamente in posizione dispari siano posti prima di quelli occupanti originariamente posizione pari. Per esempio l'insieme:

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

deve essere trasformato in:

1	3	5	7	2	4	6	8
---	---	---	---	---	---	---	---

ALGORITMO 4.2 COSTRUZIONE DI ISTOGRAMMI MEDIANTE ARRAY

Problema

Dato un insieme di voti di esami di n studenti (nell'intervallo da 0 a 100) calcolare il numero di studenti che hanno ottenuto ogni possibile votazione.

Sviluppo dell'algoritmo

Questo problema incorpora lo stesso principio dell'algoritmo 2.2 dove dovevamo conteggiare il numero di studenti promossi ad un esame. Ciò che dobbiamo fare in questo caso è ottenere la distribuzione di un certo numero di voti. Questo è un tipico problema di calcolo della frequenza.

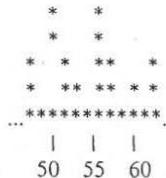
Un approccio potrebbe essere quello di definire 101 variabili $C_0, C_1, C_2, \dots, C_{100}$ ognuna corrispondente ad un particolare voto. La strategia di calcolo da adottare potrebbe essere la seguente:

finché sono stati esaminati meno di n voti

- (a) prendi il prossimo voto m ,
- (b0) se $m = 0$ allora $C_0 := C_0 + 1$;
- (b1) se $m = 1$ allora $C_1 := C_1 + 1$;
- (b2) se $m = 2$ allora $C_2 := C_2 + 1$;
- (b3) se $m = 3$ allora $C_3 := C_3 + 1$;
- .
- .
- .
- (b100) se $m = 100$ allora $C_{100} := C_{100} + 1$.

La difficoltà di questo approccio è che dobbiamo fare 101 test (di cui solo uno con esito positivo) per calcolare il valore di un particolare voto; inoltre il nostro programma è molto lungo. Sembra quindi che ci debba essere un modo più semplice per risolvere il problema.

Per iniziare la ricerca di una soluzione migliore pensiamo per prima cosa a come si potrebbe risolvere il problema senza il calcolatore. Un approccio che potremmo utilizzare è illustrato dal seguente diagramma:

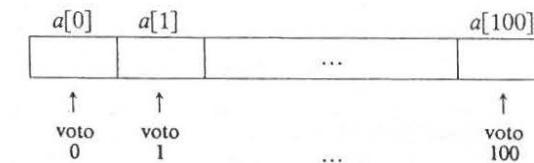


È necessario prendere un pezzo di carta per grafici o dividere un foglio da pianificazione in tracce ognuna delle quali sia identificata da un particolare valore del voto. Il passo successivo è esaminare ciascun voto e, a seconda del suo valore, porre un asterisco nella traccia corrispondente.

Se applichiamo questo passo a tutti i voti, una volta completato il lavoro, il numero di asterischi associati ad ogni traccia rappresenta il numero di volte in cui è stato assegnato quel particolare voto.

Il metodo che è stato delineato rappresenta certamente una soluzione realizzabile manualmente. La domanda che ora dobbiamo porci è: possiamo utilizzare queste idee per sviluppare una soluzione del problema usando il calcolatore? Nel metodo proposto non è necessario confrontare ogni voto con tutti i suoi possibili valori. Il valore di un particolare voto, invece, ci porta direttamente alla traccia che deve essere aggiornata. Questa procedura ad un solo passo per ogni voto sarebbe certamente avvincente se potesse essere implementata mediante un algoritmo.

È a questo punto che dobbiamo riconoscere che un array può essere vantaggiosamente impiegato nella soluzione del problema. Possiamo costruire molto facilmente un array con 101 locazioni, ognuna delle quali corrisponda ad un particolare valore del voto. Ad esempio:



Se memorizziamo in ogni posizione dell'array il "conto" del numero di studenti che hanno ottenuto un certo voto, noi avremo la soluzione del problema (ad esempio, se 15 studenti hanno ottenuto il voto 57, troveremo il numero 15 in posizione 57 quando tutti i voti saranno stati esaminati).

Ciò che dobbiamo ora considerare è come si perviene al risultato per ogni locazione dell'array. Nell'elaborare tale meccanismo vogliamo cercare di utilizzare la procedura ad un passo impiegata nella soluzione manuale. Consideriamo innanzitutto cosa succede quando si incontra un particolare voto. Supponiamo che il voto corrente da prendere in esame sia 57. Utilizzando l'array per il conteggio, possiamo a questo punto addizionare uno al valore contenuto nella locazione 57. In questo passo possiamo usare il vero valore del voto (cioè 57) per riferirci

alla locazione dell'array che vogliamo aggiornare. In questo modo il valore del voto può essere impiegato come un suffisso dell'array. Poiché è necessario incrementare di un'unità il valore precedente contenuto nella locazione 57 occorre una istruzione del tipo:

nuovo valore in locazione 57 := valore precedente in locazione 57 + 1

Poiché la locazione $a[57]$ deve giocare entrambi i ruoli di valore precedente e nuovo valore, possiamo scrivere:

$a[57] := a[57] + 1$

o, per un qualsiasi voto m ,

$a[m] := a[m] + 1$

Quest'ultima istruzione costituisce la base del nostro algoritmo di conteggio dei voti. Utilizzando il valore del voto per indirizzare l'appropriata locazione dell'array, abbiamo modellato il metodo diretto della soluzione manuale.

Per completare l'algoritmo, tutto quello che è necessario è includere i dettagli per l'input dei voti e l'output della distribuzione della frequenza dei voti. Prima di iniziare il procedimento di conteggio, l'array deve avere tutti i suoi elementi inizializzati a zero. I dettagli dell'algoritmo sono dati di seguito.

Descrizione dell'algoritmo

1. Prepara e leggi in n il numero dei voti che devono essere computati.
2. Inizializza tutti gli elementi dell'array di conteggio $a[0..100]$ a zero.
3. Finché ci sono ancora dei voti che devono essere analizzati ripeti.
 - (a) leggi il nuovo voto m ,
 - (b) incrementa di un'unità il valore in locazione m dell'array di conteggio.
4. Stampa la distribuzione della frequenza dei voti.

Implementazione in Pascal

```

program histogram (input, output);
var i {current number of marks processed},
    m {current mark},
    n {number of marks to be processed}: integer;
    a: array [0..100] of integer;

begin {compute marks frequency distribution}
  writeln ('enter number of marks n on a separate line followed by
            marks');
  readln (n);
  for i := 1 to 100 do a[i] := 0;
  {assert: n >= 0 ∧ all a[0..100] are set to 0}.
  {invariant: when i marks read, for j in range 0 = < j = < 100, all a[j]
  will represent the number of marks j in the first i read ∧ i = < n}
  for i := 1 to n do
    begin {read next mark and update appropriate array elements}
      read (m);
      if eoln (input) then readln;
      {assert: m in range 0 = < m = < 100}
      a[m] := a[m] + 1
    end;
  {assert: when n marks read, for j in range 0 = < j = < 100, all a[j]
  will represent the number of marks j in the set}
  for i := 0 to 100 do
    begin
      write (a[i]);
      if i mod 8 = 0 then writeln
    end
  end.

```

Note di progetto

1. Per un insieme di n voti sono richiesti essenzialmente n passi per generare gli istogrammi della frequenza.
2. Dopo i iterazioni il j -esimo elemento dell'array a conterrà un intero rappresentante il numero dei voti j incontrati nelle prime i posizioni. Questa relazione è valida per tutti i j compresi nell'intervallo $0 \leq j \leq 100$ e per tutti gli i tali che $1 \leq i \leq n$. Al termine, quando $i = n$, tutti gli elementi dell'array conterranno l'appropriato conteggio dei voti di tutto l'insieme.
3. Dalla definizione di loop **for** segue che entrambi i cicli terminano.
3. L'idea di indirizzamento per valore è importante per molti algoritmi per la sua efficienza.

Applicazioni

Analisi statistiche.

Problemi supplementari

- 4.2.1 Modificare l'algoritmo precedente in modo da ottenere un istogramma ogni 10% dei voti (ad esempio 0 → 10%, 11 → 20%,...) anziché per ogni singolo voto.
- 4.2.2 Si vuole generare un istogramma della distribuzione di un insieme di temperature medie giornaliere registrate nell'Antartide. Le temperature sono valori interi compresi tra -40° C e $+5^{\circ}\text{ C}$. Progettare un algoritmo che riceva in input gli n valori delle temperature e produca l'appropriata distribuzione.
- 4.2.3 Modificare l'algoritmo dei voti in modo da ottenere il voto più basso e quello medio dell'insieme. Il voto medio è quel valore che è stato attribuito ad almeno la metà dei candidati, o un voto leggermente inferiore.

ALGORITMO 4.3 RICERCA DEL VALORE MASSIMO IN UN INSIEME

Problema

Trovare il valore massimo in un insieme di n numeri.

Sviluppo dell'algoritmo

Prima di iniziare a lavorare sull'algoritmo per trovare il massimo, occorre avere un'idea chiara circa il concetto di massimo. Dopo alcune considerazioni possiamo concludere che il massimo è quel numero che è maggiore o uguale a tutti gli altri numeri dell'insieme. Questa definizione sottolinea il fatto che il massimo può non essere unico; implica inoltre che il massimo è definito solo per insiemi di uno o più elementi.

Per iniziare lo sviluppo dell'algoritmo per questo problema esaminiamo un particolare insieme di numeri. Ad esempio:

8	6	5	15	7	19	21	6	13
---	---	---	----	---	----	----	---	----

Dopo aver studiato questo esempio, possiamo concludere che per stabilire il massimo, dovranno essere esaminati *tutti* i numeri. Una seconda conclusione è che occorre fare un confronto tra le grandezze relative dei numeri.

Prima di procedere con l'algoritmo, consideriamo come si potrebbe risolvere il problema senza il calcolatore. Se abbiamo una lista con pochi numeri e vogliamo cercare il massimo, scorriamo semplicemente la lista alla ricerca della risposta. Per la lista precedente possiamo rapidamente essere in grado di rispondere che il massimo è 21. Il meccanismo che "ci sembra" di applicare è di esaminare i numeri e di selezionarne uno che "sentiamo" essere più grande degli altri.

Riesaminiamo poi i numeri per verificare la validità della nostra affermazione. Se incontriamo un numero più grande, riesaminiamo la lista confrontando i numeri con il nuovo candidato. Questo processo viene ripetuto finché non siamo soddisfatti della nostra scelta. Per liste con pochi numeri, tale processo è così rapido che generalmente non ci rendiamo conto dei dettagli del meccanismo che usiamo.

Per liste più lunghe (di un centinaio di numeri o più) non è così facile applicare la strategia che abbiamo delineato.

Dobbiamo allora semplificare il nostro approccio per prevenire la possibilità di andare al di là delle nostre capacità.

Il modo più semplice e più sistematico di esaminare ogni valore in una lista è incominciare dal primo elemento della lista, analizzarla numero per numero, finché non si raggiunge la fine. Non è però sufficiente solo analizzare ogni valore, ad ogni passo occorre operare un confronto.

Immaginiamo per un momento di avere il compito di trovare il massimo di un centinaio di numeri avendoli proiettati uno alla volta su uno schermo. Tale compito è collegato al problema che occorre risolvere per implementare l'algoritmo per il calcolatore. Quando appare sullo schermo il *primo* numero non possiamo sapere se esso sia o no il massimo. In questa situazione la cosa migliore che possiamo fare è memorizzare il candidato come se fosse il massimo. Dopo aver deciso di memorizzare il primo numero, dobbiamo ora decidere cosa fare quando apparirà sullo schermo il secondo.

Sono possibili tre situazioni:

1. il secondo numero può essere *minore* del nostro candidato temporaneo al massimo;
2. il secondo numero può essere *uguale*;
3. il secondo numero può essere *maggior*.

Se siamo in presenza della situazione (1) o (2) il nostro temporaneo candidato a massimo è ancora valido e quindi non deve essere cambiato. In questa circostanza possiamo semplicemente andare oltre e confrontare il terzo numero col nostro massimo temporaneo che chiameremo *max*.

Se il secondo numero è maggiore del primo, dobbiamo cancellare il candidato al massimo precedente e memorizzare il secondo numero; andiamo quindi oltre e confrontiamo il terzo numero con il nuovo

massimo temporaneo. L'intero processo deve continuare finché tutti gli elementi dell'insieme non siano stati esaminati. Man mano che si incontrano valori maggiori essi assumono il ruolo di massimo temporaneo. Nel momento in cui sono stati esaminati tutti i numeri il massimo temporaneo memorizzato è il massimo di tutto l'insieme: tale strategia può costituire la base del nostro algoritmo per il calcolatore.

Come per la maggior parte degli algoritmi dobbiamo preoccuparci di porre delle corrette condizioni iniziali.

Possiamo usare una procedura iterativa per esaminare tutti gli elementi dell'insieme (supponiamo che si trovino effettivamente in un array). La nostra proposta iniziale potrebbe quindi essere:

1. Finché tutti gli elementi dell'array non sono stati esaminati
 - (a) se l'elemento corrente dell'array è maggiore del massimo temporaneo, aggiorna il massimo temporaneo.

Dopo aver iniziato ci accorgiamo che esiste un problema con questo procedimento. Non esiste massimo temporaneo nel momento in cui viene considerato il primo elemento, cosicché non può essere fatto un confronto valido con esso.

Dobbiamo perciò cercare di aggirare il problema.

Ricordando l'esempio dei numeri sullo schermo, vediamo che quando appare il primo numero lo consideriamo come massimo temporaneo senza alcun confronto. Possiamo applicare la stessa idea al nostro algoritmo assegnando inizialmente alla variabile temporanea *max* il valore del primo elemento della lista. Con questo abbiamo risolto il problema minore. Tutti gli altri elementi possono essere esaminati iterativamente ed i confronti possono essere fatti con un valido massimo temporaneo. Con queste modifiche all'approccio iniziale abbiamo gli elementi essenziali per il nostro algoritmo di ricerca del massimo.

Descrizione dell'algoritmo

1. Definisci un array $a[1..n]$ di n elementi, con $n \geq 1$.
2. Assegna alla variabile temporanea *max* il valore del primo elemento dell'array.
3. Finché sono stati considerati meno di n elementi dell'array esegui
 - (a) se l'elemento successivo è maggiore del corrente massimo *max*, assegna a *max*.
4. Ritorna il massimo *max* dell'array di n elementi.

Questo algoritmo può essere implementato come una funzione che riceva in ingresso un array di lunghezza n e dia in uscita *max*, il massimo valore nell'array di n numeri.

Implementazione in Pascal

```
function amax (a: nelements; n: integer): real;
var i {array index}: integer;
    max {current maximum}: real;

begin {find the maximum in an array of n numbers}
  {assert: n>0}
  i := 1;
  max := a[i];
  {invariant: max is maximum in a[1..i]∧i=<n}
  for i := 2 to n do
    if a[i]>max then max := a[i];
  {assert: max is maximum in a[1..n]}
  amax := max
end
```

Note di progetto

1. Il numero di confronti che occorrono per trovare il massimo in un array di n elementi è $n - 1$.
2. Per ogni i tale che $1 \leq i \leq n$, quando i elementi sono stati esaminati, la variabile *max* è maggiore o uguale a tutti gli elementi compresi tra 1 e i . Questa relazione rimane invariata durante tutta la computazione. Al termine, quando $i = n$, la variabile *max* contiene il valore massimo dell'insieme. L'algoritmo termina perché ad ogni iterazione la variabile *i* viene incrementata di un'unità e quindi alla fine si raggiunge la condizione $i = n$. L'algoritmo funziona correttamente per valori di $n \geq 1$.
3. Un modo di stabilire il passo iniziale è quello di considerare il problema più piccolo possibile che possa essere risolto con l'algoritmo. In questo caso il problema più semplice è quello di trovare il massimo di un array di un numero (cioè con $n = 1$). Questo problema può essere risolto con il diretto assegnamento della variabile *max*. Stabilito il passo iniziale, problemi più ampi, (cioè con $n > 1$), possono essere risolti con l'iterazione.
4. Spesso algoritmi per trovare il massimo vengono inizializzati ponendo *max* = 0. È un metodo di programmazione sbagliato assegnare inizialmente a *max* un qualsiasi valore costante (infatti il programma darebbe esito errato se tutti i valori fossero negativi e *max* fosse stato inizializzato a zero).
5. Una generalizzazione di questo problema è quella di trovare il k -esimo valore più piccolo di un array. Tale problema verrà considerato più avanti.

Applicazioni

Diagrammi, misurazioni in scala, ordinamento.

Problemi supplementari

- 4.3.1 Progettare un algoritmo per trovare il minimo di un array.
- 4.3.2 Progettare un algoritmo per trovare il numero di volte in cui ricorre il massimo in un array di n elementi. L'array deve essere scandito una sola volta.
- 4.3.3 Progettare un algoritmo per trovare il massimo in un insieme e la posizione
 - (a) in cui ricorre la prima volta;
 - (b) in cui ricorre l'ultima volta.
- 4.3.4 Progettare un algoritmo per trovare la massima differenza assoluta tra una coppia di elementi adiacenti di un array di n elementi.
- 4.3.5 Progettare un algoritmo per trovare il secondo valore massimo in un array di n elementi.
- 4.3.6 Trovare la posizione di un numero x (se ricorre) in un array di n elementi.
- 4.3.7 Progettare un algoritmo per trovare il massimo in un insieme confrontando ogni numero con tutti gli altri.
L'algoritmo terminerà quando verrà trovato un numero che sia maggiore o uguale a tutti gli altri. Cosa succede se cercate la soluzione più efficiente per questo metodo?
- 4.3.8 Progettare un algoritmo per trovare il minimo, il massimo e quante volte entrambi ricorrono in un array di n elementi.
- 4.3.9 Il minimo ed il massimo di un array di n elementi possono essere trovati usando $(3/2)n$ confronti anziché $2n$, confronti considerando gli elementi a coppie e confrontando l'elemento maggiore della coppia con il massimo corrente ed il minore della coppia con il minimo corrente. Implementare questo algoritmo.

ALGORITMO 4.4 ELIMINAZIONE DEI DOPPI IN UN ARRAY ORDINATO

Problema

Rimuovere tutti i duplicati da un array ordinato e conseguentemente ridurre l'array.

Sviluppo dell'algoritmo

Come punto di partenza per questo progetto, focalizziamoci su un esempio specifico in modo da avere idee chiare su cosa sia esattamente richiesto.

1	2	3	4	5	6	7	8	9	10	11	12	13
2	2	8	15	23	23	23	23	26	29	30	32	32

prima della rimozione dei doppi

.....

Dopo un breve esame dell'array saremo in grado di produrre il seguente array contratto:

1	2	3	4	5	6	7	8
2	8	15	23	26	29	30	32

dopo la rimozione dei doppi

Individuato il nostro obiettivo, dobbiamo ora scoprire un meccanismo conveniente per il processo. Confrontando i due array vediamo che tutti gli elementi, a parte il primo, hanno cambiato la loro posizione. In altre parole, ogni singolo elemento dell'array originario è stato spostato il più a sinistra possibile.

Qualunque sia il meccanismo che abbiamo deciso di usare, questo dovrà essere costruito intorno alla scoperta dei doppi tra i dati originali. Si ha una coppia di doppi quando due elementi adiacenti hanno lo stesso valore. Ad ogni confronto sono possibili due sole situazioni:

1. è stata trovata una coppia di doppi;
2. i due elementi sono diversi.

È abbastanza ovvio che queste due situazioni debbano essere trattate in modo differente. Per capire ciò, consideriamo cosa succede quando si incontrano i quattro 23. Abbiamo:

15	23	23	23	23	26
[4]	[5]	[6]	[7]	[8]	

Quando il 15 è confrontato con il 23 (quarto passo) il 23 è l'elemento singolo più recente che sia stato incontrato quindi dovrebbe essere spostato il più a sinistra possibile.

Al passo successivo vengono confrontati due 23. Poiché del 23 si è già tenuto conto al passo precedente, l'unica cosa che possiamo fare è passare alla coppia successiva. Al passo [8] il 26 è confrontato con il 23.

Qui il 26 è il numero singolo incontrato più recentemente. Deve quindi essere ricollocato appropriatamente nell'array.

Uno studio del nostro esempio rivela che la posizione dell'array in cui deve essere posto l'elemento singolo incontrato più di recente è determinata ad ogni istanza dal numero di elementi singolari incontrati finora. Nel caso del 26, esso è il quinto elemento singolo, quindi deve essere posto in posizione 5. Ciò suggerisce l'uso di un contatore, il cui valore ad ogni istanza indica il numero aggiornato di elementi unici incontrati. Se i è la posizione in cui è stato trovato l'elemento singolo più recente e j indica il numero di elementi unici fino ad ora, allora un assegnamento del tipo:

$$a[i] := a[j]$$

modellerà il meccanismo di contrazione.

Ad esempio, quando si incontra il 26 e si memorizza come elemento unico più recente, troviamo la seguente configurazione:

	5		9	
2	8	15	23	23
	\uparrow		\uparrow	
	j		i	

Riassumendo i passi di base del nostro meccanismo abbiamo:

finché non sono state confrontate tutte le coppie di elementi adiacenti

- (a) se non sono uguali sposta l'elemento più a destra della prossima coppia nella posizione dell'array determinata dal calcolo dell'elemento singolo corrente.

Dopo aver stabilito il meccanismo di base, occorre progettare i dettagli del processo di inizializzazione.

Guardando l'esempio, possiamo concludere che il primo elemento dell'array non viene mai toccato. Per prendere in considerazione il fatto che i primi due elementi possono essere identici, il processo di confronto deve iniziare da essi. Per fare ciò abbiamo due possibili scelte per l'inizializzazione:

1. $i := 1$
 $a[i] = a[i + 1]?$ (confronto)
2. $i := 2$
 $a[i - 1] = a[i]?$ (confronto)

Se pensiamo a come terminare l'algoritmo, scopriamo che la seconda alternativa è migliore perché permette all'algoritmo di terminare direttamente quando i diventa uguale ad n , numero di elementi nell'array originario.

Esplorando il problema si poteva notare che se nell'array non esistono doppi, allora l'assegnamento

$$a[j] := a[i]$$

non è necessario, perché tutti gli elementi si trovano già nella loro corretta posizione. Anche se in un array sono presenti doppi, bisogna iniziare a spostare gli elementi solo dopo aver incontrato il primo. Come possono essere adattate queste situazioni? Un modo è quello di confrontare coppie di elementi finché non viene incontrato un doppio. Per questo motivo possiamo usare un ciclo del tipo:

while $a[i-1] < > a[i]$ **do** $i := i + 1$

Quando si incontra un doppio, l'unico elemento contato sarà $i - 1$. La variabile j deve avere questo valore.

Con questo ciclo esiste un problema di terminazione se nell'array non sono presenti doppi; il modo più semplice per aggirarlo è forzare la terminazione del ciclo includendo il test $i < n$.

Possiamo ora descrivere più dettagliatamente l'algoritmo.

Descrizione dell'algoritmo

1. Definisci l'array $a[1..n]$ di n elementi.
2. Assegna all'indice di ciclo i il valore 2 per permettere una corretta terminazione.
3. Confronta le coppie successive di elementi finché non incontri un doppio, quindi assegna al contatore degli elementi unici j il valore $i - 1$.
4. Finché non sono state esaminate tutte le coppie
 - (a) se la prossima coppia non è doppia, allora
 - (a.1) incrementa di uno il contatore degli elementi unici j ,
 - (a.2) muovi l'ultimo elemento della coppia nella posizione dell'array determinata dal valore del contatore degli elementi unici j .

Implementazione in Pascal

```
procedure duplicates (var a: nelements; var n: integer);
var i {at all times i-1 is equal to the number of pairs examined},
    j {current count of the number of unique elements encountered};
    integer;
```

```
begin {deletes duplicates from an ordered array}
  {assert: n>1^elements a[1..n] in non-descending order}
  i := 2;
  while {a[i-1]<>a[i]} and (i<n) do i := i + 1;
  if a[i-1]<>a[i] then i := i + 1;
```

```

{assert:  $i \geq 2 \wedge a[1..i-1]$  unique  $\wedge$  in ascending order}
j := i - 1;
{invariant: after the  $i$ th iteration  $j \leq i-1 \wedge i \leq n+1 \wedge$  there are no
equal adjacent pairs in the set  $a[1..j]$ }
while  $i < n$  do
begin {examine next pair}
i := i + 1;
if  $a[i-1] \neq a[i]$  then
begin {shift latest unique element to unique count position}
j := j + 1;
a[j] := a[i]
end
end;
{assert: there are no equal adjacent pairs in  $a[1..j]$ }
n := j
end

```

Note di progetto

- Per cancellare gli elementi doppi da un array di n elementi, occorrono $(n-1)$ confronti. Il numero di movimenti di dati (come l'istruzione $a[j] := a[i]$) richiesti sono nel migliore dei casi 0 e nel peggiore $(n-2)$. In generale tale numero sarà un valore compreso tra questi due estremi.
- Alla fine di ogni iterazione, la variabile j rappresenta il numero di elementi unici incontrati dopo l'esame della i -esima coppia di elementi. I j elementi unici incontrati sono posti nelle prime j locazioni dell'array.
Al termine, quando $i = n$ o $(n+1)$, il valore di j rappresenterà il numero di elementi unici contenuti nell'insieme dei dati originari in ingresso. L'algoritmo terminerà perché la variabile i aumenta di un valore ad ogni iterazione. L'algoritmo funzionerà correttamente per valori di $n \geq 1$. Nel caso in cui tutti gli elementi siano unici il secondo while – loop non viene compiuto.
- Può essere concettualmente conveniente in questo algoritmo pensare ai dati originari come ad un array sorgente ed ai dati finali con gli elementi doppi rimossi come ad un insieme di dati pozzi.
- Un esempio specifico è molto utile nello sviluppo dell'algoritmo.
- L'inclusione del primo while – loop previene movimenti non necessari dei dati.

Applicazioni

Problemi di compressione di dati e di elaborazione testi.

Problemi supplementari

- Rimuovere da un array ordinato tutti i numeri che ricorrono più di una volta.
- Cancellare da un array ordinato tutti gli elementi che ricorrono più di k volte.
- Fornire un esempio della configurazione di un array che porti ad $(n-2)$ operazioni di movimento di dati.
- Progettare un algoritmo per memorizzare un array ordinato avente molti elementi doppi. Si può considerare l'array privo di elementi negativi.
- Dato un array ordinato di grandi dimensioni contenente molti elementi che siano multiricorrenti, progettare un algoritmo di cancellazione dei duplicati che sia più efficiente del precedente.

ALGORITMO 4.5 PARTIZIONE DI UN ARRAY

Problema

Dato un array ordinato a caso di n elementi, dividere gli elementi in due sottogruppi tali che gli elementi $\leq x$ siano in un sottogruppo, e quelli $> x$ siano nell'altro.

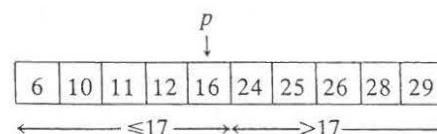
Sviluppo dell'algoritmo

Questo problema è pertinente ad alcuni algoritmi di ordinamento e di ricerca del valore medio. Per cercare di puntualizzare ciò che va fatto, possiamo considerare un particolare esempio. Dato il seguente gruppo di dati, vogliamo partizionarlo in due sottogruppi, uno contenente gli elementi ≤ 17 , l'altro quelli > 17 .

$a[1]$	$a[10]$								
28	26	25	11	16	12	24	29	6	10

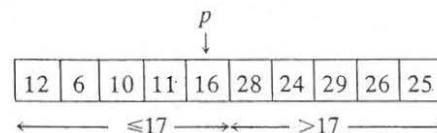
Dobbiamo chiaramente essere in grado di separare i due sottogruppi. Per fare questo potremmo porre tutti gli elementi > 17 all'inizio dell'array e quelli ≤ 17 in fondo.

Un metodo semplice per operare questo trasferimento è quello di ordinare l'array in ordine crescente, dopodiché troviamo la seguente configurazione.



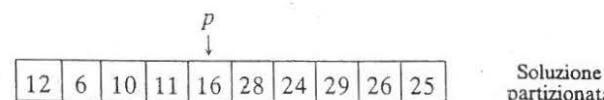
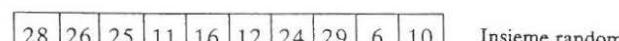
Dopo l'ordinamento possiamo facilmente trovare la posizione che partiziona gli elementi nei due sottogruppi voluti.

Con questa soluzione abbiamo ordinato i due sottogruppi oltre ad averli separati. Originariamente, il problema non richiedeva che gli elementi fossero ordinati. Per il nostro esempio, la configurazione seguente:



avrebbe potuto soddisfare benissimo alle richieste del problema. Bisogna notare che in questo gruppo, mentre i dati sono ancora partizionati in due sottogruppi, gli elementi non sono più ordinati. L'ordinamento di dati è di solito un'operazione costosa: possiamo perciò pensare che ci possa essere una soluzione del problema più semplice e meno costosa della nostra proposta originaria.

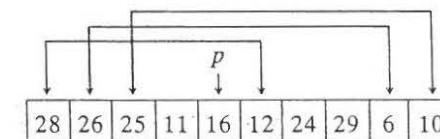
Se non sembra necessario l'ordinamento, ci possiamo allora chiedere quali altre alternativeabbiamo. Per venirne a capo, riesaminiamo e confrontiamo allora il gruppo iniziale non ordinato e la soluzione disordinata del problema.



Confrontando questi due gruppi di dati, vediamo che gli elementi all'estrema sinistra > 17 devono essere trasferiti all'estrema destra del punto di partizione p . Gli elementi ≤ 17 che si trovano a sinistra di p

non devono essere spostati perché si trovano già nella corretta partizione. Questo risparmio di spostamenti permetterà minori scambi rispetto al metodo di ordinamento. Quando siamo inizialmente in presenza di un gruppo irregolare, non sappiamo quanti elementi siano ≤ 17 . Un modo per superare ciò sarebbe di passare attraverso l'array contando tutti i valori ≤ 17 . Una volta trovato il valore di p possiamo operare un'altra scansione dell'array.

Questa volta, quando incontriamo un valore > 17 a sinistra di p , lo spostiamo a destra di p . Ad esempio, il 28 potrebbe essere posto dove si trova il 12, ed il 12 potrebbe essere posto nella posizione iniziale del 28. Questo concetto potrebbe essere esteso permettendoci quindi di terminare come illustrato qui di seguito.



Ogni volta che incontriamo un valore a sinistra che deve essere spostato nella partizione di destra, basta cercare il successivo valore a destra da spostare nella partizione di sinistra e scambiarli. Quando raggiungiamo p muovendo da sinistra saranno stati eseguiti tutti gli scambi.

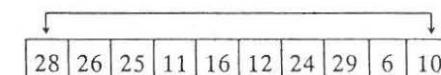
Questo metodo è molto più efficiente di quello che implica ordinamento, perché devono essere fatte solo due scansioni dell'array.

Prima di iniziare questo metodo di partizione occorre cercare di scoprire se sia o no possibile la partizione con un solo passaggio attraverso il gruppo di dati, anziché due.

Nel primo passaggio attraverso il gruppo di dati, nella nostra proposta più recente, non avviene alcuna partizione o scambio, perché non conosciamo dove dovranno essere divise le due partizioni alla fine.

Studiando ancora e più da vicino il gruppo originario, osserviamo che il 28 potrebbe sicuramente essere scambiato con il 10 senza conoscere quale sarà la partizione finale.

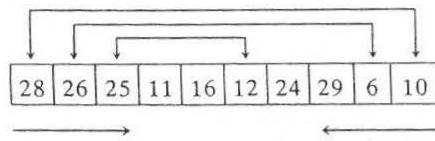
Ovvero:



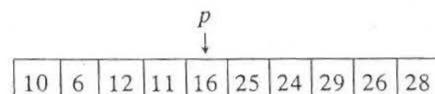
Similmente, il 26 può essere scambiato con il 6. Quello che vuole suggerire questa esplorazione, è che possiamo muovere da entrambi gli estremi. In questo modo stiamo effettivamente accrescendo entrambe le partizioni sinistra e destra verso il mezzo. Cioè:

Partizione sinistra (crescente verso destra)	\rightarrow	Partizione destra (crescente verso sinistra)
--	---------------	--

Se continuiamo questo procedimento di "pinzaggio", le due partizioni si incontreranno alla fine e quando lo faranno noi avremo la partizione desiderata del nostro insieme completo di dati. Elaborando fino in fondo l'esempio precedente, troviamo:



Con questi scambi terminiamo con il seguente insieme partizionato:



Vediamo che questo approccio ci permette di partizionare i dati con un solo passaggio attraverso l'array anziché due.

Possiamo quindi proporre il seguente meccanismo di partizione:

finchè le due partizioni non si sono incontrate,
(estendi le partizioni a sinistra e a destra verso l'interno, scambiando tra loro tutte le coppie in posizione errata nel processo).

Dopo aver delineato il meccanismo di base, dobbiamo volgere la nostra attenzione ai dettagli dell'algoritmo.

La prima considerazione è di tracciare il processo di movimento all'interno. Il movimento verso l'interno da sinistra può procedere finchè non si incontra un elemento maggiore del valore di partizione x (nell'esempio precedente $x = 17$). Ciò può essere compiuto da un ciclo del tipo:

```
while  $a[i] \leq x$  do  $i := i + 1$ 
```

Il movimento verso l'interno da destra può procedere finchè non si incontra un elemento minore o uguale ad x ; per questo possiamo usare un ciclo decrescente:

```
while  $a[j] > x$  do  $j := j - 1$ 
```

Il valore iniziale di i deve essere 1 ed il valore iniziale di j deve essere n , numero di elementi dell'array. Non appena i due cicli terminano, abbiamo trovato un elemento in posizione i che deve essere spostato nella partizione di destra, ed un elemento j che deve essere spostato nella partizione di sinistra.

$i = 1$	$a[i] > x$	$a[j] \leq x$	$j = n$
partizione sinistra		Elementi ancora da partizionare	Partizione destra

\longrightarrow i j \longleftarrow

A questo punto l' i -esimo ed il j -esimo elemento devono essere scambiati. Per lo scambio possiamo usare la tecnica standard:

```
t := a[i];
a[i] := a[j];
a[j] := t;
```

Dopo lo scambio possiamo iniziare il processo di movimento all'interno sempre alla posizione $(i + 1)$ e $(j - 1)$.

L'unica considerazione circa l'implementazione che dobbiamo fare implica la terminazione dei cicli. Il processo di pinzaggio deve terminare solo quando le due partizioni si incontrano; cioè il ciclo principale deve proseguire solo finché l'indice i è minore dell'indice j . Ovvvero:

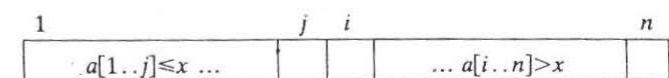
finchè $i < j$

(a) muovi i e j l'uno verso l'altro scambiando ogni coppia in posizione errata nel processo.

Applicando i concetti di movimento verso l'interno e scambio delle coppie in posizione errata, troviamo:

```
while  $i < j$  do
begin
  while  $a[i] \leq x$  do  $i := i + 1$ ;
  while  $a[j] > x$  do  $j := j - 1$ ;
  t := a[i];
  a[i] := a[j];
  a[j] := t;
  i := i + 1;
  j := j - 1
end
```

Con questa struttura a ciclo vediamo che, per come i e j mutano dopo ogni scambio, possono sorpassarsi quando le due partizioni si incontrano, ne conseguo che possiamo trovarci nella situazione seguente:



L'indice j indicherà perciò l'indice più elevato della partizione di sinistra. A questo punto possiamo credere che lo svolgimento sia completato; tuttavia, poiché ci sono due cicli che possono cambiare i e j all'interno del ciclo principale, dobbiamo cercare ulteriormente come dei casi particolari possano influenzare le condizioni di terminazione di tutti e tre i cicli. Ci sono alcune possibili configurazioni che possono essere applicate quando due partizioni stanno per incontrarsi.

Caso 1

Prima del termine

$\boxed{7 \quad 6 \quad 5}$	$x = 6$	$i < j$
$i \quad \quad j$		

Dopo il termine

$\boxed{5 \quad 6 \quad 7}$	$x = 6$	$i = j$
i, j		

Caso 2

Prima del termine

$\boxed{5 \quad 6 \quad 7}$	$x = 6$	$i < j$
$i \quad \quad j$		

Dopo il termine

$\boxed{5 \quad 7 \quad 6}$	$x = 6$	$j < i$
$j \quad \quad i$		

A questo punto esiste un problema, poiché i e j si sorpassano prima dell'ultimo scambio; ciò porterà ad una partizione non corretta dell'array. Torneremo ancora su questo problema, dopo aver considerato gli altri casi.

Caso 3

Prima del termine

$\boxed{7 \quad 5 \quad \quad}$	$x = 6$	$i < j$
$i \quad \quad j$		

Dopo il termine

$\boxed{5 \quad 7 \quad \quad}$	$x = 6$	$j > i$
$j \quad \quad i$		

Caso 4

Prima del termine

$\boxed{5 \quad 7 \quad}$	$x = 6$	$i < j$
$i \quad \quad j$		

Dopo il termine

$\boxed{\quad \quad 7 \quad 5 \quad \quad}$	$x = 6$	$j > i$
$j \quad \quad i$		

Come nel caso 2, ci sono dei problemi nel caso 4, perché i e j possono sorpassarsi prima dell'ultimo scambio.

A questo punto, proponiamoci di risolvere il problema rilevato nei casi 2 e 4. Chiaramente, nel nostro esempio, è sufficiente fare uno scambio quando $i <= j$. Un modo di prevenire una partizione sbagliata sarebbe quello di includere nel testo:

```
if  $i \leq j$  then
    "scambia  $a[i]$  e  $a[j]$ "
```

Il problema di questo nuovo test è che esso viene introdotto per far fronte a due soli casi particolari: è possibile allora un'altra risoluzione del problema? Esso sorge nei casi 2 e 4 perché gli elementi sono già in posizione corretta prima che sia stato fatto l'ultimo scambio.

Nel ciclo principale più esterno abbiamo già la condizione:

```
while  $i < j$  do
```

che non previene l'accadere dello scambio; ci possiamo però chiedere se può essere usato per prevenire lo scambio nei casi 2 e 4.

Riesaminando il meccanismo del ciclo principale vediamo che i seguenti passi di base sono ripetuti più e più volte:

1. muovi verso il centro da sinistra e da destra;
2. scambia le coppie non nella giusta partizione;
1. muovi verso il centro da sinistra e da destra;
2. scambia le coppie non nella giusta partizione;
1. ...
2. ...
- .
- .
- .

Se la condizione del ciclo era di impedire lo scambio, avrebbe dovuto essere posta direttamente prima di esso.
Cioè:

```
while  $i < j$  do
begin
     $t := a[i];$ 
     $a[i] := a[j];$ 
     $a[j] := t;$ 
     $i := i + 1;$ 
     $j := j - 1;$ 
    while  $a[i] \leq x$  do  $i := i + 1;$ 
    while  $a[j] > x$  do  $j := j - 1$ 
end
```

Anche in questo caso c'è un problema, perché al primo passaggio nel ciclo non sappiamo se $a[i]$ e $a[j]$ debbano essere automaticamente scambiati, finché non viene fatta la mossa iniziale verso l'interno da sinistra e destra per stabilire che è necessario uno scambio. Esaminando ulteriormente il meccanismo base, vediamo che il metodo più semplice per aggirare questo problema è portare la prima "mossa verso l'interno da sinistra e destra" fuori dal ciclo come passo iniziale e precondizione del ciclo. Poniamo quindi:

```

while a[i]≤x do i := i+1;
while a[j]>x do j := j-1;
while i<j do
begin
  t := a[i];
  a[i] := a[j];
  a[j] := t;
  i := i+1;
  j := j-1;
  while a[i]≤x do i := i+1;
  while a[j]>x do j := j-1
end

```

Riesaminando i quattro casi test vediamo che non vi sono altre difficoltà di sorpasso. Esiste tuttavia un ulteriore problema con questa implementazione, perché non abbiamo considerato che x possa essere maggiore o minore di tutti gli elementi. In entrambi i casi uno dei cicli – passo iniziale potrebbe provocare un avviso di locazione dell'array fuori dai limiti. La nostra unica alternativa è quella di includere delle verifiche di limite sugli indici i e j nei primi due passi. Essi diventeranno dunque:

```

while (a[i]≤x) and (i<j) do i := i+1;
while (a[j]>x) and (i<j) do j := j-1

```

Quando x è maggiore o uguale a tutti gli elementi dell'array i due cicli termineranno con $i=j=n$ e tutti gli $a[1..j] \leq x$.

Nel caso in cui, invece, x sia minore di tutti gli elementi dell'array, i cicli termineranno con $i=j=1$. In questo caso sarebbe estremamente corretto che j fosse uguale a zero. Per questo motivo, è necessario includere la seguente istruzione:

```
if a[j]>x then j := j-1
```

Lo svolgimento dell'algoritmo è ora completo.

Descrizione dell'algoritmo

- Definisci l'array $a[1..n]$ ed il valore di partizione x .
- Muovi le partizioni l'una verso l'altra finché non sia incontrata una coppia di elementi in posizione errata.
Includi come caso particolare quello in cui x sia al di fuori dei valori degli elementi dell'array.
- Finché le due partizioni non si saranno incontrate o scavalcate
 - scambia le coppie in posizione errata e prolunga entrambe le partizioni di un elemento verso l'interno;
 - prolunga la partizione di sinistra finché trovi elementi minori o uguali ad x ;
 - prolunga la partizione di destra finché trovi elementi maggiori di x .
- Ritorna l'indice di partizione p e l'array partizionato.

Implementazione in Pascal

```

procedure xpartition (var a: nelements; n: integer; var p: integer; x: real);
var i {current upper boundary for values in partition =≤x},
    j {current lower boundary for values in partition >x}: integer;
    t {temporary variable for exchange}: real;

begin {partition array into two subsets (1) elements =≤x
(2) elements >x}
{assert: n > 0}
i := 1; j := n;
while (i < j) and (a[i]≤x) do i := i+1;
while (i < j) and (a[j]>x) do j := j-1;
if a[j]>x then j := j-1;
{invariant: after the ith iteration a[1..i-1]=≤x ∧ a[i+1..n]>x
∧ i=<n+1 ∧ j>=0 ∧ i=<j+1}
while i < j do
begin {exchange current pair that are in wrong positions}
  t := a[i];
  a[i] := a[j];
  a[j] := t;
  {move inwards past the two exchanged values}
  i := i+1;
  j := j-1;
  {extend lower partition}
  while a[i]≤x do i := i+1;
  {extend upper partition}
  while a[j]>x do j := j-1
end;
{assert: a [1..i-1]=≤x ∧ a[j+1..n]>x ∧ i>j}
p := j
end

```

Note di progetto

1. Per partizionare un array in due sottogruppi occorrono al massimo $(n+2)$ confronti del tipo $a[i] \leq x$ ed $a[j] > x$. Il numero di scambi richiesti può variare tra 0 ed $[n/2]$ e ciò dipende dalla distribuzione degli elementi nell'array. Se per partizionare è stato usato un metodo di ordinamento, viene effettuato un numero di confronti dell'ordine di $n \log n$.
2. Dopo ogni iterazione i primi $(i-1)$ elementi dell'array sono $\leq x$ ed i rimanenti $(n-j)$ sono $> x$. Le variabili i e j vengono rispettivamente incrementate e decrementate in modo che le due relazioni si equilibrino. Il ciclo while esterno terminerà, perché ad ogni iterazione la distanza tra i e j diminuisce almeno di due unità, mentre le istruzioni $i := i + 1$ e $j := j - 1$ sono eseguite almeno una volta.
3. Nel progettare un algoritmo non bisogna fare più di quanto sia richiesto. Partizioni ordinate non sono necessarie per risolvere il problema.
4. Nel progetto finale il problema è risolto spostando i dati solo quando è assolutamente necessario. Questo è un algoritmo fondato sul principio del sondaggio.
5. Il progetto finale è superiore al secondo, perché raggiunge la maggior efficienza di partizione senza conoscere la posizione della stessa. La lezione che segue è di non "infastidire" la programmazione con informazioni non necessarie.
6. L'idea di lavorare verso l'interno procedendo da entrambi gli estremi dell'array, rende l'algoritmo bilanciato.
7. Se ci sono molti elementi doppi, vengono fatti alcuni scambi non necessari.

Applicazioni

Ordinamento, classificazioni statistiche.

Problemi supplementari

- 4.5.1 Un modo semplice ma meno efficiente di partizionare i dati è quello di applicare, ad ogni iterazione, le seguenti regole:

- se $a[i] \leq x$ incrementa i ;
- altrimenti se $a[j] > x$ decrementa j ;
- se la condizione non è tra le precedenti, scambia $a[i]$ con $a[j]$, incrementa i e decrementa j .

Implementare questo algoritmo (Rif. J. Reynolds, *The Craft of Programming*, Prentice Hall, London, 1981, p.126).

- 4.5.2 Progettare un algoritmo per estrarre un sottogruppo di valori in un array ordinato in modo casuale, che siano all'interno di un definito intervallo. Ciò che vogliamo è:

valori \leq	rango	valori \geq
---------------	-------	---------------

Modificare l'ordinamento per selezione (algoritmo 5.2) in modo che ordini tutti i valori minori di x .

Variando x in maniera casuale, confrontare questo algoritmo con quello di partizione.

- 4.5.3 Il problema della bandiera nazionale olandese comporta di iniziare con una fila di n secchi, (avremo secchio[1..n]), ognuno dei quali contenga un solo ciotolo che sia o rosso, o bianco, o blu.

Il nostro compito è quello di disporre i ciotoli in modo che tutti quelli rossi vengano a trovarsi prima di quelli bianchi che a loro volta devono precedere quelli blu. Progettate ed implementate l'algoritmo per risolvere il problema della bandiera nazionale olandese. (vedi anche E. W. Dijkstra *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, N.J., 1976)

ALGORITMO 4.6 RICERCA DELL'ELEMENTO K-ESIMO MINIMO

Problema

Dato un array non ordinato, determinare il k -esimo elemento minimo.

Sviluppo dell'algoritmo

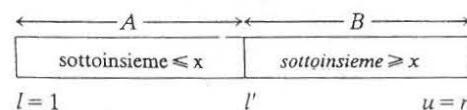
Abbiamo già visto un algoritmo per trovare il valore più grande in un insieme; il problema ora è come generalizzarlo.

Come abbiamo già visto nel problema di partizione (algoritmo 4.5), un modo per trovare il k -esimo elemento minimo, sarebbe quello di ordinare gli elementi e quindi estrarre il k -esimo valore. Possiamo, comunque, supporre che questo problema possa essere trattato in maniera simile a quello di partizione. Nel problema di partizione, sapevamo in anticipo il valore x secondo cui l'array doveva essere partizionato, ma non sapevamo quanti valori avrebbero dovuto essere posti ad entrambi i lati di x . Si tratta ora di individuare la situazione

complementare, in cui noi sappiamo come debba essere partizionato l'array, ma non conosciamo il valore di x (cioè il k -esimo valore minimo).

L'algoritmo di partizione può solamente dividere i dati in due sottogruppi quando conosce in anticipo il valore secondo cui devono essere divisi; come possiamo quindi applicare l'algoritmo di partizione all'attuale problema, senza conoscere in anticipo il valore minimo di k ?

Poichè non conosciamo in anticipo questo valore, potremmo essere tentati di scegliere un valore x a caso dall'array e partizionare l'array attorno ad x . Le variabili l ed u sono assegnate inizialmente ai valori limiti dell'array. Ad esempio:

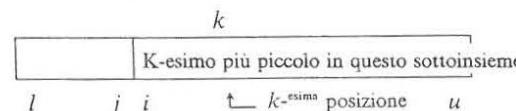


A meno che non siamo molto fortunati, non troveremo il k -esimo valore minimo con la nostra intuizione; tuttavia il valore x porterà alla divisione dell'insieme originario dei dati in due sottogruppi più piccoli. Il k -esimo elemento minimo dovrà essere in uno o nell'altro di questi sottogruppi A e B.

Se, ad esempio, fosse nel sottoinsieme B, (cioè nel sottoinsieme di elementi $\geq x$), allora potremmo tralasciare completamente il sottoinsieme A e cercare di trovare il k -esimo elemento nel sottoinsieme B. Il modo più semplice è sostituire l con l' e cominciare a cercare l'elemento k -esimo nel sottoinsieme B. Se applichiamo iterativamente questo processo di partizione a sottoinsiemi sempre più piccoli, contenenti il k -esimo elemento, otterremo finalmente il risultato desiderato.

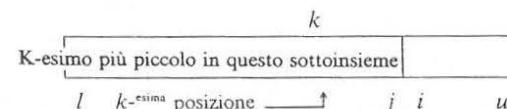
I due casi in cui è richiesto un ulteriore procedimento sono:

1. k -esimo elemento nel sottoinsieme $\geq x$:



Qui poniamo $l := i$ e ripetiamo il processo di partizione per i nuovi limiti di l ed u .

2. k -esimo elemento nel sottoinsieme $< x$:



Qui poniamo $u := j$ e ripetiamo il processo di partizione per i nuovi limiti di u ed l .

Dall'esame dei valori di i e j al termine del ciclo di partizione, riusciamo a sapere quale sottoinsieme contenga il k -esimo elemento e quindi quali limiti debbano essere aggiornati. I due test che possiamo usare sono:

```
if  $j < k$  then  $l := i$ ;
if  $i > k$  then  $u := j$ 
```

Il processo di partizione continua finchè $l < u$. Le variabili i e j devono essere resettate ai limiti aggiornati l ed u prima di iniziare una nuova fase di partizione.

Il meccanismo di base può essere posto nella forma:

finchè $l < u$ esegui i seguenti passi:

- scegli un valore di x per partizionare l'array,
- partiziona l'array in due parti contrassegnate da i e j ,
- aggiorna i limiti usando il test

```
if  $j < k$  then  $l := i$ 
if  $i > k$  then  $u := j$ 
```

Quello che dobbiamo ora decidere è come selezionare x . Poichè l'algoritmo è progettato per dati casuali non importa quale elemento dell'array sia assegnato ad x . È tuttavia allettante scegliere come ipotesi di k -esimo valore più piccolo l'elemento situato in posizione k -esima, semplicemente perché consente di trattare più efficacemente dati ordinati.

Quali sono le implicazioni di tale scelta? Se le partizioni non si incontrano in k (cioè è necessaria un'ulteriore partizione), allora bisognerà scegliere un nuovo valore di selezione nel processo. Prima di esaminare questo dettaglio, occupiamoci del processo di partizione. La strategia sviluppata nell'algoritmo 4.5 potrebbe essere applicata, (con qualche modifica), al presente problema. La differenza che possiamo anticipare in questa fase è che x dovrà essere selezionato usando:

$x := a[k]$

invece di avere x con un valore dato come nel problema precedente. Diversamente dal problema di partizione, poichè x è selezionato dall'array, potremmo evitare le complicazioni causate dal fatto che x potrebbe essere al di fuori dai limiti dell'array. Nel processo di "movimento — verso — l'interno" il ciclo:

while $a[j] > x$ **do** $j := j - 1$

termina sicuramente perché deve alla fine incorrere in x . Esiste tuttavia un problema con

```
while  $a[i] \leq x$  do  $i := i + 1$ 
```

poichè il ciclo non si ferma quando incontra il valore x .

È possibile prevenire ciò trasformando " \leq " in " $<$ ", ad esempio,

```
while  $a[i] < x$  do  $i := i + 1$ 
```

Quali sono le implicazioni di questa trasformazione? Un esame attento di alcuni esempi rivela che ciò non ha seri effetti sul processo di partizione, se non quello di causare uno scambio di valori dell'array uguali ad x (probabilmente non necessario); cioè possiamo avere valori uguali ad x nell'una o nell'altra o in entrambe le partizioni dopo un passaggio.

Ciò non presenta alcuna difficoltà rispetto alla strategia principale; inoltre, un effetto collaterale della trasformazione di $a[i] \leq x$ in $a[i] < x$ è di garantire che il valore in posizione k venga spostato ad ogni passo di partizione che non sia l'ultimo dell'algoritmo. Ciò si adatta bene alla richiesta precedente su come occupare $a[k]$. In particolare, se l' i -esimo ciclo attraversa la k -esima posizione, allora, da destra, si sposterà nella k -esima posizione, un valore inferiore rispetto alla nostra ipotesi (cioè $x = a[k]$). Il fatto che l' i -esimo ciclo abbia raggiunto per primo la k -esima posizione può significare solo che la k -esima scelta minima è troppo grande e quindi deve essere sostituita da un'ipotesi inferiore. La discussione complementare si applica se il ciclo j raggiunge per primo la k -esima posizione.

Un ulteriore esame dei dettagli per aggiornare l ed u suggerisce che debbano essere fatti dei cambiamenti per contemplare il caso in cui entrambi i e j raggiungano k allo stesso passaggio. Il modo più semplice per risolvere questo problema è quello far fare al ciclo una iterazione in più.

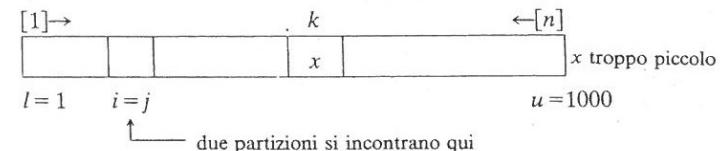
Completando i dettagli e facendo le modifiche delineate, la parte centrale dell'algoritmo di partizione può assumere la forma:

```
while  $l < u$  do
  begin {make next partitioning pass}
     $i := l$ ;  $j := u$ ;
     $x := a[k]$ ;
    while  $a[i] < x$  do  $i := i + 1$ ;
    while  $a[j] > x$  do  $j := j - 1$ ;
    while  $i \leq j$  do
      begin {exchange and extend partitions}
         $t := a[i]$ ;
         $a[i] := a[j]$ ;
         $a[j] := t$ ;
         $i := i + 1$ ;
         $j := j - 1$ ;
```

```
while  $a[i] \leq x$  do  $i := i + 1$ ;
while  $a[j] > x$  do  $j := j + 1$ 
end;
if  $j < k$  then  $l := i$ ;
if  $i > k$  then  $u := j$ 
end
```

A questo punto siamo in possesso di un utile meccanismo per trovare il k -esimo elemento minimo. L'efficacia di questo algoritmo dipende da quanto siamo fortunati con le scelte di x ad ogni fase di partizione. Ad esempio, se x è troppo piccolo, troveremo alla fine che le due partizioni si incontrano all'estrema sinistra, lontano dalla k -esima posizione.

Ad esempio:



Con $n = 1000$, $k = 500$ ed i e j che si incontrano alla posizione 20, con la scelta corrente di x abbiamo ridotto solo la dimensione del problema da $l = 1$, $u = 1000$ a $l = 21$, $u = 1000$. In questo modo, in realtà, abbiamo dovuto pagare il prezzo di un numero troppo grande di confronti con un guadagno molto piccolo o una riduzione della dimensione del problema. Potremmo chiederci, perciò, se si può fare qualcosa dopo una scelta sbagliata di x a valore k -esimo. È necessario renderci conto il prima possibile di aver fatto una scelta sbagliata riguardo al k -esimo valore. Se fossimo in grado di farlo, potremmo eliminare anche confronti che sono poco proficui. Studiando il diagramma precedente, vediamo che non appena l'indice j supera k mentre va ad incontrare i , sappiamo che la nostra scelta di x deve essere stata troppo piccola. Similmente, la discussione complementare indica che non appena i supera k mentre va ad incontrare j , sappiamo che la nostra scelta di x deve essere stata troppo grande.

Perciò, nel primo caso, (quando i incontra j in una posizione minore di k) possiamo fermare il processo di "pinzaggio" quando $j < k$ piuttosto che quando $i > j$. Il risparmio in questa terminazione anticipata è illustrato in Fig. 4.1. Un ragionamento simile può essere fatto per l'altro caso, quando i incontra j ad una posizione maggiore di k . Usando questo approccio siamo in grado di fermare le procedure di confronto e scambio non appena abbiamo rilevato che la scelta attuale x non è il valore k -esimo minimo.

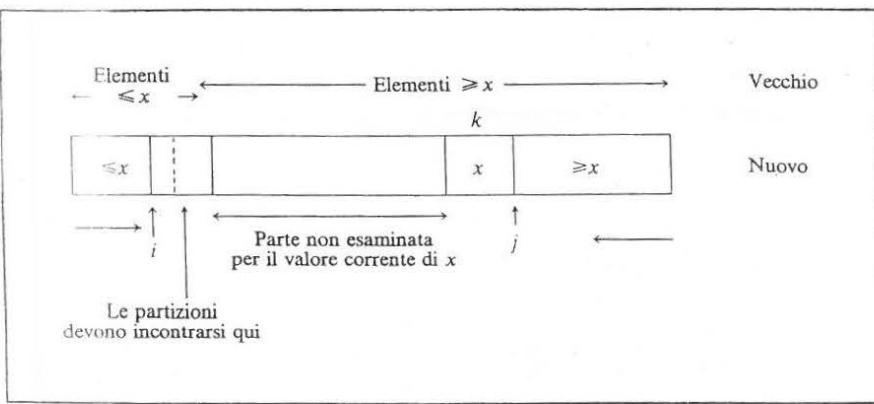


Fig. 4.1.
Partizionamento con
terminazione
anticipata.

Per implementare questo metodo alternativo di terminazione, ci servono due test, uno per controllare se i ha superato k , l'altro per controllare se j ha superato k . Possiamo usare:

```
while ( $i \leq k$ ) and ( $j \geq k$ ) do ...
```

Ancora una volta possiamo considerare cosa succede in termini di scambi quando una delle parti raggiunge la k -esima posizione. Usando la configurazione:

i	k	j
...	250	292

$\dots | 250 | 292 | 300 | \dots$

$x = 292$

Questa volta siamo in condizioni migliori rispetto all'algoritmo precedente perché non è più possibile che i due while-loop permettano a i e j di sorpassarsi prima dell'ultimo scambio nella corrente iterazione del ciclo più esterno di partizione. Ciò accade perché x si trova sempre tra le posizioni i e j . Il test condizionale di controllo degli scambi:

```
if  $i \leq j$  then scambia
```

non è più necessario. Abbiamo quindi un algoritmo più semplice ed efficiente. La descrizione dettagliata dell'algoritmo finale è data qui di seguito.

Descrizione dell'algoritmo

- Definire l'array $a[1..n]$ ed il requisito che si ricerchi il k -esimo elemento minimo.

- Finché le partizioni di destra e sinistra non si sovrappongono
 - scegli $a[k]$ come valore corrente di partizione x ;
 - assegna ad i il limite superiore l della partizione di sinistra;
 - assegna a j il limite inferiore u della partizione di destra;
 - finché i non va oltre k e j è maggiore o uguale a k
 - estendi la partizione di sinistra finché $a[i] < x$
 - estendi la partizione di destra finché $x > a[j]$;
 - scambia $a[i]$ con $a[j]$;
 - incrementa i di 1 e decrementa j di 1;
 - se il k -esimo minore è nella partizione di sinistra, aggiorna il limite superiore u della partizione di sinistra;
 - se il k -esimo minore è nella partizione di destra, aggiorna il limite inferiore l della partizione di destra.
- Ritorna l'array partizionato con gli elementi $\leq a[k]$ nelle prime k posizioni dell'array.

Implementazione in Pascal

```
procedure kselect (var a: nelements; k, n: integer);
var i {temporary extension of left partition for current guess at kth
smallest element x},
j {temporary extension of right partition for current guess at kth
smallest element x},
l {upper limit for left partition},
u {lower limit for right partition} integer;
x {current guess at kth element in array},
t {temporary variable used to exchange a[i] with a[j]} : real;

begin {finds kth smallest element in array a. on termination kth
smallest is in position k}
{assert: n>0 ∧ 1 ≤ k ≤ n}
l := 1;
u := n;
{invariant: all a[1..l-1] ≤ all a[u+1..n] ∧ l ≤ k+1 ∧ k-1 ≤ u}
while l < u do
  begin {using new estimate of kth smallest x try to extend left
  and right partitions}
    i := l;
    j := u;
    x := a[k];
    {invariant: all a[l..i-1] ≤ all a[j+1..u] ∧ i = k+1 ∧ j = k-1
    ∧ 1 ≤ k ≤ n}
    while i <= k and j ≥ k do
      begin {extend left and right partitions as far as possible, then
      exchange}
        while a[i] < x do i := i+1;
        while x < a[j] do j := j-1;
        t := a[i];
        a[i] := a[j];
        a[j] := t;
      end;
    end;
  end;
end;
```

```

a[j] := t;
i := i + 1;
j := j - 1
end;
{update limits of left and right positions as required}
if j < k then l := i;
if i > k then u := j
end
{assert: all a[1..k] = < all a[k..n]}
end

```

Note di progetto

1. L'algoritmo deve fare al massimo $(n+1)$ confronti per trovare il k -esimo minore tra n elementi. Ciò accade quando il k -esimo elemento minore è stato scelto come valore iniziale di partizione. Nel caso peggiore quando viene fatta una ricerca della mediana m ($m = [n+1]\text{div}2$) e lo spazio occupato dal problema si riduce di 1 ad ogni passo, allora vengono fatti $(n/2+1)(n/2+2)/2 = (n+2)(n+4)/8$ confronti. Questo risultato deriva dalla formula di addizione in cui la somma Sq di $1+2+\dots+q$ è $Sq = q(q+1)/2$. Nel caso norma (che è alquanto difficile da analizzare), per trovare il valore medio sono necessari $3n$ confronti. Per k molto piccolo rispetto ad n il numero di confronti è in media $2n$. Al limite, quando $k=1$ è necessario un numero di confronti vicino a $n+1+\log_2 n$. Non considereremo gli scambi richiesti in questo caso, ma occorre notare che l'algoritmo finale comporta un numero di scambi considerevolmente minore rispetto alla prima versione dell'algoritmo.
2. La prova dettagliata di correttezza di questo algoritmo è abbastanza lunga. Indicheremo perciò solo i dettagli più importanti. Per fare ciò dobbiamo caratterizzare il comportamento delle 4 variabili l, u, i e j . Dopo ogni passaggio attraverso il ciclo while più esterno, tutti gli elementi in posizione $a[1], a[2], \dots, a[i-1]$ sono minori o uguali ad x e tutti gli elementi in posizione $a[u+1], a[u+2], \dots, a[n]$ sono maggiori o uguali ad x . Al termine, gli elementi $a[1], a[2], \dots, a[k-1]$ sono minori o uguali ad $x = a[k]$ e gli elementi $a[k+1], a[k+2], \dots, a[n]$ sono maggiori o uguali ad $x = a[k]$. Dopo ogni passaggio attraverso il ciclo while ($i \leq k$) and ($j \geq k$) do tutti gli elementi in posizione $a[1], a[2], \dots, a[i-1]$ sono minori o uguali ad x e tutti gli elementi in posizione $a[j+1], a[j+2], \dots, a[n]$ sono maggiori o uguali al valore dell'elemento corrente $x = a[k]$. Al termine del ciclo più esterno, $j = k-1$ ed $i = k+1$ e la condizione precedente rimane invariata. Lo scambio non altera le relazioni tra i e j . Il ciclo while più interno termina perché x è contenuto in $a[1..n]$ e tra $a[i]$ e $a[j]$. Il ciclo while $i = k..$ termina perché ad ogni iterazione i si incrementa verso k per lo meno di un'unità e j decresce di almeno un'unità.
3. Abbiamo visto due algoritmi utili svolti per trovare in generale il k -esimo elemento minore in un insieme casuale (ed in particolare il mediano di un insieme casuale).

L'algoritmo finale è migliore per diverse ragioni. Persiste con una data scelta x del k -esimo elemento solo finché ciò è essenziale. Non appena si scopre che x non è corretto, viene considerato un nuovo x . Questa idea evita sia scambi che confronti (in particolare quando k è piccolo rispetto ad n). Il meccanismo è più semplice perché non è necessario evitare di fare degli scambi come nell'algoritmo precedente.

4. In questo problema è effettivamente applicata la strategia di incominciare con un problema ampio e ridurlo a problemi sempre più piccoli.

Applicazioni

Trovare la mediana ed i percentili.

Problemi supplementari

- 4.6.1 Implementare e confrontare i due algoritmi dati per valori di k variabili relativamente ad n . Per il confronto misurare il numero di scambi in ogni caso. Usare un generatore di numeri casuali per produrre un adeguato insieme di dati.
- 4.6.2 Per valori di k molto piccoli rispetto ad n (cioè per $k < 10$ ed n grande) è possibile progettare un algoritmo più efficiente usando l'idea seguente. Ad ogni passo dell'algoritmo il valore più grande tra i primi k elementi viene scambiato con valori progressivamente minori nell'insieme $a[k+1], a[k+2], \dots, a[n]$. Progettare ed implementare questo algoritmo e confrontarlo con l'algoritmo precedente per valori di k piccoli rispetto ad n .
- 4.6.3 Se i primi k elementi dell'array vengono posti su di una pila o una struttura ad albero, otteniamo una versione più efficiente dell'algoritmo 4.6.2. L'implementazione di questo algoritmo implica la conoscenza di alcuni concetti piuttosto sofisticati.
- 4.6.4 L'algoritmo che abbiamo descritto può essere migliorato usando un metodo di campionatura. L'idea è quella di usare un campione dell'1% e trovare il k -esimo minore come stima di x (rif. R. W. Floyd e R. L. Rivest, "Expected time bounds for selection", Comm. ACM, 18, 165 – 172, 1975). Implementare e testare questo approccio.

ALGORITMO 4.7

SOTTOSEQUENZA MONOTONA DI MASSIMA LUNGHEZZA

Problema

Dato un insieme di n numeri diversi, trovare la lunghezza della massima sottosequenza monotona crescente.

Sviluppo dell'algoritmo

Una sottosequenza monotona crescente è un sottoinsieme di numeri che sono strettamente crescenti da sinistra verso destra. Questa definizione non richiede che i numeri siano adiacenti nell'insieme originario o che la sequenza massima sia unica. Per esempio, nell'insieme seguente:

x	x	x	x	x	x				
1	2	9	4	7	3	11	8	14	6

la sequenza 1,2,4,11,14, è una sottosequenza monotona crescente. Il problema è quello di trovare la più lunga di tali sottosequenze, per una data sequenza di interi.

Sebbene non sia molto difficile distinguere ad occhio sottosequenze monotone crescenti, l'implementazione di una procedura sistematica sembra un po' più impegnativa.

Per iniziare lo svolgimento di un algoritmo, esaminiamo più attentamente la sottosequenza che abbiamo scelto e la sua relazione con la sequenza completa. La nostra sottosequenza è dunque:

1 2 4 7 11 14

Nel ricavare questa sottosequenza notiamo che molte altre sono presenti nei dati:

1 2 9
1 2
2 9
1 2 3 6

Il nostro compito nel risolvere questo problema implica la "capacità" di riconoscere sottosequenze monotone crescenti e, nel far ciò, di decidere quale sia la più lunga. Dalle sottosequenze esaminate, notiamo che esse non hanno tutte uno stesso valore iniziale e finale, ed anche più di una sottosequenza può iniziare e finire allo stesso punto. (Ad

esempio, abbiamo trovato che due sottosequenze monotone 1,2,9 e 2,9 terminano con 9). La sottosequenza più lunga terminante con 9 ha lunghezza 3. Il nostro problema è che non sappiamo dove finisce la sottosequenza più lunga.

Queste osservazioni suggeriscono che dobbiamo trovare un modo sistematico di generare ed esaminare sottosequenze per fare passi avanti nella ricerca della sottosequenza monotona crescente di lunghezza massima. Sappiamo che la più lunga sottosequenza deve terminare con un particolare valore della sequenza originale. Allo stesso tempo potremmo sospettare che il compito di trovare la più lunga sottosequenza terminante ad un particolare valore sia più semplice da risolvere rispetto al problema originario.

Se in primo luogo potessimo risolvere questo problema minore relativamente ad ogni possibile valore finale (cioè determinare le sottosequenze più lunghe terminanti in ogni posizione dell'array), saremmo in grado di selezionare da questo insieme la soluzione cercata.

Allo scopo di cercare un algoritmo per trovare la sottosequenza più lunga terminante in un punto particolare, possiamo tornare al nostro esempio specifico.

Supponiamo di voler determinare la più lunga sottosequenza monotona crescente che termini in posizione 6.

Per primo scriviamo le lunghezze delle prime 5 sottosequenze più lunghe come verifica, quindi cercheremo di elaborare un meccanismo adeguato.

1	2	3	4	5	6	indice
1	2	9	4	7	3	sequenza
1	2	3	3	4		lunghezza della sequenza

Nell'eseguire questi passi notiamo che ogni sequenza deve essere almeno di lunghezza unitaria. Vediamo inoltre che la sottosequenza terminante in posizione 4 (cioè 1,2,4) può essere solo di lunghezza 3. Per ottenere ciò dobbiamo però trascurare il 9, possiamo allora vedere che non ha senso considerare valori maggiori del valore finale (in questo caso 4) nel determinare la lunghezza della sequenza di lunghezza più lunga terminante con 4. Applicando questa osservazione alla più lunga sequenza terminante in posizione 6, troviamo che essa deve essere:

1 2 3

che è di lunghezza 3. Tale sequenza è formata dalla sottosequenza terminante in posizione 2 e dall'elemento in posizione 6. Ne consegue che per determinare la sottosequenza più lunga terminante in posizione 6, dobbiamo esaminare tutte le altre sottosequenze che terminano prima di 6 e vedere se il 3 possa essere aggiunto alla fine di queste sottosequenze. Ad esempio, abbiamo:

Sequenza	Valida/non valida	Lunghezza della vecchia sequenza
1 3	V	1
1 2 3	V	2
1 2 9 3	X	3
1 2 4 3	X	3
1 2 4 7 3	X	4

Le sottosequenze che non mantengono il criterio di monotonia crescente sono contrassegnate da una croce.

Per implementare il processo di estensione delle prime sottosequenze dobbiamo conoscere le lunghezze di tutte le sottosequenze precedenti. La sottosequenza più lunga terminante in posizione 6 deve essere derivata dalla più lunga sottosequenza precedente a cui possa essere aggiunto l'elemento in posizione 6. Lo stesso procedimento può essere usato per trovare la sottosequenza più lunga terminante in posizione 7 e così via.

Abbiamo ora la base di un metodo per trovare la sottosequenza monotonamente crescente più lunga, cioè:

1. Per ogni posizione terminale di una sottosequenza esegui
 - (a) aggiungi l'elemento corrente alla più lunga delle sottosequenze precedenti che permetta di mantenere il criterio di monotonia crescente.

Ci occorrono due cicli per implementare questo processo, uno per esaminare ogni nuova posizione termine, ed un altro per esaminare tutte le sottosequenze precedenti. Una struttura di base che possiamo usare per un array $a[1..n]$ è:

```
For i := 2 to n do
begin
  (a) For j := 1 to i - 1 do
  begin
    (a.1) "trova la più lunga sequenza monotonamente crescente nell'insieme
          a[1..i - 1] terminante con un valore < a[i]"
    end
    (b) "aggiungi l'i-esimo elemento alla appropriata sottosequenza monotonamente crescente precedente ed aggiorna la sottosequenza massima per risalirne quando necessario"
  end
end
```

Il ciclo più esterno inizia con $i = 2$, cosicché esiste sempre un valido predecessore. Il ciclo più interno deve esaminare tutti gli elementi che precedono l' i -esimo elemento corrente (cioè $a[1..i - 1]$).

Per trovare la più lunga sottosequenza monotonamente crescente entro il ciclo più interno, dobbiamo escludere gli elementi maggiori dell'ultimo elemento considerato, $a[i]$. Se $last$ viene assegnato ad $a[i]$, ciò può essere inserito nel testo come:

```
if a[j] < last then
```

"guarda se la lunghezza della j -esima sequenza è più lunga di quella incontrata finora"

Un altro semplice test condizionale è necessario per scoprire e memorizzare la lunghezza della sequenza più lunga incontrata. L'operazione di accodamento implica solo l'immagazzinamento della lunghezza della sottosequenza massima monotonamente crescente terminante in posizione i -esima. Un altro array $b[1..n]$ può essere usato a questo scopo.

Abbiamo ora elaborato a grandi linee tutti i dettagli del nostro algoritmo. Prima di procedere, ci potremmo chiedere se esiste un modo per poterne migliorare l'efficienza. Nel ciclo più interno occorre molto tempo per il riesame dello stesso insieme di dati; è necessario trovare il modo di ridurre il numero di passi eseguiti in tale ciclo. Sarebbe auspicabile esaminare le sottosequenze precedenti nell'ordine della più lunga alla più corta. Il problema è che probabilmente ciò richiederebbe l'ordinamento dei predecessori per lunghezza che è di per sé stessa un'operazione costosa (vedi Note di progetto) Quali altre alternative abbiamo? Il nostro algoritmo, così com'è, tiene conto della più lunga sottosequenza monotonamente crescente incontrata finora. A patto che si conosca la posizione in cui termina la sottosequenza più lunga, possiamo verificare direttamente se l'ultimo elemento possa essere appeso alla sottosequenza. Se ciò è possibile, non è necessario passare attraverso il ciclo più interno. Ponendo questo test prima del ciclo interno si migliorerà considerevolmente l'efficienza del nostro algoritmo, particolarmente se la più lunga sottosequenza monotonamente crescente è piuttosto lunga in relazione alla lunghezza totale della sequenza.

Nell'esempio in cui la sequenza da esaminare sia in ordine ascendente, l'algoritmo avrà uno svolgimento dell'ordine di n operazioni anziché dell'ordine di n^2 operazioni. In molte applicazioni, la sottosequenza più lunga avrà una dimensione considerevole, in cosicché questo ridimensionamento dell'algoritmo sarà molto utile. L'algoritmo può essere ora descritto dettagliatamente.

Descrizione dell'algoritmo

1. Definisci l'insieme di dati $a[1..n]$ di n elementi.
2. Assegna le condizioni iniziali per la sottosequenza terminante nella prima posizione.
3. Per le rimanenti $(n - 1)$ posizioni dell'array
 - (a) se l'elemento corrente è minore del massimo nel precedente insieme, allora

- (a.1) individua la posizione ed il valore del massimo tra i predecessori;
 - (a.2) aggiorna la posizione e la lunghezza del massimo se richiesta;
altrimenti aggiorna lunghezza e posizione del massimo e lunghezza massima trovati finora.
4. Ritorna la lunghezza della più lunga sottosequenza monotona crescente.

Implementazione in Pascal

```

procedure monotone (a: nelements; n: integer; var maxlen: integer);
var
  length: array [1..100] of integer;
  i {index for current terminating subsequence},
  j {index for predecessors of current terminating subsequence},
  maxj {length of current longest predecessor subsequence},
  pmax {position of current longest monotone increasing
  subsequence}: integer;
  current {current element to be appended to appropriate preceding
  subsequence}: real;

begin {find longest monotone increasing subsequence}
  {assert: n > 0}
  length [1] := 1;
  pmax := 1;
  maxlen := 1;
  {invariant: after ith iteration length[1..i] are lengths of lmss ending
  at a[1..i] ∧ 1 ≤ i ≤ n ∧ maxlen = length of lmss in first i
  positions ∧ pmax is where it terminates}
  for i := 2 to n do
    begin {append current element to its longest valid predecessor
    subsequence}
      current := a[i];
      if current < a[pmax] then
        begin {search for longest predecessor subsequence}
          maxj := 1;
          {invariant: after jth iteration maxj = length of lmss in a[1..j]
          terminating in a value less than a[i] ∧ j ≤ i - 1}
          for j := 2 to i - 1 do
            begin {reject subsequence with too large a terminator}
              if a[j] < current then
                begin
                  if maxj < length[j] then maxj := length[j]
                end
            end;
          end;
          length[i] := maxj + 1;
        end;
        if length[i] > maxlen then
          begin {save length and position of longest subsequence
          to date}
            maxlen := maxlen + 1;
            pmax := i
          end
      end
    end
end

```

```

else
  begin {append directly to longest predecessor}
    maxlen := maxlen + 1;
    length [i] := maxlen;
    pmax := i
  end
end;
{assert: maxlen = length of lmss in a[1..n] ∧ pmax is where it
terminates}
end

```

Note di progetto

1. Nel caso peggiore (una sequenza in ordine discendente) l'algoritmo esaminerà gli elementi $\frac{1}{2}n(n - 1) + n = \frac{1}{2}n(n + 1)$ volte.
Nel caso migliore, quando i dati sono in ordine ascendente, saranno fatti essenzialmente n test sui dati. In media, il numero di test sarà un valore compreso tra questi due estremi. Più lunga è la massima sottosequenza monotona crescente, meno confronti dovrà fare la verifica che riguarda il ciclo interno. In contrasto con l'implementazione finale, la nostra prima proposta comporterebbe sempre $\frac{1}{2}n(n - 1)$ esami di elementi.
2. Dopo l' i -esimo passo, nel ciclo esterno sarà stata determinata la più lunga sottosequenza terminante ai primi i elementi. Inoltre, dopo l' i -esimo passo, sarà stabilita la posizione finale e la lunghezza della più lunga sequenza monotona crescente tra i primi i elementi. Nel ciclo interno, dopo il j -esimo passo, sarà stabilita la più lunga sottosequenza terminante con un valore minore di $a[i]$.
3. Nello sviluppo di questo algoritmo un piccolo numero di esempi specifici svolge un ruolo importante.
4. L'idea di spezzare un grosso problema in un insieme di problemi più piccoli e più semplici è una valida strategia nel progetto di algoritmi.
5. In questo progetto abbiamo visto come sia sempre importante cercare di fare l'uso migliore di una data informazione (cioè abbiamo usato la conoscenza della sottosequenza più lunga trovata finora per migliorare l'efficienza dell'algoritmo).
6. Poiché è più semplice far riferimento ad una variabile singola piuttosto che ad una indiciaria, il termine corrente $a[i]$ (a cui ci si riferisce frequentemente) è stato assegnato ad una variabile singola *current*.

7. Con un addizionale immagazzinamento di n elementi, è possibile implementare un algoritmo più efficiente per questo problema collegando insieme tutti gli elementi con sequenze della stessa lunghezza. Possiamo facilmente esaminare le sequenze in ordine dalla più grande alla più piccola.

Applicazioni

Studio di sequenze casuali, confronti di file.

Problemi supplementari

- 4.7.1 Implementare il primo progetto che è stato proposto per trovare la più lunga sottosequenza monotona crescente.
- 4.7.2 Confrontare lo svolgimento dell'algoritmo del problema 4.7.1 con l'algoritmo precedente per
 - (a) dati casuali,
 - (b) dati con una lunga sottosequenza monotona crescente (di $0.5n$).
- 4.7.3 Progettare ed implementare un algoritmo che stampi la più lunga sottosequenza monotona crescente di un dato insieme di dati.
- 4.7.4 Progettare ed implementare un algoritmo che determini la lunghezza della più lunga sottosequenza monotona (può essere sia crescente che decrescente).
- 4.7.5 Progettare ed implementare un algoritmo che utilizzi il suggerimento della nota 7.

CAPITOLO 5

ORDINAMENTO E RICERCA

INTRODUZIONE

Le attività collegate di ordinamento (*sorting*), ricerca (*searching*) e fusione (*merging*) sono basilari in molti impieghi del computer. Si dice che da solo il sorting occupi più del 30% del tempo totale speso dai computer.

Il sorting ed il merging ci forniscono un mezzo con cui organizzare l'informazione e facilitare il reperimento di dati specifici; i metodi di searching sono progettati per trarre vantaggio dall'organizzazione dell'informazione, e pertanto ridurre lo sforzo necessario per individuare la posizione di un determinato elemento o per stabilire che esso non è presente in un certo insieme di dati.

Gli algoritmi di ordinamento organizzano gli elementi di un insieme secondo una relazione d'ordine prestabilita. I due tipi più comuni di dati sono i dati numerici e i dati stringa. La relazione d'ordine per i dati numerici implica semplicemente che gli elementi vengano disposti in sequenza dal più piccolo al più grande (o viceversa), di modo che ogni elemento risulti *minore o uguale* al suo immediato successore. Questo ordinamento è detto *in ordine non decrescente*. Gli elementi dell'insieme che segue sono disposti in ordine numerico non decrescente.

[7, 11, 13, 16, 16, 19, 23]

I dati stringa ordinati sono generalmente organizzati secondo l'ordine lessicografico o del dizionario; la lista che segue è stata disposta secondo tale ordine.

[a, abaco, asino, bambola, blu, blue]

A causa della sua importanza economica, il problema dell'ordinamento è stato studiato estesamente dal punto di vista sia teorico sia pratico. Di conseguenza, esistono numerosi algoritmi per riordinare le informazioni, che ricadono normalmente in una delle due classi seguenti:

1. Gli algoritmi più semplici e meno sofisticati sono caratterizzati dal richiedere all'incirca n^2 confronti (ossia $O(n^2)$) per ordinare n elementi.
2. Gli algoritmi avanzati, per ordinare n elementi, richiedono circa $n \log n$ confronti (ossia $O(n \log n)$). Gli algoritmi di questa specie si avvicinano alla migliore prestazione possibile nell'ordinamento di dati del tutto casuali.

È istruttivo paragonare tra loro n^2 e $n \log_2 n$ in funzione di n , come indicato in Tab. 5.1.

Tab. 5.1. Confronto della complessità di algoritmi di sort al variare di n .

n	n^2	$n \log_2 n$	$n^2/n \log_2 n$
10	100	33.2	3.01
100	10,000	664.4	15.05
1,000	1,000,000	9,966	100.34
10,000	100,000,000	132,877	752.58

Chiaramente, all'aumentare di n , i metodi avanzati affermano la loro superiorità sugli altri, come evidenziato dal rapporto $n^2/n \log_2 n$.

La superiorità di tali metodi è dovuta alla loro capacità di scambiare valori molto distanti fin dalle prime battute del sort. Si può dimostrare che, mediamente, per dati distribuiti casualmente, gli elementi debbono essere mossi per una distanza di $n/3$; i metodi più semplici e meno efficienti tendono a muovere gli elementi solo su piccole distanze, col risultato di dover eseguire molti più spostamenti prima di conseguire l'ordinamento definitivo.

Non c'è un metodo che sia il migliore per tutte le applicazioni; le prestazioni dei diversi metodi dipendono da parametri come l'ampiezza dei dati, il grado di ordine relativo già presente nei dati, la distribuzione dei valori degli elementi, la quantità di informazione associata ad ogni elemento. Per esempio, se i dati sono già quasi ordinati, il *bubblesort* (che è di solito il meno efficiente per dati disordinati), può dare risultati migliori di quelli di un metodo avanzato.

In questo capitolo esamineremo una serie di algoritmi di ordinamento, dai più semplici ai più avanzati, nell'intento di fornire una visione complessiva dell'ampia varietà di tecniche che possono essere impiegate per manipolare i dati in un computer.

ALGORITMO 5.1 FUSIONE (MERGE) DI DUE SEQUENZE ORDINATE

Problema

Fondere due sequenze di interi, entrambe con gli elementi in ordine ascendente, in un'unica sequenza ordinata.

Sviluppo dell'algoritmo

La fusione di due o più insiemi di dati è un compito che s'incontra di frequente in informatica; è più semplice del sorting poiché è possibile trarre vantaggio dall'ordine parziale dei dati.

L'analisi di due array ordinati di numeri può aiutarci a scoprire gli elementi essenziali di una procedura conveniente di fusione. Consideriamo i due array:

a:

15	18	42	51
----	----	----	----

m elementi

b:

8	11	16	17	44	58	71	74
---	----	----	----	----	----	----	----

n elementi

Con poca fatica si deduce che il risultato della fusione sarà quello di seguito indicato, ove la fonte di ogni elemento (gli array *a* o *b*) è indicata al di sopra dell'array *c*:

c:

<i>b</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>a</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>	<i>b</i>	<i>b</i>
8	11	15	16	17	18	42	44	51	58	71	74

 $(n+m)$ elementi

Che cosa possiamo imparare da questo esempio? La cosa più ovvia che possiamo osservare è che *c* è più lungo di *a* e *b*; infatti deve contenere un numero di elementi corrispondente alla somma del numero di elementi di *a* e di *b* (ossia $n+m$). Un altro fatto che emerge dall'esempio è che per costruire *c* è necessario esaminare tutti gli elementi di *a* e *b*. L'esempio mostra anche che per attuare la fusione occorre un meccanismo per selezionare in maniera appropriata elementi da *a* e da *b*. Per mantenere l'ordine nel collocare gli elementi in *c* sarà necessario eseguire dei confronti tra gli elementi di *a* e quelli di *b*. Per vedere come questo si possa fare analizziamo il problema di *merging* minimo, quello della disposizione in ordine di due elementi.

Per fondere due array di un solo elemento, tutto quel che dobbiamo fare è scegliere il più piccolo degli elementi *a* e *b* e collocarlo in *c*. Quindi si porta in *c* l'elemento più grande.

Consideriamo l'esempio seguente:

a:

15

b:

8

Poiché 8 è minore di 15 andrà per primo in *c*[1]; quindi 15 prenderà posto in *c*[2], dando come risultato: