

c:	8	15	
----	---	----	--

Dovremmo essere in grado di partire nello stesso modo nel più vasto problema di fondere m ed n elementi. Il confronto tra $a[1]$ e $b[1]$ ci consente di assegnare $c[1]$. Abbiamo quindi:

a:	15	18	42	...
b:	8	11	16	...
c:	8	...		

Dopo aver posto 8 in $c[1]$, ci occorre un modo per decidere quale elemento collocare successivamente nell'array c . Nel merge di due elementi, era stato posto il 15; tuttavia, nell'esempio allargato porre 15 in $c[2]$ lascerebbe fuori sequenza l'11. Ne consegue che per collocare l'elemento successivo in c dobbiamo confrontare il *prossimo* elemento di b (cioè 11) con il 15 e porre in c il più piccolo dei due.

Possiamo cominciare a vedere che cosa succede; nel caso generale il prossimo elemento da collocare in c risulta sempre essere *il più piccolo* tra gli elementi iniziali delle parti *non ancora combinate* di a e di b .

Dopo il secondo confronto (ossia: $a[1] < b[2]$?) abbiamo la figura 5.1(a), e ripetendo il processo (ossia confrontando $a[1]$ con $b[3]$) otteniamo la figura 5.1(b). Per conservare traccia dei punti d'inizio delle parti non ancora combinate di a e b occorrono due puntatori i e j ; in principio essi hanno entrambi valore 1: non appena un elemento è selezionato da a o da b , il puntatore corrispondente deve essere incrementato di 1. Ciò garantisce che i e j sono sempre allineati con il rispettivo punto d'inizio delle parti non ancora combinate di entrambi gli array. L'unico altro puntatore richiesto è quello che serba traccia del numero di elementi collocati fino a questo punto nell'array di destinazione. Tale puntatore, indicato con k , è semplicemente incrementato di 1 per ogni nuovo elemento aggiunto a c .

Se seguiamo passo passo il procedimento di fusione avviato sopra, perveniamo finalmente alla situazione illustrata in figura 5.1(c).

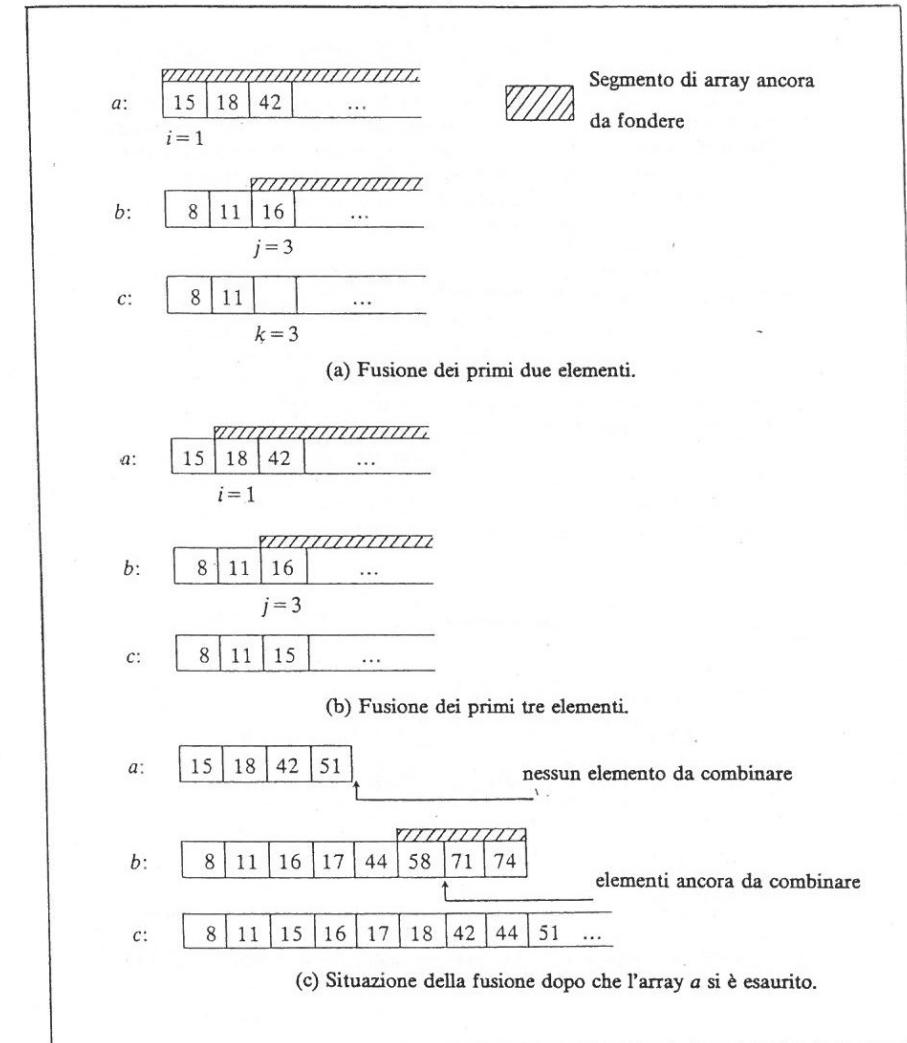
Quando abbiamo esaurito in uno degli array gli elementi da fondere, non possiamo compiere ulteriori confronti tra elementi; dobbiamo perciò disporre di un modo per determinare quando uno degli array si esaurisce: quale dei due finisce per primo dipende dai dati contingenti.

Un approccio che possiamo tentare con questo problema è includere confronti per verificare quando l'uno o l'altro array finisce. Non appena si conclude questa fase della fusione, si avvia un altro meccanismo che copia nell'array c gli elementi ancora da fondere. Una struttura complessiva che potremmo usare è la seguente:

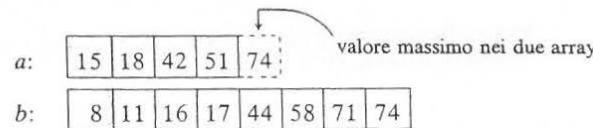
1. finché $i \leq m$ e $j \leq n$
 - (a) confronta $a[i]$ con $b[j]$ e colloca nell'array c l'elemento minore della coppia,

- Fig. 5.1.
- (a) *Fusione dei primi due elementi.*
 - (b) *Fusione dei primi tre elementi.*
 - (c) *Situazione della fusione dopo che l'array a si è esaurito.*
- (b) aggiorna i puntatori appropriati, se $i < m$
 - (a) copia in c la parte rimanente di a , altrimenti
 - (a') copia in c la parte rimanente di b .

Uno studio attento di questa proposta rivela che essa si traduce in un'implementazione piuttosto inelegante; è allora conveniente vedere se esistono alternative più semplici.



Un punto di partenza lungo questa nuova direzione è il segmento che nell'esempio viene copiato: esaminandolo, vediamo che il valore *massimo* degli array *a* e *b* finisce per essere l'*ultimo* valore da introdurre in *c*. Che cosa ci suggerisce una simile osservazione? Se per un istante immaginiamo che il valore massimo al termine dell'array *a* sia 74, la fusione potrebbe procedere fino al raggiungimento dell'estremo in *entrambi* gli array *a* e *b*. Ossia:



Abbiamo la chiave per imbastire un meccanismo di merging più semplice: se ci assicuriamo che il massimo elemento dei due array sia presente al termine di *entrambe* le sequenze, l'ultimo elemento da trasferire sarà l'ultimo elemento di *a* e l'ultimo di *b*. Sicuri di questo fatto, non dovremo più preoccuparci di quale dei due array si esaurisca per primo: continueremo semplicemente il processo di fusione finché non saranno stati introdotti $(n+m)$ elementi nell'array *c*.

La parte centrale di questa implementazione potrebbe essere:

```

if  $a[m] < b[n]$  then  $a[m+1] := b[n]$  else  $b[n+1] := a[m]$ ;
i := 1;
j := 1;
nm := n+m;
for k := 1 to nm do
begin {merge next element}
  if  $a[i] < b[j]$  then
    begin
      c[k] := a[i];
      i := i+1
    end
  else
    begin
      c[k] := b[j];
      j := j+1
    end
end

```

Quest'ultima proposta, benché corrisponda ad un'implementazione semplice e nitida, può in certi casi comportare un numero di confronti molto maggiore del necessario.

Nell'algoritmo precedente, abbiamo fatto l'uso migliore delle informazioni disponibili? Dal momento che possiamo accedere agli ultimi elementi di entrambi gli array, è facile determinare in anticipo quale dei due terminerà per primo *durante* la fusione e quale dovrà essere copiato

dopo la fusione stessa. Sulla base di questi fatti dovremmo essere in grado di ridurre i test nel *while-loop* da due ad uno, e di ridurre contemporaneamente il numero dei confronti. Un confronto tra $a[m]$ e $b[n]$ stabilirà quale dei due array finirà per primo la fase di fusione.

<i>a</i> [1]	<i>a</i> [<i>m</i>]	<i>b</i> [1]	<i>b</i> [<i>n</i>]

Avremo allora:

- se $a[m] \leq b[n]$
 - (a) combina tutti gli elementi di *a* con *b*,
 - (b) copia il resto di *b*,
 - altrimenti
- (a') combina tutti gli elementi di *b* con *a*,
- (b') copia il resto di *a*.

Per implementare questi passi di merge e di copia si può utilizzare un'unica procedura *mergecopy*; le operazioni di merge e di copia si possono anche implementare come procedure separate. Nel processo di fusione è possibile che i due array non si sovrappongano per niente: quando ciò si verifica, gli insiemi di dati *a* e *b* andranno copiati l'uno dopo l'altro. Dopo aver determinato quale dei due array finisce per primo, è assai semplice determinare se vi sia sovrapposizione fra di essi: il confronto dell'*ultimo* elemento dell'array che finisce per primo con il *primo* elemento dell'altro array stabilirà se vi sia sovrapposizione o no. Per esempio, se *a* finisce per primo, potremo utilizzare il seguente frammento:

```

se  $a[m] \leq b[1]$ 
  (a) copia l'array a,
  (b) copia l'array b,
altrimenti
  (a') combina tutti gli elementi di a con b,
  (b') copia il resto di b.

```

I dettagli di questo merge sono assai simili a quelli della seconda proposta, tranne che adesso il processo continua solamente fino a che tutti gli *m* valori dell'array *a* siano stati combinati (nel caso che *a* termini per primo). Il meccanismo di copia implica la copia diretta di un array in un altro.

Descrizione dell'algoritmo

(1) procedura *merge*

1. Definisce gli array $a[1..m]$ e $b[1..n]$.
2. Se l'ultimo elemento di *a* è minore o uguale all'ultimo elemento di *b*

- (a) combina tutti gli elementi di a con b ,
 - (b) copia il resto di b ,
 - altrimenti
 - (a') combina tutti gli elementi di b con a ,
 - (b') copia il resto di a .
3. Restituisce il risultato $c[1..n+m]$.

(2) procedura *mergecopy*

1. Definisce gli array $a[1..m]$ e $b[1..n]$ con $a[m] \leq b[n]$.
2. Se l'ultimo elemento di a è minore o uguale al primo elemento di b
 - (a) copia tutti gli elementi di a nelle prime m posizioni di c ,
 - (b) copia l'array b in c a partire da $m+1$,
 - altrimenti
 - (a') combina in c tutti gli elementi di a con quelli di b ,
 - (b') copia il resto di b in c , a cominciare dalla posizione immediatamente successiva a quella finale del merge.

(3) procedura *shortmerge*

1. Definisce gli array $a[1..m]$ e $b[1..n]$ con $a[m] \leq b[n]$.
2. Finché non è stato collocato tutto l'array a , esegue quanto segue
 - (a) se l'elemento corrente di a è minore o uguale all'elemento corrente di b
 - (a.1) copia la a corrente nella posizione corrente di c ,
 - (a.2) sposta alla posizione successiva il puntatore dell'array a ,
 - altrimenti
 - (a'.1) copia la b corrente nella posizione corrente di c ,
 - (a'.2) sposta alla posizione successiva il puntatore dell'array b .
 - (b) incrementa di uno il puntatore all'array c .

(4) procedura *copy*

1. Definisce gli array $b[1..n]$ e $c[1..n+m]$, inoltre stabilisce dove deve iniziare la copia in b (ossia j) e dove deve iniziare la copia in c (ossia k).
2. Finché non si è ancora raggiunta la fine dell'array b , esegue quanto segue
 - (a) copia un elemento dalla posizione corrente di b alla posizione corrente di c ;
 - (b) fa avanzare il puntatore j alla prossima posizione di b ;
 - (c) fa avanzare il puntatore k alla prossima posizione di c .

Implementazione in Pascal

```

procedure copy(var b: nelements; var c: npmelements; j,n: integer;
var k: integer);
var i {index for section of b array to be copied}: integer;
begin {copy sequence b[j..n] into merged output}
{assert: k=k0}

{invariant: c[k0]=b[j]∧c[k0+1]=b[j+1]∧...c[k-1]=b[i]∧1=<k0
∧1=<i=<n ∧ 1=<j=<n ∧ k0=<k=<m+n+1}
for i := j to n do
begin
  c[k] := b[i];
  k := k + 1
end
{assert: c[k0]=b[j]∧c[k0+1]=b[j+1]...∧c[k-1]=b[n]}
end;

procedure shortmerge(var a,b: nelements; var c: npmelements; m:
integer; var j,k: integer);
var i {index for the a array that is being merged}: integer;
begin {merges all of a array with b array elements <a[m]}
{assert: m>0∧n>0∧a[1..m] ordered∧b[1..n] ordered}
i := 1;
{invariant: after ith iteration a[1..i-1] merged with b[1..j-1] and
stored in c[1..k-1] ordered ∧ i=<m+1 ∧ j=<n+1 ∧
k=<m+n+1}
while i <= m do
begin
  if a[i] <= b[j] then
    begin
      c[k] := a[i];
      i := i + 1
    end
  else
    begin
      c[k] := b[j];
      j := j + 1
    end;
  k := k + 1
end
{assert: c[1..k-1] ordered ∧ it is made up of only a[1..m] and
b[1..j-1] elements ∧ k=<m+n+1 ∧ j=<n+1}
end

procedure mergecopy(var a,b: nelements; var c: npmelements; m,n:
integer);
var i {first position in a array},
j {current position in b array},
k {current position in merged array - initially 1}: integer;

```

```

begin {merges a[1..m] with b[1..n]}
  assert: m > 0  $\wedge$  n > 0  $\wedge$  a[1..m] ordered  $\wedge$  b[1..n] ordered
  i := 1; j := 1; k := 1;
  if a[m]  $\leq$  b[j] then
    begin {two sequences do not overlap so copy instead of merge}
      copy(a,c,i,m,k);
      copy(b,c,j,n,k)
    end
  else
    begin {merge all of a with b then copy rest of b}
      shortmerge(a,b,c,m,j,k);
      copy(b,c,j,n,k)
    end
  assert: c[1..m+n] ordered  $\wedge$  it is made up of only a[1..m] and
  b[1..n] elements
end

procedure merge(var a,b: nelements; var c: npmelements; m,n:
integer);

begin {merges the arrays a[1..m] and b[1..n] to give c[1..m+n]
taking advantage of which array is used up first in merge}
  if a[m]  $\leq$  b[n] then
    mergecopy(a,b,c,m,n)
  else
    mergecopy(b,a,c,n,m)
end

```

Note di progetto

- Per due array in input di dimensioni m ed n , il numero di confronti prima di terminare il lavoro può variare da 2 a $(n+m+1)$. Quando non c'è sovrapposizione negli intervalli dei due array vale la situazione di minimo.
- Nel considerare il comportamento di questo algoritmo ci concentreremo sulla procedura *shortmerge*. Al termine del passo i -esimo del while-loop i primi $(i-1)$ elementi di *a* saranno stati combinati con i primi $(j-1)$ elementi di *b*: il numero di elementi combinati dopo il passo i -esimo sarà $i+j-2$. I primi $k-1$ elementi dopo il passo i -esimo saranno in ordine non ascendente. Il loop ha termine poiché ad ogni iterazione l'uno o l'altro di i e j viene incrementato, e poiché è certo che esista un j tale che $a[m] \leq b[j]$ già prima di chiamare *shortmerge*, si creerà alla fine una situazione per cui i verrà incrementato oltre m .
- Il progetto finale cui siamo pervenuti è verosimilmente di implementazione più complicata che non le due proposte precedenti. Benché ciò sia vero, il meccanismo per l'algoritmo finale è più semplice e pulito; inoltre utilizza nel modo migliore tutte le informazioni disponibili. Quando i due array non si sovrappongono, sono evitati i confronti inutili.

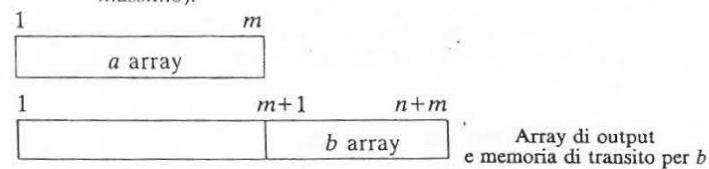
- La considerazione del problema minimo di merging ci ha fornito un punto di riferimento per isolare il meccanismo basilare di fusione.
- L'algoritmo attuale non è adatto per fondere due insiemi di dati di cui non si conoscano preventivamente le estensioni; questo caso si presenta sovente, ed occorre impiegare una versione modificata del primo algoritmo.
- Quando le dimensioni dei due insiemi di dati sono note in anticipo è possibile utilizzare soltanto due array per portare a termine il lavoro.

Applicazioni

Ordinamento, ordinamento su nastro, elaborazione dati.

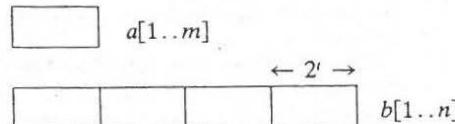
Problemi supplementari

- Implementare il primo algoritmo di merge sviluppato.
- Progettare e sviluppare un algoritmo di merge che legga i dati da due file di lunghezza non conosciuta. Utilizzare i test di end-of-file per individuare la fine degli insiemi di dati.
- Progettare e sviluppare un algoritmo di merge che utilizzi soltanto due array. Si può ritenere che le dimensioni dei due insiemi di dati siano note in partenza. Un modo interessante per risolvere il problema è collocare l'array con l'elemento massimo in modo che corrisponda alla parte finale dell'array di output. Il diagramma seguente illustra l'idea (l'array *b* contiene l'elemento massimo).



Ciò semplifica l'operazione, poiché una volta concluso il merge di *a*, i restanti elementi di *b* sono già al loro posto.

- Progettare un algoritmo per fondere tre array.
- Nel caso speciale in cui occorra "fondere" due file di m ed n elementi con $m = 1$ ed n grande, il problema è risolto in modo migliore da una ricerca binaria. È anche possibile dimostrare che le condizioni ottime per il merge si hanno quando $m = n$. Tenendo presenti questi fatti è possibile implementare un algoritmo di fusione che riunisce una ricerca binaria con le normali operazioni di merging, così da conservare le caratteristiche migliori di entrambe. Ciò si può realizzare immaginando di suddividere il più grande dei due insiemi di dati in blocchi di lunghezza uguale a quella del più piccolo. Per esempio:



ove la dimensione dei blocchi è 2^t e $t = \lfloor \log_2(n/m) \rfloor$. Il merge inizia allora procedendo a ritroso lungo b , con passi di ampiezza 2^t , finché non si scopre che l'elemento $a[m]$ è contenuto in un dato blocco. L'ultimo elemento di a può quindi essere combinato col blocco mediante ricerca binaria in t confronti. Il procedimento si ripete poi per $a[m-1]$. Implementare l'algoritmo. (Si veda: F.W.Hwang e S.Lin, "A simple algorithm for merging two disjoint linearly ordered data sets", *SIAM J. Computing*, 1, 31-39 (1972)).

ALGORITMO 5.2 ORDINAMENTO PER SELEZIONE

Problema

Dato un insieme di n interi disposti casualmente, ordinarlo in ordine non decrescente utilizzando un sort per selezione.

Sviluppo dell'algoritmo

Un'idea importante nell'ordinamento dei dati è l'impiego di un metodo di selezione per realizzare l'ordinamento richiesto. Nella sua forma più semplice, ad ogni passo del processo di ordinamento, occorre trovare il prossimo elemento più piccolo e disporlo in ordine.

Consideriamo l'array disordinato:

$a[1]$	$a[2]$	$a[3]$	\dots	$a[8]$
20	35	18	8	14

array disordinato

Quel che ci proponiamo di fare è sviluppare un meccanismo che converte l'array disordinato nella configurazione ordinata seguente:

$a[1]$	$a[2]$	\dots	$a[8]$
3	8	14	20

array ordinato

Confrontando l'array ordinato e quello disordinato vediamo che un modo per iniziare il processo di ordinamento potrebbe essere l'esecuzione dei due passi seguenti:

1. Ricerca l'elemento minimo dell'array disordinato;
2. Colloca tale elemento in posizione $a[1]$.

↓							
20	35	18	8	14	41	3	39

Per individuare l'elemento minimo si può impiegare il costrutto seguente:

```
min := a[1];
for j := 2 to n do
  if a[j] < min then min := a[j]
```

Quindi per portare il minimo in posizione $a[1]$ si può usare l'assegnazione:

$a[1] := min$

Eseguendo questi due passi sull'array disordinato otteniamo:

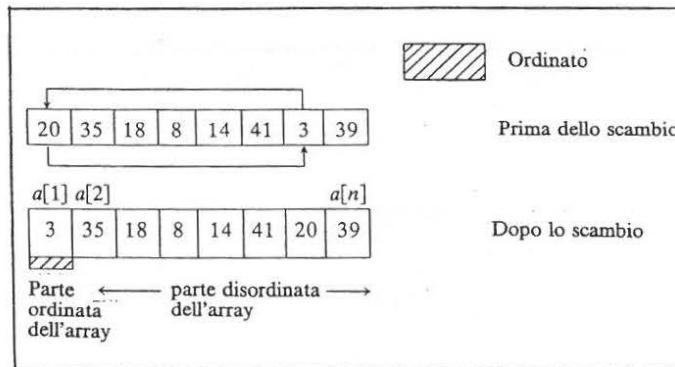
3	35	18	8	14	41	3	39
---	----	----	---	----	----	---	----

Applicando questi due passi abbiamo certamente ottenuto di mettere il 3 in posizione $a[1]$, ma così facendo abbiamo *perduto* il 20 presente in origine. Inoltre ora abbiamo due 3 nell'array mentre inizialmente ne avevamo uno solo. Per evitare questi problemi dobbiamo ovviamente procedere in modo diverso: prima di porre il 3 in posizione $a[1]$ occorre *salvare* il valore ivi contenuto, cioè il 20. Un modo semplice per farlo sarebbe quello di memorizzare $a[1]$ in una variabile temporanea (p.es. $t := a[1]$ ottiene lo scopo).

La domanda successiva è che fare del 20 una volta che il 3 è stato collocato in posizione $a[1]$. Occorre rimetterlo nell'array, ma in quale posizione? Se lo collocassimo in posizione 2 non miglioreremmo di molto le cose poiché dovremmo cercare un posto per il 35. Un esame accurato della situazione, illustrata in figura 5.2, mostra che in effetti possiamo rimetterlo nella posizione in cui abbiamo trovato il minimo. Il 3 ancora in questa posizione è già stato trasferito così che non si perde informazione.

Fig. 5.2.

*Mecanismo basiliare
di selezione e
scambio.*



Per attuare queste due modifiche ci occorre un meccanismo che non solo trovi il minimo, ma che ricordi anche la posizione dell'array in cui il minimo è collocato al momento. Ovvero, tutte le volte che il minimo è aggiornato dobbiamo salvarne la posizione. Questi passi possono essere aggiunti al codice precedente:

```
min := a[1];
p := 1;
for j := 2 to n do
  if a[j] < min then
    begin
      min := a[j];
      p := j
    end
```

Il 20 e il 3 possono essere scambiati con le due istruzioni seguenti (si osservi che il 3 è salvato nella variabile temporanea *min*), che devono essere poste *al di fuori* del loop di ricerca del minimo.

```
a[p] := a[1]; [pone 20 nella posizione p]
a[1] := min
```

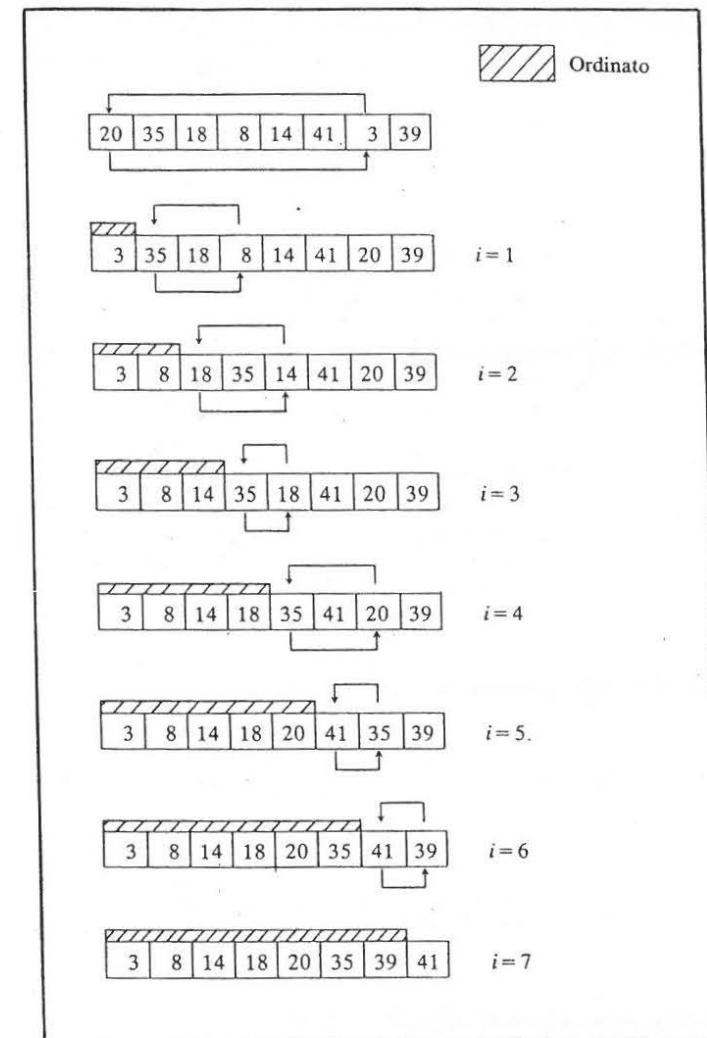
Abbiamo per ora un meccanismo che "ordina" un elemento, ma ciò che ci occorre è un meccanismo che ci consenta di ordinare *n* elementi. La stessa strategia si può applicare per ricercare e collocare nel giusto ordine il minimo tra gli elementi rimasti. Per individuarlo dobbiamo cominciare a scorrere l'array dalla posizione 2 poiché se ripartissimo dalla 1 ritroveremmo il 3 e non ci sarebbe di aiuto. I passi necessari sono perciò:

```
min := a[2];
p := 2;
for j := 3 to n do
  if a[j] < min then ...
```

Possiamo estendere questo procedimento alla ricerca del più piccolo elemento dopo i primi due, dopo i primi tre, e così via. Un modo concettualmente semplice per pensare il procedimento è quello di considerare l'array suddiviso ad ogni passo in una parte ordinata e una disordinata. Il meccanismo di selezione consiste allora nell'estendere ripetutamente di un elemento la parte ordinata, individuando l'elemento minimo successivo nella parte disordinata dell'array e collocandolo alla fine della parte ordinata. Ossia:

finché non sono stati ordinati tutti gli elementi dell'array,

Fig. 5.3.
*Ordinamento
mediante il metodo
di selezione.*



- si individuano la posizione p ed il valore $a[p]$ dell'elemento minimo rimasto nell'array non ordinato;
- si scambia l'elemento in $a[p]$ con l'elemento in *prima* posizione nella parte di array non ordinata.

Prima di considerare i dettagli dell'implementazione, si veda la figura 5.3, in cui è illustrato completamente il sort per il nostro esempio.

Si osservi che dobbiamo percorrere la parte non ordinata dell'array solo $n-1$ volte (7 in questo caso) poiché, non appena $n-1$ elementi sono stati disposti in ordine, l' n -esimo deve per definizione essere il massimo e pertanto è ordinato.

Per collocare progressivamente gli elementi nell'array ordinato occorre un loop per ricercare ripetutamente l'elemento più piccolo del rimanente array non ordinato.

Il meccanismo che possiamo utilizzare a questo proposito è:

```
for i = 1 to n-1 do
begin
  "ricerca e collega in ordine"
end
```

Il nostro sviluppo è adesso concluso.

Descrizione dell'algoritmo

- Definisce l'array di n elementi $a[1..n]$.
- Finché vi sono ancora elementi nella parte non ordinata dell'array
 - si individuano il minimo min e la sua posizione p nella parte non ordinata dell'array $a[i..n]$;
 - si scambia il minimo min nella parte non ordinata dell'array con il primo elemento $a[i]$ dell'array non ordinato.

Implementazione in Pascal

```
procedure selectionsort(var a: nelements; n: integer);
var i {first element in unsorted part of array},
    j {index for unsorted part of array},
    p {position of minimum in unsorted part of array},
    min {current minimum in unsorted part of array}: integer;

begin {sorts array a[1..n] into non-descending order using selection
method}
  {assert: n > 0 ∧ i = 0}
  {invariant: a[1..i] sorted ∧ all a[1..i] = < all a[i + 1..n]}
  for i := 1 to n - 1 do
```

```
begin {find minimum in unsorted part of array and exchange it
with a[j]}
  {assert: 1 = < i = < n - 1 ∧ j = i}
  min := a[i];
  p := i;
  {invariant: min = < all a[i..j] ∧ i = < n - 1 ∧ i = < j = < n
  ∧ i = < p = < j ∧ min = a[p]}
  for j := i + 1 to n do
    if a[j] < min then
      begin {update current minimum in unsorted part of array}
        min := a[j];
        p := j
      end;
    {assert: min = < all a[i..n] ∧ min = a[p]}
    a[p] := a[j];
    a[j] := min
  end
  {assert: a[1..n] sorted in non-descending order ∧ a permutation
  of original data set}
end
```

Note di progetto

- Nell'analizzare l'algoritmo di sort mediante selezione vi sono tre parametri importanti: il numero dei confronti eseguiti, il numero degli scambi, e il numero di volte che il minimo è aggiornato. Eseguendo il loop più interno, la prima volta sono eseguiti $n-1$ confronti, la seconda volta $n-2$, la terza $n-3$, e l'ultima volta un solo confronto. Pertanto il numero di confronti è sempre:

$$n_c = (n-1) + (n-2) + (n-3) + \dots + 1 = n(n-1)/2$$

(dalla formula di Gauss)

Il numero di scambi è sempre $(n-1)$ poiché uguaglia il numero di volte che il loop esterno è eseguito. Il calcolo del numero di volte in cui il minimo è aggiornato richiede un'analisi più dettagliata, dal momento che dipende dalla distribuzione dei dati. In media si può dimostrare che per dati distribuiti casualmente sono richiesti $(n \log n + cn)$ aggiornamenti.

- Dopo l' i -esimo passo del loop più esterno i primi i valori $a[1..i]$ sono stati disposti in ordine non decrescente (ossia $a[1] \leq a[2] \leq \dots \leq a[i]$); pertanto dopo l' i -esimo passo tutti gli elementi d'indice $> i$ (ossia $a[i+1..n]$) sono maggiori o uguali ai primi i elementi. Dopo il j -esimo passo del loop più interno risultano determinati la posizione p e il valore min dell'elemento minimo di $a[i..j]$: al termine di questo loop risultano determinati la posizione e il valore dell'elemento minimo di $a[i..n]$. Per definizione entrambi i for-loop hanno termine, e con essi l'algoritmo.
- Tra gli algoritmi semplici di ordinamento, il sort per selezione è uno dei migliori poiché riduce al minimo il numero di scambi.

- eseguiti. Ciò può essere importante se vi è una quantità significativa di dati associata a ciascun elemento.
4. In questo progetto abbiamo visto come si costruisce un intero algoritmo progettando all'inizio un algoritmo che risolve il problema più semplice. Una volta ottenuto, può essere generalizzato per fornire la soluzione completa.
 5. Il numero di confronti richiesti dal sort per selezionare può essere ridotto considerando gli elementi a coppie ed individuando simultaneamente il minimo e il massimo. Nell'implementare questo algoritmo occorre prestare qualche attenzione.
 6. Esistono maniere più sofisticate ed efficienti per attuare il processo di selezione. Ne esamineremo qualcuna più avanti.

Applicazioni

Solo per l'ordinamento di piccoli insiemi; per insiemi vasti esistono metodi molto più efficienti.

Problemi supplementari

- 5.2.1 Ordinare un array in ordine decrescente.
- 5.2.2 Implementare un sort per selezione che elimini i doppiioni durante il processo di ordinamento.
- 5.2.3 Implementare un algoritmo che incorpori l'idea della nota 5 e determinare il numero di confronti che esso esegue.
- 5.2.4 Calcolare il numero minimo di aggiornamenti richiesti dal sort per selezione per dati disposti casualmente, in ordine inverso.
- 5.2.5 Il sort per selezione può essere modificato in modo da interrompersi non appena si scopre che l'insieme dei dati è ordinato. Ciò si ottiene contando il numero di volte in cui il minimo è aggiornato ad ogni passo di selezione. Sono richiesti anche altri cambiamenti. Implementare l'algoritmo.
- 5.2.6 Implementare una funzione che esamina un array e restituisca il valore booleano *sorted*, che sarà *vero* se l'array è in ordine non decrescente e *falso* in caso contrario.

ALGORITMO 5.3 ORDINAMENTO PER INTERSCAMBIO

Problema

Dato un insieme di n interi disposti a caso, ordinarlo in ordine non decrescente con un metodo di interscambio.

Sviluppo dell'algoritmo

Quasi tutti i metodi di sort si affidano allo scambio dei dati per ottenere il desiderato ordinamento: il metodo che consideremo ora si basa pesantemente su un meccanismo di interscambio. Immaginiamo di partire con il seguente insieme di dati casuali:

$a[1]$	$a[2]$	\dots	$a[n]$
30	12	18	8 14 41 3 39



Nei dati così come sono è presente veramente poco ordine: quello che andiamo cercando sempre nel sort è un modo per aumentare l'ordine. Osserviamo che i primi due elementi sono "fuori posto" nel senso che, indipendentemente da quella che sarà la configurazione finale, il 30 dovrà comparire dopo il 12. Se si scambiano 30 e 12 avremo in qualche modo "aumentato l'ordine" dei dati; con ciò si perviene alla configurazione seguente:

$a[1]$	$a[2]$	$a[3]$	
12	30	18	8 14 41 3 39



Dall'esame della nuova configurazione, vediamo che l'ordine dei dati può essere ulteriormente aumentato confrontando e scambiando il secondo e il terzo elemento; con questo nuovo cambiamento otteniamo la configurazione:

12	18	30	8	14	41	3	39
----	----	----	---	----	----	---	----

L'analisi fatta ci suggerisce che l'ordine dell'array può essere aumentato utilizzando i passi seguenti:

1. Per tutte le coppie contigue dell'array
 - (a) se la coppia corrente di elementi non è in ordine non decrescente, scambiare tra di loro gli elementi.

Dopo aver applicato questa idea a tutte le coppie contigue dell'array otteniamo la configurazione seguente:

12	18	8	14	30	3	39	41
----	----	---	----	----	---	----	----

Analizzando con attenzione il meccanismo scopriamo che esso *garantisce* che l'elemento massimo 41 venga spinto in ultima posizione nell'array: a questo punto l'ultimo elemento può dirsi "ordinato"; ma l'array è ancora ben lungi dall'esserlo.

Se ripartiamo dal principio e applichiamo lo stesso meccanismo potremo garantire che il valore massimo escluso l'ultimo (ossia 39) finisca nella penultima posizione dell'array. In questo secondo passaggio non è necessario coinvolgere nei confronti a coppie l'ultimo elemento poiché è già al suo posto. Con analogo ragionamento, in un terzo passaggio attraverso i dati, gli ultimi due elementi sono al posto giusto e pertanto non devono essere coinvolti nella fase di confronto e scambio.

Il metodo di scambi ripetuti che stiamo sviluppando garantisce che ad ogni passaggio attraverso i dati viene disposto in ordine un elemento in più. Poiché gli elementi sono n , ciò implica che per completare il sort occorrono $n-1$ passaggi, di lunghezza decrescente. È evidente che sono richiesti $n-1$ passaggi invece di n , poiché per definizione, una volta messi in ordine $n-1$ elementi, l' n -esimo valore è al posto giusto (in questo caso sarà il primo elemento). Combinando queste osservazioni con il nostro precedente meccanismo di confronto e scambio di una coppia contigua, la struttura generale dell'algoritmo diventa:

```

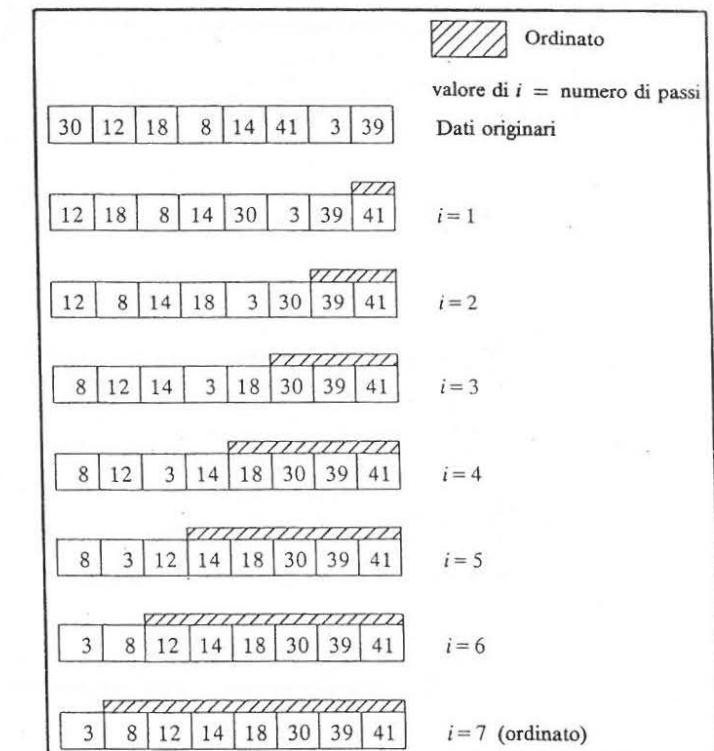
for i := 1 to n-1 do
begin
(a) Per tutte le coppie contigue dell'array
  (a.1) se la coppia corrente di elementi non è in ordine non
       decrescente, scambiare tra di loro gli elementi.
end

```

In figura 5.4 questo meccanismo è applicato al nostro set originale di dati.

if $a[j] > a[j+1]$ **then** scambia la coppia

Fig. 5.4.
Ordinamento per
interscambio.



Le nostre considerazioni finali riguardano i dettagli del loop interno, per garantire che esso operi solo sulla parte di array non ancora sottoposta a sort. Le coppie adiacenti possono essere confrontate utilizzando un test della forma:

La nostra maggiore preoccupazione qui è che l'indice non invada la parte ordinata dell'array. Poiché l'estremo superiore dell'indice ($j + 1$) ad ogni passo è direttamente funzione del numero di passi, e cioè il valore i , il limite superiore per il loop più interno dovrà essere $n-i$. Un controllo sul primo passaggio (quando $i = 1$) lo conferma: in tal caso per l'ultima coppia contigua quando $j = n-1$ l'indice $j + 1$ sarà uguale ad n come richiesto. Per l'esempio completo che abbiamo sviluppato vediamo che l'array è già completamente ordinato la volta che i raggiunge 6.

Non è difficile immaginare, per la natura di questo algoritmo di sort, che per molti insiemi di dati l'array sarà completamente ordinato prima che i raggiunga $n-1$. Possiamo trarre vantaggio da ciò? Per farlo ci occorre un criterio che ci dica quando l'array è completamente ordinato. Per cercare di ricavare tale criterio, consideriamo che cosa accade quando l'algoritmo è applicato ad un array già ordinato: in questo caso non viene eseguito alcuno scambio. Pertanto un modo abbastanza semplice per risolvere il nostro problema è quello di veri-

ficare se si sono avuti scambi oppure no durante l'ultimo passaggio: se non ci sono stati scambi, ne consegue che tutti gli elementi della parte dell'array non ancora soggetta a sort sono già in ordine non decrescente. L'operazione si può compiere con un indicatore logico *sorted*, cui viene assegnato il valore *vero* all'inizio del passo corrente e che è posto a *falso* se viene eseguito uno scambio. Un ulteriore confronto sarà richiesto al termine di ogni passo per vedere se *sorted* è ancora *vero*. Ciò modifica il loop più esterno.

Vi sono ancora altri perfezionamenti possibili per questo algoritmo, ma li lasciamo ai problemi supplementari.

Descrizione dell'algoritmo

1. Definire l'array $a[1..n]$ di n elementi.
2. Finché l'array non è ancora ordinato
 - (a) porre l'indicatore *sorted* al valore *true*;
 - (b) per tutte le coppie contigue di elementi nella parte di array non ancora ordinata
 - (b.1) se la coppia corrente di elementi non è in ordine non decrescente
 - (1.a) scambiare gli elementi della coppia.
 - (1.b) porre l'indicatore *sorted* al valore *false*.
3. Restituire l'array ordinato.

Implementazione in Pascal

```

procedure bubblesort(var a: nelements; n: integer);
var i {index for number of passes through the array},
    j {index for unsorted part of array},
    t {temporary variable used in exchange}: integer;
    sorted {if true after current pass then array sorted}: boolean;

begin {sorts array a[1..n] into non-descending order by exchange
method}
  {assert: n > 0}
  sorted := false;
  i := 0;
  {invariant: after ith iteration i < n  $\wedge$  a[n-i+1..n] ordered  $\wedge$  all
  a[1..n-i] <= all a[n-i+1..n]  $\vee$  (i < n  $\wedge$  a[1..n-i] ordered  $\wedge$ 
  a[n-i+1..n] ordered  $\wedge$  all a[1..n-i] <= all a[n-i+1..n])}
  while (i < n) and (not sorted) do
    begin {make next pass through unsorted part of array}
      sorted := true;
      i := i + 1;
      {invariant: after jth iteration j = < n - i  $\wedge$  all a[1..j] = < a[j+1]}
      for j := 1 to n - i do
        if a[j] > a[j+1] then

```

```

begin {exchange pair and indicate another pass required}
  t := a[j];
  a[j] := a[j + 1];
  a[j + 1] := t;
  sorted := false
end
{assert: all a[1..n-i] = < a[n-i+1..n]}
end
{assert: a[1..n] sorted in non-descending order  $\wedge$  a permutation
of original data set}
end

```

Note di progetto

1. I parametri significativi per analizzare questo algoritmo sono il numero dei confronti e il numero degli scambi effettuati. Il numero minimo di confronti è $(n-1)$ quando i dati sono già ordinati. Il numero massimo di confronti si ha quando vengono eseguiti $(n-1)$ passaggi: in questo caso vengono compiuti $n(n-1)/2$ confronti.
Se l'array è già ordinato non vengono effettuati scambi; nel caso peggiore ci sono tanti scambi quanti sono i confronti, vale a dire $n(n-1)/2$. Nel caso medio vengono compiuti $n(n-1)/4$ scambi.
2. Dopo l' i -esima iterazione, tutti gli elementi $a[n-i+1..n]$ sono ordinati e tutti gli $a[1..n-i]$ sono minori o uguali ad $a[n-i+1]$. Inoltre il loop più interno può decidere che gli elementi $a[1..n-i]$ sono ordinati anche per $i < n$. Per il loop più interno, dopo la j -esima iterazione l'elemento $a[j+1]$ sarà maggiore o uguale a tutti gli elementi in $a[1..j]$. La conclusione del *while*-loop è assicurata poiché i viene incrementato di 1 ad ogni passo mentre il *for*-loop, per definizione, deve sempre terminare.
3. Un punto debole di questo algoritmo è che esso si basa sugli interscambi più pesantemente della maggior parte degli altri metodi. Poiché tali scambi sono abbastanza dispendiosi in termini di tempo, questa caratteristica rende il metodo molto costoso per ordinare grossi insiemi di dati casuali. Tuttavia c'è un caso in cui il *bubblesort* (come viene normalmente chiamato) risulta efficiente: se i dati hanno solo una piccola percentuale di elementi fuori posto, il *bubblesort* può richiedere soltanto un piccolo numero di scambi e confronti.
4. L'esame delle configurazioni intermedie per l'insieme di dati dell'esempio suggerisce come l'algoritmo perda progressivamente in equilibrio e simmetria. Mentre gli elementi più grandi migrano rapidamente verso l'estremo destro dell'array, gli elementi piccoli si muovono solo molto lentamente verso l'estremo di sinistra (si osservi il movimento del 3 nell'esempio). Questo problema può essere ridotto *alternando* il senso di percorrenza dei dati ad ogni passaggio.

Applicazioni

Solo per l'ordinamento di dati in cui una piccola percentuale di elementi è in disordine.

Problemi supplementari

- 5.3.1 Confrontare il sort per selezione e il bubblesort, applicati a dati casuali, utilizzando un contatore del numero di confronti e scambi effettuati.
- 5.3.2 Implementare una versione del bubblesort che, a differenza del presente algoritmo, metta insieme l'array ordinato a partire dall'elemento più piccolo.
- 5.3.3 Progettare e implementare un algoritmo che includa il suggerimento della precedente nota 4.
- 5.3.4 Progettare e implementare un bubblesort modificato che includa interscambi a distanza fissa.
- 5.3.5 Provare a progettare un bubblesort *meno* efficiente del presente algoritmo.

ALGORITMO 5.4 ORDINAMENTO PER INSERZIONE

Problema

Dato un insieme di n interi disposti a caso, ordinarlo in ordine non decrescente con un metodo di inserzione.

Sviluppo dell'algoritmo

L'ordinamento per inserzione è uno dei modi più ovvi e naturali per ordinare le informazioni. Esso approssima molto da vicino la procedura di ordinamento spesso usata dai giocatori di carte. Alla base di questo algoritmo è l'idea di costruire la soluzione completa prendendo un elemento dalla parte non ancora ordinata ed inserendolo nella soluzione corrente parzialmente ordinata, che aumenta così di un elemento. Questo meccanismo suggerisce un sort per selezione in cui selezioniamo il minimo nella parte non ancora ordinata e lo collociamo all'estremità della parte ordinata. Abbiamo:

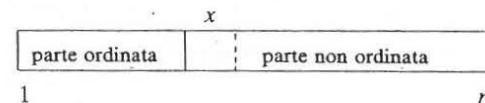
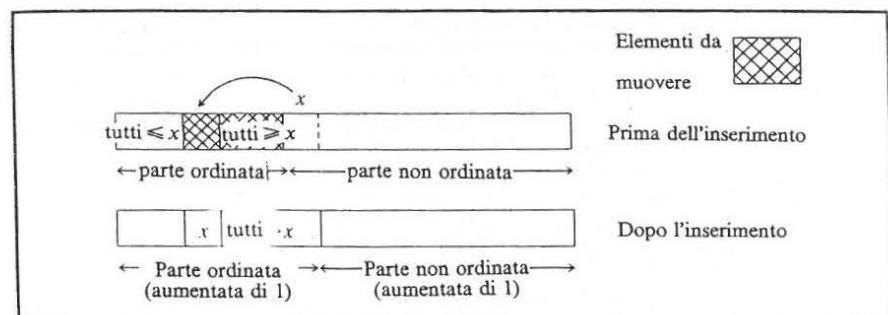


Fig. 5.5.
Meccanismo di inserzione.

Un modo alternativo, semplice e sistematico, con cui scegliere il prossimo elemento da inserire è quello di prelevare il *primo* elemento della parte non ancora ordinata (ossia la x dell'esempio). Dovremo allora inserire opportunamente la x nella parte ordinata e, dinamicamente, estendere di un elemento tale parte. In figura 5.5 è illustrato schematicamente ciò che vogliamo fare.



All'inizio è disordinato l'intero array: per partire occorre che x sia il secondo elemento ed $a[1]$ la "parte ordinata". La parte ordinata viene quindi aumentata inserendo prima il secondo elemento, poi il terzo, e così via. A questo punto la traccia per il nostro algoritmo di sort per inserzione è:

```
for i := 2 to n do
begin
  (a) scegliere il prossimo elemento da inserire ( $x := a[i]$ ),
  (b) inserire  $x$  nella parte ordinata dell'array,
end
```

Per fare spazio all'inserimento di x , tutti gli elementi maggiori di x debbono essere spostati in avanti di una posizione (area tratteggiata di figura 5.5). Partendo con $j := i$, i passi seguenti ci permettono di muoverci a ritroso nell'array e di assolvere il compito.

```
while  $x < a[j-1]$  do
begin
   $a[j] := a[j-1];$ 
   $j := j-1$ 
end
```

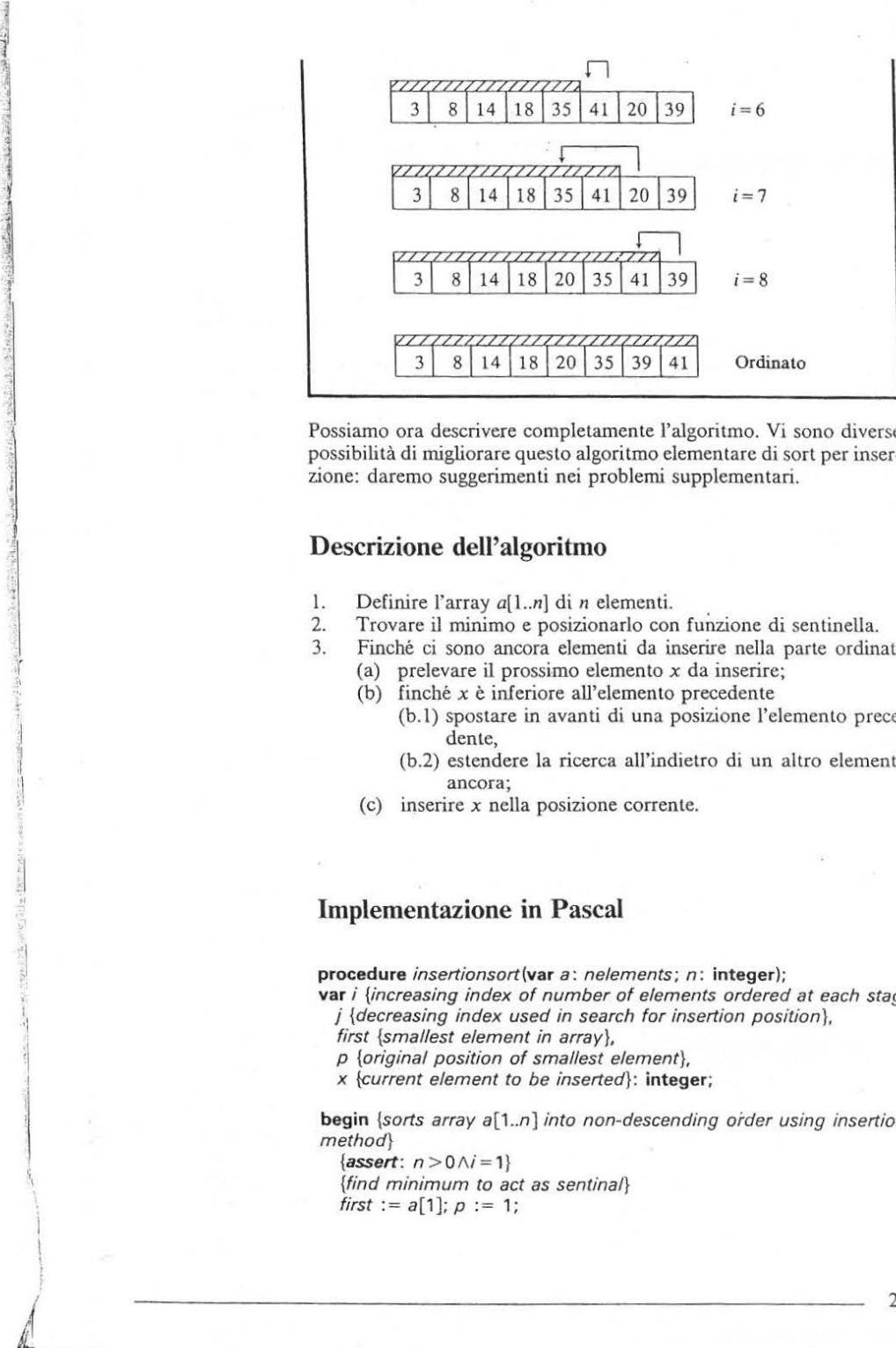
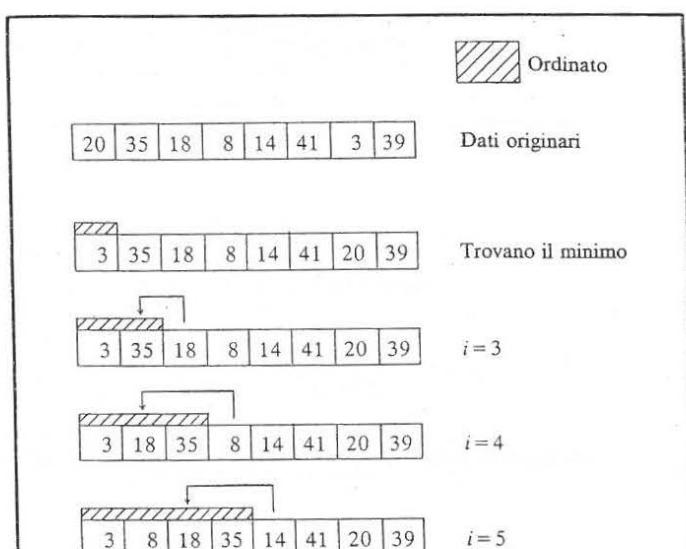
Una volta stabilito che x è maggiore o uguale ad $a[j-1]$, il loop dovrà terminare e noi sapremo che x deve essere posto in $a[j]$ per soddisfare i requisiti dell'ordinamento non decrescente. Con questo loop c'è un problema di terminazione quando accade che x sia minore di tutti gli elementi $a[1..i-1]$: in tal caso il loop produrrà un riferimento all'elemento $a[0]$. Dobbiamo perciò proteggerci contro questo inconveniente: un modo potrebbe essere quello di includere un controllo per l'indice j , scrivendo:

```
while x < a[j-1] and j > 2 do
```

Questo comporta un test più oneroso, da eseguire molto spesso. Un altro approccio che possiamo adottare per terminare correttamente il loop è quello di collocare contemporaneamente x come sentinella in $a[0]$ o $a[1]$, che obblighi il loop a concludersi. Questo è un modo di programmare piuttosto inelegante: ci domandiamo se esista un'alternativa più pulita. La nostra preoccupazione è sempre dovuta al fatto che vogliamo inserire un elemento *più piccolo* di quello che in questo momento occupa la posizione $a[1]$. Se l'elemento minimo fosse in $a[1]$ non avremmo alcuna preoccupazione circa la terminazione del meccanismo di inserimento. Questo ci suggerisce che il modo più semplice di superare il nostro problema è determinare il minimo e metterlo al suo posto *prima* di iniziare il processo di inserzione. Una volta che il minimo è al suo posto, i primi due elementi saranno in ordine e quindi inizieremo l'inserzione partendo dal *terzo* elemento. Per verificare questo schema di ragionamento, la figura 5.6 illustra il meccanismo applicato ad un esempio specifico.

485b, fc

Fig. 5.6.
Ordinamento per
inserzione.



Possiamo ora descrivere completamente l'algoritmo. Vi sono diverse possibilità di migliorare questo algoritmo elementare di sort per inserzione: daremo suggerimenti nei problemi supplementari.

Descrizione dell'algoritmo

1. Definire l'array $a[1..n]$ di n elementi.
2. Trovare il minimo e posizionarlo con funzione di sentinella.
3. Finché ci sono ancora elementi da inserire nella parte ordinata
 - (a) prelevare il prossimo elemento x da inserire;
 - (b) finché x è inferiore all'elemento precedente
 - (b.1) spostare in avanti di una posizione l'elemento precedente,
 - (b.2) estendere la ricerca all'indietro di un altro elemento ancora;
 - (c) inserire x nella posizione corrente.

Implementazione in Pascal

```

procedure insertionsort(var a: nelements; n: integer);
var i {increasing index of number of elements ordered at each stage},
    j {decreasing index used in search for insertion position},
    first {smallest element in array},
    p {original position of smallest element},
    x {current element to be inserted}: integer;
begin {sorts array a[1..n] into non-descending order using insertion
method}
  {assert: n > 0 ∧ i = 1}
  {find minimum to act as sentinel}
  first := a[1]; p := 1;
  {insertion loop}
  for i := 2 to n do
    begin
      j := i;
      while j > 1 and a[j-1] > a[j] do
        begin
          a[j] := a[j-1];
          j := j - 1;
        end;
      a[j] := x;
    end;
end;
  
```

```

for  $i := 2$  to  $n$  do
  if  $a[i] < \text{first}$  then
    begin
       $\text{first} := a[i];$ 
       $p := i;$ 
    end;
     $a[p] := a[1];$ 
     $a[1] := \text{first};$ 
  {invariant: }  $1 \leq i \leq n \wedge a[1..i]$  ordered
  for  $i := 3$  to  $n$  do
    begin {insert  $i$ th element — note  $a[1]$  is a sentinel}
       $x := a[i];$ 
       $j := i;$ 
    {invariant: }  $1 \leq j \leq i \wedge x \leq a[j..i]$ 
    while  $x < a[j-1]$  do
      begin {search for insertion position and move up elements}
         $a[j] := a[j-1];$ 
         $j := j-1;$ 
      end;
    {assert: }  $1 \leq j \leq i \wedge x \leq a[j+1..i]$ 
    {insert  $x$  in order}
     $a[j] := x;$ 
  end
  {assert: }  $a[1..n]$  sorted in non-descending order and a
  permutation of original data
end

```

Note di progetto

- Per analizzare questo algoritmo sono importanti due parametri: il numero dei confronti eseguiti (ossia $x < a[j-1]$) e il numero degli elementi dell'array che debbono essere spostati. Il while-loop più interno deve essere eseguito almeno una volta per ogni valore di i : ne viene che debbono essere effettuati almeno $(2n-3)$ confronti. All'estremo opposto, per ogni valore di i vengono compiuti al massimo $(i-1)$ confronti: utilizzando formule standard di sommatoria possiamo dimostrare che nel caso peggiore saranno richiesti $(n^2 + n - 4)/2$ confronti. Assumendo che mediamente ad ogni passo occorrono $(i+1)/2$ confronti prima che x possa essere inserito, si può dimostrare saranno allora richiesti $(n^2 + 6n - 12)/4$ confronti. La prestazione dell'algoritmo è pertanto $\Theta(n^2)$. Argomenti simili si possono impiegare per calcolare il numero delle operazioni di spostamento.
- La condizione che rimane invariata nel loop più esterno è che dopo la i -esima iterazione i primi i valori dell'array sono stati disposti in ordine non decrescente (ossia $a[1] \leq a[2] \leq \dots \leq a[i]$): dopo l' n -esima iterazione l'array è ordinato completamente. Al termine dell'iterazione di indice j nel while-loop più interno è garantito che il sottoinsieme di elementi $a[j..i]$ sia tutto $\geq x$ con $1 \leq j \leq i$. L'algoritmo funziona correttamente per tutti i valori di

$n \geq 1$. Il for-loop ha termine per definizione; la terminazione del loop più interno è assicurata poiché l'indice del ciclo è decrescente in senso stretto ed il minimo in $a[1]$ garantisce che la condizione $x < a[j-1]$ risulti alla fine falsa.

- Per piccoli volumi di dati casuali, il sort per inserzione è di norma considerato come il migliore degli algoritmi di ordinamento di classe n^2 .
- Lo schema di questo algoritmo è stato ottenuto ricavando per primo il meccanismo di inserzione in un array di dimensione $n = 2$. La generalizzazione a problemi di maggiore dimensione è quindi immediata.
- C'è una maniera più nitida di eseguire un sort per inserzione che richiede un leggero aumento nell'onere dei loop. Essa implica la ricerca in ogni istante della posizione di inserimento a partire dall'inizio dell'array. La parte centrale del sort per inserzione assume allora la forma:

```

 $\vdots$ 
for  $i := 2$  to  $n$  do
  begin {search for  $x$ 's position then insert it}
     $j := 1; x := a[i];$ 
    while  $x > a[j]$  do  $j := j+1;$ 
    for  $k := i$  down to  $j+1$  do  $a[k] := a[k-1];$ 
     $a[j] := x$ 
end

```

Applicazioni

In presenza di insiemi di dati relativamente piccoli. È usato spesso a questo scopo in algoritmi più avanzati di ordinamento (algoritmo 5.6).

Problemi supplementari

- Confrontare il sort per selezione e quello per inserzione su dati casuali. Per eseguire l'analisi comparativa utilizzare il numero di spostamento e quello di confronti.
- Un piccolo risparmio nel sort per inserzione si può ottenere con un metodo che non esegua la scelta dell'elemento *successivo* per l'inserimento. Cercare di includere questo suggerimento.
- Un risparmio di confronti e spostamenti si può ottenere inserendo ad ogni passo *più di un* elemento nella parte ordinata dell'array. Progettare un simile algoritmo che funzioni inserendo due elementi ad ogni passata del loop più esterno.
- L'algoritmo che abbiamo prodotto non trae vantaggio dal fatto che gli elementi $a[1..i]$ sono già ordinati. Utilizzare una ricerca binaria per accelerare la ricerca del punto di inserzione (vedi algoritmo 5.7).

- 5.4.5 Modificare il sort per inserzione in modo che risulti più equilibrato, consentendo l'inserzione ad *entrambi* gli estremi dell'array. Si può anche includere un meccanismo di scambio per accelerare ulteriormente l'algoritmo.
- 5.4.6 La posizione dell'ultima inserzione può essere "ricordata" ed usata per inserire il *prossimo* elemento. Implementare una versione del sort per inserzione che includa questa idea.

ALGORITMO 5.5 ORDINAMENTO PER DIMINUZIONE DI INCREMENTI

Problema

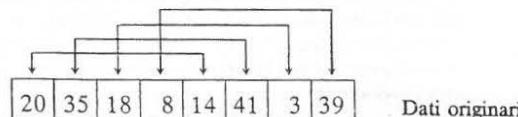
Dato un insieme di n interi disposti a caso, ordinarlo in ordine non decrescente con il metodo di Shell di inserzione con diminuzione degli incrementi.

Sviluppo dell'algoritmo

Il confronto tra un insieme di dati disposti a caso e lo stesso rior-
dinato mostra che, per un array di ampiezza n , gli elementi debbono spostarsi in media di $n/3$ posti. Questa osservazione ci suggerisce che l'evoluzione verso l'ordine finale sarà più apida se inizialmente gli elementi sono confrontati e mossi su distanze più lunghe. Questa strategia ha mediamente l'effetto di portare *prima* ciasun elemento *più vicino* alla sua posizione definitiva. Il caso peggiore per questa strategia è quando gli elementi sono confrontati e mossi su una distanza unitaria (come nel bubblesort).

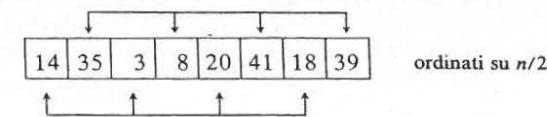
Ci domandiamo ora come implementare questa idea. Occorre un metodo che inizialmente muova gli elementi su lunghe distanze e poi, col progredire dell'ordinamento, riduca la distanza attraverso la quale gli elementi sono confrontati e mossi. Una strategia per un array di ampiezza n è quella di partire confrontando gli elementi a distanza $n/2$, e successivamente a distanza $n/4$, $n/8$, $n/16$,...1.

Consideriamo che cosa accade quando l'idea di $n/2$ è applicata ai dati seguenti.

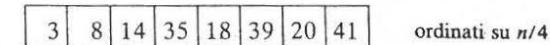


Dati originari

Dopo confronto e scambio sulla distanza $n/2$ abbiamo $n/2$ catene di lunghezza 2, "ordinate".

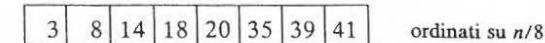


Il passo successivo è confrontare gli elementi su una distanza $n/4$ ottenendo in tal modo due catene ordinate di lunghezza 4.



Si osservi che dopo l'ordinamento su $n/4$ la "quantità di disordine" nell'array è relativamente piccola.

Nel passo finale si produce *una sola* catena di lunghezza 8 confrontando e ordinando gli elementi a distanza unitaria.



Poiché il disordine relativo dell'array è piccolo andando verso la fine del sort (in questo caso quando iniziamo l'ordinamento a $n/8$) dovremo scegliere come metodo di ordinamento delle catene un algoritmo che sia efficiente per dati parzialmente ordinati. Abbiamo già visto che il sort per inserzione e il bubblesort sono efficaci nell'ordinare dati già parzialmente in ordine. Il sort per inserzione è preferibile, poiché non si basa così massicciamente sugli interscambi. La domanda successiva da fare è: il sort per inserzione è appropriato negli stadi iniziali quando ci occorre ordinare tante catene più corte? Poiché inizialmente le catene sono corte, possiamo aspettarci che il sort per inserzione lavori bene anche in questo caso.

La considerazione successiva, e più importante, è come fare per applicare un sort per inserzione alle diverse catene. La struttura generale dell'algoritmo deve consentirci di applicare il sort per inserzione sulle distanze: $n/2$, $n/4$, $n/8$,...,1.

Possiamo implementarlo così:

```

inc := n;
while inc > 1 do
begin
  (a) inc := inc div 2;
  (b) "Ordina per inserzione tutte le catene con incremento inc"
end
  
```

I passi successivi nello sviluppo sono: stabilire quante catene devono essere ordinate per ogni valore dell'incremento e capire come accedere alle catene individuali per l'ordinamento per inserzione. Un esame accurato dell'esempio mostra che il numero di catene da ordinare è sempre uguale all'incremento stesso (ossia uguale a *inc*). Possiamo pertanto espandere il nostro algoritmo così:

```

inc := n;
while inc>1 do
  begin
    (a) inc := inc div 2;
    (b) for j := 1 to inc do
      begin
        "Ordina per inserzione tutte le catene con incremento inc"
      end
    end

```

Adesso viene il punto cruciale di calare nell'algoritmo il sort per inserzione (per una descrizione completa del sort per inserzione vedi l'algoritmo 5.4).

Nell'implementazione standard il primo elemento che cercavamo di inserire era il secondo elemento dell'array. Nel contesto attuale, per ogni catena da ordinare, esso sarà il secondo elemento della catena, la cui posizione *k* è individuata da:

k := *j*+*inc*

Gli elementi successivi di ciascuna catena che comincia da *j* si possono trovare con:

k := *k* + *inc*

finché *k* non supera *n*, numero di elementi dell'array.

Con questi perfezionamenti otteniamo:

```

inc := n;
while inc>1 do
  begin
    inc := inc div 2;
    for j := 1 to inc do
      begin
        k := j+inc;
        while k≤n do
          begin
            x := a[k];
            "Trova la posizione corrente per x"
          end
        end
      end
    end

```

```

a[current] := x;
k := k+inc
end
end

```

Nell'implementazione precedente del sort per inserzione si è usata una sentinella: nel caso attuale non abbiamo un'unica posizione (ossia *a*[1]) dove abbiano termine tutte le catene da ordinare. Ciò complica il proposito di utilizzare una sentinella per concludere il meccanismo d'inserzione. Nell'effettuare l'inserimento dobbiamo porre *x* (ovvero *a*[*k*]) nella sua posizione "ordinata" nella catena. Partendo da:

current := *k*

il primo elemento da confrontare con *x* si trova nella posizione precedente *previous*, con:

previous := *current* - *inc*

I membri della catena ancora precedenti si possono individuare con:

previous := *previous* - *inc*

Nell'algoritmo precedente, per eseguire l'inserimento adoperavamo questo loop:

while *x*<*a*[*previous*] **do**

A causa del problema delle sentinelle dobbiamo trovare un altro modo per concludere il loop. Se *previous* viene diminuito ripetutamente di *inc*, prima o poi finirà per essere minore della *j* che segna l'inizio della catena, oggetto corrente dell'inserzione. Pertanto potremo tentare l'inserimento solamente finché:

previous ≥ *j*

Saremmo tentati di scrivere la condizione richiesta nella forma:

while (*previous* ≥ *j*) and (*x*<*a*[*previous*]) **do**

Benché ciò appaia perfettamente ragionevole, una simile implementazione presenta un problema in Pascal poiché il calcolo della condizione del loop non si arresta appena si scopre che la prima condizione (*previous* ≥ *j*) non è vera. Ne deriva che è possibile l'esecuzione del test *x*<*a*[*previous*] anche quando *previous* è oltre i propri limiti. Un modo per aggirare il problema è di collocare il test nel corpo del loop e utilizzare un flag (p.es. *inserted*) per indicare se il confronto *x*<*a*[*previous*] è vero o falso.

La condizione del loop diventa così:

```
while (previous≥j) and not (inserted) do
```

L'esecuzione dell'inserimento è fatta come nell'implementazione precedente. Ad ogni passo del processo iterativo dobbiamo tener nota degli elementi *current* e *previous* della catena corrente. Ogni volta, prima che *previous* venga diminuito di *inc*, la variabile *current* deve prenderne il valore. Questo corrisponde all'uso di $a[j]$ e $a[j - 1]$ nell'algoritmo originale di sort per inserzione. Siamo adesso in grado di riassumere i dettagli dell'implementazione completa. L'algoritmo appena sviluppato è noto generalmente come *shellsort* dal nome del suo inventore D.Shell.

Descrizione dell'algoritmo

1. Definire l'array $a[1..n]$ di n elementi;
2. Porre il valore dell'incremento *inc* pari a n .
3. Finché il valore dell'incremento è maggiore di uno
 - (a) ridurre *inc* di un fattore 2;
 - (b) per tutte le *inc* catene da ordinare, eseguire con passo *inc*
 - (b.1) determinare la posizione *k* del secondo elemento della catena di indice *current*,
 - (b.2) finché non si raggiunge la fine della catena corrente
 - (2.a) utilizzare il meccanismo di inserzione per mettere a posto $x = a[k]$,
 - (2.b) spostare di uno la catena corrente aumentando *k* di *inc*.

Implementazione in Pascal

```
procedure shellsort(var a: nelements; n: integer);
var inc {stepsize at which elements are to be sorted},
    current {position in chain where x is finally inserted},
    previous {index of element currently being compared with x},
    j {index for lowest element in current chain being sorted},
    k {index of current element being inserted},
    x {current value to be inserted}: integer;
    inserted {is true when insertion can be made}: boolean;

begin {uses a diminishing increment modification to insertion sort}
  {assert: n > 0}
  inc := n;
  {invariant: n ≥= inc ≥= 1 ∧ after each iteration all inc chains with
  displacement inc are ordered}
```

```
while inc > 1 do
  begin {do insertion sorts with diminishing increments}
    inc := inc div 2;
    {invariant: 1 ≤ j ≤ inc ∧ after jth iteration first j chains with
    displacement inc are ordered}
    for j := 1 to inc do
      begin {sort all chains for current interval inc}
        k := j + inc;
        {invariant: 1 ≤ j ≤ inc ∧ 1 ≤ inc ≤ n ∧ k ≤ n
        ∧ a[j] = < a[j + inc] = < a[j + 2 * inc] = < ... = < a[k - inc]}
        while k ≤ n do
          begin {step through all members of current chain}
            inserted := false;
            x := a[k];
            current := k;
            previous := current - inc;
            {invariant: j < current ≤ k ∧ x < a[current]
            = < a[current + inc] = < ... = < a[k]}
            while (previous >= j) and (not inserted) do
              begin {locate position and perform insertion of x}
                if x < a[previous] then
                  begin {move chain member up one position}
                    a[current] := a[previous];
                    current := previous;
                    previous := previous - inc
                  end
                else
                  inserted := true
                end;
                {assert: x = < a[current + inc] = < a[current + 2 * inc]
                = < ... = < a[k]}
                a[current] := x;
                k := k + inc
              end
            end
          end
        end;
        {assert: a[1..n] sorted in non-descending order ∧ a permutation
        of original data set}
      end
end
```

Note di progetto

1. L'analisi dello shellsort si è dimostrata un problema molto difficile. Si può dimostrare che esistono scelte migliori di decremento che non la sequenza $n/2, n/4, n/8, \dots, 1$. Per la sequenza di decrementi $2^{t-1}, \dots, 31, 15, 7, 3, 1$ il numero di confronti e di spostamenti è proporzionale a $n^{1.2}$. Questo è chiaramente meglio dei metodi di sort finora considerati, ma al tempo stesso non è così buono come i metodi di sort avanzati che richiedono confronti dell'ordine di $n \log_2 n$.
2. La condizione che rimane invariata per il while-loop più esterno dello shellsort è che dopo ogni iterazione tutte le catene (in

- numero di *inc* in quel momento), formate con elementi distanziati di *inc* posizioni, sono state riordinate per inserzione. Dopo il passo *j*-esimo del **for**-loop immediatamente interno, le prime *j* catene (con incrementi *inc*) sono state riordinate per inserzione. Per il **while**-loop ancora più interno (ossia while $k \leq n$), dopo l'iterazione di indice *k*, l'elemento in posizione *k*-esima è stato correttamente inserito nella sua catena. Con il **while**-loop più interno di tutti dopo l'iterazione di indice *previous* si è stabilito se l'elemento corrente può essere inserito nella posizione *current* oppure se tutti gli elementi compreso *a[previous]* sono maggiori di *x*. Tutti questi elementi sono stati spostati di una posizione in previsione di un eventuale inserimento di *x* nella catena.
3. In questo progetto si è cercato di muovere il più presto possibile gli elementi, allo scopo di ottenere un algoritmo nel complesso più veloce.
 4. Lo sviluppo di questo algoritmo è leggermente più complesso degli altri sort che abbiamo esaminato. Visto come esempio, conferma l'importanza di un progetto *top-down* corretto e pulito.

Applicazioni

Adatto per ordinare ampi insiemi di dati, ma esistono metodi più avanzati con prestazioni migliori.

Problemi supplementari

- 5.5.1 Utilizzare il numero di confronti e spostamenti per paragonare lo shellsort con un sort standard per inserzione. Per il confronto utilizzare dati casuali.
- 5.5.2 Confrontare le prestazioni di implementazioni dello shellsort che impieghino le sequenze di decrementi $n/2, n/4, n/8, \dots 1$ e $2^{\log_2 n} - 1, \dots, 31, 15, 7, 3, 1$. Utilizzare dati casuali ed il numero di confronti e spostamenti come cifra di merito.
- 5.5.3 Implementare una versione dello shellsort che include un bubblesort al posto del sort per inserzione. Confrontare le prestazioni delle due versioni.
- 5.5.4 Progettare un algoritmo che confronti un array casuale e riordinato e che misuri la distanza media percorsa dagli elementi passando dalla disposizione casuale a quella ordinata.
- 5.5.5 Modificare lo shellsort in modo che non sia necessario il controllo $previous \geq j$ nel **while**-loop più interno.
- 5.5.6 Una versione più nitida dello shellsort si può ottenere modificando gli estremi da cui l'inserimento è eseguito ad ogni passo. Ciò richiede una leggera modifica all'idea suggerita nella nota 5 dell'algoritmo 5.4. Includere questa idea nello shellsort.

ALGORITMO 5.6 ORDINAMENTO PER PARTIZIONE

Problema

Dato un insieme di *n* interi disposti a caso, ordinarlo in ordine non decrescente utilizzando il metodo di partizione di Hoare.

Sviluppo dell'algoritmo

L'algoritmo di shellsort ci ha introdotto all'idea che in un algoritmo efficiente i dati devono essere mossi su grandi distanze nei primi passi dell'ordinamento. C'è un modo più semplice e più efficiente della strategia usata in shellsort (algoritmo 5.5). La scoperta di un meccanismo più semplice e più potente, anche sapendo che esiste, è probabilmente qualcosa fuori della portata di molti di noi. Affrontiamo tuttavia questo passo creativo ponendoci il genere di domande che possono condurci nella direzione corretta.

La nostra prima reazione all'idea di muovere i dati su lunghe distanze avrebbe potuto essere la domanda di quale sia la *massima* distanza su cui fare tale spostamento. La risposta è che dovremo scambiare il primo e l'ultimo elemento se essi sono in disordine. Analogamente, dovremo scambiare il secondo e il penultimo elemento se essi sono in disordine, e così via.

Applicando questo meccanismo all'array disordinato:

20	35	18	8	14	41	3	39
Array disordinato							

perveniamo a questa nuova configurazione:

20	3	18	8	14	41	35	39
----	---	----	---	----	----	----	----

Avendo compiuto questo primo passaggio attraverso l'array, ci domandiamo come proseguire. Il primo passaggio ci ha fornito una serie di coppie ordinate: un modo di procedere potrebbe essere quello di estendere tali coppie in segmenti ordinati più grandi. Esplorando questa idea scopriamo che si perviene a operazioni di merging, perdendo così la caratteristica di scambiare dati su lunghe distanze. A prima vista muoversi da entrambi gli estremi sembra una buona idea, ma scambiare semplicemente le coppie quando sono disordinate non è di grande aiuto. Quel che appare sbagliato in questa strategia è che non abbiamo affatto ridotto, in qualche modo facilmente riconoscibile, la

dimensione del problema da risolvere. Abbiamo perciò due alternative: possiamo cambiare il modo con cui elaboriamo gli elementi, oppure possiamo cambiare il criterio usato per coppie di elementi. Se decidiamo di continuare sulla strada precedente dobbiamo trovare un criterio alternativo per scambiare le coppie di elementi. Ma quale? Sembra un problema difficile da decidere in astratto, per cui torniamo al nostro insieme originario di dati per vedere se vi scorgiamo qualche indicazione.

Esaminando con attenzione i dati, vediamo che quando si confrontano il primo e l'ultimo dato (ossia 20 e 39), non ha luogo alcuno scambio. Tuttavia 20 è il quart'ultimo elemento in ordine di grandezza e perciò dovrebbe, in verità, essere mosso il più presto possibile verso l'estremo destro dell'array. Non ci sarebbe vantaggio scambiando 20 con 39 ma avrebbe molto senso scambiare il 20 con il 3 in settima posizione. Che cosa ci ha suggerito questo approccio? Ci ha guidato all'idea che probabilmente è una buona strategia muovere gli elementi grandi il più presto possibile verso l'estremo di destra e contemporaneamente muovere gli elementi piccoli verso l'estremo sinistro dell'array. Il nostro problema è quindi decidere in generale quali elementi siano *grandi* e quali *piccoli*.

Riflettendo per un istante, scopriamo che questo problema non ha una soluzione generale: ci siamo nuovamente impiantati! La sola possibilità rimasta è decidere *per tentativo* quale possa essere l'elemento che distinguerà gli elementi grandi da quelli piccoli. Idealmente, dopo la prima passata attraverso i dati, potremmo avere *tutti* gli elementi grandi nella metà di destra e tutti gli elementi piccoli nella metà di sinistra dell'array. Ciò corrisponde alla *partizione* dell'array in due sottoinsiemi.

Esemplicando:

Partizione di tutti gli elementi "piccoli"	Partizione di tutti gli elementi "grandi"
--------------------------------------------	-------------------------------------------

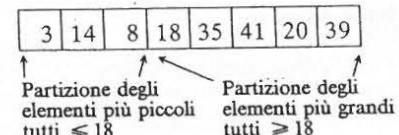
Dobbiamo decidere adesso come realizzare una tale partizione. Seguendo la discussione precedente, la sola possibilità è utilizzare un elemento dell'array. Ciò solleva la questione se sia compatibile la partizione dell'array con lo scambio dei dati su lunghe distanze nei primi passi del sort. Per tentare di rispondere alla domanda, sceglieremo per prova il 18 come valore di partizione, essendo il quarto degli otto valori. Portando avanti questa scelta, la prima cosa che scopriamo è che il 20 dovrebbe essere nella partizione degli elementi più grandi invece che in quella degli elementi più piccoli. Ossia:

20	35	18	8	14	41	3	39
↑				↑			

Se 20 è nella partizione sbagliata significa che c'è un elemento piccolo collocato erroneamente nell'altra partizione. Per progredire nell'opera di partizione dell'array, occorre disporre 20 nella partizione degli elementi più grandi. Il solo modo soddisfacente per farlo è "fargli posto" nella partizione degli elementi più grandi, trovando in tale partizione un elemento "piccolo" da scambiare con lui. Il 3 in settima posizione è un candidato per questo trasferimento. Se ci muoviamo nell'array da destra alla ricerca degli elementi piccoli e da sinistra dalla ricerca degli elementi grandi (dove grande e piccolo sono relativi a 18), avremo un modo per dividere l'array e contemporaneamente scambiare i dati su lunghe distanze, come avevamo deciso di fare in origine. A questo punto possiamo riconoscere che il primo metodo di partizione considerato nella discussione dell'algoritmo 4.6 può essere facilmente adattato alla situazione presente. Riassumendo, i passi fondamentali nell'algoritmo di partizione erano:

1. Estendere le due partizioni verso l'interno finché non si incontra una coppia non correttamente disposta.
 2. Finché le due partizioni non si sono incrociate
 - (a) scambiare le coppie non correttamente disposte;
 - (b) estendere ancora le due partizioni verso l'interno finché non si incontra un'altra coppia non correttamente disposta.

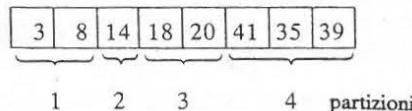
Applicando questa idea ai dati d'esempio otteniamo:



Il metodo di partizione discusso nell'algoritmo 4.6 può richiedere fino a $n + 2$ confronti per dividere n elementi. Ciò può essere migliorato sostituendo il test del loop $i \leq j$ con il test $i < j - 1$; per questa nuova implementazione, quando la terminazione si verifica per $i = j - 1$, è necessario allora eseguire uno scambio supplementare al di fuori del loop.

Gli elementi più piccoli sono a questo punto completamente separati da quelli più grandi. Come possiamo sfruttare questo risultato? Abbiamo adesso *due* partizioni i cui valori possono essere tratti indipendentemente: ossia, se ordiniamo i primi tre valori e quindi gli altri cinque, alla fine l'intero array sarà ordinato. Nell'ordinare queste due partizioni, possiamo cercare ancora di aderire all'idea di scambiare i dati sulle massime distanze; poiché le due partizioni sono indipendenti, possiamo trattarle come due problemi più piccoli, che possono essere risolti in maniera simile al problema originale. Benché per piccoli insiemi di dati (come è il caso in esame) il metodo appaia inutilmente complicato, possiamo immaginare che per insiemi molto più numerosi esso fornisca una maniera efficiente per trasferire dati su lunghe distanze.

Ciò che adesso dobbiamo stabilire è come l'idea della partizione e del trasferimento di dati su lunghe distanze si inseriscono nel nostro piano originale di ordinamento. Per indagare tale rapporto, possiamo applicare nuovamente il procedimento di partizione alle due partizioni derivate dall'insieme originale dei dati. Per rendere più facile la comprensione del meccanismo, possiamo considerare una scelta di elementi discriminatori che consenta di dividere le due partizioni a metà. (Ossia, possiamo scegliere 8 per la partizione di sinistra e 35 per quella di destra). Quando il meccanismo di partizione è applicato a queste partizioni, con 8 e 35 quali elementi discriminatori, otteniamo:



Si osservi da questo esempio che andrà eseguito uno scambio quando si incontrano elementi maggiori o uguali al valore di partizione. Torneremo su questo più avanti.

Esaminando i risultati di questi ultimi passi di partizione, vediamo che adesso ci sono quattro partizioni. Notiamo che esse sono *parzialmente ordinate* (cioè gli elementi della prima partizione sono più piccoli di quelli della seconda, che a loro volta sono più piccoli di quelli della terza, e così via).

Eseguendo una partizione banale sulle partizioni di dimensione 2 e 3, l'intero array risulterà ordinato. Ciò che abbiamo appreso da questa considerazione è che la partizione ripetuta ci fornisce un mezzo per muovere gli elementi su lunghe distanze e che, al tempo stesso, quando il processo è condotto a termine, esso produce l'ordinamento completo dei dati. Abbiamo pertanto scoperto che la partizione ripetuta può essere usata per il sort. Il nostro compito è adesso quello di ricavare la maniera di implementare questo meccanismo; prima di farlo, possiamo riassumere i progressi del nostro algoritmo. Il meccanismo di base è adesso:

finché tutte le partizioni non hanno dimensione uno

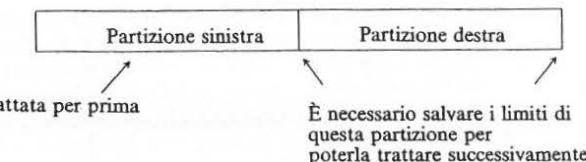
- scegliere la prossima partizione da elaborare;
- selezionare dalla partizione corrente un nuovo valore di discriminazione;
- suddividere la partizione corrente in due insiemi più piccoli parzialmente ordinati.

Adesso ci sono due punti fondamentali da risolvere. Innanzi tutto, dobbiamo trovare un metodo conveniente per scegliere i valori di partizione. La discussione precedente ci porta a concludere che il massimo che sappiamo fare è "tentare" un valore conveniente di partizione; mediamente, se i dati sono distribuiti casualmente, possiamo attenderci che un simile approccio dia un risultato ragionevole. La strategia più semplice per scegliere il valore discriminante è prendere sempre il primo valore della partizione. Un breve esame dimostra che

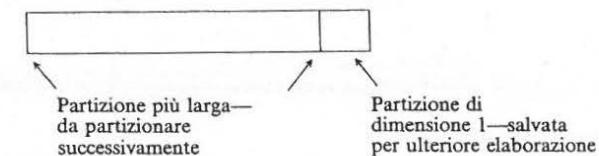
questa scelta sarebbe infelice quando i dati sono già ordinati o disposti in ordine inverso: il modo migliore per tener conto di questi casi è utilizzare il valore *a metà* della partizione; questa strategia non modifica le prestazioni per dati casuali, ma le migliora decisamente nei casi speciali più comuni che abbiamo già citato. Se *upper* e *lower* indicano i limiti dell'array per una data partizione, allora l'indice *middle* per l'elemento centrale può essere calcolato facendo la media dei due limiti (ved. algoritmo 5.7).

$$\text{middle} = (\text{lower} + \text{upper}) \text{ div } 2$$

Nell'esempio precedente abbiamo visto come i dati originari siano stati divisi in due partizioni che sarebbero state trattate in seguito. Dato che può essere considerata solo una partizione alla volta, per partizioni successive, sarà necessario salvare le informazioni sui limiti dell'altra in modo da poter essere trattata successivamente. Ad esempio:



Un'ulteriore considerazione sul meccanismo di partizionamento ripetuto suggerisce che dovremo salvare i limiti di un numero considerevole (anziché uno solo) di partizioni man mano che il processo si snoda in partizioni sempre più piccole: la maniera più semplice è memorizzarli in un array (che in realtà funziona come una pila - ved. Capitolo 7). Ciò fa sorgere la domanda sulla quantità di memoria aggiuntiva da allocare per questo array. Per poter rispondere dobbiamo considerare alla peggior situazione possibile di partizione. Nel caso peggiore, ci potremmo trovare con un unico elemento ogni volta nella partizione più piccola. Questa situazione porterebbe alla necessità di memorizzare i limiti di $n - 1$ partizioni durante l'ordinamento di n elementi. Il relativo costo sarebbe di $2(n - 1)$ locazioni dell'array. Per assicurare che il nostro algoritmo tratti correttamente tutti i casi, sembrerebbe necessario includere $2(n-1)$ locazioni supplementari, incremento costoso in termini di memoria; vogliamo quindi cercare di trovare un modo per ridurlo. Per fare questo faremo un accurato esame alla situazione peggiore in cui avevamo supposto che i limiti delle partizioni unitarie dovessero essere memorizzati ogni volta.



Il problema sorge in questo caso perché noi (inconsciamente) scegliamo sempre di trattare per prima la partizione maggiore. Se invece in questo caso potessimo "controllare" la situazione in modo da trattare prima la partizione minore, allora sarebbero necessarie solo due locazioni per salvare i limiti della partizione maggiore per il successivo reperimento. Un tale miglioramento avviene poiché la partizione di un insieme unitario non porta ad ulteriori suddivisioni prima che venga considerata quella maggiore. Con questa strategia, la situazione precedente non è più quella peggiore. Il caso peggiore accade ora quando viene trattata la più grande tra le partizioni più piccole ad ogni stadio. La partizione più grande tra le più piccole ha dimensione pari a metà di quella corrente. Se inizialmente abbiamo un array di dimensione n da ordinare, può essere dimezzato $\log n$ volte prima che venga raggiunta una partizione unitaria. Quindi, se adottiamo la strategia di trattare per prima la partizione più piccola, sarà necessaria una quantità di memoria addizionale proporzionale a solo $\log n$, cosa accettabile anche per n molto grande. (Cioè avremo bisogno solo di circa 20 locazioni di memoria per trattare un array di 2000 elementi.) Per decidere quale sia la partizione maggiore è necessario un semplice test che determini il punto d'incontro delle partizioni relativamente alla metà, ovvero: se le partizioni si incontrano a sinistra della metà allora

"considera la partizione di sinistra e salva i limiti di quella di destra"
altrimenti

"considera la partizione di destra e salva i limiti di sinistra"

Possiamo ancora riassumere i passi del nostro algoritmo aggiungendo i dettagli dei perfezionamenti appena fatti.

finché tutte le partizioni non avranno raggiunto la dimensione unitaria

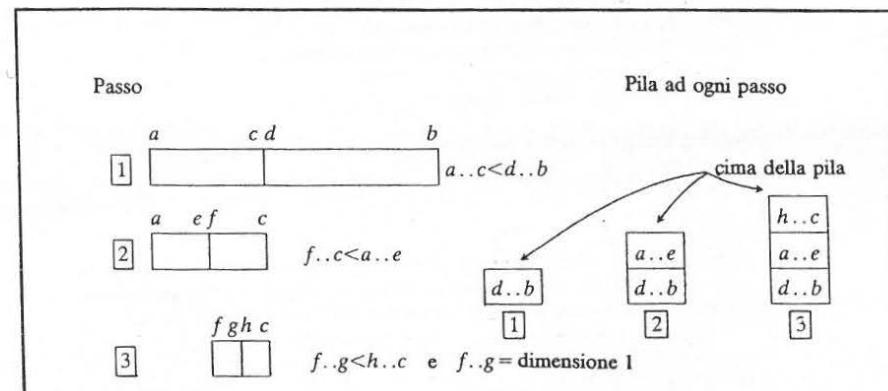
- scegli la partizione più piccola da trattare per prima;
- seleziona l'elemento al centro della partizione come valore discriminante;
- dividi la partizione corrente in due insiemi parzialmente ordinati;
- salva la partizione maggiore del caso (c) per un successivo trattamento.

A questo punto dello svolgimento non è ancora molto chiaro come termini l'algoritmo. Nella discussione precedente, abbiamo stabilito che quando viene applicato ripetutivamente il processo di partizione, arriveremo alla fine a dover partizionare un solo elemento. In accordo con questo processo di riduzione della dimensione della partizione ad uno, la maggiore delle due partizioni deve essere "messa da parte" per essere trattata successivamente. La Figura 5.7 illustra il meccanismo di partizione e l'"impilamento" della partizione maggiore in ogni caso.

Una volta raggiunta una partizione unitaria, non abbiamo altra alternativa che iniziare il processo di partizione di quella memorizzata più di recente: per questo occorre togliere i limiti dalla cima della pila.

Dall'esempio, sarà richiesto che venga partizionato per primo il segmento " $h..c$ ". Il nostro compito è di assicurare che vengano ridotte a dimensione unitaria *tutte* le partizioni; una volta assicurato ciò, non ci saranno limiti lasciati sulla pila per ulteriori trattamenti. Possiamo utilizzare questa condizione per terminare il meccanismo di partizionamento ripetuto. Il meccanismo può iniziare con i limiti dell'intero array memorizzati nella pila, si procede quindi con il processo di partizione finché la pila non sarà vuota, cosa che può essere segnalata utilizzando un puntatore alla "cima della pila". Il valore zero dell'indice dell'array indicherà che tutte le partizioni avranno raggiunto la dimensione unitaria. Vedremo più avanti (algoritmo 8.2) nell'implementazione ricorsiva del quicksort come il Pascal implementa automaticamente ed implicitamente il meccanismo della pila.

Fig. 5.7.
Meccanismo a pila
per salvare i limiti
della partizione
per un ulteriore
trattamento.



Abbiamo esplorato sufficientemente il problema da essere in grado di fornire una dettagliata descrizione dell'algoritmo. I dettagli dell'algoritmo di partizione provengono dall'algoritmo 4.6.

Descrizione dell'algoritmo

- Definisci l'array $a[1..n]$ da ordinare.
- Poni i limiti superiore ed inferiore dell'array sulla pila ed inizializza il puntatore alla cima della pila.
- Finchè la pila non è vuota
 - togli i limiti superiore ed inferiore del segmento di array dalla cima della pila;
 - finchè il corrente segmento non raggiunge la dimensione unitaria

- (b.1) seleziona l'elemento centrale del segmento di array dalla pila;
- (b.2) partitiona il segmento corrente in due rispettando il valore centrale; [*]
- (b.3) salva i limiti della partizione maggiore nella pila e comincia il trattamento della partizione minore se contiene più di un elemento.

Implementazione in Pascal

```

procedure quicksort(var a: nelements; n,stacksize: integer);
var left {upper limit of left partition a[1..left]},
    right {lower limit of right partition a[right..n]},
    newleft {upper limit of extended left partition},
    newright {lower limit of extended right partition},
    middle {middle index of current partition},
    mguess {current guess at median},
    temp {temporary variable used for exchange},
    stacktop {current top of stack}: integer;
    stack: array[1..100] of integer;

begin {sort by repeated partitioning of smallest available partition}
  {assert: n > 0}
  stacktop := 2;
  stack[1] := 1;
  stack[2] := n;
  {invariant: 1 = < i, p & I, s = < n & for all unsorted partitions defined on
  stack for which j < k we have all a[1..j] = < all a[k..l] & for all ordered
  partitions currently in array for which p < r we have all a[p..q]
  = < a[r..s] & stacktop >= 0}

  while stacktop > 0 do
    begin {partition until all partitions reduced to size one}
      right := stack[stacktop];
      left := stack[stacktop - 1];
      stacktop := stacktop - 2;
      {invariant: all a[stack[stacktop - 1]..left - 1] = < all a[left..right]
      = < all a[right + 1..stack[stacktop]] & left = < right & limits for first
      and third positions all stacked}

      while left < right do
        begin {partition the current segment according to mguess}
          newleft := left;
          newright := right;
          middle := (left + right) div 2;
          mguess := a[middle];

```

[*] Nota: Un trattamento dettagliato del meccanismo di partizione è stato fornito precedentemente nella descrizione dell'algoritmo 4.6. Per ragioni di efficienza la procedura di partizione è stata inclusa direttamente nell'algoritmo di quicksort invece che attraverso una chiamata di procedura.

```

while a[newleft] < mguess do newleft := newleft + 1;
while mguess < a[newright] do newright := newright - 1;
{invariant: all a[left..newleft - 1] = < all a[newright + 1..right]}
while newleft < newright - 1 do
  begin {exchange wrongly partitioned pair and then extend
  both segments}
    temp := a[newleft];
    a[newleft] := a[newright];
    a[newright] := temp;
    newleft := newleft + 1;
    newright := newright - 1;
    while a[newleft] < mguess do newleft := newleft + 1;
    while mguess < a[newright] do newright := newright
  end;
  if newleft <= newright then
    begin {allow for case where two partitions do not
    cross-over}
      if newleft < newright then
        begin {exchange}
          temp := a[newleft];
          a[newleft] := a[newright];
          a[newright] := temp
        end;
        newleft := newleft + 1;
        newright := newright - 1
      end;
      if newright < middle then
        begin {set up to process smaller left partition next}
          stack[stacktop + 1] := newleft;
          stacktop := stacktop + 2;
          stack[stacktop] := right;
          right := newright
        end
      else
        begin {set up to process smaller right partition next}
          stack[stacktop + 1] := left;
          stacktop := stacktop + 2;
          stack[stacktop] := newright;
          left := newleft
        end
    end
  end;
  {assert: a[1..n] sorted in non-descending order & a permutation of
  the original data set}
end

```

Note di progetto

1. L'algoritmo di quicksort è considerato probabilmente il più efficiente degli algoritmi di ordinamento interno. La sua efficienza consiste nel rimpicciolire il numero di operazioni sui dati. Una dettagliata analisi dell'algoritmo mostra che predominano i passi di

confronto. In media, per un array di n elementi, viene fatto un numero di confronti dell'ordine di $n \log_2 n$. Il caso peggiore, in cui viene sempre scelto l'elemento maggiore (o minore) come discriminante, prevede un numero di confronti dell'ordine di n^2 .

2. Per la parte di partizionamento dell'implementazione, dopo ogni iterazione abbiamo che $a[left..nleft - 1] \leq mguess$ e $a[nright + 1..right] \geq mguess$. Ad ogni passaggio attraverso il ciclo while $left < right - 1$ do tutti gli elementi dell'intervallo $a[1..left - 1]$ sono inori o uguali a quelli dell'intervallo $a[left..n]$. Similmente, tutti gli elementi nell'intervallo $a[right + 1..n]$ sono maggiori o uguali a tutti gli elementi dell'intervallo $a[1..right]$ e $left \leq right + 1$ e $right \geq left - 1$. Dopo ogni iterazione del ciclo di partizione sia $nleft$ che $nright$ saranno cambiate di almeno un valore e quindi questo ciclo terminerà. Ne consegue che terminerà anche while $left < right$ do, poiché $left$ e $right$ sono fissate da $nleft$ e $nright$. La terminazione del ciclo più esterno che dipende dallo svuotamento della pila è più difficile da capire. Senza formalità, per spiegare come avviene la terminazione, possiamo usare il seguente argomento. Un ripetuto partizionamento si risolve in limiti delle partizioni sempre più piccole che vengono posti sulla pila finché non si raggiunge una partizione di dimensione unitaria. A questo punto cessa il partizionamento e viene trattata la più piccola partizione disponibile lasciata sulla pila. Poiché tutte le partizioni considerate vengono alla fine ridotte a dimensione uno e ci sono al massimo solo n di tali partizioni, possiamo aspettarci che l'operazione di rimozione dalla pila garantirà la terminazione dell'algoritmo.
3. L'algoritmo di quicksort possiede un certo numero di attraenti attributi. Ha il vantaggio rispetto alla maggior parte degli altri $O(n \log_2 n)$ algoritmi di spostare gli elementi solo quando assolutamente necessario. (Cioè, si può dire che raggiunga il suo obiettivo applicando il principio di minimo sforzo-potente ed elegante strategia). Ciò può essere importante nel caso in cui debbano essere spostate grandi quantità di dati. (In alcuni casi ciò può essere evitato scambiando i puntatori anziché i dati).
4. L'algoritmo possiede la caratteristica di scambiare i dati su lunghe distanze nel processo di ordinamento.
5. L'algoritmo di quicksort non è superiore ad alcuni degli altri $O(n^2)$ algoritmi quando deve ordinare piccoli insieme di dati (circa = 12). L'algoritmo può quindi essere accelerato incorporando nel meccanismo un ordinamento per inserzione. Ogni volta che l'algoritmo riduce una partizione a meno di 12 elementi essa può essere ordinata per inserzione invece di essere ulteriormente partizionata.
6. L'algoritmo di quicksort è per sua natura ricorsivo. Come vedremo nel Capitolo 8 ha un'implementazione ricorsiva molto semplice.
7. Bisogna notare che quando tutti gli elementi da ordinare sono uguali, quicksort compie un considerevole numero di scambi non necessari.

Applicazioni

Ordinamento interno per insiemi con un gran numero di dati.

Problemi supplementari

- 5.6.1 Il numero di confronti richiesti dal quicksort può essere ridotto di una piccola percentuale utilizzando il valor medio di tre elementi ogni volta che venga richiesta una scelta del valore di mezzo. Un modo semplice per fare ciò è di scegliere sempre il valor medio del primo, medio e valore finale nel segmento di array da partizionare. Implementare questo raffinamento e confrontarlo con la versione originale usando il numero di confronti come misura.
- 5.6.2 È stato accenato precedentemente (in nota 5) che quicksort può essere accelerato usando l'ordinamento per inserzione ogni volta debba essere ordinato un segmento con meno di dodici elementi. Realizzare questo suggerimento.
- 5.6.3 Il suggerimento del problema precedente può risultare al di sopra per la chiamata alla procedura di ordinamento per inserzione. Un approccio alternativo e più efficiente è di posticipare l'ordinamento per inserzione finchè *tutte* le partizioni non siano state ridotte e progettare tests adatti per il confronto con l'implementazione 5.6.2.
- 5.6.4 Una significativa riduzione dei confronti può essere fatta ponendo inizialmente il valore discriminante selezionato nella prima locazione dell'intervallo da partizionare e procedendo con la partizione dalla seconda posizione. Implementare tale modifica.

ALGORITMO 5.7 RICERCA BINARIA

Problema

Dato un elemento x ed un insieme di dati in ordine numerico strettamente crescente, stabilire se x appartiene all'insieme.

Sviluppo dell'algoritmo

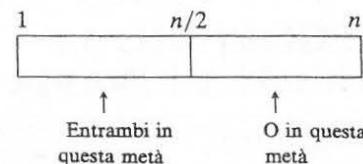
Il problema della ricerca in una lista ordinata, come un dizionario o un elenco telefonico, si presenta frequentemente in informatica.

Prima di cercare la soluzione di tale problema con il calcolatore, vediamo come esaminiamo una lista ordinata senza ricorrere ad esso. A questo proposito, consideriamo come utilizziamo un elenco telefonico per cercare il numero di qualcuno (per es. il numero telefonico del Signor J.K. Smith). Un modo per trovare questo numero sarebbe iniziare da pagina 1 dell'elenco e proseguire pagina per pagina finché non viene trovato. L'esperienza ci insegna che tale metodo è troppo lento e che invece noi usiamo un approccio completamente diverso per risolvere il problema: infatti noi non incontriamo alcuna difficoltà a localizzare velocemente il numero di telefono di una persona in un elenco con più di un milione di nomi. Ma come facciamo?

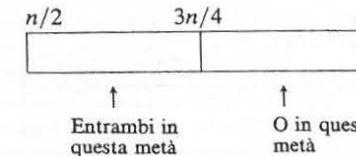
Nella ricerca del numero del Signor Smith noi non cominciamo certamente dall'inizio dell'elenco: apriamo invece l'elenco ad una pagina posta ad un po' meno di un terzo dalla fine. Lanciamo quindi uno sguardo al nome in cima alla pagina e decidiamo se proseguire o meno in un'altra pagina. Dalla posizione corrente, in dipendenza del nome trovato, applichiamo di nuovo la stessa strategia, probabilmente in scala minore. Con questi movimenti in avanti e all'indietro, troviamo velocemente la pagina contenente il nome cercato. Con questo procedimento, noi scartiamo, dopo alcune considerazioni ed il più in fretta possibile, molte pagine dell'elenco: applichiamo, infatti, quello che viene comunemente chiamato metodo di ricerca per interpolazione. È possibile formulare una strategia di questo tipo come algoritmo, ma per ora concentriamoci su un approccio un po' diverso, ma più generale, in quanto produce un risultato molto buono indipendentemente dalla distribuzione dei dati ordinati.

La strategia precedente sembra a prima vista assai difficile da formulare come algoritmo per il calcolatore. Quello che noi cerchiamo è un metodo che ci permetta di eliminare velocemente grandi quantità di dati e di arrivare sistematicamente nell'area cercata.

È facile vedere che in tutti i casi il valore dell'insieme che stiamo cercando è o nella prima metà della lista o nella seconda (può anche essere il valore centrale dell'insieme). Ad esempio:



Possiamo stabilire la metà che ci interessa confrontando il valore cercato con il valore centrale dell'insieme. Questo test eliminerà metà dei valori dell'insieme da ulteriori considerazioni. Il nostro problema è ora di dimensione pari alla metà di quello originario. Supponiamo di aver stabilito che il valore che stiamo cercando si trova nella seconda metà della lista (cioè in qualche posto tra l' $(n/2)$ -esimo e l' n -esimo valore).



Una volta ancora esso si trova o nella prima o nella seconda metà dell'insieme ridotto. Esaminando per primo il valore di posizione $n/2$, poi il valore di posizione $3n/4$, siamo stati in grado di eliminare da ulteriori considerazioni i $3/4$ dei valori dell'insieme iniziale in due soli confronti. Possiamo continuare ad applicare quest'idea di dimezzare ad ogni confronto il nostro insieme di dati, finché non incontreremo il valore cercato o stabiliremo che non è presente nell'insieme. Nella ricerca in un elenco telefonico, tutto ciò si realizza facilmente. Ci possiamo aspettare la medesima situazione da applicare al nostro algoritmo per il calcolatore.

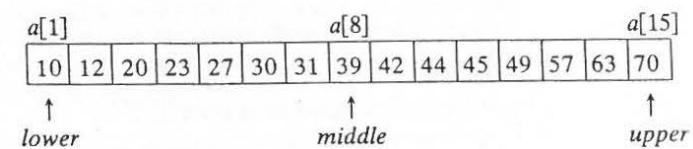
La strategia di ridurre a metà che abbiamo considerato, è uno dei metodi più largamente usati in informatica. È comunemente conosciuto come la strategia del "divide et impera". Il corrispondente metodo di ricerca, è conosciuto come algoritmo di ricerca binaria.

A questo stadio abbiamo la seguente strategia generale:

ripetitivamente

"esamina il valore centrale dei dati rimasti e sulla base di questo confronto elimina metà dell'insieme dei dati rimanenti".
finché (il valore è stato trovato o non è presente).

Consideriamo ora un esempio specifico per cercare i dettagli dell'algoritmo necessari per l'implementazione.



Supponiamo di dover esaminare un array di 15 elementi ordinati per stabilire se è presente il valore $x = 44$ e, se così, la posizione che occupa nell'array contiene un numero pari di elementi non avrà esattamente un valore centrale. Per trovare il valore di mezzo di un array di n elementi, possiamo usare:

$$middle := n \div 2$$

per $n = 15$ questa produce $middle = 7$ che non è esattamente il centro. Se aggiungiamo uno prima di compiere la divisione, troviamo:

$$middle := (n + 1) \div 2 = 8 \text{ (il centro)}$$

Ciò che fornisce $a[middle] = a[8] = 39$. Dato che il valore che stiamo cercando (44) è maggiore di 39, si dovrà trovare nell'intervallo $a[9]..a[15]$, ovvero $a[9]$ diviene il limite inferiore dell'insieme in cui potrebbe essere 44. Cioè, $lower := middle + 1$. Abbiamo allora:

$a[9]$	$a[12]$	$a[15]$
42	44	45
49	57	63

↑ ↑ ↑
lower middle upper

Per calcolare l'indice centrale questa volta dobbiamo sottrarre 8 da 15 per trovare il numero di valori rimasti (7). Possiamo poi dividerlo per 2 per trovare 3 che, aggiunto a 9 dà 12. Questo procedimento è un po' complicato. Esaminando i limiti inferiore e superiore vediamo che se li sommiamo e li dividiamo per 2, troviamo il valore 12. Controlli su altri valori, confermano che questo metodo per il calcolo del valore centrale funziona nel caso generale, cioè:

$$middle := (lower + upper) \text{ div } 2$$

Quando viene esaminato $a[12]$, si stabilisce che 55 è minore di $a[12]$; ne consegue che 44 (se è presente) deve trovarsi nell'intervallo $a[9]..a[11]$, perciò il limite superiore diventa inferiore di uno rispetto al valore centrale, cioè

$$upper := middle - 1$$

Abbiamo allora:

$a[9]$	$a[11]$
42	44
45	

↑ ↑ ↑
lower middle upper

Da questo vediamo che ad ogni confronto o aumenta il limite inferiore o diminuisce quello superiore.

Al confronto successivo, troviamo il valore cercato e la sua posizione nell'array, cioè in soli 3 confronti abbiamo localizzato il valore voluto. Si può vedere che sarebbero stati necessari altri confronti se il valore cercato fosse stato 42 o 45.

Il nostro algoritmo deve trattare la situazione in cui il valore cercato non sia presente nell'array. Quando è presente l'algoritmo termina quando il valore centrale corrente è uguale al valore cercato. Chiaramente questo test non può mai essere vero se il valore cercato non è presente. Devono essere aggiunte altre condizioni per garantire che l'algoritmo termini.

Per indagare sulla terminazione quando l'elemento non è presente, vediamo cosa succede quando cerchiamo 43, anziché 44. La procedura avanza come prima finché non troviamo la seguente configurazione:

$a[9]$	$a[11]$
42	44
45	

↑ ↑ ↑
lower middle upper

A questo punto troviamo che 43 è minore del valore centrale ed abbiamo:

$$upper := middle - 1$$

Questo porta alla situazione in cui $lower = middle$ (sono tutti = 9).

$a[9]$	$a[11]$
42	44
45	

↑
lower middle upper

(cioè $lower = 9$
 $middle = 9$
 $upper = 9$)

Il confronto successivo di 43 con $a[middle] = a[9]$ indica che il valore cercato è al di sopra del valore centrale. Troviamo quindi:

$$lower := middle + 1 = 10$$

Ora abbiamo $lower = 10$ e $upper = 9$, cioè i limiti inferiore e superiore si sono sorpassati. Un ulteriore esame alla ricerca di 45 indica che ancora una volta $lower$ diventa maggiore di $upper$. Quando il valore cercato è presente, può accadere questa situazione, ma solo nel momento in cui esso viene trovato. Poiché tutte le ricerche infruttuose passano sicuramente attraverso lo stadio in cui $upper = lower = middle$ (per il modo in cui viene calcolato il valore centrale), possiamo usare la condizione:

$$lower > upper$$

unitamente con il test di uguaglianza tra il valore dell'array ed il valore cercato per terminare l'algoritmo. Prima di terminare questa dissertazione, dobbiamo essere sicuri che l'algoritmo termina quando il valore cercato è minore del primo valore $a[1]$ o maggiore dell'ultimo $a[n]$.

Un ulteriore controllo per il caso particolare in cui l'array sia di un solo elemento, conferma che l'algoritmo funziona correttamente e termina come richiesto. A questo punto abbiamo il seguente algoritmo:

1. Definisci un array ordinato di dimensione n ed il valore cercato x .
2. Definisci l'insieme ordinato dei dati $a[1..n]$.
3. Assegna i limiti *lower* e *upper*.
4. Ripetitivamente
 - (a) calcola la posizione centrale del rimanente array;
 - (b) se il valore cercato è maggiore di quello centrale allora assegna al limite *lower*, $middle + 1$, altrimenti assegna al limite *upper*, $middle - 1$
 finché non è stato trovato il valore cercato o *lower* diventa maggiore di *upper*.
5. Modifica di conseguenza *found*.

Implementazione in Pascal

```

procedure binsch (var a:nelements; n,x:integer; var middle:integer;
  var found:boolean);
var lower, {lower limit}
  upper {upper limit}:integer;
begin
  lower := 1;
  upper := n;
  repeat
    middle := (lower + upper) div 2;
    if x > a[middle] then
      lower := middle + 1
    else
      upper := middle - 1
    until (a[middle]=x) or (lower>upper);
    found := (a[middle]=x)
end
  
```

La condizione di terminazione dell'algoritmo precedente è un po' rozza. È difficile, inoltre, provarne la correttezza. È quindi utile cercare se esiste un'implementazione più semplice. In qualsiasi ricerca, il test se il valore centrale corrente ($a[middle]$) è uguale al valore cercato si applica solo dentro il passo in cui altrimenti l'algoritmo terminerebbe (per spiegazioni vedere l'albero binario delle decisioni). Questo significa che se l'algoritmo può essere formulato in modo da terminare con una sola condizione, la soluzione sarà più elegante ed efficiente.

Un esame approfondito dell'algoritmo originario indica che non possiamo semplicemente togliere il test $a[middle] = x$ perché produrrebbe delle situazioni in cui l'algoritmo non terminerebbe correttamente. Il problema sorge perché è possibile superare il valore cercato.

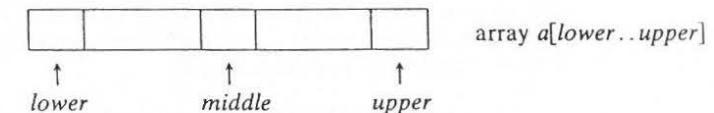
Per ottenere una migliore soluzione del problema, dobbiamo quindi conservare delle condizioni che prevengano il sorpasso. Un modo può

essere quello di assicurare che si dirigano verso la posizione di terminazione in modo da non scavalcarsi e da non superare la condizione di uguaglianza.

In altre parole, se x è presente nell'array, vogliamo trovare la seguente condizione dopo ogni iterazione:

$$a[lower] \leq x \leq a[upper]$$

Ne segue che *lower* e *upper* dovranno essere modificati in modo da garantire questa condizione se x è presente. Se operiamo in questo modo, saremo in grado di trovare un'adatta condizione di terminazione coinvolgendo solo *lower* e *upper*. La configurazione iniziale sarà:



Iniziando con:

$$middle := (lower + upper) \text{ div } 2$$

possiamo introdurre il seguente test condizionale nello sforzo di avvicinare *lower* e *upper* tra loro:

$$x > a[middle]$$

Se questa condizione è vera, allora x sarà nell'intervallo $a[middle + 1..upper]$, se è contenuto nell'array. Ne segue che con questa condizione vera, possiamo fare l'assegnamento:

$$lower := middle + 1$$

D'altra parte, se la condizione non è vera, (e quindi $x < a[middle]$) x deve cadere nell'intervallo $a[lower..middle]$, sempre se è contenuto nell'array. La variabile *upper* può quindi essere assegnata come:

$$upper := middle$$

(occorre notare che non viene fatta l'assegnazione $upper := middle - 1$ perché il test $x > a[middle]$ non è abbastanza potente da comprendere la possibilità che $a[middle] = x$.)

Abbiamo ora il problema di decidere come deve terminare questo meccanismo. Il meccanismo per cui *lower* e *upper* sono cambiati è tale che, se l'elemento x è presente uno dei due raggiunge per primo un elemento dell'array uguale ad x . Se *upper* scende ad un valore dell'array pari ad x , allora *lower* deve aumentare finché non raggiunge questo elemento dal basso. La situazione complementare sarà applicata se *lower* raggiunge per primo un elemento dell'array pari ad x . Queste due situazioni suggeriscono che la condizione di terminazione:

è probabilmente la più appropriata in presenza di x nell'array.

Il fatto che a $lower$ sia assegnato il valore ($middle + 1$) anziché solo $middle$, garantisce che $lower$ aumenterà dopo ogni passaggio attraverso il ciclo in cui viene riassegnato, è più sottile, perché viene sempre assegnato al valore corrente di $middle$. La troncatura causata dalla divisione intera:

$$middle := (lower + upper) \text{ div } 2$$

assicura che il valore $middle$ sia sempre inferiore al valore corrente di $upper$ (eccetto quando $lower = upper$). Ad esempio:

$$\begin{aligned} middle &= (2 + 4) \text{ div } 2 = 3 < 4 \\ middle &= (2 + 3) \text{ div } 2 = 2 < 3 \end{aligned}$$

Dato che $middle$ decresce in questo modo, ne consegue che $upper$ decrescerà sempre ogni volta che verrà riassegnato.

A questo punto possiamo controllare i casi particolare in cui x non è presente per assicurare la corretta terminazione dell'algoritmo. I principali casi particolari sono:

1. l'array contiene un solo elemento;
2. x è minore del primo elemento dell'array;
3. x è maggiore dell'ultimo elemento dell'array;
4. x appartiene all'intervallo $a[1..n]$, ma non è presente nell'array.

Un controllo di questi casi rivela che l'algoritmo termina correttamente quando x non è presente nell'array. Possiamo, quindi, fornire una dettagliata descrizione dell'algoritmo.

Descrizione dell'algoritmo

1. Definisci l'array $a[1..n]$ ed il valore cercato x .
2. Assegna ai limiti dell'array le due variabili $lower$ e $upper$.
3. Finché $lower < upper$
 - (a) calcola la posizione centrale del rimanente segmento di array da esaminare.
 - (b) se il valore cercato è maggiore del valore centrale corrente, allora
 - (b.1) aggiorna in conseguenza il limite inferiore altrimenti
 - (b'.1) aggiorna in conseguenza il limite superiore.
4. Se l'elemento dell'array in posizione $lower$ è uguale al valore cercato, allora
 - (a) comunica di averlo trovato altrimenti
 - (b) comunica di non averlo trovato.

Implementazione in Pascal

```
procedure binarysearch(var a: nelements; n,x: integer; var found: boolean);
var lower {lower limit of array segment still to be searched},
    upper {upper limit of array segment still to be searched},
    middle {middle of array segment still to be searched}: integer;
begin {binary searches array a[1..n] for element x}
  {assert: n>0^a[1..n] sorted in ascending order^exists k such that
  1=<k=<n ^x=a[k] if x is present}
  lower := 1;
  upper := n;
  {invariant: lower >= 1^upper = <n ^x in a[lower..upper] if present}
  while lower < upper do
    begin {increase lower and decrease upper keeping x in range if
    present}
      middle := (lower+upper) div 2;
      if x > a[middle] then
        lower := middle + 1
      else
        upper := middle
    end;
  {assert: lower = upper = k ^x = a[k] if x in a[1..n]}
  found := (a[lower]=x)
end
```

Note di progetto

1. L'algoritmo di ricerca binaria offre, in generale, un'alternativa più efficiente rispetto a quello di ricerca lineare. La sua esecuzione può essere capita più facilmente facendo riferimento ad un albero binario di ricerca. Per un array di 15 elementi, l'albero ha la forma mostrata in Fig. 5.8.
2. Ad ogni iterazione, se x è presente nell'array, $lower$ verrà incrementato e $upper$ decrementato in modo che la condizione:

$$a[lower] \leq x \leq a[upper] \text{ e } lower \leq upper$$

rimanga vera. La terminazione si avrà quando $lower = upper$ e, da qui, se x è presente nell'array, avremo:

$$a[lower] = x = a[upper]$$

3. Sono state fatte molte implementazioni dell'algoritmo di ricerca binaria (ved. ad esempio, B. Kidman, "Understanding the binary search", Aust. Comp. J., 13,7 - 12 (1981); N. Wirth, "Algorithms + Data Structures = Programs", Prentice Hall, 1976). La maggior parte di essi è o più complicata, o possiede condizioni di inizio e terminazione non usuali che li rendono di difficile comprensione.

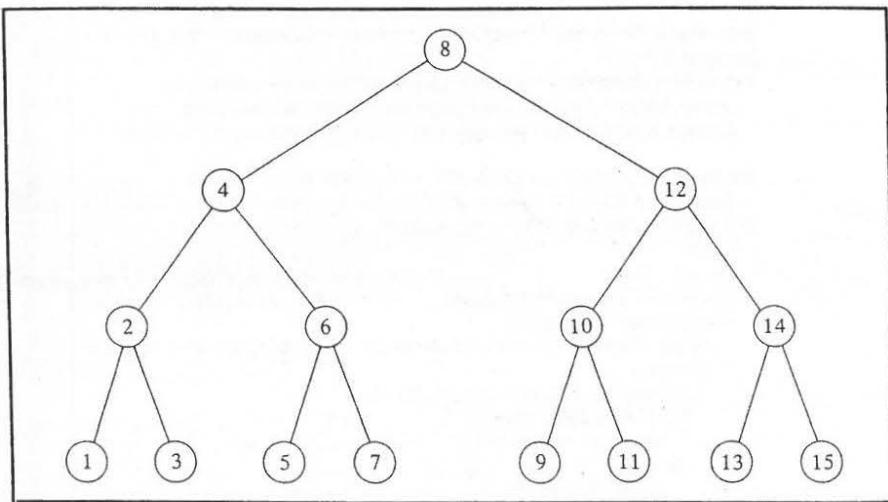


Fig. 5.8.
Albero binario di
decisione per
l'algoritmo di ricerca
binaria.

Problemi supplementari

5.7.1. Progettare ed implementare versioni di ricerca binaria che contengano le seguenti strutture di ciclo:

- (a) `while lower < upper + 1 do`
- (b) `while lower < upper - 1 do`

5.7.2. Implementare versioni di ricerca binaria che facciano le seguenti coppie di cambiamenti a *lower* e *upper*:

- (a) $\left. \begin{array}{l} lower := middle + 1 \\ upper := middle - 1 \end{array} \right\}$
- (b) $\left. \begin{array}{l} lower := middle \\ upper := middle \end{array} \right\}$

5.7.3 Implementare un algoritmo di ricerca binaria che calcoli il valore centrale usando $(lower + upper)/2$.

5.7.4 Una variazione dell'algoritmo binario non centra il suo svolgimento attorno ai limiti *lower* e *upper*. Sono invece usati due parametri alternativi, uno che punta al centro del segmento di array ancora da esaminare, ed un secondo per segnare l'ampiezza di metà di tale segmento. Un algoritmo di ricerca binaria che utilizzi questo approccio viene detto di ricerca binaria uniforme. Progettare ed implementare un algoritmo di ricerca binaria uniforme.

5.7.5 Sviluppare un algoritmo che utilizzi un generatore di numeri casuali che generi sempre numeri compresi nell'intervallo *lower..upper*. In ogni istanza il numero casuale generato potrebbe assumere il ruolo di *middle* degli algoritmi precedenti. Confrontare le esecuzioni di questo algoritmo con l'algoritmo di ricerca binaria in termini del numero di confronti fatti.

5.7.6 Un metodo di ricerca che utilizza l'interpolazione lineare può fornire un veloce rimedio quando l'insieme di dati ordinati è distribuito abbastanza uniformemente nell'intervallo dei possibili valori. Con questo metodo, la grandezza del valore *x* cercato è usata per determinare quale elemento dell'array debba essere successivamente confrontato con *x*. Quando *lower* e *upper* sono i limiti correnti del segmento di array possiamo successivamente esaminare l'elemento che sia approssimativamente ad una distanza.

$$\left(\frac{x - a[lower]}{a[upper] - a[lower]} \right) \times (upper - lower - 1)$$

oltre il valore di corrente di *lower*. Implementare un algoritmo di ricerca per interpolazione e confrontare la sua esecuzione con la log₂*n* della ricerca binaria. Usare un generatore uniforme di numeri casuali per costruire l'array *a*. Notare che tale array deve essere ordinato.

ALGORITMO 5.8 RICERCA CALCOLATA (HASH)

Problema

Progettare ed implementare un algoritmo di ricerca calcolata.

Sviluppo dell'algoritmo

Per molte persone è una sorpresa scoprire per la prima volta che esiste un metodo di ricerca in media considerevolmente più veloce della ricerca binaria e che spesso deve esaminare solo uno o due elementi prima di concludersi con successo. Poiché ci hanno detto che un tale metodo esiste, vediamo di scoprire questo algoritmo di ricerca apparentemente misterioso e rapidissimo.

Per prima cosa prendiamo un insieme specifico di dati e vediamo cosa possiamo fare con esso. Per semplificare il problema concentriamoci per primo sulla ricerca della posizione di un numero in un array.

$a[1]$	$a[8]$	$a[15]$
10 12 20 23 27 30 31 39 42 44 45 49 53 57 60		

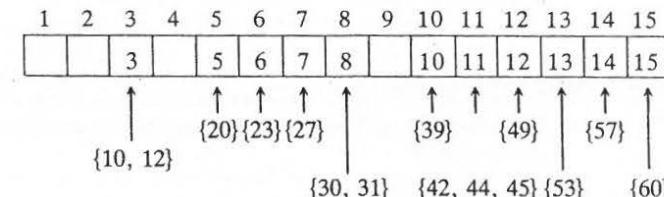
Supponiamo di voler trovare il 44 (di solito dovremmo cercare delle informazioni associate al 44). La nostra esperienza precedente con la ricerca binaria ha mostrato come ci si può avvantaggiare dell'ordinamento di un array di dimensione n per localizzare un elemento in circa $\log_2 n$ passi. L'utilizzazione dell'ordinamento non sembra un criterio abbastanza affidabile per localizzare elementi più velocemente. L'aspetto quasi incredibile del metodo più veloce è l'asserzione "deve di solito esaminare solo uno, due o tre elementi prima di terminare con successo". Per cercare di fare progressi, focalizziamoci sul caso limite che esamina solo un elemento prima di terminare. Se si prende in considerazione questo procedimento, sembra impossibile trovare un elemento tra migliaia la prima volta.

Per trovare il 44 nel nostro esempio dovremmo andare "per magia". In locazione 10 al primo passo. Nulla sta ad indicare che il 44 sia memorizzato in posizione 10: tutto ciò che conosciamo è la sua "grandezza". Ci sembra di essere arrivati ad un punto morto! Come può la grandezza del 44 aiutarci in alcun modo nella localizzazione della sua posizione in appena un passo? Riflettendoci un attimo, arriviamo alla conclusione che sarebbe possibile trovare il numero 44 in un passo, solo se si trovasse nella 44-esima posizione dell'array. Sebbene l'idea di memorizzare ogni numero nella posizione dell'array definita dalla sua grandezza ci permetterebbe di localizzarlo o di scoprirne l'assenza in un passo non sembra in generale molto pratica. Supponiamo ad esempio di voler memorizzare e trovare rapidamente un insieme di 15 numeri di telefono, ognuno di 7 cifre, come 4971667. Con tale procedimento, il numero dovrebbe essere memorizzato in posizione 4971667. Non vale la pena di costruire un array con un tale numero di locazioni solo per la memorizzazione e la ricerca di 15 numeri. Prima di abbandonare l'idea di cercare un metodo migliore, diamo un ultimo sguardo al "progresso" fatto. Per facilitare le cose, ritorniamo all'esempio precedente. Abbiamo un insieme di 15 numeri tra 10 e 60. Potremmo memorizzarli e cercarli in un array di 60 celle e recuperarli in un solo passo: in generale ciò significa che deve essere utilizzato troppo spazio. Sorge allora la domanda di come poter applicare gli stessi principi per il recupero usando uno spazio di memoria più piccolo.

Una risposta all'ultima domanda potrebbe essere procedere alla normalizzazione dei dati, ovvero applicare una trasformazione ad ogni numero prima della sua ricerca (cioè 60 potrebbe essere trasformato in modo da diventare 15, 20 in 5 e così via, attraverso una divisione intera per 4 ed un arrotondamento all'intero più vicino). Quando applichiamo questa trasformazione al nostro insieme campione di dati ed arrotondiamo all'intero più vicino, scopriamo che alcuni valori condividono-

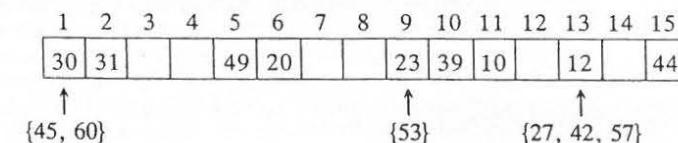
no lo stesso indice nell'intervallo tra 1 e 15, mentre altri indici di questo intervallo non vengono usati. Vediamo quindi che riducendo la dimensione dell'array da "grandezza" (60) a "occupazione" (15), abbiamo introdotto la possibilità dell'occupazione multipla. Chiaramente, meno riduciamo l'estensione, minore sarà il rischio di occupazioni multiple o "collisioni", come vengono chiamate solitamente.

Il progetto che abbiamo fatto sembra risolvere solo metà del nostro problema. L'insieme normalizzato dei valori sarebbe:



È facile immaginare situazioni in cui questo schema normalizzato introdurrà più severe situazioni di collisione (supponiamo ad esempio che il valore finale sia molto più grande di tutti gli altri dell'insieme). Queste osservazioni suggeriscono che l'approccio di normalizzazione non è probabilmente il migliore.

Le considerazioni più recenti ci fanno chiedere se esista una trasformazione alternativa di prericerca da poter applicare ai nostri dati originali. Quello che desideriamo da una tale trasformazione è che per il nostro esempio particolare produca valori nell'intervallo da 1 a 15 e non sia causa di irregolarità nella distribuzione dei dati originali. Una di tali trasformazioni alternative sarebbe quella di calcolare i valori dell'insieme originale in *modulo* 15 ed aggiungere uno (supponiamo che tutti i numeri memorizzati siano positivi). Applicando questa trasformazione (che viene usualmente chiamata "hashing", cioè calcolata) al nostro insieme di dati, otteniamo:



Questo risultato è però un pò deludente perché ci sono ancora parecchie collisioni. Con questo metodo siamo però in condizioni migliori perché si adatta meglio alle irregolarità nella distribuzione.

Precedentemente abbiamo osservato che più cerchiamo di "comprimere" un insieme di dati più facilmente ci saranno collisioni. Per provare tale affermazione vediamo se è possibile ridurre il numero di collisioni scegliendo di usare un array con 19 locazioni anziché 15 (cioè calcoleremo i nostri numeri in modulo 19 anziché in modulo 15 ed omettere il più uno). Nella maggior parte dei casi saremmo preparati all'utilizzo di un 20% in più di memoria se ciò comportasse risultati molto veloci con pochissime collisioni.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
57	20		60	23		44	45	27		10	30	12			53			
↑ {39}	↑ {42}		↑ {49}{31}															

Ancora una volta ci sono delle collisioni, ma questa volta non sono multiple. Da uno studio della tabella osserviamo che 11 dei 15 valori sono stati memorizzati in modo da essere trovati in un solo passo. L'ultima cosa da chiederci è dunque cosa fare con le collisioni. Se potessimo trovare un metodo per memorizzare i valori che si scontrano in modo da poter essere recuperati velocemente, allora troveremmo un algoritmo molto efficiente. Per studiare questo problema torniamo al nostro esempio.

La situazione iniziale è che nell'array c'è un numero di locazioni libere che può essere usato per memorizzare i quattro elementi che si scontrano. Possiamo quindi domandarci per ogni elemento quale sia la posizione migliore in cui memorizzarlo. Ad esempio, ci possiamo chiedere quale sia la posizione migliore per il 31 che si scontra con l'elemento in locazione 12. Analizzando questo caso particolare, vediamo che se 31 fosse messo nella posizione 13, saremmo in grado di trovarlo in due passi. Il modo per farlo sarebbe calcolare:

$$31 \bmod 19 \rightarrow \text{locazione } 12$$

Potremmo allora esaminare l'elemento in posizione 12 e scoprire che non è 31, dopodiché potremmo passare alla posizione successiva dell'array (cioè 13) e trovare il 31. È però possibile generalizzare questo procedimento? Come ulteriore esempio ci possiamo domandare dove debba essere posto il 49. Se la posizione 13 è occupata dal 31 dobbiamo scegliere se memorizzarlo in posizione 14 o 9, che sono le due locazioni più vicine. Dato che la nona sembra la posizione più favorevole per il 49, vediamo che ci sono dei problemi poiché in generale non sappiamo in che direzione cercare. La posizione migliore appare quindi la 14-esima. Possiamo sempre usare la regola della *ricerca in avanti* (modulo la dimensione della tabella) quando non troviamo l'elemento cercato nella posizione calcolata. Possiamo quindi concludere che le collisioni possono essere risolte memorizzando l'elemento *scontrato* nella successiva locazione libera in avanti. Applicando tale metodo al nostro insieme di dati, troviamo:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
57	20	39	60	23	42	44	45	27		10	30	12	31	49	53			
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Sono stati sottolineati gli elementi che si scontrano. La coda di ogni freccia indica dove avrebbe dovuto essere memorizzato un elemento che si scontrava. Prima di procedere ulteriormente, cerchiamo di valutare il numero "medio" di passi per recuperare un elemento da questa

tabella, supponendo che tutti gli elementi abbiano la stessa probabilità di essere ritrovati. Abbiamo:

- (a) 11 elementi localizzati in 1 passo
- (b) 3 elementi localizzati in 2 passi
- (c) 1 elemento localizzato in $\frac{1}{15}$ passi

Per questa configurazione, quindi, un elemento verrà ritrovato in media circa ogni 1.4 passi. Ciò sembra molto promettente, particolarmente se si estende tale procedimento al caso generale. Anziché seguire qui questa verifica, ammetteremo che sia vero e continueremo con lo sviluppo dell'algoritmo di ricerca calcolata che si adatti al nostro prefissato schema di memoria.

Per semplificare le cose, supporremo che l'obiettivo sia costruire una funzione di ricerca calcolata che ritorni semplicemente la posizione della chiave cercata (un numero) nella tabella hash. (Come abbiamo osservato precedentemente, noi non saremmo solo interessati alla posizione della chiave, ma piuttosto all'informazione associata ad essa cioè potremmo voler trovare l'indirizzo di una persona utilizzando una ricerca hash sul suo nome).

Fino a questo punto la nostra discussione suggerisce che i passi per la localizzazione di un elemento siano relativamente semplici. Il nostro metodo per la risoluzione delle collisioni richiede di compiere i seguenti passi principali:

1. Calcolare il valore hash modulo la dimensione della tabella della chiave cercata.
2. Se la chiave non è nell'indice dell'array corrispondente al valore calcolato allora
 - (a) compi una ricerca lineare in avanti dalla posizione corrente dell'array modulo la dimensione dell'array.

Per un elemento cercato *key* ed una tabella di dimensione *n* possiamo compiere il passo (1) usando:

$$\text{position} := \text{key} \bmod n$$

Possiamo allora usare un test della forma:

```
if table[position] < > key then ...
```

per decidere se *key* si trova o meno all'indice *position* dell'array *table*.

Per completare lo sviluppo dell'algoritmo, tutto ciò che resta da fare è definire i dettagli della ricerca. Nella costruzione della ricerca in avanti tutto ciò che dobbiamo fare è esaminare posizioni successive della tabella tenendo conto che l'indice dell'array dovrà "rifare il giro" una volta raggiunta la fine dell'array (vedi Fig. 5.9).

Possiamo usare la funzione *mod* per realizzare il “rifacimento del giro”, cioè:

$$\text{position} := (\text{position} + 1) \bmod n$$

La considerazione finale è elaborare i dettagli per la terminazione della ricerca: a questo riguardo sono importanti alcune considerazioni.

Per fare una ricerca il più efficiente possibile, dobbiamo fermare il processo o non appena è stata localizzata la chiave, o non appena ci accorgiamo che non è presente nella tabella. Un test della forma:

$$\text{if } \text{table}[position] = \text{key} \text{ then } \dots$$

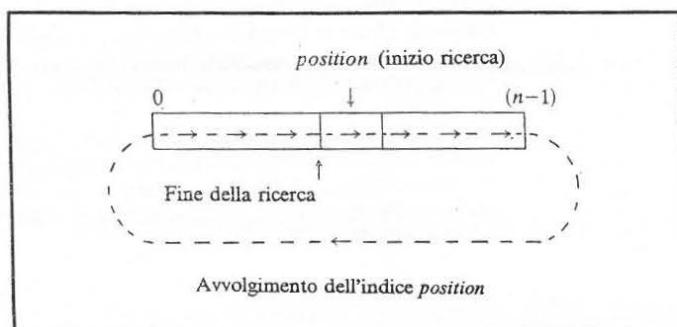
ci permetterà di rilevare la presenza di *key* nella tabella; non saremo in grado, tuttavia, di determinarne l'assenza. Per trovare un buon “test di assenza”, torniamo all'esempio precedente. Supponiamo di voler cercare nella tabella il numero 25 che non è presente. Il primo passo potrebbe essere:

$$\text{position} := 25 \bmod 19 = 6.$$

Troviamo che non è in posizione 6, quindi cominciamo a cercare in posizione 7, 8... Nel compiere questa ricerca troviamo che in posizione 9 viene incontrato un “bianco” invece di un altro numero. Cosa significa ciò?

Un “bianco” comporta una posizione *libera* in cui poter memorizzare un numero; inoltre, il modo in cui le chiavi sono inserite nella tabella ci indica che 25 non può trovarsi in alcuna posizione al di là della nona – ricordare la regola di risoluzione delle collisioni da noi proposta, “inserisci la chiave nella prima posizione libera”. Il primo incontro con una locazione non occupata può quindi essere usato per segnalare la condizione di fine per una ricerca negativa. Se stiamo usando dei numeri, non è conveniente segnare le posizioni libere con un bianco. In pratica il valore che usiamo per segnalare sicuramente una locazione non occupata dipenderà dalla natura delle chiavi e dal modo in cui è impiegata la tabella. Eviteremo quindi il problema utilizzando

Fig. 5.9.
Meccanismo di
risoluzione delle
collisioni per lo
schema hash.



una variabile *empty* passata alla procedura come parametro. Il test che potremo usare avrà allora la forma:

$$\text{if } \text{table}[position] = \text{empty} \text{ then } \dots$$

Potremmo allora proporre la seguente ricerca:

finché la chiave non è trovata e la posizione corrente non è vuota
(a) passa alla locazione successiva modulo la dimensione dell'array.

Questo sembra soddisfare le nostre richieste. Tuttavia, da un accurato studio delle condizioni di terminazione ci accorgiamo di aver trascurato la possibilità che la tabella sia piena e la chiave non sia presente. In questo caso abbiamo un ciclo potenzialmente infinito. Come possiamo evitarlo? Riguardando la Fig. 5.9 vediamo che non appena torneremo alla posizione da cui siamo partiti, sapremo che la chiave non è presente. Sembra necessario un ulteriore test. Abbiamo ora stabilito tre modi in cui può aver termine la ricerca:

1. chiave trovata;
2. non presente e posizione *empty*;
3. non presente e tabella *piena*.

La necessità di controllare queste tre condizioni ogni volta che viene esaminata una nuova posizione sembra piuttosto pesante. Ci potremmo quindi chiedere se esiste un modo per ridurre il numero di test. Può essere di aiuto l'uso fatto precedentemente di *sentinelle* per la terminazione della ricerca, di conseguenza vediamo che possono essere riuniti il primo ed il terzo test. Possiamo forzare il successo della ricerca dopo l'esame di tutti gli elementi della tabella. Il modo per farlo è memorizzare temporaneamente la chiave nella tabella nella posizione originale calcolata *dopo* aver scoperto che quel valore non uguaglia la chiave cercata. Il valore originale può essere poi rimesso al proprio posto una volta completata la ricerca. Per terminare il ciclo di ricerca può essere utilizzata una variabile Booleana *active* che è uguale a “false” quando è stata trovata la chiave o una locazione vuota. Un'altra variabile Booleana *found* può essere usata per distinguere le condizioni di terminazione. Questa variabile non dovrebbe essere assegnata quando viene trovata la chiave, ma solo dopo che la ricerca è stata completata.

È stato così completato lo sviluppo del nostro algoritmo.

Descrizione dell'algoritmo

1. Definisci la tabella calcolata da esaminare, la chiave cercata, il valore della condizione di casella vuota e la dimensione della tabella.
2. Calcola l'indice hash per la chiave modulo la dimensione della tabella.

3. Assegna una variabile booleana per terminare la ricerca.
4. Se la chiave è nella posizione indice allora
 - (a) passa alle condizioni di terminazione altrimenti
 - (b) poni una sentinella per controllare la condizione di tabella piena.
5. Finché non è soddisfatta la condizione di terminazione
 - (a) calcola l'indice successivo modulo la dimensione della tabella;
 - (b) se la chiave si trova nella posizione corrente, allora
 - (b.1) passa alle condizioni di terminazione e segnala se "trovato" è valido altrimenti
 - (b.2) se la posizione della tabella è vuota segnala la terminazione.
6. Togli la sentinella e ripristina la tabella.
7. Ritorna il risultato della ricerca.

Implementazione in Pascal

```

procedure hashsearch(var table: nelements; var position: integer;
var found: boolean; tablesize,empty,key:integer);
var temp {temporary storage for value at position start},
start {hash value index to table}: integer;
active {if true continue search of table}: boolean;

begin {uses hashing technique to search table for key}
{assert: tablesize > 0}
active := true; found := false;
start := key mod tablesize;
position := start;
if table[start]=key then
begin {key found at hash position}
active := false;
found := true;
temp := table[start]
end
else
begin {set up sentinel to terminate on full table}
temp := table[start];
table[start] := key
end;
{invariant: key not equal to values table[start..(position - 1) mod
tablesize] \ number of values checked =< tablesize + 1}
while active do
begin {linear search table from position start for key}
position := (position + 1) mod tablesize;
if table[position]=key then
begin {found key - make sure not just termination}
active := false;
if position <> start then found := true
end

```

```

else
  if table[position]=empty then active := false
end;
table[start] := temp
{assert: (table[position]=key \ position <> start \ found=true \ 
(complete table searched \ found=false))}
end

```

Note di progetto

1. Lo svolgimento dell'algoritmo di ricerca calcolata può essere caratterizzato dal numero di elementi della tabella che devono essere esaminati prima della terminazione. Questo andamento è funzione della frazione di locazioni occupate (chiamate il *fattore di carico* α). Si può dimostrare, dopo aver fatto inconfondibili affermazioni statistiche, che in una ricerca con esito positivo dovranno essere esaminate in media $[1 + (1/(1 - \alpha))]/2$ locazioni (cioè per una tabella piena per l'80% tale valore sarà meno di tre locazioni, indipendentemente dalla dimensione della tabella). Il costo di una ricerca con esito negativo è maggiore. In media, prima di incontrare una cella vuota dovranno essere esaminate $[1 + (1/(1 - \alpha_2))]/2$ locazioni (ad esempio, per una tabella per l'80%, questo ammonta a 13 locazioni).
2. Per caratterizzare il comportamento della ricerca hash, ci dobbiamo focalizzare sul ciclo di ricerca. Dopo ogni iterazione, sarà stato stabilito che *key* non è nelle locazioni da *start* a *position* o che si trova nella locazione *position*, o se la posizione corrente è vuota, che *key* non è presente nella tabella. L'algoritmo terminerà perché, con ogni iterazione del ciclo di ricerca, *position* aumenterà di un'unità modulo la dimensione della tabella; la sentinella verrà quindi incontrata sicuramente. Ciò causerà la terminazione se non era accaduta prima, dovuta al ritrovamento o ad una casella vuota.
3. Nella risoluzione di questo problema, è utile concentrarsi su una soluzione ideale ma impraticabile. Saremo in grado di passare da questa situazione ad una implementazione più pratica che conservi la maggior parte delle caratteristiche e delle proprietà dell'approccio originale.
4. Sarebbe utile che il costo di ricerche infruttuose fosse in media minore. Un modo per ottenere ciò è di inserire le chiavi nella tabella in ordine crescente alfabetico o numerico. Ad esempio, consideriamo le cinque chiavi 16,18,25,27,29 inserite in ordine numerico nella tabella seguente di dimensione 5.

0	1	2	3	4
25	16	26	18	29

Se stessimo cercando il 17 che si troverebbe **in** posizione 2, potremmo immediatamente fermare la ricerca poiché in questa posizione si trova una chiave che ricorre *numericamente* dopo il 17. La regola di stop si usa in genere per tabelle costruite usando un metodo di ordinamento. Il numero medio di locazioni da esaminare per una ricerca infruttuosa è allora *identico* a quello in caso di ricerca con esito positivo.

5. Nella maggior parte (ma non in tutte le applicazioni pratiche, le tabelle hash non possono essere riempite oltre l'80% a causa del rapido deterioramento nell'esecuzione oltre tale livello).
6. Il metodo di hashing descritto soffre del fatto che le chiavi inserite per ultime tendono a *riunirsi* intorno a quelle inserite senza collisioni. Un modo per minimizzare questo fatto è applicare quello che viene chiamato il metodo del quoziente lineare. L'*idea* è di usare il *resto* della divisione della dimensione della tabella (che deve essere un numero primo), come primo sondaggio della tabella. Il *quoziente* è allora usato come lunghezza dei passi per fare ricerche seguenti nella tabella (se la ricerca è lineare). Con questo metodo, tutti i valori della tabella vengono esaminati prima della ripetizione se la dimensione della tabella è un numero primo.

Applicazioni

Recupero veloce da tabelle grandi e piccole.

Problemi supplementari

- 5.8.1 Estendere l'algoritmo di calcolo dato in modo da operare su parole anziché su caratteri. Un modo di "calcolare" una parola è fare la somma dei valori dei caratteri della parola (cioè $a = 1$, $b = 2$, $c = \dots$ — la parola *ace* avrebbe il valore calcolato $1 + 3 + 5 = 9$).
- 5.8.2 Modificare l'algoritmo di hash dato in modo che esamini una tabella in cui le parole siano inserite in ordine alfabetico (vedi la precedente nota 4).
- 5.8.3 Non è sempre conveniente o possibile inserire degli elementi in una tabella hash nell'ordine richiesto dal problema precedente. Un approccio alternativo che raggiunge lo stesso effetto è, ogni volta che si inserisce un nuovo elemento, procedere nel modo seguente. Se x dovesse essere inserito in posizione k ed il valore in k fosse successivo ad x , allora si scambi il ruolo di x con $\text{table}[k]$ e si ripeta il processo finché non venga trovata una locazione libera. Ciò ha l'effetto di fare in modo che x preceda in ordine chiavi inserite prima, ma successive ad x in ordine alfabetico.
- 5.8.4 Implementare il metodo del quoziente lineare descritto nella nota 6 e confrontare la sua esecuzione con quella dell'algoritmo precedente per un fattore di carico dell'80%. Usare un generatore di numeri casuali per costruire l'insieme delle chiavi. Fare

test per insiemi di ricerche fruttuose e non.

- 5.8.5 Tracciare un diagramma fattore di carico contro costo di ricerca (fruttuosa e non) per fattori di carico compresi nell'intervallo dal 50% al 95% con un intervallo del 5%.
 - (a) usando la formula data nella nota 1, e
 - (b) facendo una simulazione con numeri casuali.
- 5.8.6 Usare numeri casuali per riempire una tabella per l'80% e stimare il numero di valori che si trovano nella loro posizione calcolata, alla posizione calcolata + 1, alla posizione calcolata + 2 e così via.
- 5.8.7 Se elementi vengono recuperati da una tabella hash con frequenze diverse, un'accelerazione nel ritrovamento può essere ottenuta spostando ogni elemento trovato di un posto verso la sua posizione calcolata originale facendo uno scambio. Implementare questo metodo di ricerca calcolata.

TRATTAMENTO DI TESTI E RICERCA DI CONFIGURAZIONI

INTRODUZIONE

È stato riconosciuto da molto tempo che i computer sono attrezzati meglio per eseguire cose diverse da computazioni numeriche. Una conseguenza di ciò è stata la sempre crescente applicazione dell'informatica al trattamento di testi. La natura dei dati alfabetici in contrapposizione ai dati numerici, ci costringe ad adottare un'intera serie di nuove strategie ed algoritmi computazionali che siano apprezzabilmente diversi da quelli che ci sono serviti per trattare problemi numerici. L'unità di base è ora il carattere invece che il numero.

I nostri maggiori interessi nell'elaborare un testo sono centrati attorno alla manipolazione e movimento di caratteri o alla ricerca di tipi e parole. I primi due algoritmi che considereremo in questo capitolo mostrano come viene applicata la manipolazione di testi alla preparazione di documenti con il computer. I meccanismi per manipolare testi sono generalmente semplici, sebbene si debba prestare molta attenzione ai dettagli ed alla struttura del testo.

La parte più importante dell'elaborazione di testi è forse quella basata sulla ricerca di particolari elementi all'interno di un testo. La grande quantità di testo che deve essere analizzata, e la frequenza con cui deve essere condotta la ricerca, sottolineano la necessità di algoritmi di ricerca efficienti. L'ultima metà di questo capitolo è dedicata allo studio del testo e agli algoritmi di ricerca di elementi. Questi algoritmi dimostrano l'importanza di porre attenzione alla struttura del testo ed alle informazioni disponibili nel tener conto del progetto.

ALGORITMO 6.1

DETERMINAZIONE DELLA LUNGHEZZA DI UNA LINEA DI TESTO

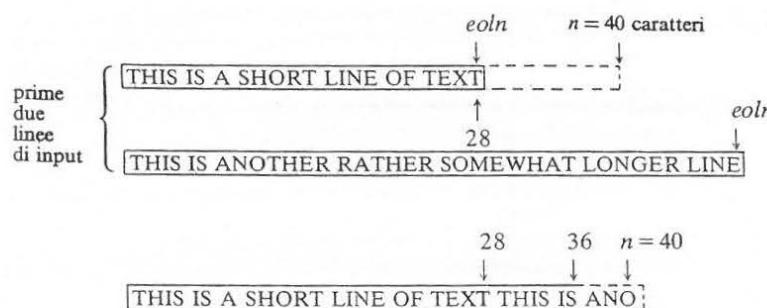
Problema

Dato un insieme di linee di un testo di lunghezza arbitraria, riformattare il testo in modo che non vengano stampate righe con un numero di caratteri superiori a n (in questo caso 40). In ogni linea in uscita devono essere stampate il maggior numero di parole per un massimo di n caratteri e nessuna parola si deve estendere tra due linee. I paragrafi devono inoltre rimanere separati.

Sviluppo dell'algoritmo

Cominciamo a pensare al progetto di questo problema cercando di isolare i meccanismi di base. Potremmo cominciare leggendo il testo finché non si incontrano n caratteri. A questo punto possono essere stampate le parole complete. Bisogna ricordare che le linee di ingresso possono contenere un numero di caratteri minore, uguale o maggiore di n .

Consideriamo in un diagramma cosa succede quando sta cominciando il processo. In questo caso supponiamo $n = 40$.



Ciò suggerisce che quando arriviamo alla fine della prima riga ed i caratteri contati non hanno ancora raggiunto il limite di $n = 40$, dobbiamo incominciare la lettura dalla riga successiva. Dopo aver posto 28 caratteri nel nostro array, abbiamo raggiunto la fine della prima riga di ingresso. Possiamo allora iniziare la lettura dalla seconda

riga per raggiungere la quota di 40 caratteri. Se posizioniamo il primo carattere della seconda riga in posizione 29, troveremo:

...OF TEXTTHIS...

Questo suggerisce che quando incontriamo un fine — riga in ingresso, dobbiamo inserire uno spazio per garantire la separazione delle parole. La cosa seguente da osservare è che arrivati al 40-esimo carattere, ci troviamo nel mezzo della parola ANOTHER. Poiché le parole non devono essere spezzate, in uscita non devono essere stampati caratteri della parola ANOTHER. La nostra prima linea in uscita sarà dunque lunga 36 caratteri.

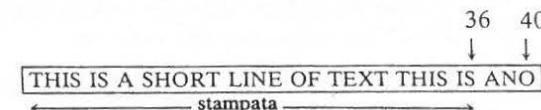


Per ottenere questo output dobbiamo sapere dove finiva l'ultima parola completa (in questo caso IS). Vi sono due modi per stabilirlo. Possiamo fare un conto all'indietro dalla posizione 41 finché non incontriamo uno spazio bianco. Possiamo ad esempio usare:

```
back := 41;
while a[back] <> ' ' do back := back - 1
```

Dobbiamo leggere un carattere in più di quelli che vogliamo in uscita per far sì che una parola possa terminare in posizione 40. Alternativamente, appena viene incontrato un nuovo spazio bianco, possiamo aggiornare una variabile in posizione corrente. Quando raggiungiamo il limite del 41-esimo carattere il valore corrente della variabile ci dirà dove finiva l'ultima parola completa. Quest'ultimo approccio è più costoso.

Sapendo dove finisce l'ultima parola si può stampare la linea corrente. Ci resta ora ANO.



Cosa possiamo fare a questo punto? Un semplice approccio sarebbe quello di muovere ANO all'inizio dell'array e quindi iniziare a riempirlo di nuovo. Allo stesso tempo possiamo contare i caratteri della nuova riga in uscita. Poiché le parole sono in media lunghe pochi caratteri, il costo di questo movimento di dati sarà piccolo e potrà essere condotto a termine nel modo seguente:

```
c := 0;
for j := back + 1 to 41 do
begin
  c := c + 1;
  a[c] := a[j];
end
```

Nella procedura di copiatura bisogna iniziare da *back + 1* per far sì che la prima parola della riga non cominci con uno spazio bianco.

Nei problemi di questo genere dobbiamo sempre essere sicuri che l'algoritmo termini correttamente. Quali sono le possibilità in questo caso? La cosa più probabile che possa accadere è di terminare nel punto in cui viene incontrato il fine — riga con una riga parzialmente riempita. L'algoritmo, come è stato sviluppato fino ad ora, stampa solo righe piene. Dobbiamo, perciò, includere dei passi che permettano la stampa di qualsiasi linea parzialmente riempita dopo l'incontro di un fine — riga.

L'algoritmo sviluppato, può essere considerato come un approccio al problema *incentrato sulla riga*. Possiamo ora riassumere la strategia principale.

1. Finché not end-of-file
 - (a) riempì la linea di uscita;
 - (b) scrivì la linea di uscita fino all'ultima parola intera;
 - (c) sposta qualsiasi parola incompleta dalla fine della riga di uscita all'inizio di essa.
2. Stampa tutte le righe riempite parzialmente.

Nel nostro algoritmo, quando una parola non si adatta completamente ad una riga, deve essere stampata sulla riga successiva. In tal modo, ognqualvolta vi sia una parola da stampare, rimane sempre la domanda: "può essere stampata sulla riga corrente o deve essere inserita nella successiva?" Queste osservazioni suggeriscono che può essere vantaggioso considerare un approccio al problema *orientato sulla parola*. Un modo di seguire questo approccio è di leggere le parole una alla volta. Dopo che è stata letta una data parola, può prevalere una delle due situazioni:

1. La parola può essere stampata sulla riga corrente perché il carattere contatto non eccede il limite di lunghezza della riga.
2. Il limite di lunghezza della riga è stato superato, quindi la parola deve essere scritta nella riga successiva.

La discussione precedente suggerisce la seguente strategia:

1. Finché not end-of-file
 - (a) leggi la prossima parola in ingresso;
 - (b) se il limite di lunghezza della riga non viene superato, allora
 - (b.1) scrivi la parola nella riga corrente
altrimenti
 - (b.1) va alla prossima riga e scrivi la parola corrente.

Confrontando questo nuovo meccanismo con quello proposto, è facile constatare che ora abbiamo un algoritmo più chiaro, più semplice e più pulito. Possiamo quindi procedere alla rifinitura del nuovo algoritmo perché non sembra esserci un'altra possibile alternativa. Nello svolgimento ci dobbiamo assicurare che il nostro metodo sia in grado di trattare la punteggiatura e di lasciare invariato ogni paragrafo

presente nel testo originale (le tabulazioni non verranno considerate).

Dopo aver analizzato una varietà di testi, arriviamo alla conclusione che, convenzionalmente, le parole sono separate da uno o più spazi e/o da un nuovo carattere. Ogni carattere di punteggiatura, che in generale segue direttamente la fine della parola, non deve essere distinto dalla parola stessa. Queste osservazioni semplificano il nostro compito.

La fase di ingresso può procedere carattere per carattere. Se il carattere corrente non è né uno spazio né un fine — riga, può semplicemente essere aggiunto all'array della parola corrente. Non appena è stata costruita la parola, si dovrà fare il conto dei caratteri. Ricordando la discussione precedente, se esso è un carattere di fine — riga, deve essere sostituito da uno spazio.

Così ridimensionato, il meccanismo di ingresso diventa:

1. Finché not end-of-file
 - (a) leggi e memorizza il prossimo carattere;
 - (b) se il carattere è uno spazio, allora
scrivi la parola corrente nella riga appropriata.

Per determinare su quale riga debba essere scritta la parola corrente, dobbiamo sommare la lunghezza della parola corrente (più gli spazi successivi) *wordcnt* alla lunghezza della riga corrente *linecnt*. Se non è stato superato il *limite* di lunghezza della riga, la parola corrente può essere copiata direttamente e si può proseguire con la parola successiva. D'altra parte, se la parola corrente esce dal margine occorre passare alla riga successiva prima di scrivere la parola. Dopo che questa è stata scritta sulla nuova linea, il numero dei caratteri della nuova riga dovrà essere pari alla lunghezza della parola appena scritta.

L'unica differenza tra le due situazioni di parola in uscita è che nell'istante in cui viene superato il limite della riga, l'output deve essere preceduto da uno spostamento sulla riga successiva e da un reset del conteggio dei caratteri. I passi necessari per la stampa della parola corrente sono:

- (a) se viene superata la lunghezza limite della riga, allora
 - (a.1) passa alla riga successiva;
 - (a.2) inizializza il contatore dei caratteri per la nuova riga *linecnt*;
- (b) copia la parola corrente più gli spazi successivi;
- (c) azzerà il contatore dei caratteri della parola pronta per essere letta.

Riprendendo le specifiche iniziali, si richiede che l'algoritmo lasci intatti i paragrafi presenti nel testo originale. Noi sappiamo che un nuovo paragrafo è contraddistinto da uno o più spazi bianchi ad inizio riga. La domanda è ora come si possa distinguere tra spazi bianchi multipli incorporati nel testo ed un nuovo paragrafo. Una piccola riflessione rivela che un *nuovo* paragrafo deve seguire direttamente un end-of-line (fine — riga). Dovremo quindi essere in grado di utilizzare la condizione di end-of-line, *eol*, seguita da uno spazio bianco, per

determinare l'inizio di un nuovo paragrafo. (Supporremo che il testo originale sia stato preordinato in modo tale che solo nuovi paragrafi abbiano spazi che li connotino come tali). La verifica del paragrafo ed il successivo procedimento saranno quindi:

se end-of-line è seguito da uno spazio singolo, allora

- va alla riga successiva,
- azzerà il contatore di carattere per la prossima riga.

Affinché questo testo sia efficace dovremo attivare un demarcatore di end-of-line, eol, nel momento in cui viene incontrata la condizione di fine-riga. Poiché la condizione di nuovo paragrafo implica la combinazione di una nuova riga seguita da uno spazio, sarà necessario posizionare il segnale dopo il controllo sul paragrafo per evitare di considerare un nuovo paragrafo quando uno spazio precede il carattere di fine-riga. Nell'adottare questa strategia occorre tener conto del fatto che il carattere ritornato in Pascal quando cerchiamo di leggere una nuova riga è uno spazio. Dobbiamo anche ricordare di disattivare il segnale di una nuova riga non appena avremo incominciato a scrivere le parole. Può essere ora descritto il completo algoritmo.

Descrizione dell'algoritmo

- Stabilisci il *limite* di lunghezza della riga e sommagli uno per considerare lo spazio bianco.
- Inizializza i contatori dei caratteri di riga e di parola a zero e il segnale di fine — riga a *false*.
- Finchè non viene trovato un fine — riga
 - leggi e memorizza il prossimo carattere;
 - se il carattere è uno spazio, allora
 - se c'è un nuovo paragrafo, allora
 - spostati sulla nuova riga e riaggiorna il contatore di caratteri di riga,
 - somma la lunghezza della parola corrente a quella della riga,
 - se la parola corrente provoca un eccesso di lunghezza della riga, allora
 - passa alla riga successiva e poni la lunghezza della riga pari a quella della parola corrente,
 - copia la parola corrente ed i suoi successivi spazi e riinizializza il contatore di caratteri
 - disattiva il segnale di fine — riga d'ingresso.
 - se trovi un fine — riga — d'ingresso, allora
 - attiva il segnale di fine — riga e passa alla riga successiva.

non alfabetici

Implementazione in Pascal

```
procedure textformat(limit: integer);
var i {index for word array},
    linecnt {count of characters on current line},
    wordcnt {count of characters in current word}: integer;
    chr {current character},
    space {the space character}: char;
    eol {flag to mark end of line condition}: boolean;
    word: array[1..30] of char; {storage for current word}

begin {reformats text so no line more than limit long and no broken words}
{assert: all words less than 31 characters ∧ limit >= maximum wordlength in text}
wordcnt := 0; linecnt := 0; eol := false; space := ' ';
limit := limit + 1;
{invariant: after current character read, all lines with maximum number of whole words = <= linecnt will have been printed ∧ all complete words read and identified will have been printed}
while not eof(input) do
begin {process character by character using space to signal end-of-word}
read(chr);
wordcnt := wordcnt + 1;
word[wordcnt] := chr;
if chr = space then
begin {possible end of current word}
if eol and (wordcnt = 1) then
begin {a new paragraph detected in original text}
writeln;
linecnt := 0
end;
linecnt := linecnt + wordcnt;
if linecnt > limit then
begin {line length limit exceeded so move to next output line}
writeln;
linecnt := wordcnt
end;
{write out current word}
for i := 1 to wordcnt do
write(word[i]);
wordcnt := 0;
eol := false;
if eoln(input) then
begin {set end-of-input-line flag to detect new paragraph}
eol := true;
readln
end
end
end;
end;
```

```

{assert: complete text reformatted and printed such that no line
contains more than limit characters and each line has the
maximum number of complete words packed on the line}
writeln
end

```

Note di progetto

- Il costo del testo riformattato è linearmente dipendente dal numero di caratteri del testo originale.
- Al punto in cui il j -esimo carattere nel testo originario è stato letto, tutte le parole complete nei primi j caratteri sono state scritte, in accordo con i requisiti stabiliti dalla variabile *limit*. Anche tutti i paragrafi all'interno dei primi j caratteri saranno stati separati come nel testo originale. Tutte le righe stampate prima del raggiungimento del j -esimo carattere saranno di lunghezza minore o uguale al valore *limit* dei caratteri. Su ogni riga in uscita sarà stampato il massimo numero di parole contenuto nel limite di lunghezza. Ad ogni iterazione viene letto un nuovo carattere dal file, così verrà infine raggiunta la condizione di fine — riga ed il ciclo avrà termine.
- Il focalizzarsi sulle parole in uscita invece che sulle righe, porta ad un algoritmo più semplice e più chiaro. Dovremo sempre stare attenti al meccanismo fondamentale che accompagnerà il progetto che vorremo svolgere.
- Occorre notare che il presente algoritmo tratta punteggiatura e spazi multipli del testo in maniera semplice. Possiamo supporre che ci siano parole di *lunghezza zero* tra spazi multipli.
- Confrontando il progetto finale con il metodo iniziale, possiamo notare di aver costruito condizioni più semplici di inizializzazione e di terminazione.
- Un diagramma, sebbene utile per la risoluzione di questo problema, ha cercato di spingerci verso una soluzione del problema accentuato sullo studio della riga piuttosto che ad una soluzione orientata sullo studio delle parole.

Applicazioni

Elaborazione testi, scrittura di relazioni e gestione di file.

Problemi supplementari

- Modificare l'algoritmo in modo da togliere gli spazi bianchi multipli diversi da quelli all'inizio di un paragrafo.

- L'algoritmo sviluppato non può riprodurre il testo originario se l'uscita viene utilizzata come ingresso. Ciò accade perché se c'è uno spazio alla fine di una riga in ingresso, questo dovrebbe essere trasformato in due spazi nel testo in uscita. Modificare l'algoritmo in modo da superare questo problema.
- Progettare un algoritmo che legga e allinei a sinistra righe di un testo (come richiesto nel nostro algoritmo). L'inizio di un nuovo paragrafo non deve essere allineato. Notiamo che si può supporre che ogni nuovo paragrafo inizi con una lettera maiuscola e sia preceduto da un certo numero di spazi bianchi.
- Progettare un algoritmo che legga righe di un testo, lo riformatti e stampi pagine con due colonne, (ognuna di quaranta caratteri) separate da un intervallo di 10 spazi. La prima colonna in uscita deve corrispondere alla prima metà della pagina in ingresso e la seconda colonna alla seconda metà. Ogni pagina in uscita deve contenere 40 righe di testo.
- Implementare il primo progetto di formattazione di testo proposto. Cercare di evitare di spostare frammenti di parole dopo che sia stata stampata la riga corrente.

ALGORITMO 6.2 ALLINEAMENTO DEI MARGINI DI UN TESTO

Problema

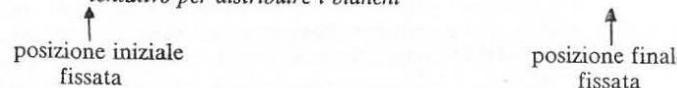
Progettare ed implementare una procedura che allinei i margini sinistro e destro di un testo evitando di spezzare parole e lasciando intatti i paragrafi. Si dovrebbe fare un tentativo per distribuire i bianchi addizionali il più regolarmente possibile nel testo allineato.

Sviluppo dell'algoritmo

Libri e documentazioni di informatica hanno quasi sempre i margini allineati. Ciò significa che *tutte* le righe hanno *lunghezza fissa*, la *prima* parola comincia sempre in una posizione fissata (eccetto all'inizio di un nuovo paragrafo) e l'*ultima* parola termina anch'essa in posizione

fissa. Ad esempio, consideriamo l'esposizione precedente del problema, allineata a sinistra e destra. Abbiamo:

Progettare ed implementare una procedura che allinei i margini sinistro e destro di un testo evitando di spezzare parole lasciando intatti i paragrafi. Si dovrebbe fare un tentativo per distribuire i bianchi



La lunghezza prefissa della riga viene raggiunta inserendo tra le parole spazi addizionali. Questo esempio pone in rilievo la necessità di permettere alla lunghezza prefissa della riga di essere considerevolmente più lunga o più corta della lunghezza media delle righe del testo non formattato. Ciò suggerisce che in alcuni casi prima dell'inizio della marginatura, sarà necessario un considerevole riordinamento del testo. Da uno studio più dettagliato dell'esempio precedente, si arriva alla conclusione che una riga può essere allineata a sinistra ed a destra quando la riga non formattata sia di lunghezza minore o uguale al valore prefissato. Sembra, quindi, che la procedura di formattazione debba avvenire in due passi:

- (a) riordina il testo in modo che le lunghezze delle righe non superino la lunghezza prefissa della riga in uscita.
- (b) allinea i margini delle righe riordinate di testo.

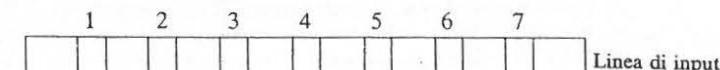
I requisiti del primo passo sono essenzialmente eseguiti dalla procedura sviluppata nel paragrafo 6.1. Il nostro compito si riduce quindi a quello di inserire il risultato della procedura dell'algoritmo 6.1 nella procedura di marginatura destra e sinistra. In questo modo il processo può essere sviluppato molto semplicemente. Non appena la procedura *textformat* produce una riga di testo, essa può chiamare direttamente la nostra nuova procedura per completare l'allineamento. Questo richiederà solo alcuni piccoli cambiamenti alla procedura originale *textformat*, che sono lasciati come esercizio al lettore.

Da qui in poi ci concentreremo sulla procedura di allineamento con l'asserzione che l'ingresso di questa procedura sia una linea di testo di lunghezza minore o uguale a quella richiesta in uscita. Il nostro problema è ora di distribuire *equamente* gli spazi bianchi addizionali tra le parole della riga di ingresso. Per semplificare il problema, supponiamo che ci sia un solo spazio separante le parole, e non vi siano spazi addizionali alla fine della riga. È semplice modificare la procedura di formattazione in modo da garantire di avere questa condizione in uscita. Possiamo anche supporre che le punteggiature nel testo seguano o precedano direttamente le parole senza spazi intermedi.

Per iniziare l'inserimento di spazi possiamo considerare alcuni esempi per cercare di comprendere quello che comporta. Da un esame approfondito del problema si arriva alla conclusione che per ogni particolare riga è possibile una delle seguenti quattro situazioni:

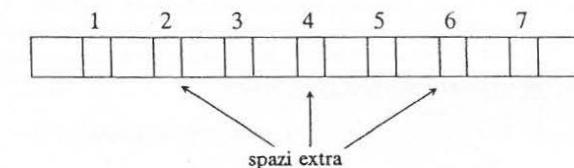
1. La riga è già della lunghezza corretta e quindi non è richiesta alcuna procedura (non devono essere aggiunti spazi).
2. Il numero di spazi extra da aggiungere per allargare la riga corrente è uguale al numero di spazi già presente nella riga. In questo caso è sufficiente aggiungere uno spazio ad ognuno di quelli già esistenti.
3. Il numero di spazi da aggiungere è maggiore di quelli già esistenti nella riga.
4. Il numero di spazi da aggiungere è minore di quelli già esistenti nella riga.

Poiché i primi due casi sono immediati, possiamo concentrarci su esempi in cui siano applicate le situazioni (3) e (4). Come per esempio del caso (3), supponiamo di dover aggiungere 10 spazi addizionali nella riga che possedeva originariamente solo 7 spazi.



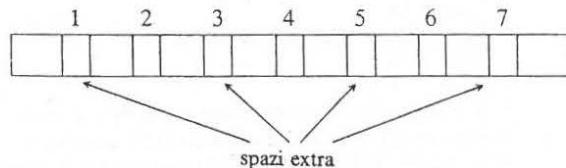
Il nostro problema è di *distribuire uniformemente* i 10 spazi extra tra le 7 posizioni collocate. Dividendo 10 per 7 scopriamo che idealmente dovrebbero essere aggiunti $1,3/7$ spazi a quelli già esistenti. Ovviamen-
te ciò non è di grande aiuto, poiché possiamo sommare solo un numero intero di spazi a quelli già esistenti. Poiché ci sono più spazi da aggiungere che posizioni esistenti, un primo passo sarebbe quello di aggiungere uno spazio ad ogni posizione esistente. Ciò soddisferà certamente il nostro criterio di uniformità e ci lascerà con soli tre spazi da allocare ($10 - 7$). Non appena ci sono meno spazi da aggiungere che posizioni disponibili, siamo nella situazione (4). Il nostro problema ora è quello di sommare *meno di uno* spazio ad ogni posizione esistente.

A questo punto potremmo aggiungere un altro spazio ad ognuna delle prime tre posizioni. Se adottiamo questa strategia, l'aspetto globale del testo in uscita tenderà a diventare asimmetrico. Dobbiamo, quindi trovare un modo per distribuire più uniformemente questi spazi extra tra le 7 posizioni. Se ci fosse un solo spazio da aggiungere, vorremmo venisse aggiunto nel mezzo, se ce ne fossero due, vorremmo che il primo fosse posto ad un terzo della riga, ed il secondo a due terzi. Quando questo approccio viene accoppiato alla variazione delle dimensioni delle parole ed alle lunghezze delle righe in ingresso, gli spazi addizionali tenderanno a mescolarsi all'ambiente. Ritornando al nostro esempio, dove dobbiamo distribuire 3 spazi in 7 posizioni, abbiamo:



In questo particolare caso il risultato migliore corrisponderebbe probabilmente agli spazi extra aggiunti dopo le parole 2, 4, e 6.

Prima di cercare di sviluppare un meccanismo, concentriamoci su di un altro esempio (distribuire 4 spazi addizionali in 7 posizioni).



“Ad occhio”, in questo caso la soluzione migliore sembrerebbe quella di distribuire gli spazi aggiuntivi dopo le parole 1, 3, 5 e 7; per questo dobbiamo sviluppare un meccanismo che sia in grado di distribuire gli spazi in questo modo. Notiamo che in entrambi i casi c’è un intervallo di due tra spazi extra consecutivi. Nel primo caso, che comporta l’inserimento di tre spazi in 7 posizioni, dovremo idealmente aggiungere uno spazio ogni $2,1/3$ (cioè $7/3$) posizioni, più vicino a due che a qualsiasi altro valore: sembrerebbe così una strategia ragionevole inserire gli spazi extra ogni due parole. Nel secondo caso, dove devono essere aggiunti 4 spazi, troviamo che idealmente potremo inserire uno spazio ogni $1,3/4$ posizioni. Ancora una volta, la soluzione migliore sembra sia l’inserimento ogni due posizioni. Con questa applicazione troviamo la sequenza 2, 4, 6, 8.

A questo punto sorge un problema, perché l’ultimo spazio extra non si adatta all’interno del nostro insieme di 7, sebbene 2 sembri l’incremento più favorevole. Nella soluzione “manuale” abbiamo usato le sequenze 1, 3, 5, 7 adottando ancora un incremento di 2, ma con la posizione iniziale spostata indietro di una posizione. Gli ultimi due casi sono uguali eccetto per le posizioni iniziali. Le ultime osservazioni suscitano almeno due puntualizzazioni: in primo luogo, la posizione iniziale scelta può influenzare la possibilità o meno di aggiungere il numero richiesto di spazi; secondariamente non esiste garanzia che per un dato incremento sarà possibile aggiungere il numero di spazi richiesto. Il nostro algoritmo dovrà provvedere a questi punti.

Supponendo di essere convinti che un metodo ripartito sia il migliore per distribuire uniformemente spazi addizionali, cerchiamo di stabilire una posizione iniziale adatta per il caso generale in cui noi conosciamo già l’incremento della ripartizione δ . Nel nostro esempio con $\delta = 2$, vediamo che la ripartizione potrebbe essere iniziata dallo spazio 1 o 2. In generale, sarebbe possibile un qualsiasi valore iniziale compreso tra 1 e δ . Ci domandiamo ora quale posizione iniziale produrrebbe la più bilanciata distribuzione di spazi addizionali.

Una posizione iniziale $next$ di:

$next := \delta \text{ div } 2$

ci fornirà un buon bilanciamento, poiché è la media della possibile portata. Tuttavia, $next$ sarà uguale a zero quando δ è uno, a causa

della divisione intera. Possiamo evitare questa situazione, usando:

$next := (\delta + 1) \text{ div } 2$

Rimane ora il secondo problema di cosa fare se la ripartizione scelta e la posizione iniziale non ci permettono di espandere la riga in modo voluto. (Ad esempio, la nostra combinazione ripartizione/posizione iniziale ci permette di inserire 5 spazi, mentre ne sono necessari 7 per produrre la lunghezza desiderata della riga in uscita. Una volta aggiunti i 5 spazi, rimane il problema di distribuire uniformemente i restanti 2.) In presenza di tale situazione dopo l’aggiunta degli spazi in accordo con la ripartizione, dovremo risolvere un problema più piccolo, cosa che potremo fare utilizzando esattamente lo stesso meccanismo. Questa possibilità di poter ripetere la distribuzione degli spazi addizionali, suggerisce la necessità di due cicli — uno per distribuire gli spazi extra conformemente alla ripartizione e l’altro per ripetere il processo per altre ripartizioni se necessario. La parte centrale del nostro meccanismo potrebbe essere:

finchè ci sono spazi extra da distribuire

- calcola l’incremento corrente della ripartizione e la posizione iniziale in accordo con il contatore degli spazi extra correnti,
- distribuisci gli spazi addizionali tramite la ripartizione e riduci opportunamente il contatore degli spazi extra.

Dopo aver sviluppato un meccanismo che tratta il caso in cui ci siano meno spazi da aggiungere che posizioni disponibili, ci dobbiamo chiedere come esso possa essere adattato al resto del meccanismo. Confrontando i casi (3) e (4) è chiaro che stiamo effettivamente usando un meccanismo di ripartizione anche nel caso (3). Non è quindi necessario trattare il caso (3) come particolare. Scegliendo un incremento di uno ed una posizione iniziale uguale a uno, il meccanismo ci permetterà di aggiungere spazi ad ognuno già esistente. Può accadere, tuttavia, di voler aggiungere più di uno spazio ad ogni spazio già esistente (cioè possiamo voler aggiungere 9 spazi in 4 posizioni). Per provvedere a questo caso può essere usata una variabile *spaceblock* indicante il numero di spazi da aggiungere ad ogni posizione del passo corrente di ripartizione. Quando ci sono *extraspaces* spazi extra da aggiungere a *nspaces* posizioni, abbiamo:

$spaceblock := extraspaces \text{ div } nspaces$

Seguendo il ragionamento precedente, l’incremento di ripartizione δ verrà calcolato usando:

$\delta := \text{round} (nspaces/extraspaces)$

L’uso di questa espressione nel caso in cui *nspaces* sia minore di *extraspaces* porterebbe l’assegnazione a δ del valore zero. Un ulteriore test:

$\text{if } \delta = 0 \text{ then } \delta := 1$

è quindi necessario per uscire da questa situazione.

Dopo aver trovato il meccanismo della procedura per la distribuzione degli spazi addizionali per marginare il testo, il prossimo compito è di costruire il resto del meccanismo attorno a questi passi.

Prima di incominciare ad aggiungere gli spazi, dobbiamo determinare il numero di posizioni della riga in cui possano venire posti. Ammettendo che le parole siano separate da un unico spazio, questo numero sarà il numero di spazi nella riga. Scegliendo questo svolgimento, abbiamo dimenticato l'esistenza di un certo numero di spazi lasciati all'inizio di un paragrafo. Prima di contare gli spazi di una data riga, quindi, dovremo passare oltre tutti gli spazi iniziali. Per precauzione, potremo anche controllare gli spazi alla fine della riga.

I passi saranno:

1. Togli gli spazi all'inizio ed alla fine della riga.
2. Azzera il contatore degli spazi.
3. Finché ci sono ancora caratteri nella riga
 - (a) se il carattere corrente è uno spazio, allora
 - (a.1) incrementa il contatore degli spazi.

Resta ora il problema di aggiungere effettivamente gli spazi se possibile. Il meccanismo sviluppato può dover compiere uno o più passaggi attraverso i dati. Una volta considerata l'idea di scorrere la riga ed aggiungere gli spazi in accordo con la corrente ripartizione, vediamo che iniziano a sorgere i problemi. I problemi incominciano non appena esiste la possibilità che sia necessario più di un passaggio attraverso la riga per ottenere la distribuzione voluta ed a causarli è l'introduzione degli spazi multipli. Quali altre alternative abbiamo? Un approccio potrebbe essere quello di determinare dove debbano essere aggiunti *tutti* gli spazi extra *prima* di cercare di aggiungerli alla riga. In proposito servirebbe un array di lunghezza uguale al numero di posizioni della riga in cui devono venire aggiunti gli spazi. Potremo allora usare questo array per memorizzare il numero di spazi extra da aggiungere ad ogni spazio già esistente. L'idea migliore sarebbe di inserire nell'array i singoli spazi al momento del conteggio. Nel caso in cui dovevano essere inseriti 10 spazi extra a 7 posizioni esistenti, nel nostro array contatore di spazi avremmo trovato i seguenti valori:

1	2	3	4	5	6	7
2	3	2	3	2	3	2

Nel caso in cui occorre aggiungere solo tre spazi:

1	2	3	4	5	6	7
1	2	1	2	1	2	1

Le introduzioni possono essere fatte iterativamente sulla tabella per ogni ripartizione usando i valori *next* e *delta*, come stabilito precedentemente, cioè

1. finché non viene passato un fine – riga

- (a) aggiungi il blocco di spazi alla posizione corrente di ripartizione;
 - (b) spostati sulla posizione seguente della ripartizione.

Il ciclo può causare l'aggiunta di un numero di spazi maggiore di quello necessario. Deve essere quindi aggiunta una condizione che assicuri di non addizionare più spazi del necessario. Ciò sarà fatto decrementando e controllando il contatore degli spazi extra.

L'ultimo compito rimanente è quello di usare la tabella del contatore degli spazi o l'array per produrre il testo di uscita marginato. Seguendo il ragionamento precedente, bisognerà scandire le righe carattere per carattere. Ogni volta che nella riga viene incontrato uno spazio, dovreмо far riferimento alla tabella del contatore degli spazi per decidere quanti spazi dovranno essere aggiunti alla posizione corrente. Se stiamo effettivamente per scrivere una riga con i margini allineati in questo modo, ogni volta in cui viene esaminato un carattere ci possiamo trovare di fronte ad una delle due seguenti situazioni: se il carattere non è uno spazio, può essere copiato direttamente, altrimenti deve essere usata la tabella per decidere quanti spazi debbano essere aggiunti prima di passare alla parola successiva. L'indice dell'array dovrà essere incrementato di uno non appena viene incontrato un nuovo spazio. In questa procedura non abbiamo provveduto agli spazi lasciati all'inizio di un nuovo paragrafo. Ciò può essere trattato nel momento in cui sia stato fatto il conteggio degli spazi iniziali della riga. Se *start* è il punto in cui incomincia il primo carattere alfabetico della riga, possiamo usare:

Dall'inizio alla fine della riga,

- (a) se il prossimo carattere è uno spazio, allora
 - (a.1) passa alla prossima posizione nella tabella del contatore di spazi
 - (a.2) inserisci il numero di spazi contenuto nella tabella altrimenti
 - (a.1) copia il carattere corrente del testo.

Qui di seguito viene fornita la completa descrizione dell'algoritmo di allineamento dei margini.

Descrizione dell'algoritmo

1. Definisci la riga da marginare, la sua lunghezza corrente (vecchia) e quella dopo la marginatura (nuova).
2. Includi un controllo per vedere se deve essere marginato.
3. Inizializza il contatore degli spazi e l'inizio alfabetico della riga.

4. Finché il carattere corrente è uno spazio
 - (a) passa al carattere successivo;
 - (b) incrementa l'inizio alfabetico della riga;
 - (c) scrivi uno spazio.
5. Dall'inizio alfabetico alla fine della riga
 - (a) se il carattere corrente è uno spazio, allora
 - (a.1) incrementa il contatore degli spazi;
 - (a.2) assegna il valore 1 alla posizione corrente della tabella — contatore degli spazi.
6. Togli tutti gli spazi alla fine della riga.
7. Determina gli spazi extra che devono essere aggiunti alla nuova riga.
8. Finchè ci sono ancora spazi extra da aggiungere ed è possibile farlo
 - (a) calcola l'incremento corrente della ripartizione dal contatore degli spazi e da quello degli spazi extra (usa valori arrotondati);
 - (b) se l'incremento è zero, assegna il valore 1 poichè gli extra saranno maggiori degli spazi;
 - (c) se gli spazi extra sono maggiori del contatore degli spazi, allora
 - (c.1) determina il blocco degli spazi usando gli spazi extra ed il contatore degli spazi;
 - altrimenti
 - (c'.1) poni la dimensione del blocco di spazi uguale ad 1;
 - (d) determina la posizione iniziale per la ripartizione;
 - (e) finchè non arrivi a fine riga e ci sono ancora spazi extra
 - (e.1) somma il blocco degli spazi alla posizione corrente della ripartizione,
 - (e.2) passa alla posizione successiva nella ripartizione,
 - (e.3) decremente il contatore degli spazi extra del valore del blocco degli spazi.
9. Dall'inizio alla fine della riga
 - (a) se il prossimo carattere è uno spazio, allora
 - (a.1) passa alla posizione successiva della tabella contatore degli spazi
 - altrimenti
 - (a'.1) stampa il carattere corrente.
10. Termina con un fine — riga.

Implementazione in Pascal

```
procedure leftrightjustify(line: nchars; oldlength,newlength: integer);
const tsize = 40;
var delta {increment for current template},
    extraspaces {number of extra spaces to complete justification},
    j {index for writing out spaces},
    ispace {index for space count template},
```

```
nspaces {number of spaces in input line},
next {current position in template array},
position {current position in line},
start {position of first non-space in line},
spaceblock {number of spaces to be added to current template};
integer;
space {the space character}: char;
template: array[1..tsize] of integer;

begin {right and left justifies a line of text to newlength by blank fill}
if oldlength>newlength then
  writeln('line too long - cannot left and right justify')
else
  begin {line can be left and right justified}
    space := ' '; start := 1;
    while (line[start]=space) and (start<=oldlength) do
      begin {allow for new paragraph with leading blanks}
        start := start + 1;
        write(space)
      end;
    nspaces := 0;
    if start <= oldlength then
      while line[oldlength]=space do oldlength := oldlength - 1;
    {assert: all leading & all trailing spaces removed from line &
    nspaces = 0}
    {invariant: start <= position <= oldlength & after current
    iteration & nspaces equals count of all spaces in range
    line[1..position]}
    for position := start to oldlength do
      if line[position]=space then
        begin {count spaces and initialize space count template}
          nspaces := nspaces + 1;
          template[nspaces] := 1
        end;
    extraspaces := newlength - oldlength;
    {invariant: extraspaces = number of spaces still to be added to
    line}

    while (extraspaces>0) and (nspaces>0) do
      begin {set up template according to current space count}
        delta := round(nspaces / extraspaces);
        if delta = 0 then delta := 1;
        if extraspaces>nspaces then
          spaceblock := extraspaces div nspaces
        else
          spaceblock := 1;
        next := (delta + 1) div 2;
        {invariant: after current iteration extraspaces = number of
        spaces still to be added to line & spaceblock spaces to be
        added after (next - delta) word & 1 = < next = < nspaces +
        delta & delta >= 1}

        while (next<=nspaces) and (extraspaces>0) do
          begin {designate space position for current template
          size}
```

```

template[next] := template[next]+spaceblock;
next := next + delta;
extraspaces := extraspaces - spaceblock
end
end;
ispace := 0;

for position := start to oldlength do
if line[position]=space then
begin {write extra spaces if any at current location}
ispace := ispace + 1;
for j := 1 to template[ispace] do
write(space)
end
else
write(line[position]);
writeln
end
{assert: current line left and right justified to length newlength}
end

```

Note di progetto

- Le tre fasi di iterazione di questo algoritmo, il conteggio degli spazi, la costruzione della ripartizione e la stampa della riga del testo marginata, mostrano un funzionamento lineare. La fase di uscita che può dominare quando la lunghezza della riga di uscita è maggiore di quella in ingresso, è lineare rispetto alla lunghezza dell'output.
- Ci sono alcuni cicli di cui bisogna considerare il funzionamento. Il primo è il ciclo che legge gli spazi all'inizio del nuovo paragrafo. Dopo la prima iterazione di questo ciclo, saranno stati scritti i primi ($start - 1$) spazi della riga. Per il ciclo di conteggio degli spazi, dopo la prima iterazione, i primi $position$ spazi della riga saranno stati controllati. Anche dopo la corrente iterazione il numero degli spazi nei primi $position$ caratteri sarà $nspaces$ (esclusi gli spazi già inclusi). Le prime $nspaces$ posizioni dell'array *template* conterranno questi.
Dopo ogni iterazione del ciclo di costruzione della ripartizione degli spazi, il numero di spazi extra che dovranno ancora essere inseriti sarà *extraspaces*. La ripartizione con ampiezza di incremento *delta* sarà stata costruita alla fine della corrente iterazione. Sarà stata posizionata iniziando dallo spazio ($\delta + 1$) $\text{div } 2$. Dal ciclo più interno di costruzione del vettore dopo l'iterazione corrente, blocchi di spazi di dimensione *spaceblock* saranno stati aggiunti alle prime posizioni ($next - \delta$) del vettore in accordo al corrente incremento che inizia in posizione ($\delta + 1$) $\text{div } 2$. Il rimanente numero di spazi da inserire dopo la corrente iterazione sarà *extraspaces*.
Tenendo conto del ciclo *for* più esterno nella scrittura della riga allineata dopo la corrente iterazione i caratteri in prima posizione

nella vecchia riga saranno stati scritti tutti insieme agli spazi originali ed inseriti che dovevano essere scritti dopo le parole *ispace* della riga corrente. Dopo l'iterazione corrente del ciclo *for* più interno, saranno stati inseriti i primi j spazi dopo la parola *ispace* nella riga corrente.

Riguardo alla terminazione, il primo ciclo *while* termina poiché ad ogni iterazione sarà stato considerato un altro carattere della riga. Il ciclo *for* per il conteggio degli spazi terminerà per definizione. Il ciclo *while* più interno necessario per costruire il vettore termina poiché ad ogni iterazione *next* verrà incrementato di almeno un unità finché $\delta \geq 1$. Ad ogni iterazione del ciclo più esterno di costruzione del vettore, il ciclo più interno di costruzione del vettore, (che termina), sarà stato iniziato e questo riduce sempre gli spazi extra da aggiungere. Da ciò segue che il ciclo più esterno termina. Gli altri cicli terminano tutti per definizione di ciclo *for* e quindi l'intero algoritmo termina poiché terminano tutti i suoi cicli.

- In questo problema abbiamo visto come, quello che viene inizialmente considerato come caso particolare possa, con alcuni accurati progetti, essere incorporato nel meccanismo generale.
- Usando una tabella extra per memorizzare le posizioni degli spazi sarebbe possibile migliorare significativamente l'efficienza del ciclo di espansione.
- L'algoritmo tratta correttamente il caso in cui c'è solo una parola per riga.
- L'algoritmo si semplifica non addizionando gli spazi extra finché non sia conosciuta la loro completa distribuzione.
- È possibile progettare algoritmi più sofisticati di marginazione a sinistra ed a destra che producano un risultato esteticamente migliore.

Applicazioni

Preparazione di documenti — applicazione limitata.

Problemi supplementari

- Modificare l'attuale algoritmo in modo che mentre aderisca all'idea di base di distribuzione vettoriale, non aggiunga spazi extra multipli in una posizione prima che sia stato aggiunto uno spazio extra in ogni posizione.
- Incorporare il suggerimento della nota 4.
- Implementare una versione dell'algoritmo che ripassi la riga marginata alla procedura chiamante.
- Includere nel presente algoritmo i tests che assicurino che la linea non sia allargata se debbano essere aggiunti un numero maggiore del doppio degli spazi esistenti.

- 6.2.5 Progettare ed implementare un semplice algoritmo per la marginatura che non richieda l'uso di una tabella degli spazi.
- 6.2.6 Progettare ed implementare un algoritmo che rovesci il progetto di marginatura rimuovendo gli spazi multipli. I paragrafi devono però essere lasciati intatti.
- 6.2.7 Progettare ed implementare un algoritmo di allineamento dei margini a destra e sinistra che inserisca gli spazi extra in un primo tempo dopo la parola più lunga, in seguito dopo la seconda parola più lunga e così via. Nell'implementazione, facendo alcune supposizioni, cercate di evitare di fare un ordinamento. Questo approccio produce in genere un risultato esteticamente migliore.

ALGORITMO 6.3 RICERCA DI PAROLE CHIAVE IN UN TESTO

Problema

Trovare il numero di volte in cui una determinata parola ricorre in un testo assegnato.

Sviluppo dell'algoritmo.

Il problema di ricerca in un testo e le sue relative variazioni, sono comunemente incontrate in programmazione. Per fare un esempio, possiamo cercare di determinare se la parola SENTENCE ricorra o meno nel testo:

THIS IS A SENTENCE OF TEXT

Noi riusciamo a risolvere visivamente il problema in maniera molto semplice, senza renderci conto del meccanismo applicato. Per progettare un algoritmo per il calcolatore, dobbiamo cercare di rendere ciò più esplicito. Come punto di partenza dobbiamo in qualche modo confrontare i caratteri della parola con quelli del testo in esame.

Per esaminare un testo mediante il calcolatore abbiamo la limitazione di esaminare un carattere alla volta. Ci occorre quindi un modo per decidere quando è stato fatto un riscontro.

Prima di iniziare con questa strategia, occorre avere una chiara idea di quale sia esattamente il significato di parola e di testo. L'osservazione di una qualsiasi parte di un testo ci mostra che una parola è una sequenza consecutiva di caratteri alfabetici. Una parola in un testo può essere preceduta e seguita da uno o più caratteri non alfabetici (come spazi, punteggiature, caratteri di fine – riga, caratteri di fine – file). Un esame più dettagliato mostra che la prima parola è un caso particolare, poiché può o meno essere preceduta da uno o più caratteri non alfabetici.

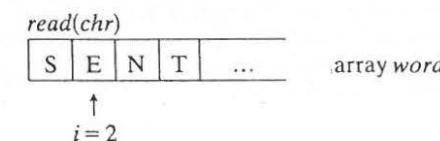
Siamo ora in una migliore condizione per definire cosa si intenda per uguaglianza di due parole. Due parole possono essere definite uguali, quando concordano carattere – per – carattere dal primo all'ultimo. Da ciò segue che se il numero di uguaglianze carattere – per – carattere è uguale alla lunghezza sia della parola cercata che di quella del testo, abbiamo un'identità. Ad esempio:

...S E N T E N C E ...	(parola del testo – lunghezza = 8)
↓ confronto ↓	
...S E N T E N C E ...	(parola cercata – lunghezza = 8)

Una *disuguaglianza* tra due parole significa quindi che le due parole non sono le stesse carattere – per – carattere o, in altre parole, che il numero di uguaglianze carattere – per – carattere è minore della lunghezza delle due parole. A questo punto dobbiamo stare un po' attenti perché ad esempio la parola SENT è contenuta nella parola SENTENCE, ma non possiamo dire che ci sia uguaglianza tra le due parole. Torneremo su questo problema più avanti. Dopo aver definito le grandezze fondamentali di questo problema, dobbiamo ora trovare il modo di implementare l'algoritmo di ricerca nel testo.

Poichè dobbiamo ogni volta essere in grado di accedere alla parola che stiamo cercando nel testo, la possiamo memorizzare in un array. Durante la ricerca della parola SENTENCE, essa può essere memorizzata nelle prime otto posizioni di un array chiamato word. Il nostro problema sarà ora quello di esaminare tutte le parole del testo e vedere se ognuna a turno uguaglia la parola SENTENCE. Questo può essere fatto esaminando la prima parola del testo, poi la seconda e così via finché non si arriva alla fine del testo.

Consideriamo per primo il caso in cui ci sia uguaglianza tra il primo carattere del testo e della parola. Il passo successivo può essere quello di confrontare il secondo carattere del testo con quello della parola (occorre tener presente che il nostro algoritmo tratterà anche parole di un solo carattere). Per fare ciò possiamo leggere il carattere successivo e spostare il puntatore i dell'array word in seconda posizione. Ad esempio abbiamo:



Dobbiamo ora confrontare il *chr* appena letto con *word*[2]. Se il carattere *chr* è una E il processo di confronto può proseguire con l'incremento di *i* e la lettura di un nuovo *chr* e così via. Se si raggiunge il valore 9 (per l'esempio precedente), sapremo che è stata trovata una parola uguale a quella data.

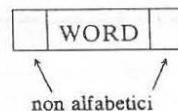
Una volta stabilito che l'elemento cercato è stato completamente identificato, dobbiamo procedere ulteriormente per decidere se l'elemento è una parola o la parte di una parola. Ricordando le precedenti definizioni, basta controllare se l'elemento sia o meno preceduto e seguito da un carattere non alfabetico.

La parte centrale della strategia risulta:

1. Assegna ad *i* la prima posizione di *word*.
2. Finché non si raggiunge il fine – file
 - (a) leggi il prossimo carattere del testo;
 - (b) se il carattere corrente del testo uguaglia l'*i*-esimo carattere della parola, allora
 - (b.1) incrementa *i* di un'unità;
 - (b.2) se l'elemento è uguale, allora vedi se è un'identità di parole e compi le azioni appropriate
 - altrimenti occupati della situazione di disuguaglianza.

Nella lettura del testo bisogna tener conto dei fine – riga.

Siamo arrivati ai problemi del controllo della condizione di identità della parola e del trattamento delle disuguaglianze. Dalla trattazione precedente, sappiamo che, in generale, le parole sono precedute e seguite da caratteri non alfabetici.



È abbastanza facile, dopo che è stata trovata l'identità di due elementi, stabilire se il carattere successivo è alfabetico, semplicemente facendo il test:

chr in *alphabet*

dove in *alphabet* è stato posto l'insieme dei caratteri alfabetici.

Il problema di controllare il carattere che precede l'elemento corrente, non è così facile. In qualche modo bisogna trovare un accorgimento per ricordare il carattere che precede l'elemento. I caratteri candidati a precedere una parola sono solo quelli che non uguagliono il carattere corrente dell'array, cioè sono quelli trovati in una situazione di disuguaglianza. Ogni volta che troviamo una disuguaglianza, dovremmo salvare il carattere corrente del testo nel caso sia necessario

decidere in seguito se ci sia stata o meno un'uguaglianza tra parole. Il carattere salvato non verrà modificato durante un confronto di elementi, poiché ciò sarà fatto solo in caso di disuguaglianza.

Quando si trova una disuguaglianza, occorre riposizionare il puntatore dell'array *word* sul primo carattere. I passi che devono essere fatti quando si ha una disuguaglianza sono quindi:

altrimenti (disuguaglianza)

- (a) salva il carattere corrente del testo in *pre*,
- (b) posiziona il puntatore all'array *word* sul primo carattere.

Per considerare il caso in cui la prima parola non sia preceduta da un carattere non – alfabetico, dobbiamo inizializzare la variabile *pre* ad un carattere non – alfabetico. Anche il puntatore all'array deve essere inizializzato in questo modo.

Nella nostra discussione non ci siamo addentrati in ciò che deve essere fatto quando si trova un'identità di parole. Risulta che questa situazione è molto simile al caso della disuguaglianza. Tuttavia ora il carattere che segue la parola confrontata deve assumere il ruolo del carattere che precede la parola successiva. Il puntatore all'array *word* dovrà essere riposizionato in prima posizione e infine il contatore di uguaglianza dovrà essere aggiornato. Abbiamo quindi:

se c'è uguaglianza, allora

- (a) incrementa il contatore di uguaglianza di un'unità,
- (b) assegna a *pre* il carattere letto più di recente,
- (c) posiziona il puntatore *i* sul primo elemento dell'array *word*.

Può essere ora implementato l'algoritmo di ricerca di parole.

Descrizione dell'algoritmo

1. Definisci la parola e la lunghezza, *wlength* della parola da ricerare.
2. Inizializza il contatore delle uguaglianze *nmatches*, assegna il carattere precedente ed assegna al puntatore *i* dell'array *word*, il valore 1.
3. Finché non arrivi al fine – file
 - (a) finché non arrivi al fine – riga
 - (a.1) leggi il prossimo carattere;
 - (a.2) se il carattere corrente del testo *chr* è uguale all'*i*-esimo carattere della parola, allora
 - (2.a) incrementa *i* di uno,
 - (2.b) se c'è un'identità parola-elemento, allora
 - (b.1) leggi il prossimo carattere *post*
 - (b.2) se i caratteri precedenti e seguenti non sono alfabetici, allora
 - (2.a) aggiorna il contatore *nmatches*

- (b.3) riinizializza il puntatore *i* dell'array *word*
 - (b.4) salva il carattere successivo *post* come carattere precedente
- altrimenti
- (2'.a) salva il carattere corrente del testo come carattere precedente per il confronto,
 - (2'.b) rimetti il puntatore dell'array *word* in prima posizione;
- (b) leggi oltre il fine – riga.
4. Ritorna il contatore di parole uguali *nmatches*.

L'implementazione è stata progettata per trattare solo caratteri alfabetici minuscoli.

Implementazione in Pascal

```

procedure wordsearch(word: nchars; wlength: integer; var
nmatches: integer);
type letters = 'a'..'z';
var i {position in search word array}: integer;
    chr {current text character},
    pre {candidate for character preceding current word},
    post {candidate for character following current word}: char;
    alphabet: set of letters;
begin {counts the number of times a word occurs in a given text}
{assert: wlength >0}
alphabet := ['a'..'z'];
pre := ' '; i := 1;
while not eof(input) do
begin {read and process next character}
{invariant: (i=<wlength  $\wedge$  last(i-1) characters read in text
match word [1..i-1])  $\vee$  (a wordmatch  $\wedge$  i=1)  $\wedge$ 
nmatches = number of word matches in first (i-1) characters
read}
while not eoln(input) do
begin
    read(chr);
    if chr=word[i] then
        begin {see if partial match is now complete}
            i := i + 1;
            if i>wlength then
                begin
                    read(post);
                    if (not (pre in alphabet)) and (not (post in
alphabet)) then
                        begin {a word match}
                            nmatches := nmatches + 1;
                        end
                    i := 1;
                    pre := post
                end
            end
        else
end
end
end

```

```

begin {a mismatch}
pre := chr;
i := 1
end
end;
readln;
end
{assert: nmatches = number of times word found in text}
end

```

Note di progetto

1. L'istruzione principale di questo algoritmo è il confronto del carattere corrente del testo con quello della parola. Il numero di passaggi attraverso questo test è uguale al numero di caratteri cercati nel testo.
2. Al punto della ricerca in cui sono già stati letti ed elaborati i primi *j* caratteri saranno state conteggiate tutte le parole del testo che coincidono completamente con quella cercata nei primi *j* caratteri. Dopo ogni iterazione, il valore della variabile *i* indicherà che i precedenti *i* – 1 caratteri del testo uguaglano i primi *i* – 1 caratteri della parola cercata se siamo in una situazione di parziale identità. Ad ogni iterazione la variabile *pre* è assegnata al carattere che è potenzialmente un carattere precedente alla successiva parola da esaminare.
Alla fine dell'iterazione in corso, la variabile *post* è o indefinita, o contiene il carattere direttamente successivo ad un'uguaglianza. L'algoritmo termina perché ad ogni iterazione viene letto un nuovo carattere e così comunque viene raggiunta la fine del file.
3. Il presente algoritmo usa un meccanismo molto semplice nella parte principale della computazione.
4. L'algoritmo può essere usato nella ricerca per prefissi e suffissi di parole ma non tratta il problema generale della ricerca di stringhe, anche se è basato su un confronto degli elementi che precedono la parola.
L'algoritmo, ad esempio, non registrerà un'uguaglianza nella ricerca dell'elemento *abcabdacb* nel testo *abcabcabdacb*.
5. A livello di costo, abbiamo sviluppato un algoritmo con funzionamento lineare nel caso peggiore, rispetto al comportamento quadratico che è possibile nel caso peggiore del corrispondente algoritmo di identificazione di elementi.
6. Nella ricerca della risoluzione del problema, è utile incominciare considerando il problema più piccolo, quello della ricerca di una sola parola nel testo. Nell'implementazione, abbiamo contato sul fatto che il Pascal ritorna uno spazio bianco quando viene incontrato un carattere di fine – riga.
7. È possibile implementare algoritmi più efficienti per risolvere sia il problema corrente che problemi più generali di riconoscimento di elementi.

Applicazioni

Ricerca in testi limitati.

Problemi supplementari

- 6.3.1 Modificare questo algoritmo in modo che termini dopo aver trovato la prima parola uguale completa.
- 6.3.2 Progettare ed implementare un algoritmo di ricerca di parole che dopo aver trovato una disuguaglianza nella parola corrente legga i caratteri all'inizio della parola seguente, prima di trovare una nuova uguaglianza.
- 6.3.3 Progettare ed implementare un algoritmo che ricerchi un testo e salvi la parola che abbia ottenuto il migliore confronto parziale (oltre alla totale uguaglianza) con la parola cercata.
- 6.3.4 Progettare ed implementare un algoritmo che stampi una lista di tutte le parole di un testo che contengano la parola cercata come prefisso.

ALGORITMO 6.4 EDITAZIONE DI UNA LINEA DI TESTO

Problema

Progettare ed implementare un algoritmo che ricerchi una linea di testo individuata da un particolare elemento o sottostringa. Una volta trovato l'elemento, esso deve essere sostituito con un altro dato.

Sviluppo dell'algoritmo

La necessità di sostituire una stringa con un'altra ricorre molto frequentemente nella preparazione di programmi e documentazioni. Visto al livello più semplice, il problema è sostituire *tutte* le ripetizioni di un particolare elemento su una riga con un altro. Il meccanismo per la soluzione manuale di un tale problema è semplice, ad esempio

supponiamo di voler sostituire *wrong* con *right* nella linea successiva:

the two *wrongs* in this line are *wrong* (originale)

Tutto quello che dobbiamo fare è esaminare la riga, trovare l'elemento *wrong* da cambiare e sostituirlo con il nuovo elemento *right*. Per il nostro esempio, avremo:

the two *rights* in this line are *right* (riga stampata)

Un'indagine preliminare suggerisce che ci sono due fasi nel problema dell'editing di un testo. Il primo stadio implica la localizzazione dell'elemento che deve essere sostituito, il secondo considera la vera e propria *sostituzione* con il nuovo elemento.

Il compito sarebbe più semplice se, nel progettare il nostro algoritmo, ci avvantaggiassimo del modo naturale in cui si divide il problema. È meglio, quindi, lavorare inizialmente sulla ricerca. Nel caso precedente sulla ricerca di chiavi in un testo si era giunti alla conclusione che il metodo di ricerca proposto non avrebbe trattato casi come la ricerca dell'elemento *abcabdabc* nel testo *abcabcaabdabc*. Nel nostro algoritmo di editing di un testo, dobbiamo essere in grado di trattare tutti i casi. Un buon metodo per iniziare l'algoritmo potrebbe essere con un esame del problema che ci siamo appena posti.

Abbiamo:

abcabdabc	(elemento)
abcabcaabdabc	(testo)
↑	
disuguaglianza	

Il nostro vecchio algoritmo di ricerca, confrontato con questo problema, procederebbe con il confronto del sesto carattere del testo e dell'elemento. Quindi, avendo incontrato una disuguaglianza, continuerebbe con il confronto del settimo carattere del testo con il primo dell'elemento. Nell'adottare questa strategia, non esiste nell'attuale esempio la possibilità di trovare un'identità. Si può vedere che il problema sorge poiché, dopo aver trovato la disuguaglianza, abbiamo proceduto con il confronto del *primo* carattere dell'elemento con il *settimo* del testo. Se, invece, avessimo rimediato alla disuguaglianza attraverso il confronto del *primo* carattere dell'elemento con il *secondo* del testo e così via, non sarebbe stata persa la uguaglianza richiesta.

Il meccanismo che stiamo cercando di implementare può essere pensato come *posizionamento dell'inizio del campione sul primo carattere del testo, sul secondo e così via*. In ogni posizione dovrà essere determinato il grado di identità tra campione ed il testo. In questo modo potrà essere evitato il rischio di perdere un'uguaglianza. Nell'esempio precedente ci sono solo quattro posizioni in cui possa essere individuato l'elemento relativamente al testo. Essi sono:

Posizione	testo	
(1)	abcabca	(testo)
(2)	bcabdabc	disuguaglianza
(3)	abcabdabc	disuguaglianza
(4)	abcabdabc	disuguaglianza

La parte centrale della strategia di ricerca dell'elemento è quindi:

finchè ci sono ancora posizioni nel testo che potrebbero contenere l'elemento

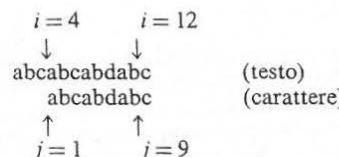
- (a) "posiziona" l'elemento nella prossima posizione del testo;
- (b) guarda se è verificata una completa uguaglianza nella posizione corrente del testo;

Dal modo in cui interagiscono la lunghezza dell'elemento $patlength$ e l'ampiezza del testo, $textlength$, possiamo concludere che ci sono:

$$textlength - patlength + 1$$

posizioni in cui può essere posto l'elemento nel testo.

Dopo aver posizionato l'elemento in una data posizione i del testo, la mossa successiva è quella di esaminare l'estensione dell'identità nella particolare posizione i del testo. Per fare ciò dobbiamo essere in grado di passare attraverso i caratteri consecutivi dell'elemento e, allo stesso tempo, di proseguire nel controllo del testo. Per cercare di stabilire il meccanismo di confronto richiesto, torniamo all'esempio precedente e consideriamo il caso in cui l'elemento sia posizionato col primo carattere nella quarta posizione del testo. La variabile j può essere usata per caratterizzare l'estensione del confronto con esito positivo.



Nella determinazione dell'estensione dell'uguaglianza in quarta posizione, vogliamo che entrambi gli indici del testo e dell'elemento, seguano le successioni:

testo	elemento	spostamento
4	1	3
5	2	3
6	3	3
.	.	.
.	.	.
12	9	3

Nel suo svolgimento, la sequenza del testo viene spostata di $i - 1$ relativamente alla sequenza dell'elemento. È possibile adattare questo spostamento al nostro confronto usando:

`if pattern[j] = txt[i + j - 1] then "estendi il confronto"`

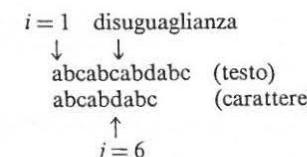
L'incremento di j servirà a proseguire il confronto. Infatti compirà il lavoro un ciclo che si estenda oltre la lunghezza dell'elemento ma si fermi in corrispondenza di una disuguaglianza.

Abbiamo ora un metodo per confrontare l'elemento ed il testo ma non siamo ancora in grado di decidere quando sia stato fatto un confronto completo o cosa fare quando viene rilevata una disuguaglianza. Riferendoci ancora al nostro esempio, vediamo che il modo più semplice per fare un test di confronto è di controllare se j sia o meno andato al di là della lunghezza dell'elemento (nell'esempio precedente, se j avesse raggiunto 10 avremmo saputo che il confronto era stato completato con esito positivo nella posizione corrente del testo). Il test da applicare sarebbe:

`se $j > patlength$ allora`

"confronto completato, quindi passa all'operazione di stampa"

La situazione di disuguaglianza, che è probabilmente la situazione più comune, chiaramente non richiede che sia fatta alcuna operazione di editing. Nel nostro esempio, l'esame della prima posizione dell'elemento chiarisce cosa debba essere fatto in una situazione di disuguaglianza.



In questo esempio, prima di poter testare l'uguaglianza dell'inizio di un elemento situato in posizione 2, il puntatore all'elemento j deve essere resettato al valore uno.

Questa indagine più recente, suggerisce che il meccanismo di confronto è più semplice di quello che avevamo originalmente presunto. Ogni volta che confrontiamo un carattere di un testo con uno di un elemento, prevale una delle due situazioni — *la coppia di caratteri è uguale o è diversa*. Se c'è un'uguaglianza allora siamo obbligati a fare un test addizionale per vedere se abbiamo riconosciuto un elemento completo. Una volta visto il problema sotto questo aspetto, scompare la necessità di un ciclo separato per i confronti. La strategia centrale di ricerca si è quindi evoluta come segue:

```

finchè  $i \leq textlength - patlength + 1$ 
(a) se  $pattern[j] = txt[i + j - 1]$  allora
    (a.1) incrementa  $i$  di 1,
    (a.2) se l'identità è completa allora esegui l'editazione
        altrimenti
            (a'.1) reinizializza il puntatore  $j$  all'elemento,
            (a'.2) muovi l'elemento nella posizione successiva del testo incrementando  $i$ .

```

Abbiamo ora raggiunto il punto in cui potremo cercare e scoprire un'uguaglianza dell'elemento nel testo. Il problema rimanente è quello di formulare il passo di editazione.

Per cominciare questo problema, ritorniamo al nostro esempio originale in cui avevamo sostituito *wrong* con *right*. È chiaro che in generale la linea edita sarà costruita con parti comuni alla riga originale. Nell'esempio le parti comuni sono sottolineate.

the two wrongs in this line are wrong (originale)
the two rights in this line are right (editata)

In generale, non ci possiamo aspettare che la linea in ingresso e quella editata abbiano la stessa lunghezza poiché l'elemento originale e quello sostituito possono essere di lunghezza diversa. Questo suggerisce che sarà probabilmente più semplice creare una nuova copia delle parti comuni nella produzione della linea edita. Il passo di editazione implicherà una sequenza di passi di *copiatura* e di *sostituzione di elementi*, con la sostituzione che ha luogo quando l'algoritmo di ricerca trova una completa identità, mentre la copiatura viene fatta negli altri casi. Così, nel produrre la linea editata dobbiamo copiare sia dalla riga originale che dall'elemento sostituito. La copiatura dalla riga originale sarà quindi necessaria per procedere via via nella ricerca. La domanda a cui si deve ora rispondere è, come è possibile integrare l'operazione di copiatura nella ricerca? È evidente che la copiatura può aver luogo ogni volta che viene trovata una disuguaglianza ed un conseguente spostamento dell'elemento relativamente al testo. Nel caso in cui siano stati trovati due caratteri uguali, non è così evidente il da farsi. Esaminando più accuratamente questa situazione, vediamo che i verrà incrementato solo quando è stata riscontrata una totale identità e quindi in una situazione di parziale uguaglianza non è necessaria alcuna copiatura. Una uguaglianza completa segnala la necessità di copiare non dalla riga originale, ma dall'elemento da sostituire. Le due situazioni di copiatura di cui ci dobbiamo occupare sono:

- (a) Se trovi una disuguaglianza, copia dalla riga originale.
- (b) Se trovi una totale uguaglianza copia dal nuovo elemento.

Consideriamo per prima la situazione di disuguaglianza. Una proposta per la copiatura potrebbe essere:

$newtext[i] := txt[i]$

Questo, tuttavia, non considera il fatto che la nuova riga crescerà in proporzione diversa se il vecchio ed il nuovo elemento non hanno la stessa lunghezza. La variabile di posizione dell'elemento i sarà ancora adatta, ma sarà anche necessaria una nuova variabile k per la riga da editare che può crescere con un diverso andamento. La copiatura in situazione di disuguaglianza avrà ora la forma:

$newtext[k] := txt[i]$

Quando incontriamo una completa uguaglianza dobbiamo copiare un elemento completo invece di un solo carattere, come accade nella situazione di editazione. Il nuovo elemento *newpattern* deve essere inserito nella posizione immediatamente successiva a quella dell'ultimo carattere della riga del testo (cioè dopo la posizione k). Poiché deve essere copiato un certo numero di caratteri, il modo migliore per farlo è con un ciclo:

```

for  $l := 1$  to  $newpatlength$  do
begin
     $k := k + 1$ ;
     $newtext[k] := newpattern[l]$ 
end

```

Una volta copiato l'elemento, dobbiamo saltare la posizione del testo occupata dal vecchio elemento. Possiamo fare ciò aumentando i della vecchia lunghezza dell'elemento:

$i := i + patlength$

A questo punto dobbiamo anche riposizionare il puntatore all'elemento cercato.

Abbiamo stabilito un meccanismo di editazione che sostituisca un elemento in posizione qualunque ed un numero qualsiasi di volte nella riga da editare. Abbiamo anche costruito un meccanismo per copiare dalla riga originale del testo. Da un esame più accurato di questo meccanismo vediamo che esso può trascurare la copiatura degli ultimi caratteri del testo a causa del valore minore dell'indice di posizione rispetto al numero di caratteri del testo originale. Dobbiamo, quindi inserire dei passi per copiare questi caratteri "lasciati indietro". Ovvvero:

```

while  $i \leq textlength$  do
begin
     $k := k + 1$ ;
     $newtext[k] := txt[i]$ ;
     $i := i + 1$ 
end

```

Una volta incorporati questi requisiti nello schema di ricerca di elementi avremo completato l'algoritmo.

Descrizione dell'algoritmo

1. Assegna la riga di testo, l'elemento da cercare, quello da sostituire e le loro lunghezze in caratteri.
2. Assegna i valori iniziali di posizione del testo vecchio, del nuovo e dell'elemento da cercare.
3. Finché non sono state esaminate tutte le posizioni dell'elemento nel testo
 - (a) se i caratteri correnti del testo e dell'elemento sono uguali, allora
 - (a.1) estendi gli indici sulla coppia successiva di caratteri del testo/elemento
 - (a.2) se l'uguaglianza è completa, allora
 - (2.a) copia il nuovo elemento nella posizione corrente della riga edita,
 - (2.b) "salta" il vecchio elemento del testo,
 - (2.c) riposiziona il puntatore all'elemento cercato
 - altrimenti
 - (a'.1) copia il carattere corrente del testo nella posizione successiva del testo edito,
 - (a'.2) riposiziona il puntatore all'elemento cercato
 - (a'.3) muovi l'elemento nella posizione successiva del testo.
4. Copia i caratteri lasciati indietro nella riga originale del testo.
5. Ritorna la riga edita di testo.

Implementazione in Pascal

```

procedure textedit (var text,newtext: nchars; var pattern,newpattern:
nchars; var newtextlength: integer; textlength,patlength,newpatlength:
integer);
var i {position of start of search pattern in text};
    j {pointer for search pattern and displacement for current text
position},
    k {current number of characters in the edited text line},
    l {index for newpattern}: integer;

begin {searches for and replaces pattern by newpattern in text
returning edited line newtext}
{assert: textlength,patlength,newtextlength,newpatlength > 0}
i := 1; j := 1; k := 0;
{invariant: i = <= j = <= patlength + 1 ^ newtext[1..k] established ^
((j = <= patlength ^ last j - 1 characters in text match pattern[1..j - 1])
V (j = 1 ^ text[i..i + patlength] replaced by newpattern in newtext))}

while i <= textlength - patlength + 1 do
begin
  if text[i+j-1]=pattern[j] then
    begin{a pattern and text character match}
      j := j + 1;
    if j>patlength then
      begin {a complete pattern match has been made}
        for l := 1 to newpatlength do

```

```

          begin {copy new pattern to edited line}
            k := k + 1;
            newtext[k] := newpattern[l];
          end;
          i := i + patlength;
          j := 1
        end
      end
    else
      begin {a mismatch so copy current text character to edited
line and reset}
        k := k + 1;
        nextext[k] := text[i];
        i := i + 1;
        j := 1
      end
    end;
  while i <= textlength do
  begin {copy left over characters from text to edited line}
    k := k + 1;
    newtext[k] := text[i];
    i := i + 1
  end;
{assert: newtext[1..k] represents edited text with all
non-overlapping occurrences of pattern replaced by newpattern}
newtextlength := k
end

```

Note di progetto

1. L'istruzione principale di questo algoritmo è il confronto del carattere corrente del testo con il carattere corrente dell'elemento. L'algoritmo di ricerca di elementi, da solo, ha potenzialmente funzionamento quadratico nel caso peggiore, poiché il numero di confronti è di poco minore del prodotto delle lunghezze dell'elemento e del testo. Il presente algoritmo compie un numero minore di confronti a causa dello spostamento fatto quando viene trovata un'uguaglianza. La situazione peggiore per una data lunghezza di un elemento avviene quando non viene trovata una completa identità, ma un massimo numero di uguaglianze parziali (ad esempio, supponiamo di cercare *aab* nel testo *aaaaaaaa...*). Il funzionamento medio dell'algoritmo potrebbe essere vicino a lineare rispetto alla lunghezza del testo.
2. Al punto della ricerca in cui l'elemento è stato posto in corrispondenza delle prime *i* posizioni del testo, saranno stati trovati e sostituiti dal nuovo elemento tutti gli elementi del testo nelle prime *i* posizioni che uguaglano completamente l'elemento cercato. Dopo ogni iterazione il valore della variabile *j* indicherà che i precedenti *j* - 1 caratteri del testo cominciante da *i* uguaglano i primi *j* - 1 caratteri dell'elemento cercato nel caso in cui prevalga una situazione di parziale uguaglianza o disuguaglianza. Dopo ogni iterazione in cui sia stata riscontrata l'uguaglianza, *j* verrà posto uguale ad

1. Ad ogni iterazione la variabile k rappresenta il numero di caratteri della riga editata di testo. Al termine di ogni iterazione, i è rimasto costante e j è aumentato, oppure i è aumentato e $j = 1$. La variabile j può aumentare con i costante solo finché i non supera il valore $patlength$, per cui i viene incrementato. Così, sebbene la variabile del ciclo i non aumenti ad ogni iterazione, ci sarà sempre soltanto un numero finito ($patlength$) di iterazioni non produttive prima che i venga incrementato verso il limite di terminazione. Da ciò segue che il ciclo principale termina così come i due cicli minori di copiatura.
3. Questo algoritmo tratta il caso in cui l'elemento sia più lungo del testo.
4. Il progetto si semplifica se si considera il problema di porre l'elemento in una sequenza di posizioni del testo.
5. Nell'editing della linea di testo, abbiamo supposto che le sovrapposizioni non debbano essere considerate, saltando il testo confrontato quando viene fatto un edit. In alcune applicazioni questa ammissione può non essere appropriata.
6. Come vedremo più avanti nel capitolo, esistono algoritmi di ricerca di elementi molto più efficienti.

Applicazioni

Editazione limitata di un testo.

Problemi supplementari

- 6.4.1 Implementare una versione del corrente algoritmo di ricerca che conti il numero di volte che un dato elemento ricorre in un testo. La vostra implementazione può provvedere al fatto che l'elemento cercato può avere sottosegmenti che si ripetono.
- 6.4.2 Progettare ed implementare un algoritmo che cancelli tutte le ripetizioni di un elemento particolare da un testo.
- 6.4.3 Usando un testo ed elementi in lingua Inglese determinare l'andamento medio dell'algoritmo di ricerca di elementi 6.4.1.
- 6.4.4 Cercare di implementare un algoritmo di ricerca di elementi che abbia andamento in caso peggiore lineare invece di quadrattico.
- 6.4.5 Dati due insiemi ordinati di numeri A e B, determinare se l'insieme A è o meno contenuto in B.
- 6.4.6 Implementare un algoritmo che stampi tutti gli elementi che siano della stessa lunghezza dell'elemento cercato, ma che differiscono dall'elemento al massimo di un carattere.

ALGORITMO 6.5 RICERCA LINEARE DI UN ELEMENTO

Problema

Progettare ed implementare un algoritmo di ricerca di un elemento con un procedimento che sia linearmente dipendente dalla lunghezza della stringa o del testo cercato. Può essere fatto il conteggio del numero delle volte che l'elemento cercato ricorre nella stringa.

Sviluppo dell'algoritmo

Abbiamo già visto un semplice algoritmo di confronto di elementi usato nel problema dell'editing di un testo (algoritmo 6.4). Il semplice algoritmo rivela un funzionamento vicino a quello lineare in applicazioni quali l'esame di un testo in Inglese. Questo significa che il numero di confronti fatti sui caratteri è di poco maggiore della lunghezza del testo. L'efficienza di questo algoritmo può, tuttavia diminuire seriamente in altre applicazioni in cui vengano considerati alfabeti più corti; e, nel caso peggiore possibile, il numero di confronti di caratteri sarà solamente di poco inferiore al prodotto delle lunghezze della stringa e dell'elemento cercato. È quindi desiderabile avere un algoritmo di ricerca con un andamento che sia sempre linearmente dipendente dalla lunghezza della stringa, senza che importi la dimensione dell'alfabeto.

Per iniziare questo più efficiente algoritmo di ricerca, esaminiamo più dettagliatamente il semplice algoritmo di uguaglianza di elementi.

abcabdabc	(elemento cercato)
abcabcabdabc...	(stringa su cui cercare)
↑	
disuguaglianza	

Nel nostro esempio, il confronto tra l'elemento e la stringa procede finché non troviamo la disuguaglianza tra la **d** e la **c** in sesta posizione. Quando ciò accade nell'algoritmo precedente, il processo riprende con il confronto del secondo carattere della stringa con il primo carattere dell'elemento. Nel fare questo passo dobbiamo *riesaminare dei caratteri della stringa che avevamo già visto prima*. È questo continuo riesame dei caratteri della stringa che contribuisce all'inefficienza del semplice algoritmo di ricerca di elementi. Il nostro compito è quello di costruire un algoritmo di ricerca di elementi tale che non sia necessario riesaminare i caratteri precedenti di una stringa quando viene rilevata una disuguaglianza.

Nel condurre la semplice ricerca dell'elemento, l'idea era quella di "mettere" l'elemento cercato in posizioni consecutive del testo e di stabilire la dimensione dell'uguaglianza. Riferendoci all'esempio, possiamo vedere che il problema sorge quando viene trovata una uguaglianza parziale. Se non vogliamo riesaminare tutti i caratteri precedenti del testo dobbiamo trovare il modo di usare le informazioni ottenute prima della disuguaglianza. Controlliamo ancora più dettagliatamente il nostro esempio per cercare di scoprire se esiste un modo di agire alternativo dopo la disuguaglianza in sesta posizione.

abcabdabc	(elemento cercato)
abcabca...babc...	(stringa su cui cercare)
↑	
	disuguaglianza

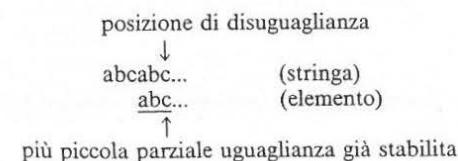
Per compiere questa esplorazione, possiamo porre l'elemento in posizioni successive relativamente alla stringa. Ad esempio:

Posizione	posizione diversa	
↓		
(2)	abcabc...	(stringa)
(3)	abcab...	(elemento)
(4)	abca...	(elemento)
	abc...	(elemento)

Dall'esempio, per le posizioni già esaminate per la parziale uguaglianza nella prima posizione della stringa, si può vedere che la seconda e la terza posizione non offrono possibilità di uguaglianza. La quarta posizione offre la possibilità di uguaglianza perché il quarto carattere della stringa uguaglia il primo carattere dell'elemento e così via. Dopo aver scoperto che l'elemento è in grado di stabilire un'uguaglianza a partire da una posizione (la quarta), precedente a quella della disuguaglianza (la sesta), possiamo cercare di capire che cosa significhi ciò. Esaminando più da vicino la stringa campione, vediamo che la ragione per cui una uguaglianza può incominciare *prima* della posizione diversa è perché l'*inizio* della stringa è ripetuto prima della disuguaglianza. È questa parziale identità con la parte ripetuta dell'elemento che ci consente di non ricominciare la ricerca subito dopo la disuguaglianza confrontando il primo carattere dell'elemento con il settimo della stringa. Dobbiamo ora chiederci, cosa abbiamo imparato da questa osservazione che ci permetta di non riesaminare ognuno dei precedenti caratteri della stringa?

Se vogliamo evitare ogni riesame dei caratteri precedenti della stringa allora, direttamente dopo la disuguaglianza al sesto carattere, dovranno essere considerati il sesto o il settimo carattere. Abbiamo inoltre stabilito che non è adatto confrontare il primo carattere dell'elemento con il settimo della stringa, poiché ciò potrebbe farci perdere un'uguaglianza. Ci chiediamo ora se esista qualche soluzione a questo dilemma. Esaminando ancora una volta il nostro esempio con l'elemento collocato in diverse posizioni, vediamo che sarebbe conveniente confrontare il terzo carattere dell'elemento con il sesto della stringa

poiché non verrebbe introdotto il rischio di perdere una possibile uguaglianza con l'elemento collocato con l'inizio al quarto carattere.



Nel fare questo nuovo confronto abbiamo preso atto della *più piccola* parziale uguaglianza che esiste tra l'elemento e la stringa quando viene trovata la disuguaglianza al sesto carattere della stringa. Nell'adottare questa strategia, siamo stati in grado di evitare di riguardare i caratteri della stringa che precedono quello diverso.

Sappiamo ora di poter evitare di tornare indietro nella stringa, annotando la più piccola parziale uguaglianza esistente al momento dell'incontro di due caratteri diversi. Il compito successivo è di rifinire e sviluppare questa idea. Nell'esempio precedente, la più piccola uguaglianza parziale esistente quando si trova la disuguaglianza è *ab*. Se stiamo per utilizzare il concetto appena sviluppato, ci occorre un modo per trovare i piccoli gruppi di elementi uguali (se esistono) in ogni posizione possibile dell'elemento prima di una disuguaglianza. Il nostro metodo di sviluppo deve essere indipendente dalla stringa su cui effettuare la ricerca, quindi occorrerà stabilire queste uguaglianze parziali dall'elemento stesso. Per trovarle, consideriamo più da vicino un altro elemento

abcabcacab

Se troviamo una disuguaglianza tra il *primo* carattere dell'elemento ed il carattere corrente della stringa, possiamo confrontare il primo carattere dell'elemento con il carattere *successivo* della stringa (Fig. 6.1(a)). Se, però, il primo carattere dell'elemento è uguale a quello della stringa ed è seguito da una disuguaglianza, (fig. 6.1(b)), possiamo confrontare ancora una volta il primo carattere dell'elemento con il successivo carattere della stringa.

Come ulteriore esempio, supponiamo che i primi cinque caratteri dell'elemento siano uguali a quelli della stringa, e ci sia una disuguaglianza al sesto carattere. (Fig. 6.1(c)).

In quest'ultima situazione, dopo aver preso in esame la disuguaglianza al sesto carattere, l'estensione dell'uguaglianza sarà di due caratteri, per il modo in cui *ab* è ripetuto nell'elemento. Ne segue che possiamo continuare confrontando il terzo carattere dell'elemento con il sesto (come mostrato) della stringa. Si noti che in questo caso non abbiamo proseguito nella stringa al momento di fare il confronto successivo.

La tabella 6.1 fornisce le più piccole uguaglianze parziali che esistono quando viene trovata una disuguaglianza.

situazione corrente

a b c a b c a c a b ... b a ... ↑ disuguaglianza	(elemento) (stringa)
-----------------------------------------------------------	-------------------------

passo successivo

a b c a b c a c a b ... b a ...	(elemento shiftato di una posizione) (stringa)
------------------------------------	---------------------------------------------------

(a) Disuguaglianza al primo carattere dell'elemento.

situazione corrente

a b c a b c a c a b a * a ... ↑ disuguaglianza	(elemento) (stringa)
---------------------------------------------------------	-------------------------

(dove "*" indica un qualsiasi carattere diverso da "b")

passo successivo

a b c a b c a c a b a * a ...	(elemento) (stringa)
----------------------------------	-------------------------

(b) Disuguaglianza al secondo carattere dell'elemento.

situazione corrente

a b c a b c a c a b ... a b c a b * ... ↑ disuguaglianza	(elemento) (stringa)
-------------------------------------------------------------------	-------------------------

(dove "*" indica un qualsiasi carattere diverso da "c")

passo successivo

a b c a b c a c a b ... a b c a b * ...	(elemento) (stringa)
--------------------------------------------	-------------------------

(c) Disuguaglianza al sesto carattere dell'elemento.

Fig. 6.1.

Per determinare questi elementi parzialmente uguali, necessari per uscire da una situazione di disuguaglianza occorre costruire il processo appena svolto manualmente. Questo equivale a fare una copia dell'elemento e considerarla come una stringa fissa. L'altra copia viene posta in posizioni relative alla prima copia. Ogni volta che viene riscontrata una parziale uguaglianza (come, ad esempio, quella che inizia in quarta posizione), le posizioni che sono parte della parziale uguaglianza (cioè 5,6 e 7 nell'esempio), non vengono considerate come posizioni in cui cercare parziali uguaglianze. La ragione è che potrebbero soltanto

Tab. 6.1. Tabella delle parziali uguaglianze.

Posizioni diverse										
	1	2	3	4	5	6	7	8	9	10
Elemento Uguaglianze parziali più piccole	a	b	c	a	b	c	a	c	a	b
	0	0	0	1	2	3	4	0	1	2

fornire uguaglianze parziali più piccole di quella già trovata. Una piccola riflessione rivela che dovremo sempre considerare la *più grande* uguaglianza parziale per evitare di perdere complete uguaglianze alla ripresa da una situazione di disuguaglianza. Possiamo riassumere la procedura per trovare tutte le parziali uguaglianze, come segue:

finché devono essere ancora trovate parziali uguaglianze

(a) continua il confronto dalla posizione iniziale corrente e salva i piccoli gruppi parzialmente uguali finché non incontri una disuguaglianza;

(b) se hai incontrato una disuguaglianza, allora

- (b.1) assegna zero parziali uguaglianze per la posizione corrente,
- (b.2) riazzerà il contatore per considerare parziali uguaglianze nella posizione successiva.

Dopo aver calcolato come trovare le informazioni sulle parziali uguaglianze, necessarie per l'algoritmo di ricerca lineare, cerchiamo di sviluppare un algoritmo di ricerca che ne faccia uso.

All'inizio sembra chiaro che dovremo trattare in maniera diversa le situazioni di uguaglianza e disuguaglianza tra l'elemento e la stringa. Quello che dobbiamo fondamentalmente fare è considerare il posizionamento dell'elemento relativamente alla stringa, come imposto dalle uguaglianze parziali, complete e dalle disuguaglianze che vengono rilevate. La strategia generale potrebbe essere:

finché non sono state considerate tutte le posizioni di parziale uguaglianza

(a) se i caratteri correnti della stringa e dell'elemento sono uguali, allora

- (a.1) aumenta la parziale uguaglianza,
- altrimenti

(a.1) occupati della situazione di disuguaglianza.

Per aumentare la parziale uguaglianza, occorre possedere un meccanismo che passi alla posizione successiva sia della *stringa* che dell'*elemento*. Insieme a questo incremento abbiamo bisogno di un test per le complete identità. Questo può essere un semplice test nella forma:

se l'indice parziale supera la lunghezza dell'elemento, allora
"abbiamo trovato due elementi uguali".

Ogni volta che tale condizione è soddisfatta, dobbiamo conteggiare il riscontro dell'uguaglianza. Bisogna rispondere però anche alla domanda di come "continuare" dopo una situazione di identità. Abbiamo considerato finora solo come uscire da una situazione di disuguaglianza. Dallo studio della tabella dell'esempio precedente, vediamo che la dimensione della *parziale uguaglianza* sarà *ab... o*, in altre parole, i primi due caratteri dell'elemento una volta trovata l'uguaglianza. Da ciò deriva che la ripresa dopo il rilevamento di un'identità non dovrà essere trattata in modo diverso da quelle dopo una disuguaglianza.

Sorge ora la domanda su come dovremo usare la tabella delle *parziali uguaglianze* per riprendere il lavoro dopo una disuguaglianza. Nella discussione preliminare su come ricavare le uguaglianze parziali, abbiamo trovato che non tutte le situazioni avrebbero dovuto essere trattate in maniera identica. Esse si dividono in due tipi, quelle in cui la *prima parziale uguaglianza* è *zero*, e quelle in cui esiste una *finita parziale uguaglianza* prima del riscontro di una disuguaglianza. Nel caso più recente non ci sono movimenti in avanti nella stringa quando incomincia la ricombinazione poiché nella situazione con zero parziali combinazioni basta semplicemente passare al carattere *successivo* della stringa e confrontarlo con il primo carattere dell'elemento cercato. Il confronto parziale con esito zero è indicato nella tabella da uno zero, e quindi questa situazione può essere facilmente scoperta. Per decidere sui dettagli del meccanismo necessari per uscire da un caso di disuguaglianza dovendo continuare con una combinazione diversa da zero, consideriamo un caso specifico.

Supponiamo di voler riprendere dopo una disuguaglianza al settimo carattere con l'elemento usato precedentemente. In questo caso la combinazione parziale alla ripresa è *abc* ed a questo punto il *quarto* carattere dell'elemento dovrà essere confrontato con il carattere corrente della stringa al passo successivo. Da un esame della tabella 6.1, vediamo che il valore in *sesta* posizione (quello *immediatamente precedente* alla disuguaglianza) provvede al recupero della parziale uguaglianza. Se la tabella delle combinazioni parziali è chiamata *recover* e la variabile *match* fornisce la posizione dell'elemento in cui viene rilevata la disuguaglianza, allora la seguente istruzione fornisce la nuova posizione dell'elemento.

```
match := recover[match - 1] + 1
```

L'“1” deve essere aggiunto al valore recuperato perché è il carattere successivo dell'elemento al di là della corrente uguaglianza parziale che deve essere confrontata con il carattere corrente della stringa. Il meccanismo necessario per trattare la condizione di disuguaglianza è quindi:

se non c'è una parziale uguaglianza, allora
 (a) passa al prossimo carattere della stringa
 (b) posiziona il puntatore all'elemento all'inizio
 altrimenti

```
match := recover[match - 1] + 1.
```

La ripresa dopo una completa identità è trattata in modo simile. Nell'implementazione può essere usata una procedura separata per la ripresa.

Nella costruzione della tabella delle parziali combinazioni non è assegnata la prima posizione poiché corrisponde a spostamento nullo. L'indice del recupero può assumere i valori zero e uno, quindi *entrambe* queste posizioni potrebbero avere registrato il valore zero.

Siamo ora nella situazione di poter descrivere dettagliatamente gli algoritmi per la costruzione della tabella *recover* e per la conduzione della ricerca lineare.

Descrizione dell'algoritmo

In questo paragrafo viene fornita la descrizione dell'algoritmo di ricerca e di quello di recupero.

(1) Algoritmo di costruzione della tabella delle parziali combinazioni

1. Definisci l'elemento da cercare.
2. Assegna lo spostamento iniziale tra l'elemento e se stesso pari ad 1.
3. Inizializza le posizioni zero e prima dell'array delle combinazioni parziali a zero.
4. Finché non saranno state considerate tutte le posizioni relative dell'elemento
 - (a) se il carattere corrente dell'elemento e quello dell'elemento spostato sono uguali, allora
 - (a.1) salva il livello corrente di parziale uguaglianza,
 - (a.2) passa alla posizione successiva dell'elemento fisso e di quello spostato
 - altrimenti
 - (a'.1) disuguaglianza, allora poni uguale a zero la parziale uguaglianza,
 - (a'.2) posiziona il puntatore all'inizio dell'elemento spostato,
 - (a'.3) muovi l'inizio dell'elemento spostato nella successiva posizione disponibile.
5. Ritorna la tabella delle combinazioni parziali.

(2) Algoritmo lineare di ricerca di elementi

1. Definisci l'elemento che deve essere cercato e la stringa in cui cercare insieme alle loro lunghezze.
2. Assegna i valori iniziali dell'elemento e della stringa e azzera il contatore delle uguaglianze.
3. Finché non sono state esaminate tutte le posizioni dell'elemento relativamente alla stringa
 - (a) se i caratteri correnti della stringa e dell'elemento sono uguali

- (a.1) incrementa gli indici per la successiva coppia,
 - (a.2) se c'è una completa combinazione, allora
 - (2.a) aggiorna il contatore delle combinazioni complete,
 - (2.b) riazzera la posizione del recupero dalla tabella delle combinazioni parziali
 - altrimenti
 - (a'.1) riazzera la posizione di recupero della tabella delle combinazioni parziali.
4. Ritorna il contatore del numero delle combinazioni complete dell'elemento nella stringa.
- (3) Procedura per il recupero dalle disuguaglianze e dalle complete combinazioni
1. Definisci la tabella parziale delle combinazioni, la posizione corrente della stringa e la posizione dell'elemento.
 2. Se non ci sono combinazioni parziali più piccole, allora
 - (a) passa alla posizione successiva della stringa,
 - (b) torna all'inizio dell'elemento
 - altrimenti
 - (a') riprendi il lavoro dopo una disuguaglianza o una ugualanza completa usando la tabella per memorizzare le nuove più piccole combinazioni possibili nella posizione corrente della stringa.
 3. Ritorna la più piccola combinazione parziale e la posizione corrente dell'elemento.

Implementazione in Pascal

```

procedure kmpsearch (pattern: nchars; string: nchars; var recover:
  ntchars; var nmatches; integer; patlength, slength: integer);
var position {current position in string},
  match {one more than extent of current partial match}: integer;

procedure restart (recover: ntchars; var match, position: integer);
begin {uses partial match table to recover from mismatches and
  complete matches}
  {assert: mismatch or complete match ∧ match = position in pattern
  ∧ position = position in string}
  match := recover[match - 1] + 1;
  if match = 1 then
    {no smaller partial match so move to next position and restart}
    position := position + 1
  {assert: (match - 1)=smaller partial match ∧ match = position in
  pattern ∧ position = position in string}
end;

```

```

procedure partialmatch (pattern: nchars; var recover: ntchars;
  patlength: integer);
var position {current starting position of displaced pattern},
  match {one more than extent of current partial match}: integer;

begin {sets up partial match table used by linear pattern search}
  {assert: patlength > 0}
  position := 2; match := 1;
  recover[0] := 0; recover[1] := 0;
  {invariant: pattern[1..match - 1] matches pattern [position - match
  + 1..position - 1] ∧ recover[position - 1] = match - 1 ∧
  position <= patlength + 1}
  while position <= patlength do
    begin
      if pattern[position] = pattern[match] then
        begin
          recover[position] := match;
          match := match + 1;
          position := position + 1
        end
      else
        begin
          recover[position] := 0;
          match := 1;
          position := position + 1
        end
    end
  {assert: complete partial match table recover[0..patlength]
  constructed}
end;

```

```

begin {pattern search algorithm – uses partial match table to achieve
  linearity}
  partialmatch(pattern,recover,patlength);
  {assert: partial match table constructed ∧ patlength, slength > 0}
  position := 1; match := 1;

  {invariant: (match < patlength ∧ string[position - match
  + 1..position - 1] matches pattern[1..match - 1]) ∨ (a pattern match
  ∧ nmatches = number of overlapping matches in first position - 1
  characters of string)}
  while position <= slength do
    begin {see if current partial match can be extended}
      if pattern[match] = string[position] then
        begin {extend match and test for complete match}
          match := match + 1;
          position := position + 1;
          if match > patlength then
            begin {count and recover from complete match}
              nmatches := nmatches + 1;
              restart (recover, match, position)
            end
        end
    end

```

```

    end
  else {recover from mismatch}
    restart (recover, match, position)
end
{assert: nmatches = number of times pattern found in string}

```

Note di progetto

- Nell'algoritmo per la costruzione della tabella delle combinazioni parziali, il passo principale è il confronto tra i caratteri dell'elemento fisso e quelli dell'elemento spostato. Questo passo viene eseguito solo $(m - 1)$ volte per un elemento di lunghezza m . L'algoritmo è quindi linearmente dipendente dalla lunghezza dell'elemento.
Nell'algoritmo di ricerca prevale ancora una volta il passo del confronto di caratteri. Il numero di volte in cui viene eseguito è stabilito dalla proporzione in cui *match* e *position* aumentano verso la fine della stringa. Poiché per una stringa di lunghezza n ci possono essere al massimo n spostamenti in avanti sia dell'elemento che della stringa, allora possono essere fatti al massimo $2n$ confronti. L'algoritmo di ricerca ha quindi funzionamento lineare o proporzionale a $O(n)$.
- Per la procedura di uguaglianza parziale, dopo l'iterazione per un dato valore di *position* saranno state stabilite le più piccole combinazioni parziali per i primi $a = (position - 1)$ valori della tabella *recover*. Al termine, quando *position* supera *patlength*, è stata costruita l'intera fase di recupero. Anche dopo l'iterazione con il valore corrente di *position* saranno stati confrontati con l'elemento spostato i primi $(position - 1)$ caratteri.
Dopo l'iterazione con il valore corrente di *match* rimarrà invariato il fatto che i primi $(match - 1)$ caratteri dell'elemento spostato uguaglano l'elemento fisso, iniziando dalla posizione $(position - 1)$ dell'elemento fisso e proseguendo all'indietro di $(match - 1)$ posizioni. L'algoritmo termina perché ad ogni iterazione *position* si incrementa di un'unità e quindi supererà sicuramente il valore *patlength*.
Nell'algoritmo di ricerca, nel momento in cui sono stati esaminati i primi *position* caratteri, saranno stati già conteggiati tutti gli elementi nei primi $a = position$ caratteri della stringa che uguaglano completamente l'elemento cercato. Ciò sarà rappresentato dal valore di *nmatches*. La stessa considerazione applicata a *match* nell'algoritmo delle combinazioni parziali è attinente all'algoritmo di ricerca. Il discorso nella nota 1 conferma che l'algoritmo termina dopo al massimo $2n$ passi per una stringa di lunghezza n .
- Sono stati molto utili esempi specifici per rivelare osservazioni sottili e piuttosto ambigue necessarie alla risoluzione di questo problema.

- L'algoritmo tratta correttamente anche elementi di lunghezza unitaria.
- Anche nel nostro algoritmo, all'inizio, abbiamo lasciato un certo numero di inefficienze a beneficio di chiarezza e semplicità. Ad esempio, la tabella *recover* avrebbe potuto avere il valore "1" contenuto direttamente all'interno dalla sua costruzione.
- La funzione di ripresa che abbiamo sviluppato, sebbene sia probabilmente la più semplice da comprendere, non è quella che minimizza il numero di confronti fatti dall'algoritmo. Ad esempio, notiamo che quando si trova:

abcabc...

e viene rilevata una disuguaglianza in sesta posizione, non esiste modo di fare un "recupero" in terza posizione che è pure destinata a dare una disuguaglianza.

- È possibile, costruendo una adatta macchina a stati finiti, sviluppare una versione di questo algoritmo che esamini *una sola volta* i caratteri della stringa.
- L'algoritmo che abbiamo implementato è una variazione di quello dovuto a Knuth, Morris e Pratt (ved. D.E.Knuth, J.H.Morris e V.R.Pratt, "Fast pattern matching in strings", SIAM J.Computing, 6, 323-50 (1977)).
- Aho e Corasick forniscono un'applicazione ed un'implementazione molto elegante dell'algoritmo di Knuth, Morris e Pratt (ved. A.V.Aho e M.J.Corasick, "Efficient string matching: an aid to bibliographic search", Comm. ACM., 18, 333-40 (1975)).
- Bailey e Dromey hanno progettato un algoritmo sub-lineare di ricerca di elementi per alfabeti piccoli (ved. T.A.Bailey e R.G.Dromey "Fast string searching by finding subkeys in subtext", Inf. Proc. Letts., 11, 130-3 (1980)).

Applicazioni

Ricerca di elementi in sistemi di alfabeti limitati e ricerca di parole chiave multiple in un testo.

Problemi supplementari

- In molte applicazioni non viene riscontrata l'uguaglianza per il primo carattere dell'elemento. Cercare di ristrutturare l'algoritmo in modo da trarre vantaggio da questo fatto.
- Cercare di costruire una funzione migliore di recupero sulla base delle osservazioni della nota 6.
- È possibile implementare una versione più chiara e più efficiente dell'algoritmo che non debba fare distinzione tra recupero dopo un'uguaglianza zero e un'uguaglianza parziale. A questo

scopo può essere usata una seconda tabella *delta* formata da zeri e uni appropriatamente posti. La variabile *position* sarà allora aggiornata utilizzando:

position := *position* + *delta*[*match* - 1]

Modificare l'algoritmo di combinazione parziale in modo che costruisca in maniera appropriata la tabella “*delta*” e che operi dei cambiamenti nell'algoritmo di ricerca in modo da trarre vantaggio della tabella “*delta*”.

- 6.5.4 Utilizzando l'idea presente nell'algoritmo di ricerca lineare, cercare di sviluppare un algoritmo di ricerca sub-lineare che sia controllato da un meccanismo che esamini solo ogni k caratteri della stringa. Il valore di k può essere considerevolmente minore della metà della lunghezza dell'elemento. (Ved. T.A.Bailey e R.G.Dromey, "Fast string searching by finding subkeys in sub-text," Inf. Proc. Letts., 11, 130-3 (1980))

ALGORITMO 6.6 RICERCA SUB-LINEARE DI ELEMENTI

Problema

Progettare ed implementare un algoritmo che cerchi una particolare parola chiave o *elemento* per un dato testo e registri il numero di volte che questi ricorrono nel testo stesso.

Sviluppo dell'algoritmo

Nella ricerca della parola SENTENCE nel testo:

THIS IS A RATHER SLOW ATTACK AT TEXT

un approccio potrebbe essere quello sviluppato qui di seguito. Controlla se il primo carattere della parola è uguale al primo carattere del testo (cioè confronta S con T). Se non sono uguali, confronta il primo carattere della parola con il secondo del testo (cioè S con H) e così via. Quando il primo carattere della parola uguaglia il carattere corrente del testo, (cioè S di SENTENCE uguaglia S di SLOW) confronta il secondo carattere della parola con il carattere successivo del testo (in questo caso E con L) e così via. È chiaro come questo concetto possa essere esteso alla ricerca di complete uguaglianze tra parole. La domanda a

Fig. 6.2.
Posizione iniziale
successiva
dell'elemento
SENTENCE in un
dato testo.

cui dobbiamo rispondere a questo punto è “esiste un modo migliore per implementare un algoritmo di ricerca in un testo?” Nel metodo descritto ci sembra di fare una quantità di lavoro non necessario, ma allo stesso tempo è difficile pensare che ci possano essere altre alternative.

Nell'esempio precedente abbiamo iniziato guardando se la parola SENTENCE poteva essere la prima del testo, ovvero:

THIS IS A RATHER SLOW... (testo)

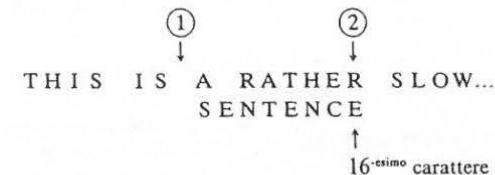
SENTENCE (parola)

Se la parola SENTENCE fosse stata la prima del testo, il primo carattere del testo sarebbe stato S. Di conseguenza, il secondo sarebbe stato E..., e l'ottavo carattere una E.

Poiché la parola SENTENCE non inizia al primo carattere del testo, possiamo chiederci, "dove potrebbe iniziare la parola SENTENCE?" Osservando dettagliatamente la struttura della parola e del testo, concludiamo che non può assolutamente iniziare prima dell'ottavo carattere. Ciò è vero poiché l'ottavo carattere è uno spazio bianco e *non ci sono spazi nella parola SENTENCE* (fig. 6.2).

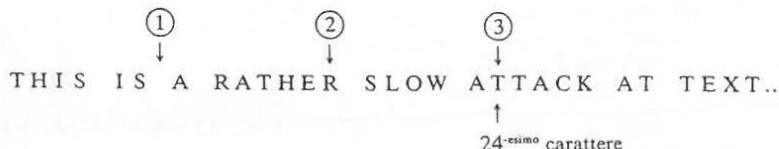
Avremmo potuto stabilire più facilmente che la parola SENTENCE non iniziava in prima posizione confrontando per primo l'ultimo carattere della parola con l'ottavo del testo. Nel fare questo avremmo trovato il carattere "bianco" che non è contenuto nella parola SENTENCE. Poiché lo spazio bianco non è contenuto nella parola, possiamo utilizzare ancora la stessa strategia, cioè guardare la posizione successiva in cui potrebbe essere l'ultimo carattere di SENTENCE (cioè il carattere in posizione 16 del testo).

Ad esempio

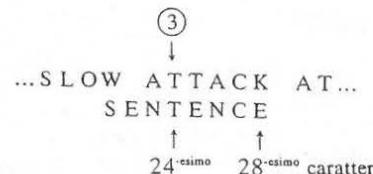


Se la parola SENTENCE terminasse al carattere in posizione 16, allora il 16-esimo carattere del testo sarebbe una E. Noi troviamo invece una R che non è contenuta in SENTENCE. Ciò significa che la posizione successiva in cui potrebbe terminare la parola è la 24-esima.

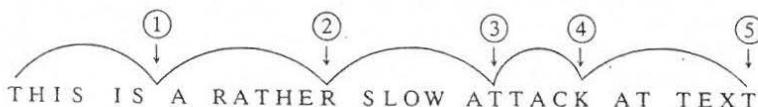
Ad esempio,



Nella 24-esima posizione del testo troviamo una T. Questo carattere è diverso dalla E di SENTENCE, quindi SENTENCE non può terminare in posizione 24. La domanda è ora, qual'è la posizione successiva che può contenere la fine della parola? Questo caso è alquanto differente dai precedenti due perché T è contenuta nella parola SENTENCE. È possibile che la T appena incontrata nel testo possa appartenere alla parola SENTENCE. Cioè, la parola SENTENCE può essere contenuta nel testo nel modo seguente:



Per controllare se la T in posizione 24 appartiene alla parola SENTENCE, possiamo ancora guardare nella posizione in cui ci si aspetta di trovare l'ultimo carattere della parola (in questo caso in posizione 28). In posizione 28 troviamo una K invece di una E. Da ciò è possibile concludere che la T in 24-esima posizione non appartiene realmente alla parola SENTENCE. Possiamo inoltre fare un altro salto di otto caratteri dalla posizione 28, perché K non è contenuto in SENTENCE. A questo punto ricapitoliamo i passi attraverso il testo.



Utilizzando la strategia di verificare sempre la posizione in cui potrebbe essere il carattere finale della parola cercata siamo giunti al termine dopo aver esaminato solo cinque caratteri (come indicato) invece di 36 (cioè abbiamo confrontato meno del 15% dei caratteri del testo). C'è un notevole miglioramento rispetto allo schema che comporta il confronto di tutti i caratteri del testo.

Dalla discussione e dall'esempio precedente potremmo prevedere che in molti casi nella ricerca di parole chiave o elementi dovrà essere esaminata solo una piccola parte dei caratteri del testo. Una volta capito questo punto, siamo al di là della metà del progetto di un algoritmo efficiente di ricerca.

Cerchiamo ora di utilizzare i risultati che abbiamo ricavato dalla discussione in modo da realizzare un'efficiente implementazione.

Quello che abbiamo imparato può essere riassunto come segue:

1. È una buona idea focalizzarsi sull'ultimo carattere della parola o dell'elemento invece che sul primo carattere.
2. Se viene incontrato un carattere del testo non contenuto nella parola chiave cercata, possiamo *saltare* un numero di caratteri uguale alla lunghezza della parola prima di confrontare di nuovo l'ultimo carattere.
3. Se in qualsiasi punto incontriamo un carattere presente nella parola, allora passiamo ad esaminare la posizione in cui sarebbe l'ultimo carattere della parola se il carattere corrente del testo appartenesse veramente alla parola cercata (come la situazione in posizione 24). Vedremo come trattare caratteri multipli più avanti.

Utilizzando queste informazioni vogliamo cercare di costruire un meccanismo che ci permetta di avanzare il più velocemente possibile ad ogni istanza, soggetto alla condizione che non ci debba essere alcun rischio di evitare delle parole uguali.

La strategia di base della parte centrale dell'algoritmo è quindi:

finché non trovi la fine del testo

- (a) se il carattere corrente del testo non indica la fine-parola, allora
 - (a.1) passa alla posizione indicata dall'identità del carattere corrente del testo
 - altrimenti
 - (a'.1) fa un confronto all'indietro carattere per carattere del testo e dell'elemento,
 - (a'.2) se trovi una completa uguaglianza, allora incrementa il contatore degli elementi uguali.

Dalla precedente discussione possiamo concludere che in ogni posizione deve essere applicata una delle tre situazioni di base:

1. il carattere del testo trovato *non è presente* nella parola chiave;
2. il carattere trovato *è presente* nella parola chiave;
3. il carattere trovato *è uguale* all'ultimo carattere della parola chiave.

Abbiamo già trattato la prima situazione, quindi esaminiamo ora più dettagliatamente gli altri due casi.

Supponiamo di voler cercare la parola SEPTEMBER e di aver appena trovato il carattere T del testo.

Ad esempio:

S E P T E M B E R	(elemento)
? ? ? T ? ? ? ? ?	(testo)
5	↑

L'esame di SEPTEMBER mostra che possiamo sicuramente fare un salto di cinque caratteri dopo la T senza alcun rischio. Se non viene trovata una R dopo l'ultimo salto, allora T non può essere parte del segmento ..T...R di SEPTEMBER. Quindi si può continuare ad avanzare in conformità al riconoscimento dell'ultimo carattere incontrato. Analogamente possiamo giungere alla seguente lista di salti:

incontrata	S	allora	salta	a	8
"	E	"	"	"	(7, 4, o 1)?
"	P	"	"	"	6
"	T	"	"	"	5
"	M	"	"	"	3
"	B	"	"	"	2
"	R	"	"	"	0?

A questo punto sorge il problema di cosa fare quando viene trovata una E o una R. Vediamo che associati ad E ci sono tre possibili salti.

Ad esempio:



Possiamo quindi decidere quando sia necessario un salto di 7, 4 o 1.

La situazione in cui ci troviamo è:

?? E ??? ... X	(testo)
SEPTEMBER	caso (1) salto 1
SEPTEMBER	caso (2) salto 4
SEPTEMBER	caso (3) salto 7

Se noi facciamo un salto di 7 posizioni, e in realtà si doveva applicare il caso (1), dovremo continuare dopo aver trovato una X che non è contenuta in SEPTEMBER. Ricordando che non volevamo rischiare di perdere delle parole chiave contenute, dobbiamo concludere che deve essere applicata la seguente regola. Quando viene incontrato un carattere del testo che ricorre più volte nella parola chiave, bisogna sempre fare il più piccolo salto associato a quel carattere. Questo garantisce che non ci saranno possibilità di perdere qualche confronto (nell'esempio precedente dopo aver incontrato una E avrebbe dovuto essere fatto un salto di 1).

L'ulteriore situazione da considerare è cosa fare quando incontriamo un carattere del testo che sia uguale all'ultimo della parola chiave. Non c'è modo di avanzare senza rischiare di perdere un confronto. La nostra unica alternativa è di cominciare un confronto all'indietro finché non troviamo la parola chiave o finché non troviamo una disugualanza. Dopo la terminazione del confronto all'indietro, dobbiamo riprendere a confrontare in avanti.

Se l'ultimo carattere trovato ricorre una sola volta nella parola chiave, allora può essere fatto un salto pari alla lunghezza della parola. Se l'ultimo carattere ricorre più di una volta nella parola chiave, come nella parola SESSION, allora si deve sempre fare un salto dall'ultima occorrenza del carattere multiplo (in questo caso 4).

Possiamo ora precisare i salti associati a tutte le lettere della parola SEPTEMBER.

Ad esempio:

8 1 6 5 1 3 2 1 9	- salti
S E P T E M B E R	- caratteri

Per poter utilizzare queste informazioni in una ricerca, dobbiamo essere in grado di accedere ad esse velocemente. Cioè, ogni volta che troviamo una E vogliamo sapere immediatamente che l'ampiezza del salto che possiamo fare è 1. Un modo per fare questo potrebbe essere quello di avere un array di 26 celle, di cui la prima contenente il salto per la A, la seconda quello per la B, e così via. La procedura in questo caso consisterebbe, ogni volta che viene incontrata una B, nel guardare nella cella 2 trovando il salto associato a B e così via. Una soluzione migliore potrebbe essere l'uso del valore numerico di ogni carattere direttamente come indice dell'array che contiene i valori del salto (ad esempio, usando l'insieme di caratteri ASCII per l'esempio precedente, la locazione 65 (cioè A), conterebbe 9, la locazione 66 il 2, e così via).

Per creare la tabella dei salti possiamo iniziare col riempire tutte le locazioni con un valore del salto pari alla lunghezza della parola chiave. Possiamo poi procedere alla modifica della tabella per i caratteri contenuti nella parola chiave cominciando da quello più a sinistra e muovendo verso destra. In questo caso noi poniamo, nella cella dell'array corrispondente al valore della chiave, il salto richiesto per il raggiungimento dell'ultimo carattere della parola chiave. Questo può essere implementato nella maniera seguente se l'array "parola" contiene i caratteri della parola chiave.

```
for i := 1 to wlength-1 do
begin
  p := ord(word[i]);
  skip[p] := wlength-i
end;
p := ord(word[wlength]);
skip[p] := -skip[p]
```

Notiamo che tutti i caratteri multipli faranno il salto *più piccolo* associato a quel carattere della parola. Il salto associato all'ultimo carattere della parola è negativo per facilitare la scoperta della necessità di un confronto all'indietro.

L'esame successivo serve per stabilire come debba essere usata la tabella dei salti nella conduzione della ricerca. Sappiamo che il processo ha inizio dall'esame del carattere in posizione *wlength* del testo (ad esempio supponiamo $i = wlength$). Il passo successivo serve per identificare il valore numerico del carattere del testo in posizione *i*-esima. Per questo possiamo usare:

```
nxt := ord(txt[i])
```

Il valore *nxt* diventa allora l'indice della tabella dei salti. Se il valore corrispondente è positivo, non siamo alla fine della parola, quindi possiamo passare al carattere in posizione

```
i := i + skip[nxt]
```

Altrimenti se il valore della tabella dei salti è negativo, abbiamo raggiunto l'ultimo carattere e quindi dobbiamo fare dei "confronti all'indietro".

L'effettivo processo di confronto deve essere fatto direttamente carattere — per — carattere. Possiamo ora riassumere i passi dell'algoritmo.

Descrizione dell'algoritmo

1. Definisci la parola ed il testo da cercare.
2. Fissa la tabella dei salti.
3. Azzera il contatore delle parole chiave trovate.
4. Assegna alla posizione del carattere *i* il valore della lunghezza della parola chiave.
5. Finché la posizione del carattere corrente < della lunghezza del testo,
 - (a) prendi il valore numerico *nxt* del carattere corrente in posizione *i*,
 - (b) inseriscilo nella tabella dei salti in posizione *next*,
 - (c) se il valore del salto per il carattere corrente è maggiore di zero, allora
 - (c.1) incrementa la posizione corrente del valore del salto
 - altrimenti
 - (c'.1) confronta all'indietro il testo e la parola,
 - (c'.2) se sono uguali aggiorna il contatore delle corrispondenze,
 - (c'.3) riprendi il lavoro dopo una diseguaglianza.
6. Ritorna il contatore delle uguaglianze.

Implementazione in Pascal

```
procedure quicksearch (text,word: tc; tlength, wlength: integer; var nmatches: integer);
const asize = 127;
type
  ascii = array[0..127] of integer;
var i {index for text search},
    j {index for matching backwards in text},
    k {index for matching backwards in word},
    nxt {ordinal value of current character}: integer;
    match {if true when k=0 then a word match}: boolean;
    skip: ascii;

begin {set up skip table}
  {assert: wlength > 0 ∧ tlength > 0 ∧ wlength = < tlength ∧ text and pattern ascii}
  setskips (word, skip, wlength, asize);
  {assert: skip table established for the search pattern word}
  nmatches := 0;
  i := wlength;
  {invariant: i points to next place where pattern word could end ∧ wlength = < i = < tlength + wlength ∧ nmatches = number of times word matched in first i characters}
  while i <= tlength do
    begin {use skip table to drive search for pattern}
      nxt := ord(text[i]);
      if skip[nxt]>0 then
        i := i + skip[nxt]
      else
        begin {maybe at end of word so match backwards}
          {assert: last character in word found in text}
          j := i - 1
          k := wlength - 1
          match := true;
          {invariant: word[k+1..wlength] matches text[j+1..i] ∧ k >= 0 ∧ j >= 0 ∧ i = < tlength}
          while (k>0) and (match=true) do
            begin
              if text[j]=word[k] then
                begin {move back one character in word and text}
                  j := j - 1;
                  k := k - 1
                end
              else
                match := false
            end;
          end;
          {assert: (k=0 ∧ match=true) ∨ (k>0 ∧ match=false)}
          if match=true then nmatches := nmatches + 1;
          {recover allowing for negative skips}
          i := i - skip[nxt]
        end
    end
  {assert: nmatches = number of times word found in text}
end
```

```

procedure setskips (word: tc; var skip: ascii; wlength, asize: integer);
var i {index for skip table array},
    j {index for characters in search word},
    p {ascii value of current character in word}: integer;

begin {set skips associated with all characters in alphabet}
  {assert: wlength > 0}
  for i := 0 to asize do
    skip[i] := wlength;
  {assert: j = 0 ∧ all ascii characters given skip length of wlength}
  {invariant: skip table adjusted according to first j characters in
  word}
  for j := 1 to wlength - 1 do
    begin {use ascii as index for skip table}
      p := ord(word[j]);
      skip[p] := wlength - j
    end;
  {assign negative skip to last character to differentiate from others}
  p := ord(word[wlength]);
  skip[p] := -skip[p]
end

```

Note di progetto

- È difficile il calcolo del funzionamento medio dell'algoritmo a causa della sua dipendenza dalla distribuzione di probabilità del testo e della parola o elemento cercato. Nel caso migliore, quando nessuno dei caratteri dell'elemento di lunghezza m è contenuto nel testo di lunghezza n , ci si aspetta un numero pari a n/m confronti di caratteri. Questo è il limite inferiore del numero di confronti. A causa del modo in cui è stato implementato il confronto all'indietro, nel caso peggiore ci aspetteremo $O(nm)$ confronti. È possibile calcolare il valore atteso dei confronti per una ricerca in un dato testo, se conosciamo la probabilità di tutti i caratteri dell'alfabeto usato nel testo. In questo caso abbiamo:

$$\text{N}^{\circ} \text{ di confronti} = \frac{n}{\sum s_j * P_j}$$

dove s_j è il salto associato al j -esimo carattere alfabetico dell'elemento o della parola da cercare e P_j è la probabilità di trovarla. Questa stima può essere ulteriormente corretta moltiplicando il numero di confronti per la probabilità dell'ultimo carattere della parola. Questo porta ad una correzione del *primo ordine* del numero di confronti all'indietro. Con l'introduzione di questa correzione troviamo che la formula precedente fornisce risultati molto accurati per un testo in Inglese. In molti casi è relativamente vicina a n/m .

- Per l'algoritmo di ricerca, dopo che è stato esaminato l' i -esimo carattere, sono state trovate tutte le combinazioni *complete* dell'ele-

mento con il testo e $nmatches$ rappresenterà il conteggio delle uguaglianze fino ad ora. Lo *skip* associato all' i -esimo carattere sarà maggiore di zero se l' i -esimo carattere non è l'ultimo carattere della parola. Se l'ultima occorrenza dell' i -esimo carattere del testo è in k -esima posizione del carattere del testo di lunghezza m , allora deve essere fatto un salto di $(m-k)$. Se l' i -esimo carattere del testo è uguale all'ultimo carattere dell'elemento, allora finché i caratteri del testo e della parola saranno uguali verrà compiuto uno spostamento all'indietro di un carattere per entrambi. L'algoritmo termina perché ad ogni iterazione del ciclo più esterno viene compiuto un passo positivo nel testo. Termina anche il ciclo più interno di confronto perché ad ogni iterazione viene fatto un progresso verso il ritrovamento di una uguaglianza o una disuguaglianza.

- Nell'algoritmo viene compiuto un tentativo di fare l'uso migliore dell'informazione acquisita ad ogni passo. (Potremmo, tuttavia, fare un uso migliore dei confronti all'indietro.)
- Quello che sembra il metodo migliore di affrontare il problema (cioè confrontando per primo il primo carattere), può portare solo ad un algoritmo inefficiente.
- Sono spesso molto efficienti algoritmi basati su tabelle (come in questo caso).
- L'algoritmo è più generale di uno per il ritrovamento di una parola. Può essere classificato più correttamente come un algoritmo di ricerca di elementi.
- L'algoritmo è sviluppato per lavorare con caratteri ASCII nel caso della parola chiave in ingresso.

Applicazioni

Elaborazione di testi, ricerca di elementi e di parole chiave.

Problemi supplementari

- Progettare un programma principale che chiami la procedura *quicksearch*. Il programma deve essere in grado di leggere e cercare blocchi successivi di un testo.
- Riprovarle la procedura precedente conteggiando il numero di confronti che devono essere fatti per parole di lunghezza diversa (provare per lunghezze di 5, 10, 15 caratteri).
- Modificare la procedura *quicksearch* in modo che l'indirizzamento tabellare dei caratteri sia sostituito dall'uso dei set.
- Un approccio diverso ad una ricerca rapida di un testo è di passare attraverso il testo con salti uguali alla lunghezza della parola. Non appena troviamo una disuguaglianza durante il confronto della parola cercata, saltiamo all'inizio della parola successiva. Per fare questo è necessario costruire nel testo i

collegatori "salta — alla — parola — successiva".
Ad esempio:

11 THIS IS A16 SENTENCE TO BE6 SKIP 0 SEARCHED

Lo "0" è usato per indicare un fine — riga. Nel condurre questa pre-elaborazione, è spesso prudente non considerare parole brevi (a cui non si è in genere interessati nella ricerca). Quando vengono ignorate parole di tre o più caratteri è necessario, nella conduzione della ricerca, esaminare solo il 10-20% dei caratteri del testo. L'esecuzione di questo algoritmo verrà frequentemente confrontata con esito favorevole con l'algoritmo appena sviluppato.

- 6.6.5 Progettare i necessari algoritmi di preelaborazione e ricerca.
Un altro algoritmo veloce di ricerca comporta il passaggio attraverso il testo con un confronto solo ogni m caratteri (dove m è la lunghezza dell'elemento cercato) ed un'azione appropriata per confermare l'uguaglianza ogni volta che vengono incontrati dei caratteri del testo che appartengono all'elemento. (Ved. C. Lakos e A. Sale, "Is disciplined programming transferrable?", Aust. Comp. J., 10, 87-93 (1978)).
Implementare questo algoritmo.

CAPITOLO 7

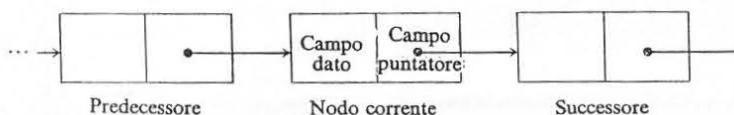
ALGORITMI PER STRUTTURE DINAMICHE DI DATI

INTRODUZIONE

Nella scienza della computazione è fondamentale la necessità di metodi flessibili ed efficienti per la memorizzazione, la manipolazione e l'accesso a grandi quantità di informazioni. Per far fronte a questa richiesta è stata sviluppata una grande quantità di metodi molto potenti per memorizzare le informazioni; tali *strutture dati* o *strutture informative* sono accompagnate da un insieme di algoritmi ugualmente potenti che possono essere usati per il loro accesso e manipolazione.

Fino ad ora l'unica struttura di cui ci siamo occupati è stato l'array. Le strutture dati che considereremo ora sono diverse, in quanto immagazzinano e memorizzano i dati dinamicamente: la quantità di memoria usata ogni volta è quindi direttamente proporzionale alla quantità di informazioni immagazzinata a quel punto della computazione. Con queste strutture diventa possibile allocare nuove registrazioni, quando necessario, e scartare vecchie informazioni, quando non siano più necessarie durante l'evoluzione della computazione. Con gli array, al contrario, deve essere predefinita una quantità fissa di memoria che rimane tale attraverso l'esecuzione del programma: un array è quindi definito come una struttura dati *statica*. Le strutture dati *set* e *record* come sono usate in Pascal, cadono anch'esse sotto il nome di strutture dati statiche. Come però vedremo, tali strutture statiche giocano un ruolo importante nella creazione delle strutture dati dinamiche; tipicamente un record, un set o un array, costituiscono le *unità di memorizzazione fondamentali* di una struttura dinamica di dati. Tali unità di base di memorizzazione sono dette di norma nodi (o elementi o registrazioni) della struttura dati. Questi nodi sono collegati in molti modi ed in relazione per formare la struttura dati. L'informazione sul collegamento per un particolare nodo è contenuta all'interno del nodo stesso. Ciò che caratterizza una data struttura è il modo in cui sono disposti i collegamenti tra i dati. Esistono tre tipi di collegamenti: lineare, gerarchico o ad albero, e a rete (o in senso matematico a grafo). I collegamenti vengono chiamati link, puntatori o riferimenti.

La più semplice di queste classi è quella *lineare*. Per i membri di questa classe c'è di norma un *solo* puntatore associato ad ogni nodo (nodi terminali hanno il puntatore *nullo*). Questo puntatore mette in relazione il nodo corrente con quello successivo nella lista. Il risultato è una struttura a catena con ogni nodo costituito dai due campi dato e puntatore. Ad esempio:



All'interno di questa classe i membri si distinguono per le limitazioni che impongono all'inserimento e alla cancellazione di nodi ed elementi: i membri più importanti di questa classe sono le *pile*, le *code* e le *liste ordinate*.

Per le pile e le code è predefinito dove debbano avvenire inserimenti e cancellazioni. Per una pila, tutti gli inserimenti vengono fatti sulla cima della pila e similmente tutte le cancellazioni vengono fatte dalla stessa estremità; una coda, invece, permette gli inserimenti da un'estremo e le cancellazioni dall'altro. Per una lista collegata, il punto in cui viene fatta un'inserzione o una cancellatura dipende dal particolare percorso considerato e dalle sue relazioni con la lista. Prima che venga fatto un inserimento o una cancellazione è quindi necessaria una ricerca nella lista.

Tra la classe gerarchica di strutture dinamiche, la struttura più largamente usata è quella *ad albero binario ordinato*. Questa struttura differisce da quelle lineari in modo tale che, sebbene i nodi possano avere un predecessore (o antenato, se ci si riferisce all'analogia con la famiglia degli alberi), essi possono avere due successori. In questo modo ogni nodo contiene il dato ed il puntatore al nodo successivo (ved. Fig 7.1). La struttura dati ad albero binario è *ricorsiva* [*]. Ciò significa che è possibile definire un albero di una data dimensione in termini di due alberi più piccoli che rivelino le proprietà dell'albero. Questa natura ricorsiva degli alberi ha implicazioni importanti in relazione agli algoritmi usati per il trattamento degli alberi. Frequentemente, algoritmi ricorsivi si addicono naturalmente al trattamento di strutture ad albero definite ricorsivamente. Una caratteristica della struttura dati ad albero è che permette un rapido accesso all'informazione ed efficienti inserimenti e cancellazioni. Come per le liste collegate, prima di un inserimento o di una cancellazione è generalmente necessario un esame dei nodi dell'albero.

Le più complicate tra le strutture dati ad albero che abbiamo menzionato sono le reti o, come sono chiamate di solito, i grafi. Un *grafo* è in realtà una generalizzazione della struttura ad albero che permette i cicli. La differenza sta nel fatto che ogni nodo del grafo può avere più di

[*] Vedi capitolo 8

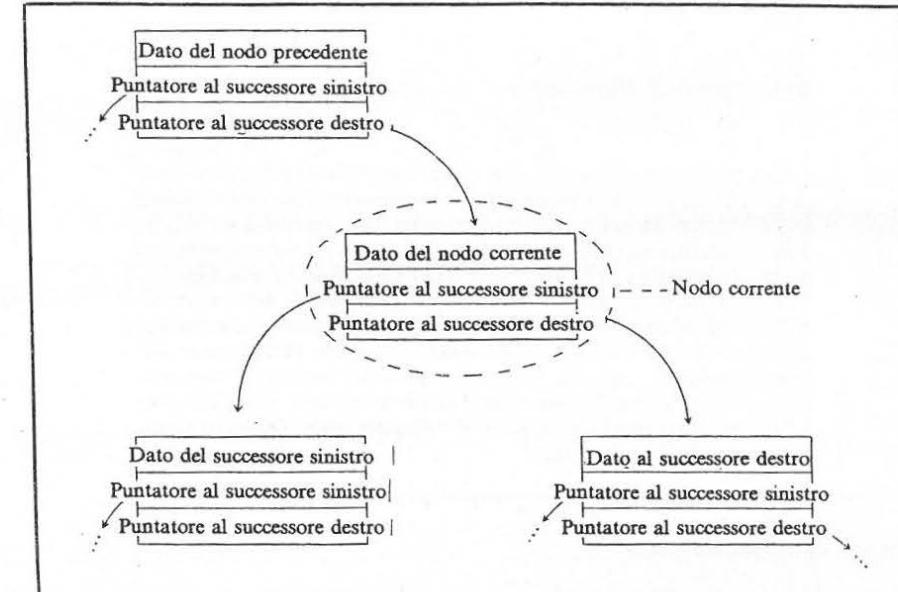


Fig. 7.1.
Struttura a puntatori
per un albero binario
ordinato.

un predecessore e più di un successore (in molti casi non viene fatta distinzione tra predecessori e successori). Le operazioni di ricerca, inserzione e cancellazione nei grafi non sono di solito così importanti. Nei grafi, invece, siamo generalmente interessati alla costruzione di percorsi attraverso il grafo che colleghino i nodi in modi speciali. Nei grafi è perciò possibile trovare una "distanza" associata ai puntatori tra coppie di nodi. Nel presente capitolo ci interesseremo solo alle liste di puntatori, alle pile, alle code e agli alberi binari. I diversi algoritmi associati ai grafi implicano uno studio più avanzato.

ALGORITMO 7.1 OPERAZIONI SULLE PILE (STACK)

Problema

Implementare due procedure, una per aggiungere elementi ad una pila e la seconda per rimuoverli.

Sviluppo dell'algoritmo

La maggior parte di noi conosce come funzionano le pile di lavoro in un ufficio. Le cartelle appena ricevute vengono impilate una sull'altra sopra una scrivania, dopodichè un membro del personale dell'ufficio toglie la cartella in cima alla pila per occuparsene. Le due operazioni possibili per la pila dell'ufficio sono illustrate in Fig. 7.2 e in Fig. 7.3.

Molte volte in scienza dell'elaborazione, (ad esempio nel trattamento delle chiamate di procedura, ricorsione, compilazione) è necessario modellare dei meccanismi che funzionino in modo molto simile alle pile dell'ufficio. Una *pila*, in scienza dell'informazione è una lista ordinata di zero o più elementi che permette inserimenti e cancellazioni ad una sola estremità chiamata *cima*. (Non può essere fatta una cancellazione in una pila vuota).

Fig. 7.2.
Inserimento in pila
("push"):
(a) Prima
dell'inserimento
dell'elemento 5;
(b) Dopo
l'inserimento
dell'elemento 5.

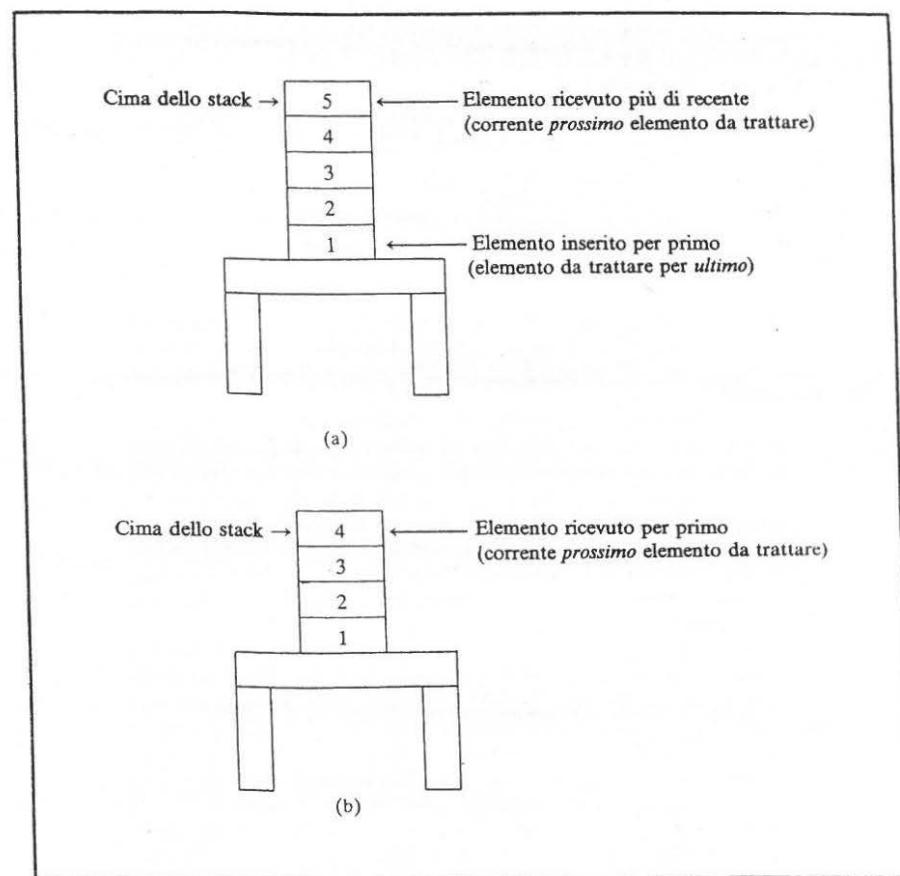
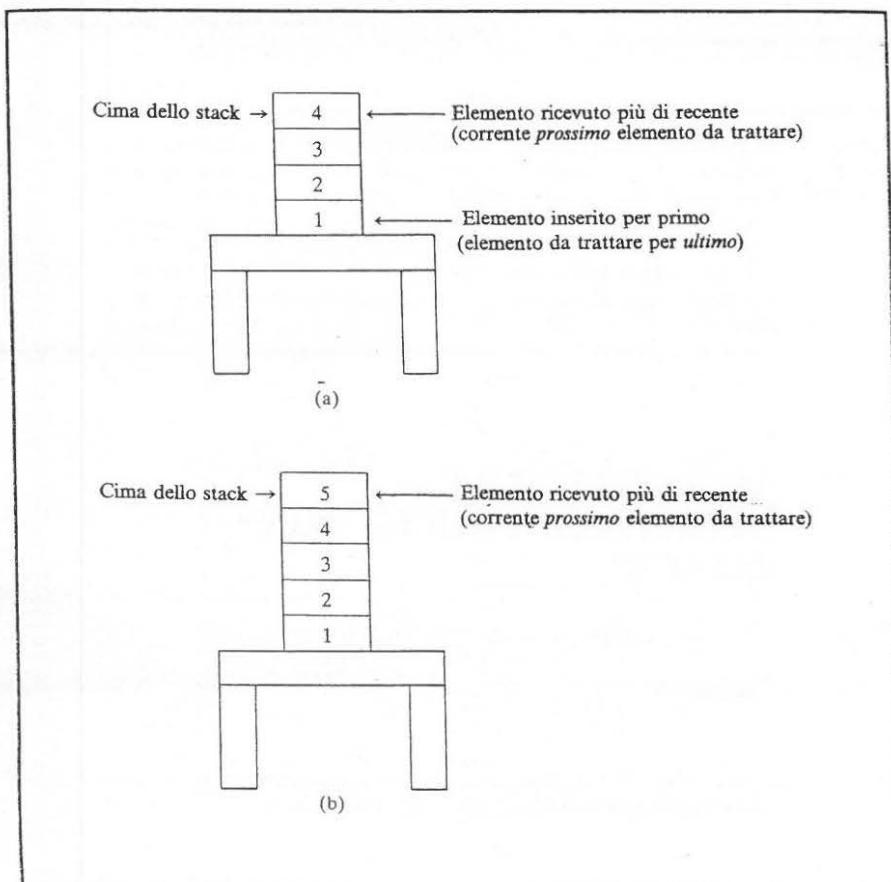


Fig. 7.3.
Cancellazione da una
pila ("popping"):
(a) Prima della
cancellazione
dell'elemento 5;
(b) Dopo la
cancellazione
dell'elemento 5.

Dalla precedente descrizione, l'array sembra soddisfare i requisiti per la costruzione di una pila: vediamo allora come è possibile l'uso di un array per costruire una pila.

L'introduzione di una nuova registrazione su una pila corrisponde ad aggiungere un nuovo elemento ad un array. Per semplicità supponiamo di voler costruire una pila di interi. L'introduzione (*push*) di un elemento sarà data da

```
stackpointer := stackpointer + 1;
stack [stackpointer] := item
```

Nel fare la push abbiamo trascurato di effettuare un controllo per vedere se la pila abbia o meno lo spazio per l'introduzione di tale controllo, l'implementazione in Pascal del push di una pila sarà quella descritta qui sotto (abbiamo tralasciato la descrizione scritta di questa procedura a causa della semplicità dei passi che la compongono).

```

procedure push (var stack:nelements; var stackpointer:integer;
item,maxsize:integer);
begin {pushes items onto an array stack}
  if stackpointer<maxsize then
    begin {push item onto stack}
      stackpointer := stackpointer+1;
      stack[stackpointer] := item
    end
  else
    writeln('stack overflow')
end

```

La rimozione di un elemento dalla cima della pila viene trattata semplicemente come l'inserimento ed è equivalente alla rimozione di un elemento di un array. Le istruzioni centrali sono:

```

item:=stack [stackpointer];
stackpointer:=stackpointer - 1

```

Ora non ci occorre controllare che non ci sia spazio, ma occorre piuttosto un test che verifichi se nella pila vi sia almeno un elemento da poter togliere. Il valore zero del puntatore alla pila, *stackpointer*, segnalerà l'assenza di elementi se l'array *stack* ha come limiti [1..maxsize]. L'implementazione Pascal della procedura di popping è descritta qui di seguito.

```

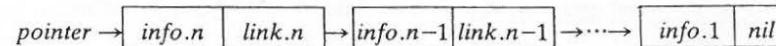
procedure pop (var stack:nelements; var item,stackpointer:integer);
begin {pops items from an array stack}
  if stackpointer>0 then
    begin {pop item off top of stack}
      item := stack[stackpointer];
      stackpointer := stackpointer-1
    end
  else
    writeln('stack underflow')
end

```

In questa procedura occorre notare che ogni volta che un elemento viene "rimosso" dalla cima della pila, rimane invariato nell'array; tuttavia, a causa del modo in cui viene utilizzato il puntatore alla pila, non fa più parte della pila corrente. Da queste osservazioni si rivela l'esistenza di un punto debole nell'uso di un array per implementare una pila; tipicamente per il modo in cui vengono usate, durante l'esecuzione di un programma le pile possono crescere o diminuire piuttosto drammaticamente; inoltre è di solito necessario immagazzinare un elevato numero di informazioni in ogni elemento della pila. In conseguenza di ciò le pile possono potenzialmente consumare grandi quantità di memoria.

Nell'implementazione di una pila tramite un array, è necessario, al momento della definizione delle variabili, preallocare la dimensione massima della pila. Un modo molto più desiderabile di usare una pila potrebbe essere far sì che occupi in ogni punto dell'esecuzione una quantità di memoria pari al numero di elementi contenuti in quel momento. Un certo numero di linguaggi di programmazione, compreso il Pascal, sono in grado di provvedere a questo tipo di allocazione dinamica di memoria. Nell'adottare quest'ultimo approccio, costruiremo la pila in modo tale che si avvantaggia naturalmente della sua natura dinamica.

Procedendo nell'implementazione della pila dinamica dovremo avvantaggiarci di linguaggi che favoriscono l'uso di allocazioni di memoria a struttura dinamica. In Pascal ciò implica l'uso del meccanismo a puntatori accoppiato alla struttura a liste ordinate; a questo scopo viene utilizzata una catena di record, ognuno dei quali è composto da due campi: uno per la memorizzazione dell'informazione (detto *info*) e l'altro per il collegamento (o *link*) che punta al suo predecessore nella catena. La struttura sarà:



La dichiarazione di tipo necessaria per l'implementazione di questa struttura sarà:

```

type stackpointer = ↑stackelement;
stackelement = record
  info : integer;
  link : stackpointer
end

```

L'introduzione del campo puntatore di tipo *stackpointer* in ogni record di tipo *stackelement* ci permette di costruire una lista di record. L'ulteriore dichiarazione necessaria è la variabile *pointer* che deve essere del tipo *stackpointer* e che può essere definita con una dichiarazione standard di variabile fatta nella procedura chiamante.

All'inizio il meccanismo per la rimozione e l'inserimento di elementi dalla pila è simile alla versione per il trattamento degli array, ma con l'approfondimento del problema sorge la necessità di introdurre nuovi passi per mantenere i collegamenti e per creare e cancellare i record.

Per comprendere meglio la procedura di linking occorre esaminare cosa succede non appena i primi elementi vengono inseriti nella pila. Prima dell'inserzione il puntatore non punterà ad alcun elemento, il che può essere esplicitato dall'inizializzazione:

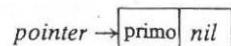
```

pointer:=nil

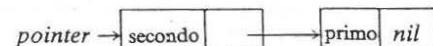
```

che fornirà il valore del puntatore precedente all'inserimento del primo

record. Una volta creato il primo record, esso conterrà il primo componente di collegamento della pila; bisogna decidere come debba venire assegnato questo collegatore per poter trattare la pila. Ora che sulla pila è posto un record, il puntatore pointer dovrà puntare a questo record, ed il record stesso (che è pure l'ultimo elemento della pila) punterà "niente". Avremo:



Dopo l'inserimento di un nuovo elemento nella pila, avremo:



In questo modo non appena viene inserito un nuovo elemento nella pila, il *puntatore* deve essere aggiornato per puntare all'elemento stesso ed il *link* del nuovo componente deve puntare all'elemento che era precedentemente in cima alla pila. Gli assegnamenti saranno:

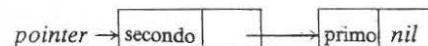
```

link := pointer;
pointer := newelement
  
```

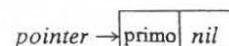
Per creare il nuovo elemento è necessaria una chiamata alla procedura Pascal *new*. Per far riferimento al nuovo componente bisogna utilizzare l'istruzione Pascal *with*.

Per eliminare un elemento dalla cima della pila, basta semplicemente "invertire" il processo appena descritto.

La prima mossa sarà:



e la situazione finale:



Questo implica riposizionare il puntatore in cima alla pila, in modo che punti *all'elemento puntato da quello in cima alla pila*. Il dato sulla cima della pila deve essere memorizzato ed il primo elemento rimosso con la funzione Pascal *dispose*. L'assegnamento per aggiornare il puntatore sarà:

```

pointer := link
  
```

L'implementazione può essere fatta usando ancora una volta l'istruzione *with*. Per controllare se la pila è vuota, dobbiamo verificare il puntatore *nil*. Può ora essere fornita la descrizione dettagliata sulle operazioni di push e pop.

Descrizione dell'algoritmo

(1) Inserimento di un elemento in una pila

1. Definisci il dato da inserire.
2. Crea il nuovo record ed inserisci il dato.
3. Posiziona il puntatore dal nuovo elemento verso il successore.
4. Aggiorna il puntatore alla cima della pila.

(2) Rimozione di un elemento dalla cima di una pila

1. Se la pila non è vuota, allora
 - (a) togli il puntatore al primo elemento,
 - (b) recupera il dato in cima alla pila,
 - (c) posiziona il puntatore al successore dal primo elemento,
 - (d) togli il primo elemento dalla pila
 - altrimenti
 - (a') scrivi che la pila è vuota.
2. Restituisce il dato in cima alla pila.

Implementazione in Pascal

```

procedure push (data: integer; var pointer: stackpointer);
var newelement {pointer to new element created}: stackpointer;
begin {pushes data onto top of stack}
  {assert: pointer points to top of stack}
  new(newelement);
  with newelement^ do
    begin
      info := data;
      link := pointer
    end;
  pointer := newelement
  {assert: newelement pushed onto stack ∧ pointer points to top of
  stack}
end

procedure pop(var data: integer; var pointer: stackpointer);
var topelement {pointer to pop element on stack}: stackpointer;
begin {pops data off top of stack}
  {assert: pointer points to top of stack}
  if pointer <> nil then
    begin
      topelement := pointer;
      with topelement^ do
        begin
          info := data;
          link := nil
        end;
      dispose(pointer)
    end
  else
    data := 0
end
  
```

```

begin
    data := info;
    pointer := link
end;
dispose(topelement)
end
{assert: top element removed from stack ∧ pointer points to new
top of stack}
else
    writeln('stack underflow')
end

```

Note di progetto

- Le operazioni di una pila implicano sostanzialmente un passo composto e quindi la loro complessità è di poco interesse. Riguardo all'efficienza in termini di memorizzazione l'implementazione mediante puntatori è migliore perché utilizza una quantità di memoria pari al numero di elementi contenuti nella pila; con l'implementazione facente uso degli array, al contrario, la quantità di memoria per contenere la dimensione massima della pila viene mantenuta *dall'inizio alla fine* dell'esecuzione del programma.
- La condizione che rimane invariata per entrambe le operazioni di *push* e *pop*, è che dopo una transazione il puntatore alla pila viene aggiornato in modo tale che punti sempre alla cima della pila.
- L'implementazione delle operazioni sulla pila con l'uso dei puntatori è in armonia con la natura dinamica della pila.
- Nell'implementazione mediante puntatori, non è necessario definire un limite superiore alla dimensione della pila, tuttavia è concepibile che la dimensione della pila fissata dal sistema possa eccedere.

Applicazioni

Traccia e passaggio di parametri in chiamate di procedure, compilatori, ricorsione.

Problemi supplementari

- Implementare il programma principale che fornisca e riceva i dati dalle procedure *push* e *pop*.
- Progettare ed implementare una procedura che riceva in ingresso una stringa di caratteri rappresentante un'espressione aritmetica, ad esempio:

$$y := (a + b) * (c + d) / ((x * y) / (w + z))$$

e stabilire se è "ben-formata". Per "ben formata" si intende che le parentesi debbano essere bilanciate in accordo con le convenzioni algebriche (notare che l'espressione precedente ha un ugual numero di parentesi aperte e chiuse, ma non è ben - formata).

- È spesso necessario, nello svolgimento di operazioni sulle pile, avere a disposizione stringhe a lunghezza variabile. Implementare le procedure di *push* e *pop* che trattino stringhe di questo tipo memorizzate in una pila consistente in un array di lunghezza fissata di caratteri.
- Progettare ed implementare delle procedure per il trattamento di due pile all'interno di un array.

ALGORITMO 7.2 AGGIUNTE E CANCELLAZIONI IN UNA CODA

Problema

Progettare ed implementare delle procedure per la gestione di una coda soggetta ad inserimenti e cancellazioni.

Sviluppo dell'algoritmo

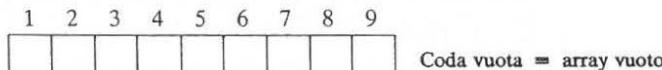
Sorge spesso la necessità di modellare dei processi simili a quello che osserviamo nella coda di una mensa. In questa coda, quando qualcuno viene servito, tutti gli altri si spostano di una posizione. I nuovi arrivati prendono posto alla fine della coda. In un sistema operativo, ad esempio, il lavoro dello "scheduler" è di trattare una coda delle richieste dei programmi.

Prima di iniziare la risoluzione di questo problema, occorre avere le idee chiare su cosa si intenda per *coda* nel contesto della scienza dell'elaborazione. Una *coda* è definita come una lista lineare in cui tutti gli inserimenti vengono fatti ad un estremo (quello inferiore) e tutte le cancellazioni vengono fatte all'altra estremità (la superiore). Le code sono quindi usate per memorizzare dati che devono essere elaborati nello stesso ordine del loro arrivo.

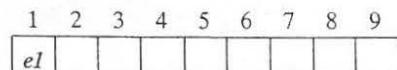
Schematicamente abbiamo:



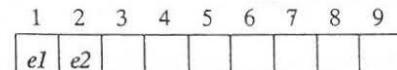
Per modellare una coda in un computer, occorre una struttura dati idonea. La definizione e l'esempio suggeriscono che un array possa essere quello che stiamo cercando: visualizziamo questa idea con un altro esempio. Consideriamo una coda vuota corrispondente ad un array privo di elementi attivi.



Il posto più ovvio dove inserire il primo elemento *e1* della coda è la prima cella dell'array.



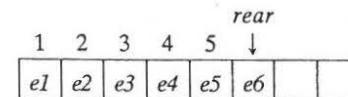
Il secondo elemento *e2*, (purchè *e1* non sia stato cancellato), può essere inserito nella seconda cella dell'array.



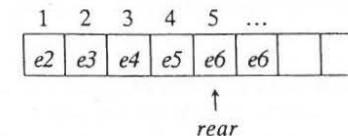
Quello a cui siamo interessati, durante l'inserimento di elementi all'estremità di una coda, è di conoscere in ogni momento dove debba essere aggiunto l'elemento successivo. Una soluzione potrebbe essere quella di cominciare all'inizio della coda e passare attraverso gli elementi finchè non viene trovata una cella vuota; al crescere della coda, tuttavia questo procedimento diventa sempre più inefficiente. Un esame più dettagliato del problema rivela che possiamo avere una traccia di dove sia la fine della coda utilizzando semplicemente una variabile il cui valore indichi sempre l'estremità della coda. Con questo suggerimento, l'inserimento di un elemento alla fine della coda comporta due passi:

1. aggiornare l'indicatore di fine-coda;
2. inserire il nuovo elemento nella posizione puntata dall'indicatore di fine-coda.

La nostra idea circa gli inserimenti in una coda sembra fattibile, quindi procediamo per vedere come possono essere trattate le cancellazioni. Se iniziamo con la configurazione seguente:



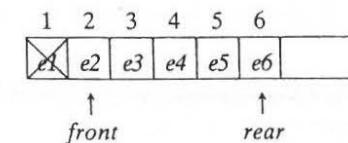
allora il primo elemento da eliminare sarà *e1*. È possibile ultimare questa eliminazione spostando ogni elemento di una posizione verso l'inizio, e modificando l'estremità inferiore, ovvero:



rear

Con questo meccanismo è necessario muovere tutti gli elementi della coda ogni volta che viene effettuata una cancellazione; con l'allungarsi della coda questo diventa un passo relativamente costoso. Possiamo quindi chiederci se esista un approccio meno costoso. Nel porci questa domanda ci siamo in realtà chiesti se esiste un modo di trattare una coda senza dover muovere tutti gli elementi ogni volta che viene eliminato il primo.

Da un più accurato esame dell'esempio, vediamo che se gli elementi non devono essere mossi, allora l'unica alternativa è muovere *l'estremità superiore della coda*; a questo scopo viene introdotta una variabile avente un ruolo complementare a quello della variabile inferiore. Così, ad esempio, dopo l'eliminazione del primo elemento, avremo:



front

rear

Abbiamo così trovato il meccanismo per gli inserimenti e le cancellazioni da una coda. Ogni volta che vengono fatte ulteriori operazioni di inserimento e cancellazione, gli ingressi correnti della coda vengono spostati sempre più a destra (verso la fine a suffisso più alto) nell'array. Per fare un esempio, all'inizio potremmo avere i seguenti parametri per la coda:

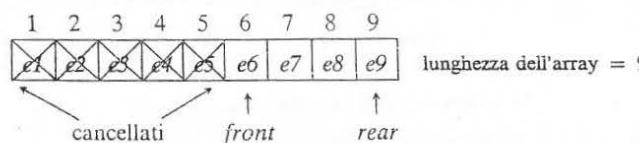
$$\begin{aligned} \text{front} &= 1 \\ \text{rear} &= 25 \end{aligned}$$

$$qlength = 25 \quad (qlength = front - rear + 1)$$

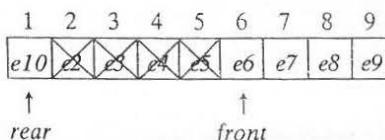
e, dopo un certo numero di operazioni:

front	=	87
rear	=	102
qlength	=	16

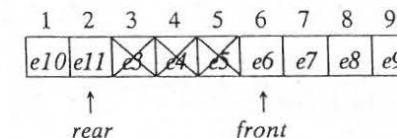
Questi esempi indicano che dopo un certo periodo di tempo occorrerà supplire a costi di memorizzazione molto alti. Questo uso eccessivo di memoria rende impraticabile la nostra attuale implementazione. Idealmente la cosa migliore sarebbe quella di impiegare solo una quantità di memoria pari alla massima lunghezza della coda fissata in anticipo. Questo obiettivo sembra ragionevole perché, a causa del modo in cui avvengono gli inserimenti e le cancellazioni, la lunghezza della coda è relativamente stabile. Dal primo esempio è chiaro che una coda può essere contenuta in un array maggiore o uguale alla sua lunghezza massima spostando gli elementi come avevamo proposto in precedenza. La domanda che rimane è se sia possibile operare in questo modo senza dover muovere tutti gli elementi dopo ogni cancellazione. Una possibile idea sarebbe quella di aspettare che venga raggiunta la fine dell'array, quindi spostare tutti gli elementi il più possibile verso l'inizio; questa proposta è decisamente migliore della precedente. Prima di uno spostamento supponiamo di avere:



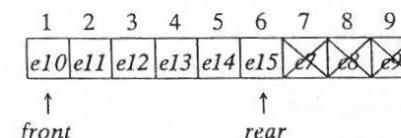
In questa configurazione non ci sono più posti per l'inserimento alla fine dell'array ma le prime 5 locazioni sono libere; possiamo quindi, spostare e6 in posizione 1, e7 in posizione 2, e così via. Il problema in questo caso è che esso implica un notevole numero di spostamenti. Dobbiamo allora chiederci di nuovo se esistano alternative migliori. Poiché esiste spazio libero all'inizio dell'array, potremmo memorizzare e10 in posizione 1, ma che conseguenze può avere una cosa di questo genere nel mantenimento della continuità della coda? Un'azione di questo tipo significherebbe portare la fine della coda prima dell'inizio della coda stessa, cioè:



Questa situazione sembra piuttosto particolare; vediamo allora cosa succede quando cerchiamo di inserire un altro elemento, e11. Applicando il procedimento originale per l'inserimento, troviamo:



Questa soluzione sembra permetterci di mantenere la coda a patto di memorizzare in modo appropriato la fine dell'array. Vediamo che sicuramente, dopo un certo numero di inserimenti e cancellazioni, verrà raggiunta di nuovo la configurazione in cui l'inizio è dietro la fine:



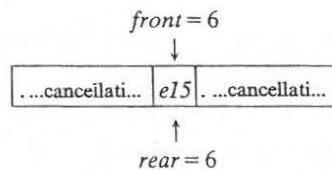
Abbiamo così una coda che si avvolge intorno ad un array: in questo caso ci si riferisce ad essa come ad una coda circolare. Il procedimento di inserzione e cancellazione è stabile finché la dimensione della coda non supera quella dell'array. Per conservare la coda ogni volta che viene fatta un'inserzione o una cancellazione, occorre controllare se le marce di inizio e fine supereranno i limiti della dimensione dell'array. Ogni volta che accade una cosa del genere, la marcatura corrispondente dovrà essere posta pari a 1.

Dopo aver costruito il meccanismo di base per il trattamento di una coda in un array, il compito successivo è di esaminare più attentamente il progetto degli algoritmi di inserimento e cancellazione. Dopo un più accurato esame della struttura dati, vediamo che ogni volta che si vuole fare un inserimento si può trovare un certo numero di situazioni:

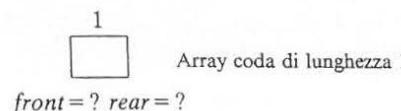
1. la coda può essere vuota;
2. la coda può essere già piena;
3. è stata raggiunta la fine dell'array prima dell'inserimento;
4. non sussiste alcuna delle condizioni precedenti (caso più favorevole).

Possiamo anticipare che alcuni di questi casi possono coinvolgere altre cose oltre l'inserzione diretta nella posizione successiva. Un caso importante da trattare è quello in cui si cerca di fare un inserimento a coda piena. Dagli esempi precedenti il modo migliore sembrerebbe quello di controllare se le variabili di inizio e fine sono uguali. Tuttavia, non appena iniziamo a pensare a questa implementazione, ci accorgiamo che ci potrà essere un'intera serie di situazioni in cui la coda

contenga solo un elemento e sia presente la condizione $front = rear$. Ad esempio:

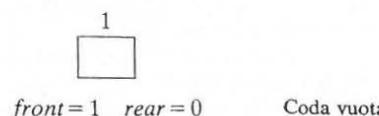


È quindi necessario trovare un altro modo per distinguere tra una coda con un solo elemento e una piena. Per semplificare il problema consideriamo una coda di minima lunghezza (la coda avente lunghezza massima = 1) per vedere se è possibile distinguere i due casi. La differenza tra le due situazioni coinvolge un solo elemento, quindi sarà più semplice osservare cosa succede.

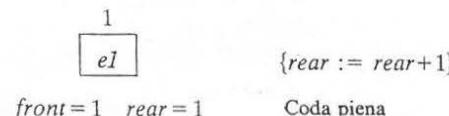


In genere quando viene fatto un inserimento, $rear$ viene incrementato di una unità e l'elemento viene posto in coda. Se noi applicassimo questa regola nel caso attuale $rear$ dovrebbe essere inizializzato a zero. Ci si pone ora la domanda di cosa fare con $front$ prima e dopo la cancellazione: per l'array ad un elemento esistono solo due possibilità. Prima dell'inserimento la variabile $front$ può valere solo 0 o 1. Se sceglieremo $front = 1$ avremo:

Prima dell'inserimento

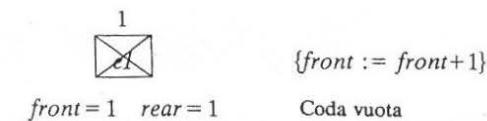


Dopo l'inserimento



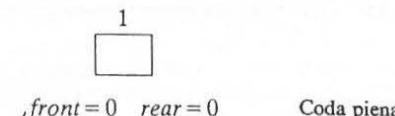
Dopo aver distinto le due situazioni di coda piena e vuota, vediamo cosa succede quando cancelliamo un elemento. Questo passo comporta l'incremento di $front$ di un'unità, ma, poiché ci porterebbe al di là del limite dell'array, deve essere riposizionato all'inizio dell'array. A questo punto sorge il problema dovuto al fatto che l'inizio e la fine dell'array sono in medesima posizione e quindi ancora una volta, dopo una cancellazione in questo modo, termineremmo con:

Dopo la cancellazione

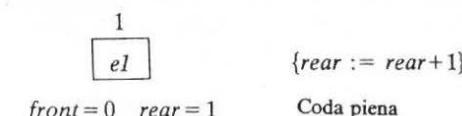


Questo contraddice la situazione iniziale, perciò consideriamo l'altra alternativa cominciando con $front = 0$.

Prima dell'inserimento

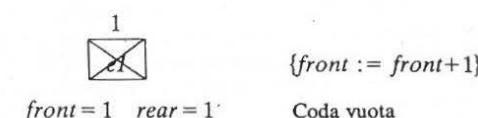


Dopo l'inserimento



Cerchiamo ora di fare una cancellazione.

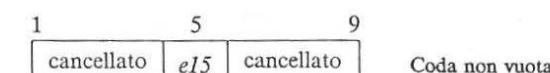
Dopo la cancellazione



La situazione di coda vuota dopo la cancellazione è ora diversa da quella di coda piena. Anche dal confronto del caso di coda vuota dopo la cancellazione e quello prima dell'inserimento, vediamo che in entrambi i casi quando la coda è vuota l'inizio e la fine coincidono. Dopo aver trovato un meccanismo per trattare le code di lunghezza unitaria, ci occorre generalizzarlo per code più lunghe.

Come passo iniziale in questa direzione, consideriamo il caso in cui nell'array di dimensione maggiore di uno ci sia un solo elemento.

Prima della cancellazione

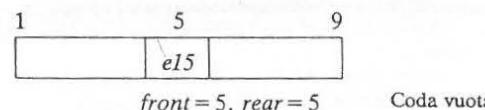


Dopo la cancellazione (che conterrà la condizione $front = front + 1$) vogliamo raggiungere la condizione $front = rear$ per indicare che la coda è vuota. Quindi, se prima della cancellazione avevamo:

$$front = 4 \text{ e } rear = 5$$

otterremo poi la condizione desiderata. Questa situazione è analoga al caso dell'array di lunghezza unitaria.

Dopo la cancellazione



Con questa configurazione, sappiamo che non è possibile un'altra rimozione poiché $front = rear$.

I passi per la cancellazione saranno:

se $front = rear$ allora

"la coda è vuota, quindi non cancellare"

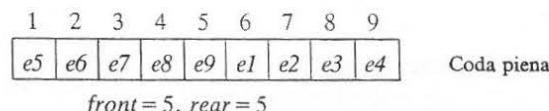
altrimenti

"incrementa $front$ assicurandoti di rimanere entro i limiti dell'array".

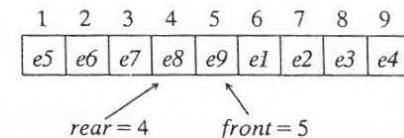
Per assicurarci che $front$ rimanga nei limiti dell'array, non appena superi $arraylength$ deve essere reinizializzato a 1, cioè:

```
front := front + 1;
if front > arraylength then
    front := 1
```

Dobbiamo ora vedere se il caso di coda piena possa essere trattato in maniera simile. Le condizioni per specificare una coda vuota richiedono che $front$ sia posizionato ad un passo dal primo elemento della coda e che $rear$ punti l'ultimo elemento della coda. Considerando la successiva coda piena, dove è stato applicato questo criterio:

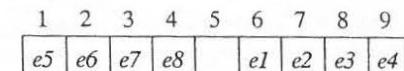


In questo caso sorgono dei problemi poiché $front$ e $rear$ sono uguali: questo corrisponde alla condizione che abbiamo cercato di usare per segnalare una coda vuota. Se applichiamo il meccanismo di inserimento, in questo caso scriveremo sopra il primo elemento della coda, perciò vogliamo che $front$ e $rear$ non siano uguali in caso di coda piena, vogliamo quindi che $rear$ sia = 4.



Coda piena?

Se questa è la configurazione corrispondente ad una coda piena, saremo in grado di memorizzare solo 8 elementi in 9 posizioni. Allora, quando troviamo la configurazione seguente:



Coda piena

$$rear = 4 \quad front = 5$$

cerchiamo di inserire un altro elemento, per primo incrementiamo $rear$ di un'unità e compiamo un test per vedere se la coda è piena. In questa situazione la condizione $front = rear$ indicherà che la coda è piena.

I passi per aggiungere un elemento alla fine della coda saranno quindi:

1. Incrementa $rear$ assicurandoti che non superi i limiti dell'array.
2. Se $front = rear$ allora
 - (2.a) coda piena, quindi compi i passi necessari
 - altrimenti
 - (2.b) aggiungi un nuovo elemento alla fine della coda.

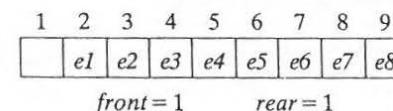
Possiamo assicurarsi che $rear$ rimanga entro i limiti dell'array usando un meccanismo simile a quello per limitare $front$. Se la coda è piena, $rear$ può essere uguagliato al suo valore precedente in modo da non confondere i due casi piena e vuota. Nel fare queste operazioni dobbiamo tener conto dei limiti dell'array, cioè:

se la coda è piena allora

- (a) scrivi il messaggio "la coda è piena";
- (b) $rear := rear + 1$;
- (c) se $rear = 0$ allora $rear := arraylength$.

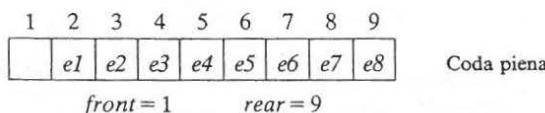
Il passo (c) è necessario per il caso seguente:

Prima delle modifiche di rear



Coda piena

Dopo le modifiche di rear



La descrizione dell'inserimento e della cancellazione da una coda è stata delineata. Si può supporre che *front* e *rear* siano inizializzati ad uno per la coda inizialmente vuota.

Descrizione dell'algoritmo

(1) Inserzione in coda

1. Definisci la coda, i dati da inserire, la lunghezza dell'array, *front* e *rear*.
2. Incrementa *rear* in modo che rimanga sempre entro i limiti dell'array.
3. Se *front* non è uguale a *rear*, allora
 - (3.a) inserisci il dato nella prima posizione libera della coda, altrimenti
 - (3.a') la coda è piena, quindi segnalalo in uscita,
 - (3.b) riassegna a *rear* il valore iniziale tenendo conto dei limiti dell'array.
3. Ritorna la coda aggiornata ed il puntatore *rear*.

(2) Cancellazione dalla coda

1. Definisci la coda, la lunghezza dell'array, i puntatori all'inizio e alla fine.
2. Se *front* non è uguale a *rear*, allora
 - (2.a) incrementa *front* in modo che rimanga sempre entro i limiti dell'array,
 - (2.b) togli l'elemento all'inizio della coda, altrimenti
 - (2.a') la coda è vuota quindi manda in uscita un messaggio.
3. Ritorna la coda aggiornata, l'elemento tolto dalla coda ed il nuovo puntatore *front*.

Implementazione in Pascal

```
procedure qinsert (var queue: nelements; var rear: integer;
data, arraylength, front: integer);
```

```
begin {inserts data at rear of circular queue—rear points to last
element}
{assert: rear points to end of queue  $\wedge$  1  $=$  < rear  $=$  < arraylength}
rear := rear + 1;
if rear > arraylength then
  rear := 1;
if front < > rear then
  queue[rear] := data
{assert: element added to rear of queue  $\wedge$  rear points to end of
queue  $\wedge$  1  $=$  < rear  $=$  < arraylength}
else
  begin {queue full so signal and readjust rear to previous value}
    writeln('queue full—cannot insert');
    rear := rear - 1;
    if rear = 0 then
      rear := arraylength
  end
end
```

```
procedure qdelete (var queue: nelements; var data, front: integer;
arraylength, rear: integer);

begin {deletes elements from front of queue—front precedes first
element}
{assert: front pointer precedes first element  $\wedge$ 
1  $=$  < front  $=$  < arraylength}
if front < > rear then
  begin {remove front element}
    front := front + 1;
    if front > arraylength then
      front := 1;
    data := queue[front]
  end
{assert: element removed from front of queue  $\wedge$  front pointer
precedes first element  $\wedge$  1  $=$  < front  $=$  < arraylength}
else
  writeln('queue empty—cannot delete')
end
```

Note di progetto

1. Le operazioni sulla coda comportano essenzialmente un passo composto, quindi la loro complessità in termini di tempo è poco interessante.
2. La condizione che rimane invariata per entrambe le operazioni è quella che dopo una transazione il puntatore *rear* punta ancora all'ultimo elemento della coda e *front* punta alla posizione immediatamente precedente al primo elemento della coda. Se la coda è vuota questa condizione è segnalata da *front* = *rear* dopo un tentativo di cancellazione. Inoltre, se la coda è piena, tale condizione è segnalata da *front* e *rear* dopo un tentativo di inserzione.

3. Con la presente implementazione deve essere lasciata libera una posizione dell'array (cioè in un array di lunghezza 10 possono essere contenuti solo 9 elementi di una coda). È possibile implementare algoritmi di inserimento e cancellazione in coda che permettano l'allocazione di *tutti* gli elementi dell'array in condizione di coda piena. La penalità in questo caso è data dall'aumento dei test durante l'inserimento e la cancellazione.
4. Come abbiamo osservato nella nota 3, è spesso possibile allocare la memoria per una più efficiente implementazione.
5. La considerazione del problema più piccolo della coda offre degli spunti molto utili per la risoluzione del problema generale.
6. Il metodo alternativo per mantenere *front* e *rear* all'interno dei limiti dell'array sarebbe quello di utilizzare un incremento del tipo:

front := (*front* + 1) mod *arraylength*

7. Nella presente dissertazione non abbiamo considerato l'implementazione di una coda tramite liste ordinate. In molte applicazioni tale implementazione è la più appropriata ed è anche più semplice da utilizzare. Tale implementazione è lasciata come esercizio per il lettore.

Applicazioni

Problemi di simulazione, lavoro di scheduling.

Problemi supplementari

- 7.2.1 Modificare il presente algoritmo in modo che la funzione *mod* sia usata per mantenere *front* e *rear* entro i limiti dell'array. Deve essere usata particolare attenzione nella specifica dei limiti dell'array (ved. nota 6).
- 7.2.2 Progettare ed implementare un algoritmo di inserzione e cancellazione da una coda che permetta che tutti gli elementi dell'array possano essere occupati nel caso di coda piena (ved. nota 3).
- 7.2.3 Scelta una coda di lunghezza iniziale *n*, usare un generatore di numeri casuali per simulare inserzioni e cancellazioni (ad esempio tra 0 e 0.5 → inserimenti, tra 0.5 e 1 → cancellazioni) e determinare quale sia la dimensione minima dell'array che possa contenere questa coda senza causare uscita dai limiti.
- 7.2.4 Una "doppia coda" è una lista lineare che consente inserimenti e cancellazioni in *entrambe* le estremità. Implementare le procedure per il trattamento di una "doppia coda".
- 7.2.5 Implementare una coda come una lista ordinata lineare in modo tale che occupi una quantità di memoria proporzionale alla dimensione corrente della coda. (Ved. algoritmi 7.1, 7.3 e 7.4.).

ALGORITMO 7.3 RICERCA IN UNA LISTA

Problema

Progettare ed implementare un algoritmo di ricerca in una lista lineare ordinata di una data chiave alfabetica o di un nome.

Sviluppo dell'algoritmo

Nelle strutture dati a pila e a coda, è sempre determinato in anticipo dove l'elemento corrente debba essere inserito o prelevato; questa situazione favorevole, tuttavia, non viene riscontrata quando occorre compiere queste operazioni su una lista lineare di puntatori. Prima di effettuare operazioni su tale tipo di lista, è necessario compiere una *ricerca* nella lista per stabilire la posizione in cui deve essere fatto lo scambio: questo è il problema che considereremo per primo. Un tale algoritmo di ricerca sarà necessario più avanti, quando verranno considerate le operazioni di inserimento e cancellazione in una lista.

Prima di iniziare il progetto dell'algoritmo, consideriamo un esempio che definisce il problema che vogliamo risolvere. Supponiamo di avere una lunga lista di nomi in ordine alfabetico, come mostrato in Tabella 7.1 e di voler inserire il nome DAVID.

Il compito che dobbiamo svolgere prima che venga fatto l'inserimento è di localizzare *dove* debba essere esattamente inserito DAVID. Dopo un esame della lista dei nomi troviamo con facilità che DAVID deve essere inserito *tra* DANIEL e DROVER. A questo punto non ci interesseremo di come debba avvenire l'inserimento (o la cancellazione), ma ci concentreremo piuttosto sullo sviluppo dell'algoritmo di ricerca di una struttura a lista puntata. Nel decidere dove inserire DAVID, quello che abbiamo fatto è stato esaminare la lista finché non

Location	Name	
1	AARONS	
2	ADAMS	
3	DÁNIEL	← DAVID deve essere
4	DROVER	inserito qui
5	ENGEL	
6	FRY	
7	GRAY	
⋮	⋮	

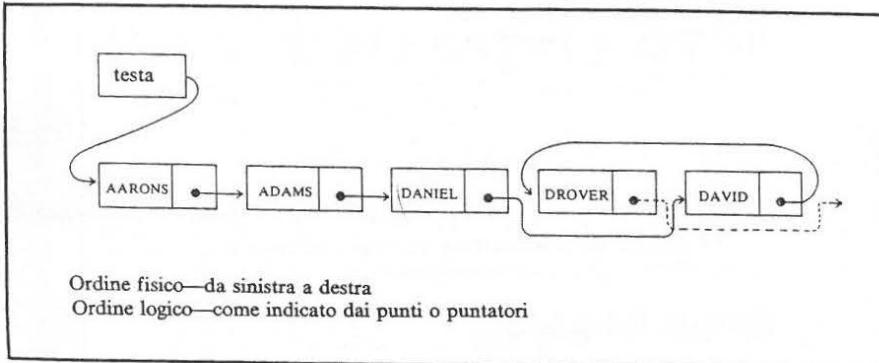


Fig. 7.4.

Differenza tra ordine logico e ordine fisico in una lista ordinata.

abbiamo trovato un nome successivo in ordine alfabetico (nel caso dell'inserimento) o uno uguale (nel caso della cancellazione). Con l'ipotesi di lista ordinata, non sono necessarie ulteriori ricerche. La parte centrale del nostro algoritmo di ricerca è quindi:

1. Finchè il nome cercato risulta in base all'ordine alfabetico successivo al nome corrente della lista,
(a) passa al nome successivo della lista.

Questo algoritmo non considera il fatto che il nome cercato possa figurare *dopo* l'ultimo nome della lista che è potenzialmente infinita, ma questo fatto verrà analizzato più avanti. Fino ad ora il nostro svolgimento è stato semplice ed il problema sarebbe risolto se la lista da esaminare fosse una lista lineare invece che una lista lineare di puntatori.

Il motivo per cui un insieme di nomi viene mantenuto ordinato mediante una lista a puntatori deriva dal fatto che essa permette operazioni efficienti pur mantenendo l'ordine alfabetico. Come vedremo questo non vale per una lista di nomi memorizzata in un array lineare.

A questo punto possiamo considerare la rappresentazione tramite puntatori di una lista ordinata ed il conseguente algoritmo di ricerca. Nella struttura dati a lista di puntatori ogni record o nome ha un campo puntatore associato che punta alla registrazione *successiva* della lista, in un ordine logico (in genere alfabetico). Da questa definizione vediamo che non ci deve essere una corrispondenza diretta tra il modo in cui i nomi sono ordinati fisicamente ed il loro ordine logico. Ciò che rende possibile operare efficienti inserimenti e cancellazioni è proprio questa diminuzione della corrispondenza tra ordine logico e fisico. La differenza è illustrata in figura 7.4.

Nel condurre l'esame di una lista di puntatori occorre seguire l'ordine logico piuttosto che quello fisico. La ricerca in una lista ordinata come quella appena descritta deve cominciare dal primo elemento in ordine logico. Se tutti i nodi della lista hanno la struttura:

```
listnode = record
    listname : nameformat;
    next : listpointer
end
```

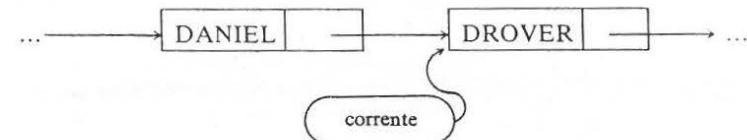
allora sarà necessario usare l'informazione di collegamento *next* per procedere lungo la lista (se la ricerca non termina al primo elemento). Questo corrisponde ad assegnare al campo puntatore del nodo corrente il valore del puntatore *next*:

```
current: = next
```

Se percorriamo in questa maniera l'ordine logico della lista, raggiungeremo sicuramente la posizione desiderata corrispondente a quella in cui il nome cercato o esiste o può essere inserito. Supponendo terminata la ricerca, bisogna chiederci quale informazione debba essere rimandata alla procedura chiamante per permettere un aggiornamento della lista o per indicare se il nome sia o meno presente. Per terminare la ricerca può essere usata una variabile booleana *notfound* che viene posta uguale a *true* quando la procedura di ricerca viene chiamata e che possa successivamente diventare *false*. Nella nostra ricerca, al momento della terminazione, avremo il puntatore al nodo corrente. Cioè:

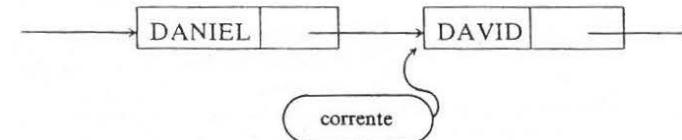
Situazione di inserimento

(per inserire DAVID)



Situazione di uguaglianza o cancellazione

(per trovare o cancellare DAVID)



In entrambe le situazioni, il puntatore al nodo corrente è attualmente l'informazione contenuta nel "nodo DANIEL". Per fare sia un inserimento o una cancellazione l'informazione del puntatore memorizzata nel "nodo DANIEL" deve essere cambiata (cioè nel caso di un inserimento il puntatore a DANIEL dovrebbe puntare al nodo DAVID, invece che a DROVER). Il nostro meccanismo di ricerca potrebbe quindi trattenere il puntatore al nodo precedente e restituirlo alla procedura chiamante quando la ricerca termina; ciò può essere effet-

la condizione *current = null* diventerà vera quando verrà raggiunta la
trovato o superato. Il ciclo terminerà, poiché alla fine (se non prima)
notfound rimarrà "true", a meno che il nome cercato non sia stato
nodo successivo dopo la corrente iterazione. La variabile booleana
previously è *current* puntava allo stesso nodo corrente ed al
precedente in ordine alfabetico prima del nome cercato. I puntatori
che tuttavia non contienevano spettivamente al nodo corrente,
ricorrono in ordine alfabetico nei nodi che precedono quello corrente.
2. Al termine di ogni iterazione nel nodo di ricerca, sarà stato stabilito
esattamente tutti i nodi. Vediamo quali sono:

Note di progetto

```

else found := false
if searchname = current.listname then found := true
if current <> null then
    assert: searchname is equal to or occurs before the name
    assert: searchname is equal to head of list
end;
current := next
begin {move pointers to next node}
previous := current;
previou
else
notfound := false
if searchname <= listname then
    with current do
        while notfound and (current <> null)
            previous node if previous node <> null
            {warning: searchname occurs alphabetically after name at,
            found := false;
            assert: listhead points to head of list}
            previous := null; notfound := true; current := listhead;
        end;
    begin {searches ordered linked list for the presence of searchname}
        previous := listpointer; listhead := listpointer; var found := listhead;
        procedure listsarch{searchname: nameformat; var current, previous;
        listpointer; listhead: listpointer; var found: boolean};
        begin
            next := listpointer;
            if previous = listnode;
                type nameformat = packed array[1..namesize] of char;
                const namesize = 20;
                record
                    listname : nameformat;
                end;
            end;
        end;
    end;
end;

```

Prima di formare la procedura di ricerca, occorre definire le dichiarazioni
nomi sino a lunghezza inferiore a 20 caratteri. Si supporta che i
riconosciuti di tipo necessarie nella procedura chiamante. Le dichiarazioni
sono:

Implementazione in Pascal

1. Assegna al puntatore al nodo precedente il punto della ricerca
2. Aggiorna il puntatore al nodo precedente in modo
che punti al prossimo nodo.
3. Finché la ricerca può procedere e non siamo alla fine del file
varabile booleana il valore true per la controllazione della ricerca
ca.
4. Stabilisci se il nome è stato o no trovato.
5. Ritorna i puntatori al nodo precedente e la situazione
trovata.

(a.1) assegna al puntatore al nodo precedente il punto
alimento

(a.2) aggiorna il puntatore al nodo precedente in modo
che punti al prossimo nodo.

(a.3) assicura il puntatore al nodo precedente della
lista

(a.4) se il nome cercato precede all'alfabeticamente il nome corrente della
lista

3. Finché la ricerca può procedere e non siamo alla fine del file
varabile booleana il valore true per la controllazione della ricerca
ca.

2. Inizializza il puntatore al nodo precedente a null ed assegna alla
lista *listhead*. Per provvedere al punto in cui il nome deve essere inserito
di ricerca dovrà fornire un puntatore al primo elemento della
lista *listhead*, o incontando la posizione (virtuale o
reale) del nome cercato, o riconoscendo la posizione (virtuale o
reale) di ricerca terminando o raggiungendo la fine della lista. Alla procedura
il ciclo di ricerca terminerà o raggiungendo la posizione (virtuale o
reale) del nome cercato o riconoscendo la fine della lista, avremo:

Il ciclo di ricerca terminerà o raggiungendo la posizione (virtuale o
reale) del nome cercato, o incontando la posizione (virtuale o
reale) di ricerca terminando o raggiungendo la fine della lista. Alla procedura
il ciclo di ricerca terminerà o raggiungendo la posizione (virtuale o
reale) del nome cercato, o incontando la posizione (virtuale o
reale) di ricerca terminando o raggiungendo la fine della lista, avremo:

Se WILSON fosse l'ultimo nome della lista, avremo:
stabilità è segnalata dal campo puntatore che in questo caso vale null.
della lista non ha successori, il suo puntatore puntata a niente: questa
WILSON è noi siamo cercando ZELMAN. Poiché l'ultima nome
cercando sia oltre la fine della lista (cioè se è l'ultimo nome della lista è
ricerca chiude di poter trattare il caso in cui il nome che stiamo
ricerca chiude di poter trattare il caso in cui il nome cercando sia
l'ultimo nome della lista. Il meccanismo di
terminazione che è stato menzionato in precedenza. Il meccanismo di
terminazione da fare riguarda il problema della
sequenza passi prima di muovere verso il nodo successivo:
tutto (supponendo di usare l'instruzione with) usando ad ogni nodo i

Descrizione dell'algoritmo

Il algoritmo può ora essere descritto dettato-
cessere inizializzato a null. L'algoritmo può ora essere descritto dettato-
prima del nome della lista, il puntatore al nodo precedente deve
lista *listhead*. Per provvedere al punto in cui il nome deve essere inserito
di ricerca dovrà fornire un puntatore al primo elemento della
lista *listhead*. Alla procedura si invia il puntatore alla lista e il nome
del nome cercato. Oltre a ciò si invia il nome cercato (cioè se è l'ultimo nome
della lista non ha successori, il suo puntatore puntata a niente: questa
WILSON è noi siamo cercando ZELMAN). Poiché l'ultima nome
cercando sia oltre la fine della lista (cioè se è l'ultimo nome della lista è
ricerca chiude di poter trattare il caso in cui il nome che stiamo
ricerca chiude di poter trattare il caso in cui il nome cercando sia
l'ultimo nome della lista. Il meccanismo di
terminazione che è stato menzionato in precedenza. Il meccanismo di
terminazione da fare riguarda il problema della
sequenza passi prima di muovere verso il nodo successivo:
tutto (supponendo di usare l'instruzione with) usando ad ogni nodo i

WILSON null ← WATSON

Se WILSON fosse l'ultimo nome della lista, avremo:
stabilità è segnalata dal campo puntatore che in questo caso vale null.
della lista non ha successori, il suo puntatore puntata a niente: questa
WILSON è noi siamo cercando ZELMAN. Poiché l'ultima nome
cercando sia oltre la fine della lista (cioè se è l'ultimo nome della lista è
ricerca chiude di poter trattare il caso in cui il nome che stiamo
ricerca chiude di poter trattare il caso in cui il nome cercando sia
l'ultimo nome della lista. Il meccanismo di
terminazione che è stato menzionato in precedenza. Il meccanismo di
terminazione da fare riguarda il problema della
sequenza passi prima di muovere verso il nodo successivo:
tutto (supponendo di usare l'instruzione with) usando ad ogni nodo i

current := next
previous := current;

fine della lista. L'ulteriore condizione di terminazione in cui il nome cercato viene trovato o ricorre alfabeticamente dopo il corrente nome della lista, forzerà la terminazione prima che venga raggiunta la fine della lista.

3. Un perfezionamento del presente algoritmo di ricerca richiede di comportare l'uso di una "sentinella" che consenta la terminazione della ricerca nel caso in cui il nome cercato sia oltre la fine della lista. Questo comporterebbe collegare l'ultimo nodo alla sentinella, che può essere posta uguale al nome cercato. Una volta fatto ciò, il test:

current < > nil

può essere abbandonato e sostituito da un test "esterno" al ciclo per stabilire se sia stato trovato il nome o la sentinella.

4. L'algoritmo di ricerca che abbiamo costruito può essere usato insieme alle procedure per l'inserimento e la cancellazione in una lista poiché fornisce le informazioni necessarie per queste modifiche.
5. L'algoritmo di ricerca binaria *non può essere usato* per esaminare le liste di puntatori, anche se ordinate, poiché non esiste corrispondenza tra ordine logico e ordine fisico; ne segue che una struttura a lista di puntatori impone una ricerca sequenziale. Per liste molto lunghe possono essere utilizzate vantaggiosamente altre alternative di memorizzazione (ad esempio alberi).

Applicazioni

Trattamento di liste brevi di puntatori.

Problemi supplementari

- 7.3.1 Modificare l'algoritmo di ricerca in modo che esamini una lista di cui non si possa ipotizzare l'ordinamento.
- 7.3.2 Progettare un algoritmo che esamini una lista che contenga una sentinella (vedi nota 3).
- 7.3.3 Progettare ed implementare un algoritmo di esame di una lista che utilizzi gli array sia per i nomi che per i puntatori.
- 7.3.4 Progettare ed implementare un algoritmo che riceva in ingresso una lista ordinata di nomi da cercare. Il compito richiede di stabilire e riferire l'assenza o la presenza di questi nomi in una seconda lista ordinata molto più lunga.

ALGORITMO 7.4 INSERIMENTO E CANCELLAZIONE DA UNA LISTA ORDINATA

Problema

Progettare ed implementare delle procedure per inserire e cancellare elementi da una lista lineare ordinata di puntatori.

Sviluppo dell'algoritmo

Nell'algoritmo precedente era già stata contemplata l'idea che le liste di puntatori sono efficienti per il mantenimento dell'ordine logico di insiemi di dati soggetti a frequenti inserimenti e cancellazioni. Cerchiamo ora di capire perché le liste di puntatori rivelano questa efficienza prima di procedere con gli algoritmi di inserimento e cancellazione.

Fig. 7.5.
Inserimento di un elemento in una lista ordinata:
(a) *Prima dell'inserimento di DAVID;*
(b) *Dopo l'inserimento di DAVID.*

Locazione	Nome
1	AARONS
2	ADAMS
3	BORDEN
4	CLOVER
5	DANIEL
6	DROVER
:	:
100	ZELMAN

(a)

Locazione	Nome
1	AARONS
2	ADAMS
3	BORDEN
4	CLOVER
5	DANIEL
6	DAVID
7	DROVER
:	:
101	ZELMAN

(è stato inserito
tutti slittati
di una posizione)

(b)

Come esempio del tipo di problema che ci proponiamo di affrontare, supponiamo di avere una lista ordinata di cognomi. Dobbiamo considerare le operazioni di inserimento di un elemento nella lista in modo da rispettarne l'ordine e la cancellazione di un elemento dalla lista. Supponiamo di voler inserire il nome DAVID nella lista mostrata in Fig. 7.5 (a).

Tale lista potrebbe essere rappresentata in un calcolatore mediante un array di nomi. Dalla figura 7.5 (b) vediamo che per inserire DAVID nella lista, tutti gli elementi che sono successivi in ordine alfabetico al nome devono essere fatti avanzare di una posizione nell'array. Al crescere delle dimensioni della lista il numero di dati che devono essere spostati diventa eccessivo, ne segue che in questo caso non è conveniente l'uso di un array per il trattamento di una lista di nomi soggetta a frequenti operazioni di inserimento e cancellazione. Con una disposizione di questo tipo ci aspettiamo di compiere in media uno spostamento di metà elementi per ogni operazione.

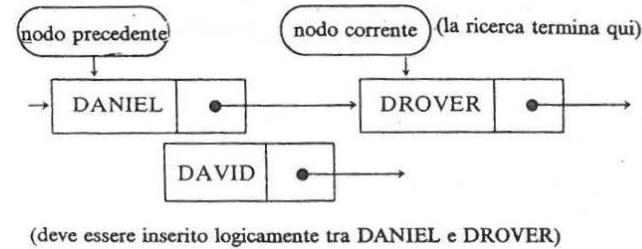
Fig. 7.6.
Inserimento di un
elemento in una lista
ordinata di puntatori:
(a) Inserimento fisico
di DAVID;
(b) Inserimento
logico di DAVID.

Locazione fisica	Nome	Puntatore (locazione fisica del successore)
1	AARONS	2
2	ADAMS	3
3	BORDEN	4
4	CLOVER	5
5	DANIEL	6 ← nodo precedente
6	DROVER	7 ← nodo corrente (dove termina la ricerca logica)
:	:	:
100	ZELMAN	[?]
101	DAVID	[?] successore logico di DANIEL

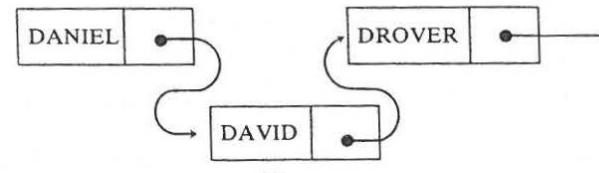
(a)

Locazione fisica	Nome	Puntatore (locazione fisica del successore logico)
1	AARONS	2
2	ADAMS	3
3	BORDEN	4
4	CLOVER	5
5	DANIEL	101
6	DROVER	7
:	:	:
100	ZELMAN	nil
101	DAVID	[6]

(b)



(a)



(b)

Fig. 7.7.
Inserimento di un
elemento nel mezzo
di una lista
di puntatori:
(a) Prima
dell'inserimento;
(b) Dopo
l'inserimento.

Chiaramente, se dobbiamo trattare una lista soggetta a frequenti operazioni di inserimento e cancellazione, è necessario minimizzare la quantità di movimenti di dati. La situazione più desiderabile è quella in cui *nessun* elemento viene spostato dopo un inserimento o una cancellazione. È certamente possibile inserire DAVID dopo ZELMAN alla fine della lista, senza dover spostare alcuno degli altri membri della lista; in questo modo, però la lista non è più in ordine alfabetico. A questo punto la nostra esigenza è quella di trovare un meccanismo che ci permetta di inserire fisicamente i nuovi elementi alla fine della lista, ma allo stesso tempo mantenere l'ordine logico. Possiamo a questo scopo associare ad ogni nome un campo puntatore che punti alla posizione fisica del suo successore logico nella lista: una volta fatto ciò, l'ordine fisico perde rilevanza. Se partiamo dalla configurazione di Fig. 7.6 (a), allora i passi per l'inserimento logico di DAVID nella lista sono mostrati in Fig. 7.6(b).

Dopo l'inserimento logico di DAVID vediamo che DANIEL è collegato con la posizione fisica di DAVID ed il puntatore di DAVID punta alla posizione fisica di DROVER. L'inserimento di un nuovo elemento ha comportato solo un piccolo cambiamento a quella che era la struttura originale della lista. Una struttura a puntatori permette quindi inserimenti e cancellazioni a basso costo.

Sarebbe possibile utilizzare degli array per implementare strutture dati a liste di puntatori, ma la memorizzazione è più pratica ed economica usando puntatori e record. Con quest'ultimo approccio, è necessaria una quantità di memoria pari al numero dei nodi contenuti

nella lista, mentre con l'implementazione con l'uso degli array era necessario preallocare una quantità fissa di memoria.

Il nostro compito è ora considerare il meccanismo di inserimento e cancellazione mediante l'uso di puntatori. Come abbiamo già visto, prima di poter fare una operazione (inserimento o cancellazione), è necessario trovare l'esatta posizione della lista dove poterla fare. La procedura illustrata nell'algoritmo 7.3 può essere utilizzata per trovare la posizione necessaria.

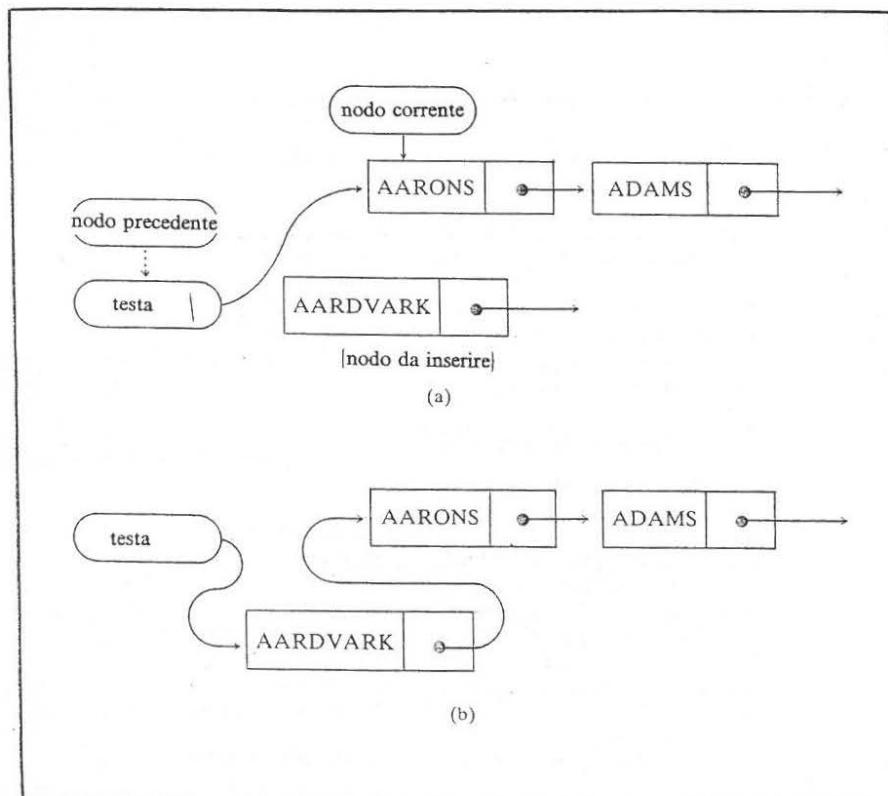
Per l'inserimento (o la cancellazione) devono quindi essere compiuti due passi:

1. Esamina la lista in ordine logico finché non trovi un nome che non precede (in ordine alfabetico) quello da inserire.
2. Inserisci il nome (o cancellalo).

La ricerca è stata già considerata, quindi possiamo procedere con lo svolgimento della seconda parte del meccanismo.

Per evitare confusioni possiamo considerare per primo l'inserimento. Usando l'esempio precedente, la struttura della lista prima e dopo

Fig. 7.8.
Inserimento di un
elemento prima del
primo elemento della
lista di puntatori:
(a) Prima
dell'inserimento;
(b) Dopo
l'inserimento.



l'inserimento del nome DAVID è data rispettivamente in Fig. 7.7(a) e (b). Per permettere l'inserimento, il puntatore di DANIEL deve essere aggiornato per puntare al nodo inserito DAVID ed il puntatore del nuovo elemento deve puntare al nodo che seguiva in ordine logico DANIEL prima dell'operazione. I cambiamenti dei puntatori saranno quindi:

puntat. al nodo prec:=posizione del nodo inserito
puntat. del nodo inserito:= pos. nodo corrente

Sarà necessario far precedere questi passi dalla creazione di un nodo, che può essere effettuata dalla funzione *new* del Pascal. Da un controllo di questo meccanismo su alcuni esempi, è possibile rilevare che esso funziona correttamente nel caso più generale; vogliamo tuttavia essere sicuri che esso funzioni correttamente anche nel caso in cui l'elemento debba essere inserito: prima del primo elemento, dopo l'ultimo e come primo elemento della lista.

In Fig. 7.8 viene mostrato il caso in cui il nodo da inserire debba essere posto prima del primo elemento della lista. Vediamo che l'inserimento prima del primo nodo richiede che il puntatore alla testa della lista venga aggiornato, poiché non c'è nessun puntatore al nodo precedente. L'inserimento davanti al primo nodo deve perciò essere trattato in maniera particolare: dobbiamo scoprire quando possa essere fatto. L'algoritmo di ricerca in una lista può fornire l'informazione necessaria. Quando terminiamo la ricerca al primo nodo della lista, vediamo che il puntatore del nodo corrente punta alla testa della lista. Possiamo quindi utilizzare il seguente test ed aggiornamento del puntatore:

se il puntatore del nodo precedente è uguale a *nil*, allora
(a) aggiorna la testa della lista in modo che punti all'ultimo nodo inserito.

Il puntatore del nodo inserito punterà al suo successore come nel caso generale. Dopo aver trattato questo caso particolare, possiamo ora considerare i rimanenti due casi menzionati in precedenza; utilizzando esempi e diagrammi troviamo che sono già compresi nella struttura generale.

I passi per l'aggiornamento del puntatore dopo l'inserimento di un nuovo nodo (con l'inclusione dei casi particolari) sono quindi:

1. Predisponi il puntatore del nodo inserito in modo che punti al suo successore
2. Se il puntatore del nodo precedente non punta alla testa della lista, allora
 - (a) predisponi il puntatore del nodo precedente in modo che punti al nodo inserito
 - altrimenti
 - (a') regola la testa della lista in modo che punti all'ultimo nodo inserito.

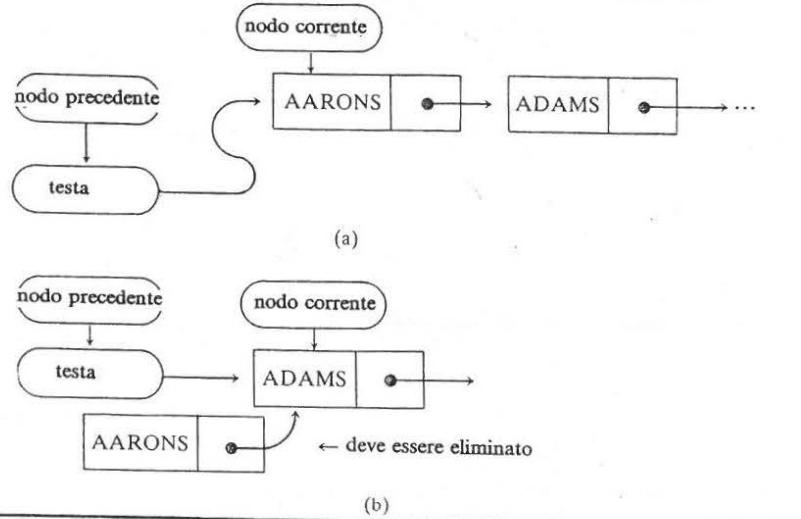


Fig. 7.9.
Cancellazione di un
elemento da una lista di
puntatori:
(a) Prima della
cancellazione;
(b) Dopo la
cancellazione.

Siamo ora in grado di descrivere in maniera completa i passi di ricerca ed inserimento, ma rimanderemo questa descrizione finché non sarà stato considerato il caso della cancellazione di un elemento da una lista.

Come prima cosa potremmo supporre che sia necessario lo stesso procedimento sia per la cancellazione che per l'inserimento: un esempio servirà a chiarire cosa comporti questa affermazione. Supponiamo di voler cancellare il nome DAVID dalla lista; con riferimento alla Fig.7.9 vediamo che il puntatore del nodo precedente (in questo caso quello di DANIEL), deve essere modificato in modo che punti al successore al nodo appena cancellato, ovvero:

puntatore al nodo precedente: = puntatore del nodo cancellato

Sarà inoltre necessario un passo per recuperare la memoria occupata dal nodo cancellato, a questo proposito può essere usata la funzione Pascal *dispose*.

Come per l'inserimento, possiamo prendere in esame casi non troppo frequenti di cancellazione, come quella del primo elemento, dell'ultimo elemento e dell'unico elemento di una lista. In Fig.7.10 sono mostrati i diagrammi della cancellazione del primo elemento di una lista.

In questo caso particolare è il puntatore alla testa della lista che deve essere aggiornato e non quello al nodo precedente: questo caso deve quindi essere analizzato e trattato adeguatamente, cioè:

1. Se il puntatore al nodo precedente non punta a *nil*, allora

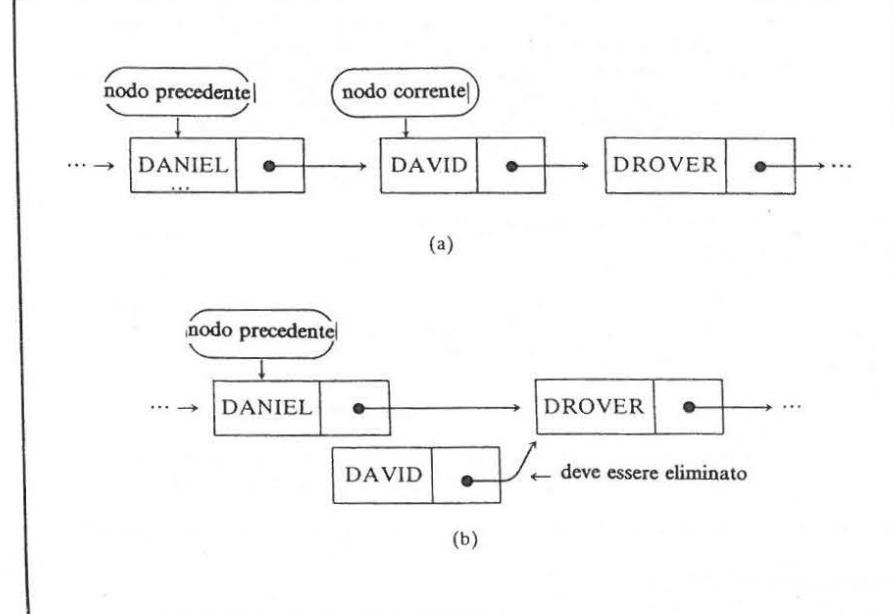


Fig. 7.10.
Cancellazione del
primo elemento da
una lista di
puntatori:
(a) Prima della
cancellazione;
(b) Dopo la
cancellazione.

(a) regola il puntatore del nodo precedente in modo che punti al successore del nodo cancellato
altrimenti
(a') regola la testa della lista in modo che punti al successore del nodo cancellato (cioè il nuovo nodo iniziale della lista).

Un'indagine circa i rimanenti due casi particolari porta alla conclusione che essi non necessitano di speciali trattamenti.

Siamo ora pronti per la completa descrizione delle procedure di inserimento e cancellazione.

Descrizioni degli algoritmi

(1) Inserzione in lista

1. Assegna il nome da inserire ed il puntatore alla testa della lista.
2. Inizializza il puntatore al nodo precedente a *nil* e quello al nodo corrente alla testa della lista.
3. Esamina la lista per trovare la posizione del nome da inserire e restituisci i puntatori ai nodi predecessori e successori logici.
4. Crea il nuovo nodo.
5. Memorizza nel nuovo nodo il nome da inserire.
6. Regola il puntatore del nodo inserito in modo che punti al suo successore logico.
7. Se non viene inserito un nuovo nome all'inizio della lista, allora

- (a) regola il puntatore del nodo precedente in modo che punti al nodo inserito
altrimenti
- (a') regola il puntatore alla testa della lista in modo che punti al nodo inserito in testa alla lista.
- 8. Ritorna la lista aggiornata e la testa della lista.

(2) Cancellazione da una lista

1. Definisci il nome da cancellare ed il puntatore alla testa della lista.
2. Inizializza a *nil* il puntatore al nodo precedente ed il puntatore al nodo corrente alla testa della lista.
3. Ricerca nella lista il nome da cancellare e ritorna i puntatori al predecessore logico ed al nodo stesso (se trovato).
4. Se è stato trovato il nome da cancellare, allora
 - (a) se il nodo da cancellare non è la testa della lista, allora
 - (a.i) fa in modo che il puntatore del nodo precedente punti al successore del nodo da cancellare
altrimenti
 - (a'.1) fa in modo che il puntatore alla testa della lista punti al successore del nodo da cancellare;
 - (b) elimina il nodo
altrimenti
 - (a') stampa che il nome non può essere cancellato.
5. Ritorna la testa della lista e la lista aggiornata.

Implementazione in Pascal

La descrizione del record è:

```
const namesize = 20;
type nameformat = packed array[1..namesize] of char;
listpointer = ^listnode;
listnode = record
  listname : nameformat;
  next : listpointer
end;
```

(1) Inserzione in lista

```
procedure listinsert (newname: nameformat; var listhead:
listpointer);
var previous {pointer to previous node},
current {pointer to current node},
newnode {pointer to new node to be inserted}: listpointer;
found: boolean; {true if newname found}
```

```
begin
  {assert: listhead points to head of list}
  previous := nil; current := listhead;
  listsearch (newname, current, previous, listhead, found);

  {assert: newname occurs alphabetically earlier than name at current node if list not empty}
  new(newnode);
  newnode^.listname := newname;
  if previous <> nil then
    begin
      newnode^.next := previous^.next;
      previous^.next := newnode
    end
  else
    begin
      newnode^.next := listhead;
      listhead := newnode
    end
  {assert: node containing newname inserted in list in alphabetical order}
end
```

(2) Cancellazione dalla lista

```
procedure listdelete (oldname: nameformat; var listhead: listpointer
var previous {pointer to previous node},
current {pointer to current node}: listpointer;
found {true if name for deletion (oldname) found}: boolean;

begin
  {assert: listhead points to head of list}
  previous := nil; current := listhead;
  listsearch (oldname, current, previous, listhead, found);
  {assert: oldname is equal to or occurs before name at current node if a name present}
  if found then
    begin
      if previous <> nil then
        previous^.next := current^.next
      else
        listhead := current^.next;
        dispose(current)
    end
  {assert: node containing oldname removed from list and link restored}
  else
    writeln('not present—cannot delete', oldname)
  end
```

Note di progetto

1. Gli algoritmi di inserimento e cancellazione in lista sono dominati dall'esame della lista. Il costo della ricerca è proporzionale alla lunghezza della lista e richiede in media l'esame di metà della lista e, nel caso peggiore quello dell'intera lista.
2. La correttezza di questi algoritmi dipende dalla correttezza dell'algoritmo di ricerca; entrambi gli algoritmi di inserimento e cancellazione, mantengono l'ordine logico della lista determinato dall'insieme dei puntatori associati.

Applicazioni

Trattamento di liste relativamente piccole ordinate che sono soggette a frequenti operazioni di inserimento e cancellazione.

Problemi supplementari

- 7.4.1 Progettare ed implementare un algoritmo che inserisca elementi alla fine di una lista. Per effettuare l'inserimento, l'algoritmo non necessita di un esame della lista.
- 7.4.2 Progettare ed implementare algoritmi di inserimento in lista e cancellazione che utilizzino array per memorizzare sia le informazioni che i puntatori. La memoria inutilizzata (prima e dopo le cancellazioni) può essere considerata come una lista libera, cioè può essere collegata insieme in una lista. Ogni volta venga cancellato un nuovo elemento, esso viene posto alla fine della lista libera e lo spazio per l'inserimento di un nuovo elemento nella lista viene preso dalla coda della lista libera.
- 7.4.3 Progettare ed implementare un algoritmo che fonda due liste ordinate di puntatori.
- 7.4.4 In alcune applicazioni è necessario estrapolare dalla lista informazioni più frequenti di altre. In queste condizioni il modo migliore di costruire tale lista è quello di mettere le informazioni più richieste all'inizio della lista, e le successive in ordine decrescente fino alle meno richieste: tale ordine non è quasi mai conosciuto in anticipo. Per fare in modo che una lista raggiunga tale ordine, ogni volta che un elemento viene richiesto, esso viene scambiato con il suo predecessore: con questo schema gli elementi più richiesti verranno posti all'inizio della lista. Progettare un algoritmo che esamini e tratti una lista di questo tipo.
- 7.4.5 Un diverso modo di affrontare il problema 7.4.4 è di spostare sempre un elemento all'inizio della lista dopo che è stato richiesto. Progettare ed implementare algoritmi che esaminino e trattino una lista di puntatori in questa forma.

7.4.6 Seguendo il tema dei precedenti due problemi, aggiungere ad ogni nodo un parametro indicante la frequenza con cui un elemento è richiesto in modo che ogni elemento venga spostato più in alto nella lista solo quando la sua frequenza di richieste è maggiore di quella del suo predecessore. Tale parametro deve poter essere aggiornato.

7.4.7 Progettare ed implementare un algoritmo per trattare liste con puntatori doppi. In tali liste ogni nodo ha due puntatori (esclusi il primo e l'ultimo) uno che punta al predecessore e l'altro al successore. (Questo considera il caso di liste trasversali con direzioni sia in avanti che all'indietro).

ALGORITMO 7.5 RICERCA IN UN ALBERO BINARIO

Problema

Progettare ed implementare un algoritmo per esaminare un albero binario ordinato per una data chiave o nome.

Sviluppo dell'algoritmo

Abbiamo visto in precedenza, nella descrizione dell'algoritmo di ricerca binaria (algoritmo 5.7), come possano essere trovate facilmente le informazioni in un array ordinato. La sola difficoltà in una struttura dati di questo tipo è che è molto costosa da mantenere se devono essere fatti frequentemente inserimenti e cancellazioni. La struttura dati a lista di puntatori, (algoritmi 7.3 e 7.4), evita tale difficoltà ma, nel farlo, permette unicamente la ricerca sequenziale e non la più efficiente ricerca binaria. Dato che la ricerca sequenziale è costosa, in modo particolare per liste lunghe, sarebbe più desiderabile essere in possesso di una struttura dati che permetta inserimenti e cancellazioni efficienti, ma che conduca anche una ricerca efficiente: fortunatamente esiste una struttura dati che ce lo permette.

Abbiamo visto precedentemente che l'algoritmo di ricerca binaria (algoritmo 5.7), era molto efficiente nella ricerca in un array ordinato, poiché il meccanismo di ricerca trattava i dati ordinati come se fossero organizzati ad albero binario. In questo contesto, in un array ordinato di n elementi, l'individuazione di uno qualsiasi di essi si tradurrebbe

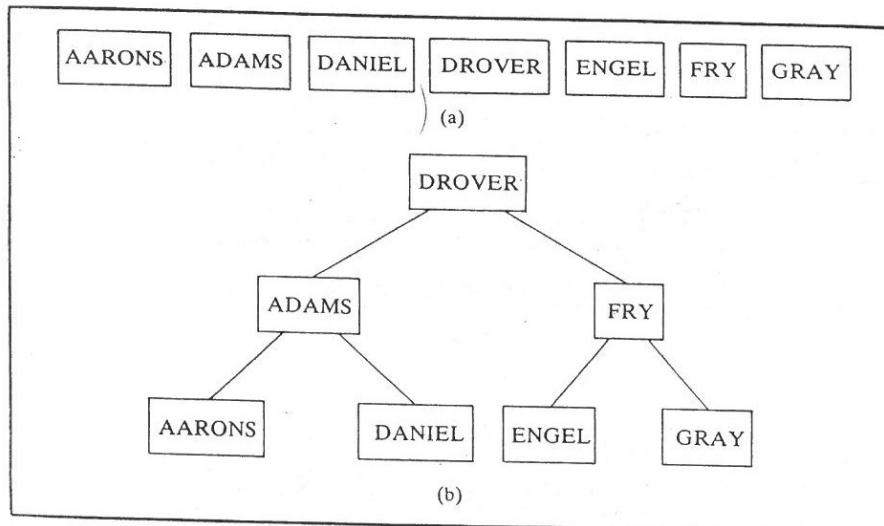
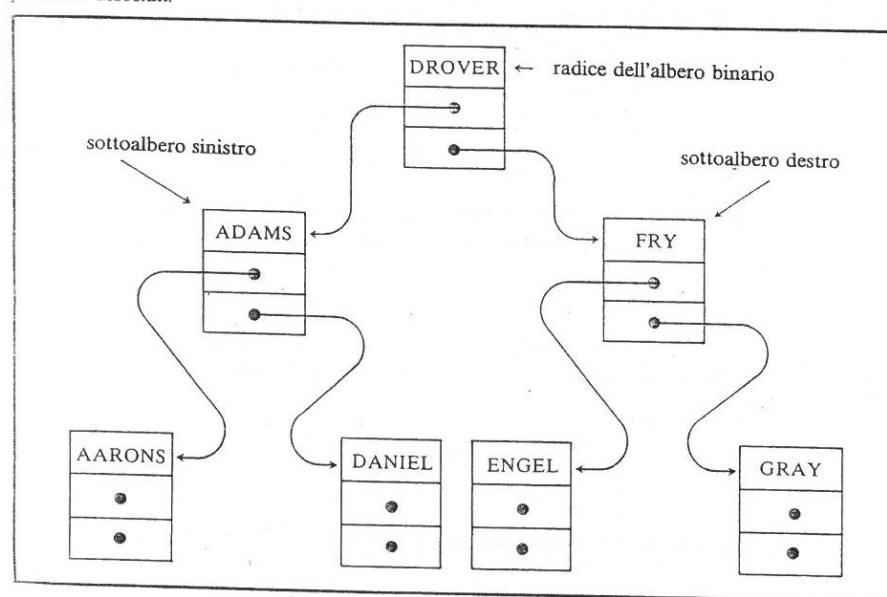


Fig. 7.11.
(a) Dati ordinati;
(b) Corrispondente
albero binario.

nell'esame di al massimo $\lceil \log_2 n \rceil$ elementi; ovvero, il meccanismo di ricerca sovrappone al file ordinato un albero immaginario. La sola alternativa sarebbe quella di costruire l'albero usando un insieme di puntatori (Fig. 7.11).

Per costruire esplicitamente la struttura dati corrispondente alla struttura ad albero binario sovrapposta, vediamo che ogni nodo dell'albero puntaori associati.



dovrà avere associati due puntatori, uno che punti alla radice del sottoalbero sinistro e l'altro alla radice del sottoalbero destro (ved. Fig. 7.12).

La struttura dati così creata è un *albero binario ordinato*. A questo punto vale la pena di definire cosa sia un *albero binario ordinato*: esso è un insieme finito di nodi che può essere o vuoto, o consistente di un nodo radice come antenato di due distinti alberi chiamati *sottoalbero sinistro* e *sottoalbero destro* del nodo radice. Ogni nodo del sottoalbero sinistro, ogni volta venga confrontato con uno del sottoalbero destro, soddisfa una relazione ordinata (alfabetica o numerica) dell'albero; i sottoalberi destro e sinistro possono avere qualsiasi livello vuoto.

Possiamo ora chiederci cosa abbiamo guadagnato da una siffatta rappresentazione dei dati. Prima di cercare di rispondere a questa domanda, dovremo riprendere in considerazione le strutture a liste puntate. Abbiamo visto che il vantaggio di tali strutture rispetto a quelle ordinate era che permettevano inserimenti e cancellazioni con pochi cambiamenti rispetto alla struttura dati iniziale. Come vedremo più avanti, inserimenti e cancellazioni usando una struttura ad albero binario ordinato, sono operazioni efficienti e relativamente semplici.

Il nostro attuale interesse è di sviluppare un algoritmo che esamina una struttura ad albero come quella appena descritta. Supponiamo di voler cercare nell'albero binario ordinato il nome DANIEL.

Ci si dovrebbero aspettare considerevoli somiglianze tra l'algoritmo di ricerca binaria per un insieme ordinato di dati e la ricerca in un albero binario ordinato. Se cerchiamo il nome DANIEL utilizzando l'algoritmo di ricerca binaria, cominceremo con il confronto del nome centrale, DROVER, con DANIEL. Considerando ora una struttura ad albero binario ordinato, ciò è equivalente al confronto del nome cercato, DANIEL, con il nome alla radice dell'albero. Il confronto dei due nomi nel caso di ricerca binaria, indica che il nome DANIEL si trova nella *prima metà* della lista di nomi. Il nome centrale, ADAMS, della prima metà della lista, sarà il successivo da esaminare. In modo equivalente, nel caso di struttura ad albero binario, il nome ADAMS, sarà la radice del sottoalbero sinistro. Per localizzare questo nome senza usare una computazione (come nella ricerca binaria), è possibile impiegare il puntatore a sinistra del nodo radice che punta alla radice del sottoalbero sinistro. Supponendo di aver già dato una appropriata definizione del nodo e di essere all'interno di un'istruzione *with*, avremo:

current: = left

I passi per la ricerca di DANIEL saranno:

DANIEL < DROVER → vai al sottoalbero sinistro (radice ADAMS)

DANIEL > ADAMS → vai al sottoalbero destro (radice DANIEL)
DANIEL = DANIEL → termine della ricerca

Ad ogni passo lo spostamento in uno dei due sottoalberi (destro o sinistro) dipenderà dal nodo corrente e dal nome cercato, cioè,

se il nome del nodo ricorre alfabeticamente prima del nome cercato, allora

- (a) spostati alla radice del sottoalbero sinistro
- altrimenti
- (a') spostati alla radice del sottoalbero destro.

Questi passi ci forniscono un mezzo per attraversare l'albero in maniera desiderata, ma non ci consentono di fermarci sia quando il nome cercato è stato trovato, sia quando la ricerca non ha avuto buon esito, di conseguenza la precedente istruzione composta potrebbe essere eseguita solo una volta stabilito che al nodo corrente non esiste uguaglianza.

Allora avremo:

se il nodo corrente è quello cercato, allora

- (a) ricerca terminata
- altrimenti
- (a') spostati al sottoalbero sinistro o destro.

Sarà inoltre necessario considerare il caso in cui la ricerca non fornisca esito positivo; per conoscere quello che occorre fare in questo caso, supponiamo che il nome cercato DAVID non sia contenuto nell'albero. Seguendo il percorso alla ricerca di DAVID, vediamo che sicuramente raggiungeremo il nome DANIEL, al cui punto decideremo che DAVID si trova nel sottoalbero destro: tuttavia DANIEL non possiede sottoalberi destri. Ai nodi senza successore viene assegnato un puntatore con valore *nil* che ne indica la mancanza. La ricerca può quindi continuare finché non troveremo il nome cercato o un nodo avente come puntatore al percorso che vogliamo seguire uguale a *nil*.

Poichè vogliamo usare l'algoritmo di ricerca in un albero come parte degli algoritmi di inserimento e cancellazione, dovremo utilizzare pure un puntatore al nodo precedente. La ragione di ciò sarà chiarita più avanti, ma per ora sarà sufficiente dire che tali operazioni sono simili a quelle di inserzione e cancellazione in lista che richiedono l'uso di un puntatore al nodo precedente. Un istruzione nella forma:

previous: = *current*

eseguita *prima* dell'aggiornamento della variabile corrente del nodo, salverà l'informazione richiesta. Abbiamo dunque stabilito come possono essere ottenute le informazioni necessarie per l'inserimento e la cancellazione; l'unico compito rimastoci è quello di stabilire se la chiave cercata sia o meno stata trovata.

Abbiamo già visto che non c'è alcun modo di far terminare il ciclo di ricerca se non quello di trovare un'uguaglianza o di arrivare alla fine di un percorso dell'albero. Il nostro compito è, quindi, di determinare un modo per terminare la ricerca nell'attuale situazione: ciò ci riconduce

alla domanda su come terminare dopo il riscontro di un'uguaglianza.

Finchè non è stato trovato il nome cercato, la ricerca rimane in uno stato di non trovato che ci permette di continuare se esiste ancora un possibile percorso nell'albero. Tuttavia, non appena viene trovata l'uguaglianza, la ricerca passa nello stato di trovato: possiamo realizzare questo meccanismo utilizzando una variabile booleana *notfound* che rimanga vera finchè non è stato trovato l'elemento. Un test sullo stato di questa variabile prima del termine della ricerca ci renderà possibile determinare se la ricerca sia stata o meno positiva. Possiamo usare:

found: = *not (notfound)*

È stato quindi completato lo sviluppo dell'algoritmo di ricerca in un albero binario ordinato.

Descrizione dell'algoritmo

1. Definisci la chiave cercata ed il puntatore alla radice dell'albero.
2. Assegna lo stato di non trovato e posiziona il puntatore del nodo corrente alla radice e quello del nodo precedente a *nil*.
3. Finchè non viene trovata la chiave cercata ed esiste nell'albero ancora un percorso valido da seguire
 - (a) se la chiave cercata non è nel nodo corrente, allora
 - (a.1) salva il puntatore al nodo corrente nel puntatore al nodo precedente,
 - (a.2) se la chiave cercata è nel sottoalbero sinistro, allora
 - (2.a) fa che il puntatore al nodo corrente punti alla radice del sottoalbero sinistro,
 - altrimenti
 - (2'.a) fa che il puntatore del nodo corrente punti alla radice del sottoalbero destro
 - (a'.1) pon la condizione di non trovato uguale a falso per segnalare la terminazione della ricerca.
4. Definisci se la chiave è stata o meno trovata.
5. Ritorna i puntatori al nodo precedente, al nodo corrente ed una variabile indicante se la ricerca ha avuto o meno successo.

Implementazione in Pascal

```
const namesize = 20;
type nameformat = packed array[1..namesize] of char;
treepointer = ^treenode;
```

```

treenode = record
    treename : nameformat;
    left : treepointer;
    right : treepointer
end;

procedure treesearch (searchname: nameformat; var current,
previous: treepointer; root: treepointer; var found: boolean);
var notfound {if false indicates search must terminate}: boolean;

begin {searches an ordered linked binary tree for the presence of
searchname}
    {assert: root points to root of tree}
    previous := nil; current := root; notfound := true;
    {invariant: after current iteration if searchname were (or is) present
it would be in subtree with root current node and predecessor
previous node}
    while notfound and (current<>nil) do
        with current↑ do
            if searchname<>trenname then
                begin {save previous node and proceed to appropriate
subtree}
                    previous := current;
                    if trenname>searchname then
                        current := left
                    else
                        current := right
                end
            else
                notfound := false;
            {assert: searchname found at current node ∨ could be inserted in
subtree of previous node}
            found := not notfound
    end

```

Note di progetto

- Il costo della ricerca in un albero binario ordinato è proporzionale al numero di nodi da esaminare prima del termine della ricerca: tale numero dipende dalla struttura dell'albero. Nel peggior caso possibile (in cui vengono collegati tutti i puntatori a destra o a sinistra) la struttura dell'albero può degenerare in una lista lineare di puntatori: in questo caso viene esaminata in media la metà dei nodi. Una cosa interessante da fare è determinare la dimensione di una struttura ad albero "media"; un modo per determinare tale parametro è di considerare alberi formati da dati scelti in maniera casuale di dimensione n . È ora necessario calcolare la media della ricerca di tutti gli elementi per i risultanti $n!$ alberi. L'analisi è piuttosto complessa, ma porta alla conclusione che in media sarà

necessario esaminare poco meno di $2\log_2 n$ nodi. Ciò regge il confronto con i $\lceil \log_2 n \rceil$ nella ricerca binaria e bilancia perfettamente i casi ad albero.

- Al termine di ogni iterazione del ciclo di ricerca sarà stato stabilito che il nome cercato si trova nel nodo *corrente* (se è presente), o nel sottoalbero la cui radice è il nodo *corrente*. Il puntatore *current* punterà al nodo successivo da esaminare ed il puntatore *previous* punterà a quello più recentemente esaminato. La variabile booleana *notfound* rimarrà vera dopo ogni iterazione finché non verrà trovata la chiave (*key*). Il ciclo terminerà poiché alla fine (se non prima) la condizione *current = nil* sarà vera quando non ci saranno più ramificazioni dell'albero da seguire. L'ulteriore condizione di terminazione, quella in cui la chiave è uguale al *searchname* può forzare una precoce terminazione del ciclo.
- Questo algoritmo è strutturalmente simile a quello di ricerca in lista e quello di ricerca binaria.
- L'algoritmo di ricerca appena sviluppato può essere usato unitamente alle procedure di inserimento e cancellazione da un albero.
- Un perfezionamento simile a quello suggerito per la ricerca in lista può aumentare l'efficienza della ricerca: questo comporterebbe la creazione di un nodo sentinella e la certezza che tutti i nodi siano collegati a questo una volta terminati i loro percorsi. Per la maggior parte delle applicazioni non vale la pena di attuare tale perfezionamento.
- Nel capitolo seguente, che tratta la ricorsione, verranno discussi diversi metodi di visita di un albero binario ordinato.

Applicazioni

Trattamento di file di grandi dimensioni con elementi soggetti a frequenti ricerche, inserimenti, cancellazioni.

Problemi supplementari

- Progettare ed implementare un algoritmo di ricerca in un albero che usa array per i nomi e per gli insiemi di puntatori a sinistra ed a destra.
- Implementare un algoritmo di ricerca in un albero che impieghi una sentinella come descritto in nota 5. Supporre che l'albero sia stato costruito in modo tale che il nodo sentinella esista ma non sia stato definito.
- Progettare un algoritmo che incrementi un contatore memorizzato in ogni nodo ogni volta che questo viene incontrato.
- Progettare ed implementare un algoritmo che visiti una struttura ad albero che può avere più di due nodi che si diramano da ogni nodo. Come parte del progetto sarà necessario costruire una struttura dati adeguata per la memorizzazione di tali alberi.

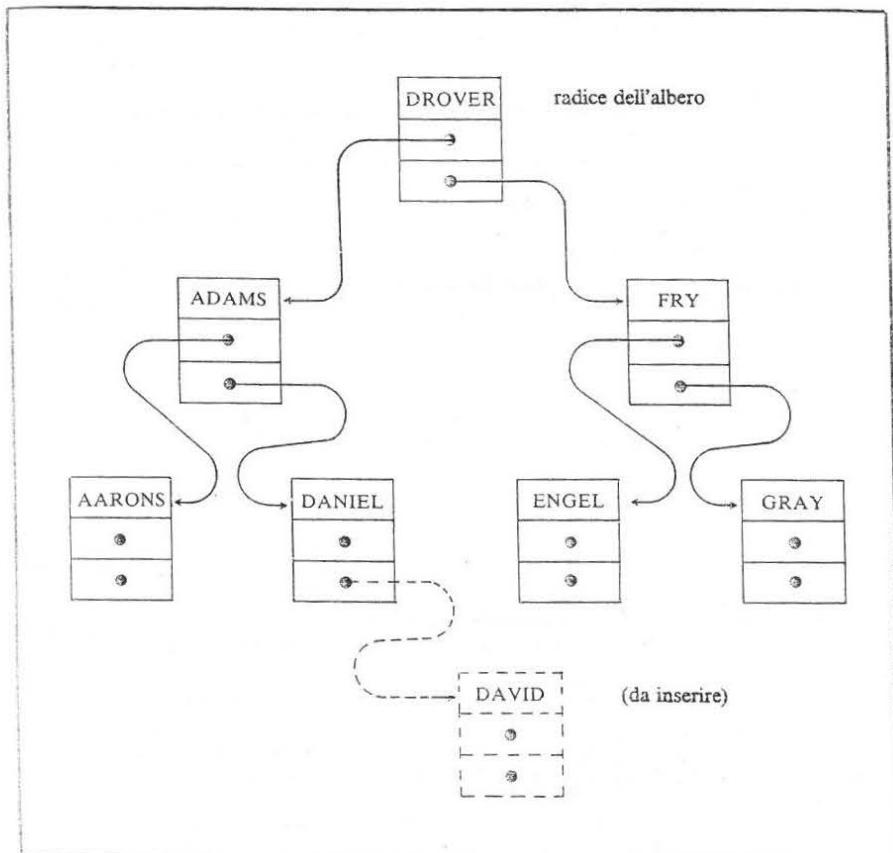
ALGORITMO 7.6

INSERIMENTO E CANCELLAZIONE IN UN ALBERO BINARIO

Problema

Fig. 7.13.
*Inserimento di un
albero binario
ordinato.*

Progettare ed implementare procedure per inserire e cancellare elementi da un albero binario ordinato con puntatori.



Sviluppo dell'algoritmo

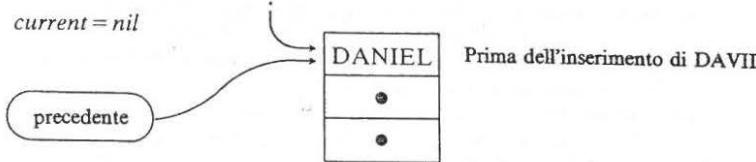
Dopo aver considerato le procedure di inserimento e cancellazione per liste di puntatori possiamo anticipare che le corrispondenti operazioni per un albero binario hanno una base simile: ovvero, ci potremo aspettare di fare per prima una ricerca e successivamente modificare alcuni puntatori per poter eseguire l'inserimento e la cancellazione. La natura della struttura ad albero può, tuttavia, impedire che gli algoritmi divengano uguali a quelli che utilizzano strutture a lista.

Come abbiamo fatto per il problema delle liste, occupiamoci per prima cosa del meccanismo di inserimento in un albero: consideriamo quindi il caso in cui vogliamo inserire il nome DAVID nell'albero binario ordinato illustrato in Fig. 7.13.

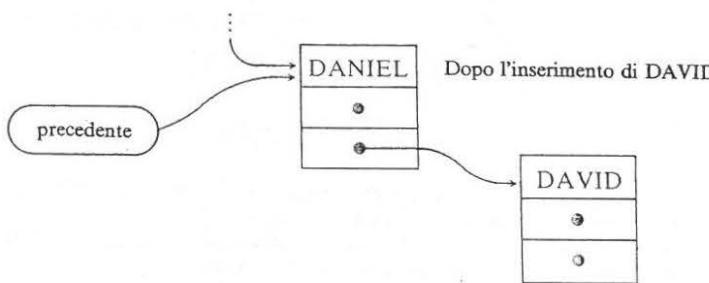
La prima domanda a cui dobbiamo rispondere prima di poter compiere l'inserimento è *dove* debba essere inserito il nome DAVID nell'albero. Potremmo posizionarlo arbitrariamente in posizioni differenti e controllare se rimane soddisfatta la condizione per gli alberi binari: tale approccio risulta tuttavia arbitrario e dispendioso in termini di tempo. Potremmo quindi anticipare che esiste un modo migliore di affrontare il problema. La nostra esperienza nell'aggiornamento di alberi binari ordinati riguarda la ricerca di un nome *già presente* nell'albero; siamo dunque in grado di esplorare un albero binario per localizzare un nome o stabilire che non è presente al suo interno. La domanda successiva potrebbe essere quella di sapere se questo possa in qualche modo aiutarci. Possiamo certamente cercare il nome DAVID nell'albero, ma ciò come può aiutarci? La nostra ricerca segue il percorso:

- | | |
|---------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DAVID < DROVER
DAVID > ADAMS
DAVID > DANIEL | spostati al sottoalbero sinistro (radice ADAMS)
spostati al sottoalbero destro (radice DANIEL)
spostati al sottoalbero destro (radice <i>nil</i> = termine) |
|---------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|

In questo caso la ricerca termina senza successo poiché DAVID non è presente, ma se fosse stato posizionato alla radice del sottoalbero destro di DANIEL la ricerca avrebbe avuto esito positivo; così, indirettamente la ricerca ha risposto alla domanda su dove inserire il nome DAVID - ovvero l'elemento da inserire deve essere posto nella posizione in cui l'algoritmo di ricerca si sarebbe aspettato di trovarlo se fosse stato presente. La prima parte dell'inserimento in un albero comporta quindi una ricerca per trovare la posizione in cui inserire l'elemento nell'albero. Ritornando all'algoritmo di ricerca, vediamo che per il nostro esempio, il puntatore *previous* punterà al nodo di DANIEL ed il valore di *current* sarà *nil* al termine della ricerca, cioè:



Per inserire DAVID sarà necessario creare un sottoalbero *destro* per la radice DANIEL; questo comporta lo spostamento del puntatore destro del nodo *precedente* (cioè quello di DANIEL), che dovrà puntare al nuovo nodo DAVID, ovvero:



A questo punto è possibile riassumere i passi fondamentali del meccanismo:

1. trova dove deve essere inserito il nuovo nome cercandolo,
2. crea un nuovo nodo e memorizza in esso il nome da inserire. Poiché non vi saranno sottoalberi ponì i suoi puntatori sinistro e destro a *nil*;
3. aggiorna i puntatori del nodo precedente in modo che puntino al nodo inserito.

Il meccanismo di ricerca è stato discusso dettagliatamente in precedenza quindi non ne parleremo più a lungo a parte confermare che può essere usata la procedura *treeseach* sviluppata nell'algoritmo 7.5. La creazione del nuovo nodo ed il meccanismo di memorizzazione sono semplici: dobbiamo risolvere quindi il solo problema di aggiornare i puntatori del nodo precedente. Esaminando ancora una volta il meccanismo di ricerca notiamo che al punto in cui la ricerca giunge al termine (supponendo che il nome non sia già presente) l'unica informazione che abbiamo è il puntatore al nodo precedente ed nodo corrente con valore *nil*. Prima di poter fare l'inserimento è necessario sapere se il nuovo nodo debba essere inserito nel sottoalbero sinistro o destro del nodo precedente; tuttavia tale informazione è andata perduta durante la ricerca. Nel nostro esempio, ci troviamo nel nodo DANIEL e dobbiamo decidere se DAVID debba essere inserito nel sottoalbero destro o sinistro: un semplice test che utilizza il nome nel

nodo precedente e quello da inserire risolverà il problema permettendoci di aggiornare l'appropriato puntatore, cioè, all'interno del contesto del nodo precedente possiamo applicare il passo:

se nomealb > nomenuovo, allora

(a) inserisci il nuovo nome nel sottoalbero sinistro aggiornando il puntatore sinistro del nodo precedente,

altrimenti

(a') inseriscilo nel sottoalbero destro aggiornando il puntatore destro del nodo precedente.

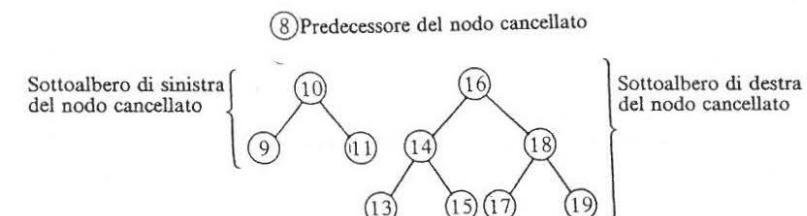
Nella situazione di albero inizialmente vuoto, *non applichiamo* questi passi, dobbiamo invece aggiornare il puntatore alla radice in modo che punti al nodo inserito più di recente.

Possiamo ora fornire la completa descrizione dell'algoritmo di inserimento nell'albero; prima però studiamo dettagliatamente l'algoritmo di cancellazione.

Il fatto che il nodo da cancellare dall'albero binario sia collegato ai sottoalberi destro e sinistro, suggerisce che il meccanismo della cancellazione debba essere più complicato di quello per l'inserimento. All'inizio ci prenderemo quindi il disturbo di considerare *tutte* le situazioni in cui possa essere richiesta una cancellazione: queste sono illustrate in Fig.7.14, dove il nodo che deve essere cancellato è individuato da un asterisco (*).

Sarebbe molto difficile trattare questo numero apparentemente molto grande di situazioni come casi particolari dell'operazione di cancellazione; il nostro obiettivo in questo contesto deve quindi essere un algoritmo generale in cui vi sia il numero minore possibile di casi particolari. Un possibile inizio può essere quello di cercare delle *similarità* invece che differenze tra i vari casi (cioè, i casi (a), (b), (c) e (d) di Fig.7.14 considerano tutti la cancellazione del nodo radice).

Un approccio potrebbe essere quello di considerare inizialmente solo i casi più semplici e cercare di raggrupparli insieme; ma è in genere più facile progettare un algoritmo che tratti il caso più complesso con la speranza che comprenda anche i casi più semplici. Per questo motivo, cominciamo con l'esame del caso (1) in cui si vuole cancellare il nodo contrassegnato dal numero 12: come passo iniziale in questa direzione, rimuoviamo tutti i collegamenti che implicano il nodo 12. Abbiamo dunque:



Il nostro albero è ora diviso in tre sottoalberi.

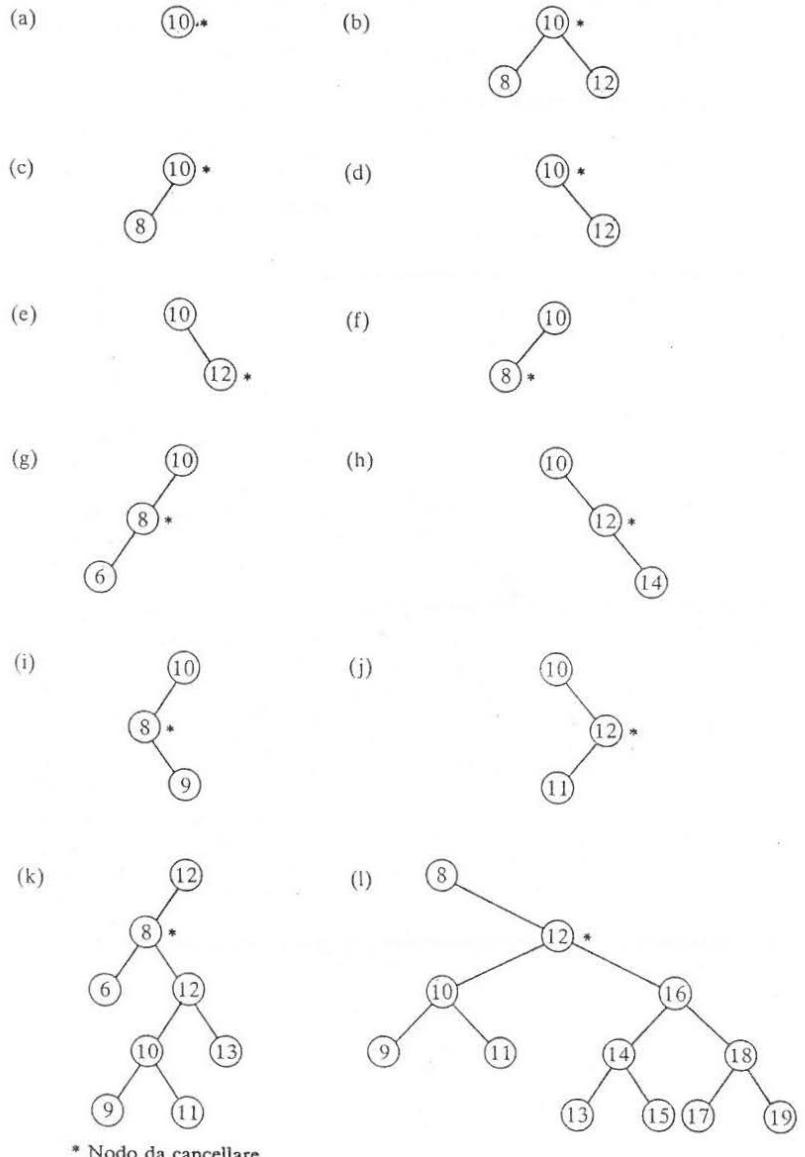
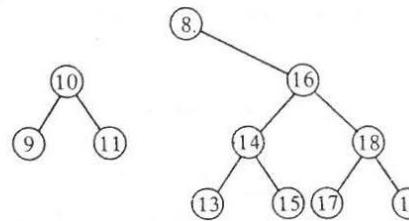
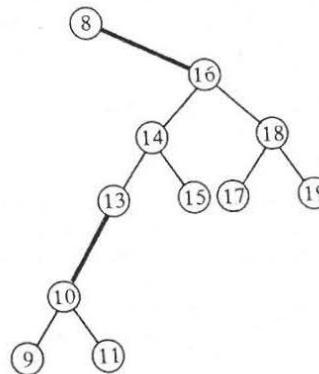


Fig. 7.14.
Casi di cancellazione
da un albero.

Dobbiamo ora decidere come debbano essere collegati questi sottoalberi in modo che il nuovo albero conservi la sua struttura caratteristica. Nel caso delle liste di puntatori, questa operazione comportava unicamente il collegamento del predecessore del nodo cancellato con il suo successore. Nel caso dell'albero binario, esiste un solo nodo predecessore (o antenato; il nodo (8) del nostro esempio) come nella lista, ma ci sono due nodi successori (il (10) ed il (16)). Dopo aver esaminato la situazione notiamo che entrambi i sottoalberi con radice (10) e (16) sono candidati al ruolo di nuovo sottoalbero destro del nodo (8). A questo punto non sembra aver molta importanza quale di questi sottoalberi debba essere collegato al nodo (8): collegiamo quindi (8) a (16).



Sorge ora la domanda di cosa fare con il sottoalbero di radice (10); esso non può diventare il sottoalbero sinistro di (8), poiché violerebbe la definizione di albero binario ordinato (vedi per la definizione, l'algoritmo 7.5). Gli unici nodi cui sarebbe possibile attaccarlo sono dunque (13), (15), (17) e (19). Un esame di tali possibilità rivela che l'unico ruolo possibile per questo sottoalbero è quello di essere collegato al nodo (13) come sottoalbero sinistro: tutte le altre possibilità violerebbero la definizione di albero binario. Dopo la rimozione del nodo (12), il nuovo albero assume quindi la forma:



I due collegamenti necessari per la ricostruzione dell'albero sono marcati da una linea ingrossata.

Ora, il nostro compito è quello di estrarre da questo esempio delle considerazioni di carattere generale: per questo confrontiamo l'albero

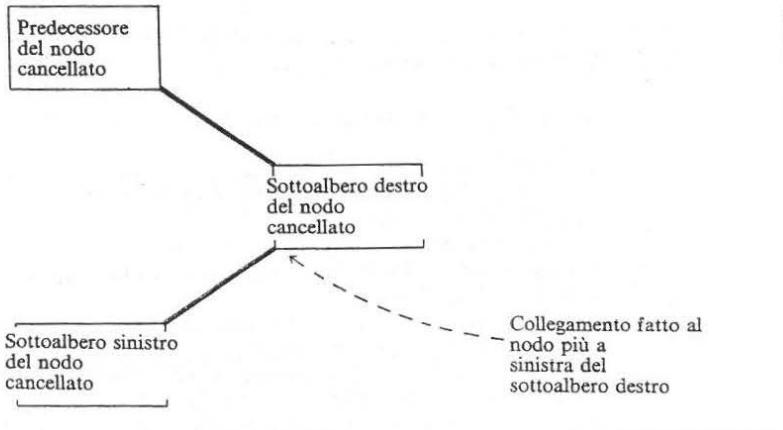


Fig. 7.15.
Rappresentazione schematica della cancellazione di un nodo da un sottoalbero sinistro.

prima e dopo la cancellazione del nodo (12). Da questo confronto possiamo concludere che devono aver luogo i seguenti passi:

- se deve essere cancellato un nodo in un sottoalbero destro allora
- collega il nodo predecessore del nodo considerato al sottoalbero destro del nodo cancellato;
 - collega il sottoalbero sinistro del nodo cancellato al nodo *più a sinistra* del sottoalbero risultante dal passo (a).

La posizione *più a sinistra* menzionata nel passo (b) è l'unica posizione in cui inserire il sottoalbero sinistro che non viola la definizione di albero binario ordinato. Questo è mostrato nello schema a blocchi illustrato in Fig. 7.15.

Da un ulteriore sguardo all'insieme di diagrammi, vediamo che le variazioni dello schema attuale dopo la cancellazione sono:

- Il nodo cancellato non può avere un sottoalbero destro.
- Il nodo cancellato non può avere un sottoalbero sinistro.
- Il nodo cancellato non può avere sottoalberi destro e sinistro.

I corrispondenti diagrammi per questi casi sono illustrati in Fig. 7.16. Notiamo che nel caso (c') il collegamento destro di (10) dovrà, alla fine, avere valore *nil*.

Il caso (a') è quello che può causare dei problemi; rimanderemo però queste considerazioni finché non avremo descritto i dettagli del caso più generale in cui il nodo cancellato possiede entrambi i sottoalberi sinistro e destro.

Per iniziare il trattamento del meccanismo di cancellazione, riconsideriamo il punto in cui il nodo da cancellare sia stato localizzato dalla

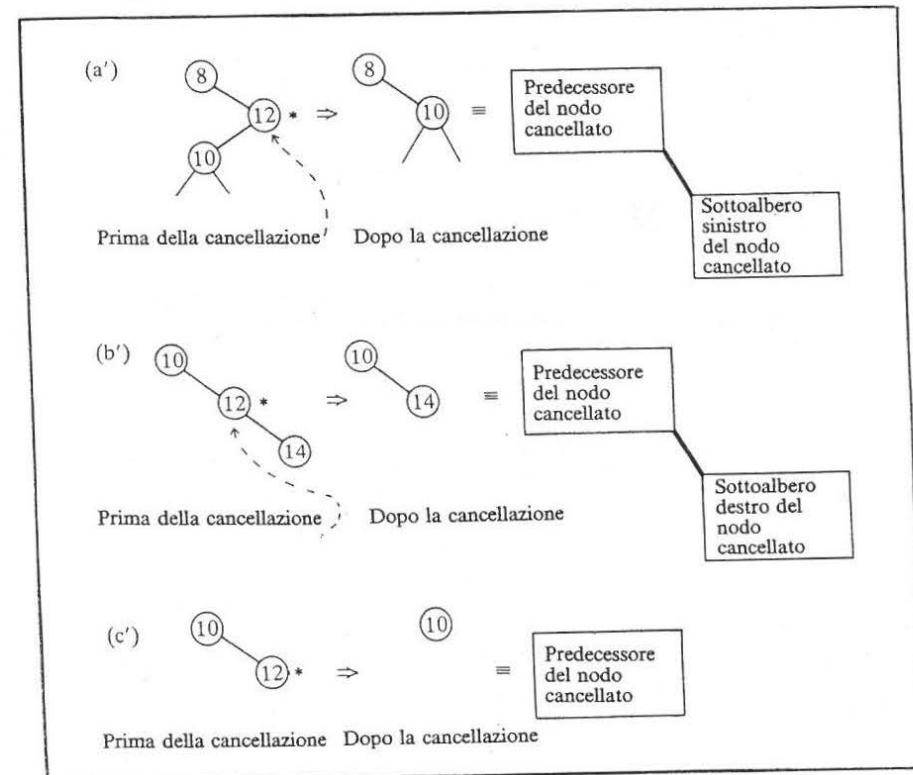


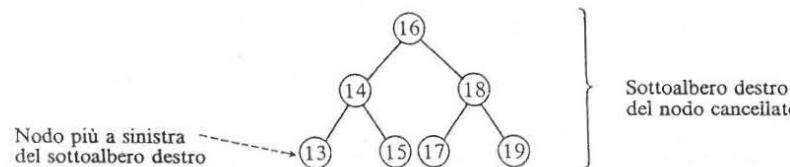
Fig. 7.16.
Altri casi di cancellazioni da un sottoalbero destro.

procedura *treesearch* in tal caso, l'unica informazione posseduta era la coppia di puntatori al nodo *previous* e *current* (cioè quello da cancellare). Ci possiamo ora chiedere se queste sono tutte le informazioni necessarie per ricomporre l'albero dopo la cancellazione del nodo. Dopo uno studio dell'esempio precedente vediamo che siamo in possesso di informazioni sufficienti per collegare il predecessore del nodo cancellato al sottoalbero destro di tale nodo (se esiste). L'istruzione:

```
if previous <> nil then
    previous↑.right := current↑.right
else
    root := current↑.right
```

ci permetterà di svolgere questo compito tenendo conto di *root = current*. La figura 7.17 illustra tale fatto in termini di puntatori. A questo punto non possediamo, tuttavia, informazioni sufficienti per collegare il sottoalbero sinistro del nodo cancellato al nodo *più a sinistra* del sottoalbero destro del nodo cancellato (riferimento: situazione (1) nel sovramenzionato insieme di esempi). Ciò che ci impedisce di fare in

modo diretto tale collegamento è che non conosciamo l'identità del nodo più a sinistra del sottoalbero destro; con riferimento all'esempio precedente, abbiamo:



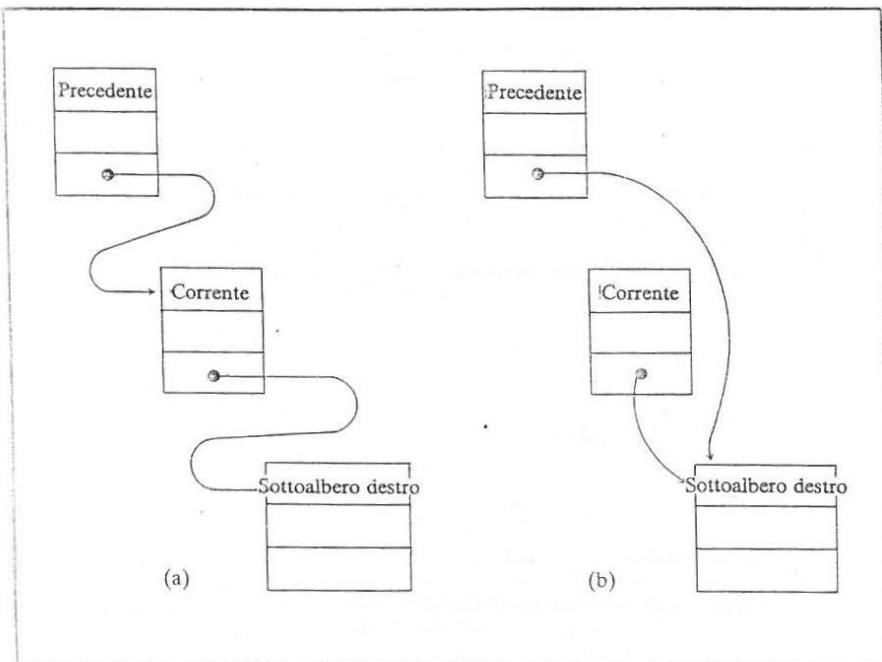
Il nostro compito è ora quello di trovare tale nodo del sottoalbero destro; per poterci muovere nel sottoalbero è necessaria un'istruzione del tipo:

$newcurrent := current \uparrow .right$

Dopo tale assegnamento $newcurrent$ punterà alla radice del sottoalbero destro.

Per trovare il nodo più a sinistra, possiamo iniziare dalla radice e seguire tutti i collegamenti a sinistra finché non incontriamo un nodo con un collegamento a sinistra di valore *nil*: questo sarà il nodo cercato

Fig. 7.17.
(a) Prima della cancellazione;
(b) Dopo la cancellazione.



(il (13) nel nostro esempio). I passi da compiere in tale ricerca sono:

1. Prendi il puntatore alla radice del sottoalbero destro e salvalo.
2. Finché il puntatore *current* è diverso da *nil*
 - (a) salva il puntatore corrente,
 - (b) assegna il puntatore corrente al suo collegamento a sinistra.

È necessario salvare il puntatore corrente prima che venga ripristinato, per impedire che il puntatore al nodo più a sinistra venga posto uguale a *nil* al termine del ciclo di ricerca; sarà inoltre necessario che la nostra procedura tenga conto che il ciclo potrebbe non essere eseguito del tutto (nel caso che il nodo da cancellare non abbia un sottoalbero destro).

Una volta trovato il puntatore al nodo più a sinistra del sottoalbero destro è solo questione di collegare il puntatore sinistro di questo nodo alla radice del sottoalbero sinistro del nodo cancellato, cioè se *newprevious* punta al nodo più a sinistra del sottoalbero destro, avremo (a patto che il nodo *current* non sia il nodo *root* nel qual caso *root* := *current*↑.left)

$newprevious \uparrow .left := current \uparrow .left$

In precedenza abbiamo trovato, considerando il caso (a'), che il passo appena descritto non si sarebbe potuto compiere nel caso in cui il nodo cancellato non fosse provvisto del sottoalbero destro; questa situazione può essere scoperta con un controllo se *current*↑.right = *nil*. In presenza di questa situazione sarà necessario porre il sottoalbero sinistro del nodo cancellato come sottoalbero destro del nodo precedente, cosa che può essere fatta usando:

$previous \uparrow .right := current \uparrow .left$

purchè *current* non sia *root*, nel qual caso *root* diviene *current*↑.left (riferimento al caso (a') per il corrispondente diagramma).

Abbiamo ora trovato un meccanismo per cancellare un nodo dal sottoalbero destro: esso comprende pure il caso in cui il nodo non abbia il sottoalbero destro e quello in cui li abbia entrambi. È compito semplice stabilire che gli altri casi sono compresi in questa struttura.

Dopo aver considerato tutte le possibilità per la cancellazione di un nodo dal sottoalbero destro, il nostro nuovo compito è quello di considerare tutti i casi di cancellazione di un nodo dal sottoalbero sinistro. Cominciando la trattazione di questo nuovo problema, giungiamo presto alla considerazione che esso può essere trattato come l'immagine simmetrica del problema appena risolto. Alcune considerazioni rivelano che per il meccanismo di cancellazione di un nodo da un sottoalbero sinistro occorre semplicemente sostituire destro con sinistro nel precedente svolgimento.

Siamo ora in possesso dei meccanismi per rimuovere elementi da entrambi i sottoalberi; l'unico problema che rimane è specificare la procedura che ci permetta, primo di trovare il nodo da cancellare, poi di chiamare le procedure adeguate. L'algoritmo deriva direttamente da questa descrizione.

Siamo ora al punto in cui tutte le procedure sopra sviluppate possono essere descritte.

Descrizione dell'algoritmo

(1) Inserimento in un albero binario

1. Definisci il nome da inserire ed il puntatore alla radice dell'albero.
2. Trova, utilizzando la procedura *treeseach*, la posizione in cui deve essere inserito il nome.
3. Crea un nuovo nodo e memorizza in esso il nome da inserire. Al momento dell'inserimento tale nodo non avrà sottoalberi, così i suoi puntatori destro e sinistro devono essere posti a *nil*.
4. Se l'albero non è inizialmente vuoto, allora
 - (a) se il nome dell'albero viene in ordine alfabetico dopo il nuovo nome, allora
 - (a.1) collega il puntatore sinistro del nodo precedente al nuovo nodo
 - altrimenti
 - (a'.1) collega il puntatore destro del nodo precedente al nuovo nodo,
 - altrimenti
 - (a') collega il puntatore alla radice al nuovo nodo.
5. Ritorna l'albero aggiornato ed il suo puntatore alla radice.

(2) Cancellazione da un albero binario

1. Definisci il nome da cancellare ed il puntatore alla radice dell'albero.
2. Ricerca nell'albero il nome da cancellare e ritorna il puntatore al nodo ed al suo predecessore, se esiste.
3. Se il nodo da cancellare è stato trovato, allora
 - (a) se il nodo è nel sottoalbero sinistro, allora
 - (a.1) chiama la procedura di cancellazione di un nodo dal sottoalbero sinistro
 - altrimenti
 - (a'.1) chiama la procedura di cancellazione di un nodo dal sottoalbero destro,
 - altrimenti
 - (a') segnala che il nome da cancellare non è presente.
4. Ritorna l'albero da cui è stato cancellato un nodo (se presente).

(3) Cancellazione da un sottoalbero destro

1. Definisci i puntatori al nodo corrente da cancellare, al suo predecessore ed alla radice dell'albero.
2. Collega il nodo predecessore del nodo corrente al suo sottoalbero destro.
3. Prendi il puntatore della radice del sottoalbero destro del nodo corrente ed assegna come nuovo puntatore al nodo corrente.
4. Trova il nodo più a sinistra del sottoalbero destro seguendo i collegamenti a sinistra finché non viene incontrato un *nil*.
5. Se il nodo da cancellare ha un sottoalbero destro allora
 - (a) fa in modo che il sottoalbero sinistro del nodo da cancellare diventi il sottoalbero sinistro del nodo più a sinistra nel sottoalbero destro e modifica la radice se necessario
 - altrimenti
 - (a') fa in modo che il sottoalbero sinistro del nodo da cancellare diventi il sottoalbero destro del suo predecessore e modifica la radice se necessario.
6. Elimina il nodo cancellato.
7. Ritorna l'albero da cui è stato cancellato il nodo (se presente).

Implementazione in Pascal

(1) Inserimento in un albero binario

```

procedure treeinsert(newname: nameformat; var root: treepointer);
var current {pointer to current node},
    previous {pointer to node above current node},
    newnode {pointer to newly inserted node}: treepointer;
    found {if true means name already present so no insertion}:
    boolean;

begin
  {assert: root points to root of tree}
  treeseach (newname, current, previous, root, found);
  if not (found) then
    begin
      {assert: previous node is predecessor node for newname to be
      inserted if tree not empty}
      new(newnode);
      with newnode^ do
        begin
          treename := newname;
          left := nil;
          right := nil
        end;
      if root<>nil then
        with previous^ do
          if treename > newname then
            begin
              if left = nil then
                left := newnode
              else
                treeseach (newname, current, previous, left, found);
                if not (found) then
                  with previous^ do
                    if right = nil then
                      right := newnode
                    else
                      treeseach (newname, current, previous, right, found);
            end;
    end;
  end;
end;

```

```

    left := newnode
  else
    right := newnode
  else
    root := newnode
end
{assert: previous node linked to newnode containing inserted
newname maintaining ordered binary tree}
else
  writeln('The name ', newname, ' is already present')
end

```

(2) Cancellazione da un albero binario

```

procedure treedelete {oldname: nameformat; var root: treepointer};
var current {pointer to the node to be deleted},
previous {pointer to the ancestor of node to be deleted}:
treepointer;
found {true if name to be deleted is in tree}: boolean;
begin
{assert: root points to root of tree}
treeresearch {oldname, current, previous, root, found};
if found then
{assert: current node points to node containing oldname}
  if previous <> nil then
    if oldname < previous^.treename then
      leftsubtree {current, previous, root}
    else
      rightsubtree {current, previous, root}
  else
    rightsubtree {current, previous, root}
{assert: oldname removed from tree and links restored to
maintain ordered binary tree}
else
  writeln ('The name', oldname, 'is not present in tree')
end

```

(3) Cancellazione da un sottoalbero destro

```

procedure rightsubtree {current, previous: treepointer; var root:
treepointer};
var newcurrent {pointer to leftmost node examined so far in right
subtree},
newprevious {finally contains pointer to leftmost node in right
subtree}: treepointer;

begin {removes current node from binary tree—current node has a
"smaller" ancestor}
{assert: root points to root of tree}
newprevious := previous;
newcurrent := current^.right;
{link predecessor node of the current node to its right subtree}
if previous <> nil then
  previous^.right := current^.right

```

```

else
  root := current^.right;
{assert: newcurrent points to root of right subtree of current node}
{invariant: newcurrent points to leftmost node examined so far in
right subtree of current node ^ newprevious points to its
predecessor}
  while newcurrent <> nil do
    begin {find leftmost node in right subtree of predecessor node}
      newprevious := newcurrent;
      newcurrent := newcurrent^.left
    end;
    if current^.right <> nil then
      {left subtree of current node becomes left subtree of leftmost
node in right subtree}
      if newprevious <> nil then
        newprevious^.left := current^.left
      else
        root := current^.left;
    else
      {no right subtree so left subtree becomes right subtree of
predecessor node}
      if previous <> nil then
        previous^.right := current^.left
      else
        root := current^.left;
    dispose {current}
{assert: current node removed and links restored to maintain
ordered binary tree}
end

```

Note di progetto

- Il costo dell'inserimento e della cancellazione in un albero binario ordinato è proporzionale al numero di nodi che devono essere esaminati prima che venga fatto un inserimento o una cancellazione; da ciò segue che la discussione sul suo svolgimento seguirà la stessa traccia fornita nella nota 1 dell'algoritmo di ricerca in un albero (algoritmo 7.5). Per un albero di n nodi costruito in maniera casuale ci possiamo quindi aspettare che vengano esaminati in media $2 \log n$ nodi.
- Nella descrizione dell'andamento degli algoritmi di inserimento e cancellazione da un albero, non considereremo l'algoritmo di ricerca che è già stato trattato nel paragrafo 7.5. Le modifiche dell'albero compiute nell'algoritmo di inserimento dipendono dal puntatore *previous* passato a questa procedura da quella di ricerca. Se prima dell'inserimento l'albero era vuoto, la radice dovrà puntare al nuovo nodo inserito, se invece non lo era, allora la procedura utilizzerà le informazioni memorizzate in quello che dovrebbe essere l'antenato del nuovo nodo, per decidere se esso debba essere inserito nel suo sottoalbero destro o sinistro. I cambiamenti operati sull'albero tramite l'algoritmo *rightsubtree* di cancellazione, dipendono dai valori

dei puntatori *previous* e *current* ritornati dalla procedura di ricerca. È possibile caratterizzare l'andamento del processo iterativo usato in questa procedura nel modo seguente: dopo ogni iterazione, il puntatore *newprevious* punterà al nodo più a sinistra del sottoalbero destro del nodo da cancellare appena esaminato. Il puntatore *newcurrent* punterà al nodo alla radice del sottoalbero sinistro del nodo puntato da *newprevious*. Al termine *newprevious* punterà al nodo più a sinistra nel sottoalbero destro del nodo da cancellare. Al termine dell'algoritmo, se l'albero conteneva solo il nodo da cancellare, il puntatore *root* sarà posto uguale a *nil*. Se è stato cancellato il nodo *root*, allora la nuova radice dell'albero sarà la radice del sottoalbero destro. Se è stato cancellato un nodo foglia (cioè un nodo con *nil* a sinistra ed un sottoalbero destro), allora il nodo predecessore avrà come puntatore *nil* dove prima aveva un puntatore al nodo cancellato. Se è stato cancellato un nodo interno (cioè un nodo con entrambi i sottoalberi), allora il puntatore destro del predecessore sarà cambiato e punterà alla radice del sottoalbero destro del nodo cancellato. Il sottoalbero sinistro, diventerà il sottoalbero sinistro del nodo più a sinistra del sottoalbero destro del nodo cancellato. Se il nodo cancellato non possiede il sottoalbero destro, allora il suo sottoalbero sinistro assumerà il ruolo di sottoalbero destro del suo predecessore. Lo svolgimento della procedura *leftsubtree* può essere descritta in maniera simile alla precedente.

Lo svolgimento della procedura *treedelete* è governato dall'andamento delle procedure *treesearch*, *rightsubtree* e *leftsubtree*.

La procedura *treeinsert* termina poiché termina la procedura di ricerca. Le tre procedure per la cancellazione terminano per le medesime ragioni. Il ciclo di *rightsubtree* termina poiché ad ogni iterazione viene esaminato un nuovo nodo, e dato che l'albero è finito verrà sicuramente raggiunto un nodo con puntatore *nil*: ciò forza la terminazione del ciclo.

3. In precedenza abbiamo considerato gli algoritmi di inserimento e cancellazione per alberi binari ordinati; abbiamo pure considerato un'unica possibile rappresentazione di tali alberi. Come possiamo immaginare, esiste un certo numero di algoritmi e rappresentazioni che possono essere utilizzati a questo scopo: tratteremo alcuni di questi algoritmi nelle note di progetto e nei problemi supplementari.
4. Nel presente algoritmo per la cancellazione di un nodo, abbiamo cercato di trarre vantaggio dalla simmetria del problema; come risultato abbiamo costruito procedure separate per la cancellazione di un nodo da un sottoalbero sinistro o destro. Le limitazioni del Pascal e la struttura dati scelta non facilitano il passaggio delle informazioni in modo da utilizzare la stessa procedura in entrambi i casi. La maggior parte degli altri algoritmi di cancellazione rivelano un'asimmetria tra le cancellazioni da un sottoalbero sinistro ed uno destro; tale differenza può, dopo un certo periodo di tempo, introdurre delle asimmetrie nella struttura dell'albero alquanto indesiderabili.

5. Per la localizzazione della posizione dell'inserimento e del nodo da cancellare è stata utilizzata una procedura distinta: questo perché entrambi i processi di inserimento e cancellazione comportano una fase iniziale di ricerca separata dalla manipolazione successiva dei nodi dell'albero.
6. Gli algoritmi di inserimento e cancellazione possono essere implementati elegantemente con l'uso della ricorsione. Gli algoritmi ricorsivi, tuttavia tendono a nascondere (almeno ai principianti) alcune delle sottigliezze dei meccanismi di inserimento e cancellazione in un albero (vedere il Capitolo 8 per la discussione sulla ricorsione).
7. L'algoritmo di inserimento può essere usato per l'ordinamento dei dati. Per poter acquisire gli elementi dell'albero in ordine è necessario attraversare l'albero in un ordine particolare. È necessario per ricavare gli elementi dell'albero in sequenza ordinata. Quello che viene chiamato un *attraversamento simmetrico* dell'albero. Gli algoritmi per l'attraversamento di un albero (incluso quello simmetrico) saranno discussi nel prossimo capitolo sulla ricorsione.
8. Negli algoritmi per l'inserimento e la cancellazione descritti, esiste il rischio che l'albero abbia una profondità maggiore di $\log_2 n$ per n nodi memorizzati. Per evitare tale rischio, sono stati sviluppati algoritmi che incorporano un meccanismo di bilanciamento. Questi algoritmi, che sono l'estensione logica di quelli di inserimento e cancellazione da noi descritti, consentono operazioni di $O(\log_2 n)$ passi, anche nel caso peggiore. Il concetto di base che sta dietro il bilanciamento è quello di assicurare per ogni nodo che il peso dei suoi sottoalberi sinistro e destro differisca al massimo di uno.

Applicazioni

Trattamenti di un alto numero di informazioni che possano essere cambiate facilmente e ricercate rapidamente (come le prenotazioni aeree).

Problemi supplementari

- 7.6.1 Progettare ed implementare algoritmi di inserimento e cancellazione da un albero che usino degli array per i nomi e per gli insiemi di puntatori a destra e a sinistra. Una lista libera con puntatori viene usata per controllare le locazioni inutilizzate. Occorre essere in grado di progettare un algoritmo tale che solo una procedura sia necessaria per rappresentare quelle di *leftsubtree* e *rightsubtree* descritte precedentemente.
- 7.6.2 Progettare un algoritmo di cancellazione da un albero che tratti in modo simile i sottoalberi sinistro e destro. Può essere utilizzata una variazione della procedura *rightsubtree* che cancelli nodi anche dai sottoalberi a sinistra. La Fig.7.18 illustra come tale procedura può trattare la cancellazione del nodo (11).

ALGORITMI RICORSIVI

INTRODUZIONE

La ricorsione è una tecnica di programmazione completamente diversa dai metodi incontrati nei capitoli precedenti. In molti problemi discussi è stata rilevata l'utilità di chiamare una o più procedure e funzioni all'interno del *corpo* della procedura principale.

Ad esempio:

```

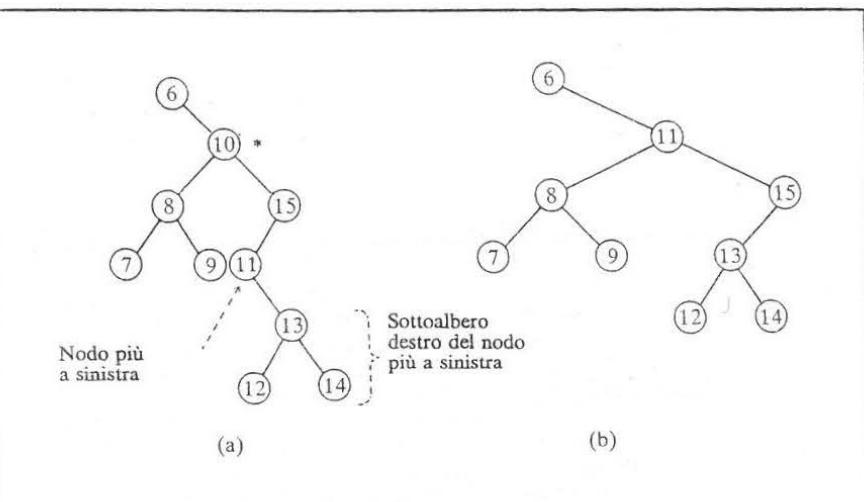
procedure external( ... )
  :
begin
  :
  internal( ... )
  :
end
```

Sotto queste condizioni le istruzioni nella procedura *external* sono eseguite secondo il flusso di controllo della procedura come influenzate dalle condizioni di ingresso e dai dati. Al punto della procedura *external* in cui si incontra la chiamata di *internal*, l'esecuzione della procedura *external* viene sospesa: a questo punto ha inizio l'esecuzione della procedura *internal* che prosegue fino alla terminazione. Direttamente dopo la terminazione della procedura *internal*, la procedura *external* riprende l'esecuzione dal punto in cui si era interrotta, tenendo conto di tutti gli eventuali cambiamenti alle variabili apportati dalla chiamata di *internal*. Anche se la procedura di *internal* chiamasse un'altra procedura *innermost* all'interno del corpo delle sue istruzioni, la sequenza di esecuzione dell'intero processo, che implica le tre procedure annidate, proseguirebbe in ordine naturale, cioè, per il nostro esempio:

1. *external* viene eseguita finché non si incontra una chiamata ad *internal*, al cui punto la procedura *external* sospende l'esecuzione.
2. Non appena *external* si sospende, inizia l'esecuzione di *internal* che continua finché non si incontra una chiamata alla procedura *innermost*: a questo punto anche *internal* si sospende (ora sia *internal* che *external* sono sospese).

Fig. 7.18.
(a) Prima della cancellazione;
(b) Dopo la cancellazione.

- 7.6.3 Un ulteriore algoritmo di cancellazione comporta la ricerca nel sottoalbero destro del nodo più a sinistra; tale nodo viene poi spostato nella posizione del nodo cancellato e viene sostituito dal suo sottoalbero destro. Questo meccanismo di trasformazione è molto utile poiché non modifica la lunghezza media dell'albero. La Fig. 7.19, in cui deve essere cancellato il nodo (10), spiega il meccanismo. Implementarlo.
- 7.6.4 Progettare ed implementare un algoritmo che converta un qualsiasi albero multiplo (in cui ogni nodo può avere più di due successori) nel corrispondente albero binario ordinato.



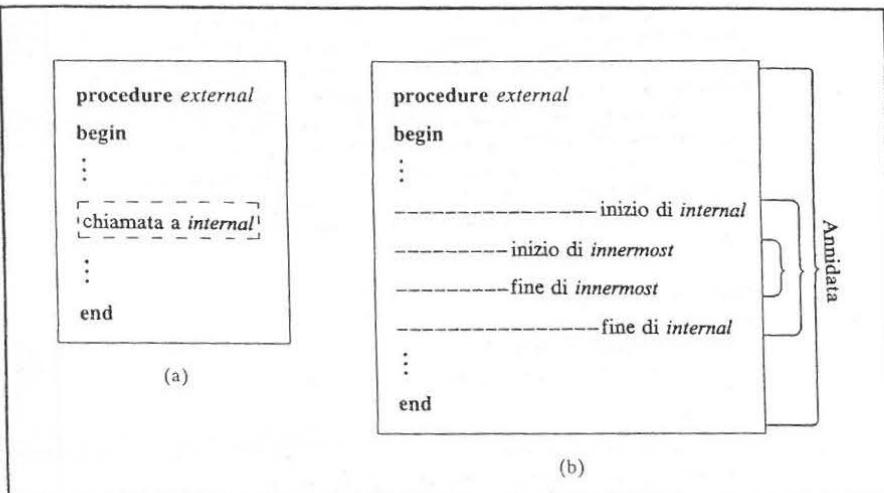


Fig. 8.1.
(a) Procedura principale originaria;
(b) Visione estesa.

3. Con *internal* ed *external* sospese, inizia l'esecuzione di *innermost* che continua finché non si raggiunge la condizione di terminazione.
4. Nel momento in cui termina *innermost*, la procedura *internal* riprende da dove si era sospesa. (A questo punto *innermost* ha terminato ed *external* è ancora sospesa.) La procedura *internal* proseguirà ora fino alla fine.
5. Dopo il termine sia di *innermost* che di *internal*, riprende la procedura principale dal punto in cui si era interrotta alla chiamata di *internal* e prosegue fino alla condizione di terminazione, punto in cui ha termine anche l'intero processo.

L'intero processo descritto è relativamente semplice e "facile da usare. Un altro modo abbastanza semplice di vedere l'intero processo è di immaginare che *ogni* procedura chiamata sia sostituita dall'insieme di istruzioni che la costituiscono. Il risultato è una versione estesa della procedura principale in cui le procedure chiamate si presentano a struttura annidata. Nel nostro esempio, l'espansione della procedura *external* assumerebbe la forma illustrata in Fig.8.1.

Abbiamo ora posto le fondamenta necessarie a permetterci di introdurre i concetti di ricorsione e di procedure ricorsive (e funzioni ricorsive).

Un particolare caso di una procedura chiamante un'altra procedura (o funzione) è quello in cui la procedura chiama *se stessa*. Algoritmi implementati in questo modo vengono detti *ricorsivi*. È possibile dividere le classi degli algoritmi ricorsivi in alcune importanti sottoclassi, come descritto qui di seguito.

Ricorsione lineare

Il tipo più semplice di algoritmo ricorsivo è quello in cui viene fatta una sola chiamata ricorsiva interna nella procedura (o funzione). Gli algoritmi ricorsivi di questo tipo sono detti *lineari*. Molti algoritmi ricorsivi lineari possiedono la seguente struttura generale (sebbene possa cambiare l'ordine dei passi)

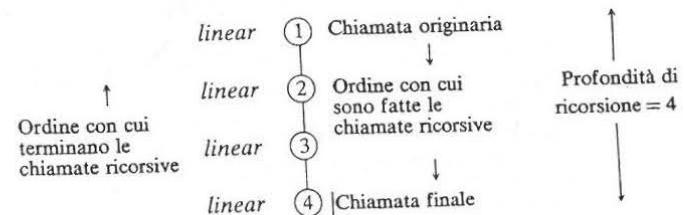
```

procedure linear ...
begin
  if "è soddisfatta la condizione di terminazione" then
    (a) restituisci i risultati
  else
    begin
      (a') compi delle azioni
      (b') fa una chiamata ricorsiva di linear
    end
end

```

Per questo tipo di procedure, *l'ultima* chiamata ricorsiva incontrata fatta prima della condizione di terminazione è la prima a terminare. Il seguente meccanismo ad albero illustra questa affermazione. I nodi si riferiscono a chiamate ricorsive.

Ricorsione lineare



Probabilmente gli esempi più semplici e più conosciuti di ricorsione lineare sono il calcolo del fattoriale e del massimo comun divisore. Gli algoritmi di ricorsione lineare derivano direttamente dalle loro definizioni ricorsive. Alcune possibili implementazioni sono:

```

function nfactorial(n:integer):integer;
begin
  if n = 0 then
    nfactorial := 1
  else
    nfactorial := n * nfactorial(n-1)
end

```

```

procedure fact(n:integer; var nfactorial:integer);
var t : integer;
begin
  if n = 0 then
    nfactorial := 1
  else
    begin
      fact(n-1, t);
      nfactorial := n * t
    end
  end
function gcd(n, m:integer):integer;
begin
  if m = 0 then
    gcd := n
  else
    gcd := gcd(m, n mod m)
end

```

Nei precedenti capitoli abbiamo visto che i calcoli del fattoriale e del massimo comun divisore hanno entrambi semplici ed efficienti soluzioni iterative. Non ci soffermeremo quindi su questi esempi. Per assicurarsi di aver capito gli esempi, il lettore dovrebbe effettuare il calcolo manuale di queste funzioni per alcuni inputs specifici.

Un'ultima osservazione deve essere fatta sulla ricorsione lineare. Con gli algoritmi ricorsivi lineari, esiste di solito un rischio maggiore di cadere in un livello molto profondo di ricorsione (ad esempio, copiando una lista molto lunga ricorsivamente). Ogni chiamata ricorsiva richiede il salvataggio di un insieme aggiuntivo di parametri, variabili locali e collegamenti. Quindi, potenzialmente, un algoritmo ricorsivo può essere molto più costoso nell'uso della memoria rispetto ad una soluzione iterativa. Ciò tende anche a rendere gli algoritmi ricorsivi più lenti rispetto alle loro leggermente meno concise controparti. Quando ci allontaniamo dalla ricorsione lineare, diventa molto meno rilevante l'argomento spazio-costo.

Ricorsione binaria

Un primo incremento in sofisticazione nell'uso della ricorsione si trova nel meccanismo detto di ricorsione binaria. La ricorsione binaria è un caso particolare della più generale *ricorsione non lineare*. A causa della sua vasta applicazione ed importanza in scienza dell'elaborazione, è importante discutere la ricorsione binaria separatamente. Si dice che un algoritmo impiega la *ricorsione binaria* se compie *due* chiamate ricorsive interne di se stesso. Molti problemi di informatica cadono in questo contesto. Il problema viene risolto prima dividendolo in *due*

problemi più piccoli che a loro volta vengono risolti dividendoli in sottoproblemi ancora più piccoli e così via, ricorsivamente. Un numero di problemi che comprendono strutture dati definite ricorsivamente (come alberi binari) sono risolti più semplicemente e naturalmente usando la ricorsione binaria. Nella prima metà di questo capitolo considereremo dettagliatamente alcuni algoritmi che utilizzano la ricorsione binaria. Una struttura comunemente usata per algoritmi che impiegano la ricorsione binaria è data qui di seguito. Per una data implementazione possono cambiare i dettagli e l'ordine dei diversi passi.

```

procedure binary(...)
begin
  if condizione di terminazione then
    (a) compi alcune azioni e/o ritorna dei valori
  else
    begin
      (a') compi alcune azioni,
      (b') compi una chiamata ricorsiva di binary che risolva uno dei due
            problemi minori,
      (c') compi un'altra chiamata ricorsiva di binary per risolvere
            l'altro problema minore,
    end
  end

```

Un esempio molto semplice di ricorsione binaria comporta il calcolo ricorsivo della sequenza di Fibonacci. Come per gli esempi lineari, la soluzione iterativa è più semplice ed efficiente. Tuttavia, come vedremo più avanti, le soluzioni iterative che risolvono problemi di ricorsione binaria sono spesso molto più complicate e richiedono l'uso di pile. Vedremo inoltre che l'interpretazione della ricorsione binaria è in genere più semplice in termini di meccanismo ad albero binario.

Ricorsione non lineare

La *ricorsione non lineare* è largamente usata in informatica. Si dice che un algoritmo è di *ricorsione non lineare* quando usa un certo numero di chiamate ricorsive interne entro la procedura. In questi sistemi, le molteplici chiamate ricorsive interne, vengono solitamente generate mettendo un'unica chiamata ricorsiva all'interno di un ciclo. Nella seconda metà di questo capitolo discuteremo dettagliatamente alcuni algoritmi ricorsivi non lineari. La seguente struttura generale si adatta a molti algoritmi di ricorsione non lineare.

```

procedure nonlinear ...
begin
  for j := 1 to n do
    begin

```

```

(a) compi alcune azioni
(b) if "non incontrata condizione di terminazione" then
    (b.1) fa una chiamata ricorsiva di nonlinear,
    else
        (b'.1) compi alcune azioni
end
end

```

Come per la ricorsione binaria, questi meccanismi sono più comprensibili in termini di strutture ad albero.

Mutua ricorsione

Un tipo più insolito di ricorsione è quella chiamata mutua ricorsione. In questo caso una procedura chiama se stessa indirettamente attraverso un'altra procedura che si richiama indirettamente attraverso la prima procedura. Le due (o più) procedure sono in un certo senso concatenate tra loro. Un semplice, ma significativo modello di ricorsione mutua è il seguente.

```

procedure mutual(...)
begin
    .
    .
    chiamata interna di another
    .
    .
end
procedure another (...)

begin
    .
    .
    chiamata interna di mutual
    .
    .
end

```

Esistono molte variazioni di questo tema di base; il compilatore Pascal usa una trasmissione ricorsiva che è una forma di ricorsione mutua. In questo capitolo abbiamo scelto di non trattare oltre la mutua ricorsione.

Nel nostro primo approccio con la ricorsione, scopriremo probabilmente che la difficoltà nel capirla sta nel cercare di avere, nello stesso momento, un quadro completo di ciò che sta accadendo e in che successione. Una volta superato ciò per mezzo di diagrammi, ecc. l'intero

concetto diviene chiaro e molto semplice da usare. Non ci occuperemo di tutti gli aspetti della ricorsione, ma piuttosto ci concentreremo sugli esempi più semplici ed attinenti, con lo sforzo di acquisire «gli aspetti fondamentali dell'argomento».

Problemi come il calcolo del fattoriale e l'algoritmo del MCD non verranno considerati in dettaglio, perché non mostrano tutti gli aspetti e la potenza della ricorsione.

ALGORITMO 8.1 VISITA DI UN ALBERO BINARIO

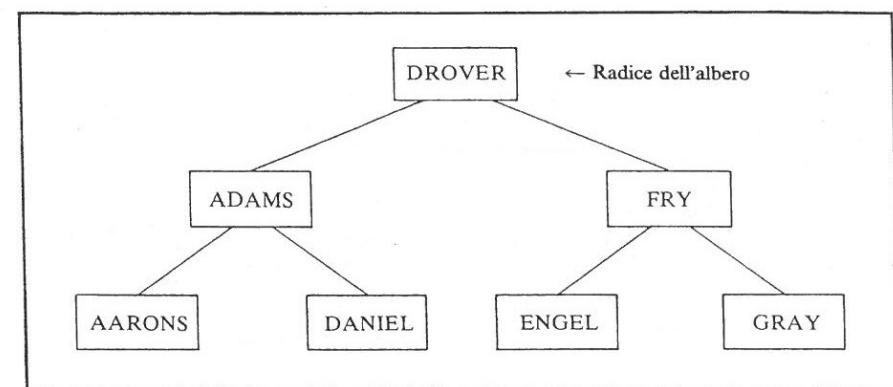
Problema

Progettare procedure ricorsive per la visita in ordine simmetrico, anticipato e differito di un albero binario.

Sviluppo dell'algoritmo

Esistono molti modi in cui poter esaminare sistematicamente ed una sola volta tutti i nodi di un albero binario ordinato. Alcuni di questi metodi di visita sono importanti a causa dell'ordine che essi impongono ai dati memorizzati nell'albero. Nella spiegazione precedente, (Capitolo 7), abbiamo visto un modo di memorizzare un insieme di nomi in un albero binario in modo che un dato nome potesse essere ritrovato in media dopo $O(\log 2 n)$ passi. Con questi dati si potrebbero

Fig. 8.2.
Un albero binario ordinario.



voler stampare i nomi in ordine alfabetico o lessicale. Per cercare di capire come farlo, consideriamo l'esempio mostrato in Fig.8.2.

L'ordine lessicale per questo insieme di nomi è:

AARONS, ADAMS, DANIEL, DROVER, ENGEL, FRY, GRAY.

Quando i nomi memorizzati in un albero binario ordinato sono stampati in ordine lessicale, si dice che è stata fatta una "visita anticipata" dei nodi dell'albero.

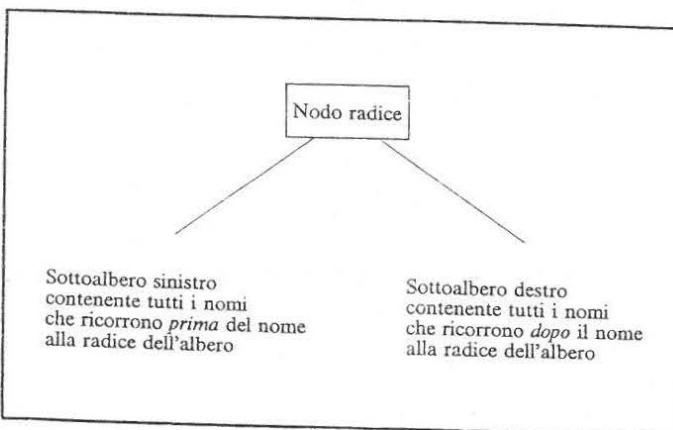
L'informazione che possediamo *prima* di iniziare la stampa dei nomi è solo un puntatore al nodo radice (cioè il nodo contenente DROVER). A questo punto sorge un'incertezza, poiché ovviamente DROVER non può essere stampato per primo. La nostra unica alternativa, dato che non possiamo stampare questo nodo per primo, è "salvarlo" (o il suo puntatore) per evitare di dover passare di nuovo per il nodo radice (occorre ricordare la nostra precedente limitazione di voler esaminare *una sola volta* tutti i nodi dell'albero). Dopo l'esame dell'albero, vediamo che tutti i nomi che seguono in ordine alfabetico DROVER sono nel suo sottoalbero destro. La situazione di base che si presenta è quella mostrata in Fig.8.3.

Da ciò possiamo dedurre che tutti i nodi del sottoalbero sinistro dovranno essere stampati prima del nome alla radice; tutti quelli del sottoalbero destro dovranno essere stampati dopo. L'ordine di stampa è quindi:

1. Stampa tutti i nomi nel sottoalbero a sinistra della radice.
2. Stampa il nome radice.
3. Stampa tutti i nomi nel sottoalbero destro della radice.

Tornando al nostro esempio, troviamo che quando ci spostiamo nel sottoalbero sinistro della radice, incontriamo un nodo con il nome ADAMS che non può ancora essere stampato per *prim*o. Il nome

Fig. 8.3.
Rappresentazione
schematica dei
sottoalberi
di un albero binario
ordinato.



ADAMS dovrà, tuttavia, essere stampato prima del nodo radice DROVER. Il problema che ora sorge è cosa fare una volta incontrato il secondo nome ADAMS che non può ancora essere stampato. Ancora navolta l'unica alternativa è memorizzarlo e cercare nel sottoalbero sinistro di radice ADAMS. Da un esame più accurato di questa situazione, scopriamo che quello che dobbiamo fare (per quanto concerne la stampa) quando ci troviamo nel nodo contenente ADAMS è essenzialmente quello che dovevamo fare in corrispondenza del nodo DROVER; cioè, non appena si arriva al nodo contenente ADAMS possiamo proseguire con il processo stampando prima tutti i membri del suo sottoalbero sinistro, poi ADAMS e quindi i membri del suo sottoalbero destro. La sequenza di passi suggerisce che per svolgere il problema può essere usata la ricorsione, perché muovendoci dalla radice al sottoalbero sinistro e così via noi applichiamo lo stesso processo più e più volte. Se potessimo formulare il problema ricorsivamente, saremmo in grado di evitare il reale salvataggio di nomi incontrati in anticipo rispetto al loro ordine di stampa.

In precedenza abbiamo visto che i passi necessari alla stampa dei nomi in ordine lessicale erano:

1. Attraversa e stampa tutti i nomi nel sottoalbero sinistro;
2. Stampa il nome alla radice dell'albero;
3. Attraversa e stampa i nomi del sottoalbero destro.

Per costruire la nostra procedura ricorsiva dovremo perfezionare i passi (1) e (3) e definire le condizioni di "terminazione per una chiamata ricorsiva. Per iniziare il processo di stampa del sottoalbero sinistro possiamo far in modo che la radice del sottoalbero diventi la radice dell'albero a cui applicare di nuovo i passi dall'(1) al (3). Questo significa fare una chiamata ricorsiva della nostra procedura originale a 3 passi. Se chiamiamo *inorder* la nostra procedura ricorsiva, allora una chiamata ricorsiva nella forma:

inorder(current↑.left)

ci permetterà di discendere alla radice del sottoalbero sinistro. Analogamente, una chiamata ricorsiva del tipo:

inorder(current↑.right)

ci permetterà di discendere alla radice del sottoalbero destro del nodo corrente. I passi necessari per la chiamata ricorsiva di *inorder* sono:

1. *inorder(current↑.left)*;
2. scrivi il nome nel nodo corrente;
3. *inorder(current↑.right)*.

Possiamo vedere che sarà necessario un certo numero di chiamate ricorsive prima di incontrare il nome AARONS (il primo nome da stampare). Quando raggiungiamo il nodo contenente AARONS e cerchiamo di applicare i tre passi:

- Attraversa e stampa i nomi del sottoalbero sinistro di AARONS;
- Stampa il nome AARONS;
- Attraversa e stampa i nomi del sottoalbero destro di AARONS.

Troviamo che il nodo AARONS non ha il sottoalbero sinistro poiché il suo puntatore sinistro vale *nil*. Ne segue che ogni volta cerchiamo di trattare un albero (o sottoalbero) con radice *nil*, il processo ricorsivo deve terminare. Considerando che la radice del sottoalbero destro di AARONS è anch'esso *nil* vediamo che tutto ciò che potrebbe accadere durante il trattamento del sottoalbero di radice AARONS è:

- Non c'è il sottoalbero destro quindi termina la ricorsione del ramo sinistro.
- Stampa la radice del sottoalbero AARONS.
- Non c'è il sottoalbero destro, quindi termina la ricorsione del ramo destro.

Il risultato della visita del nodo contenente AARONS è semplicemente la stampa del nome, come richiesto. I tre passi nella procedura ricorsiva possono quindi essere protetti da un'istruzione del tipo:

se la radice del sottoalbero corrente non è *nil*, allora
(a) procedi con la procedura ricorsiva a tre passi.

Abbiamo ora una completa formulazione della procedura ricorsiva di stampa in ordine lessicale o alfabetico dei nomi di un albero binario ordinato. Ciò corrisponde ad un attraversamento simmetrico dei nodi dell'albero.

Prima di essere completamente soddisfatti della corretta formulazione del nostro algoritmo, ne seguiremo accuratamente i passi attraverso un breve esempio. Il risultato di questa traccia dell'esempio considerato precedentemente è mostrato in Fig.8.4.

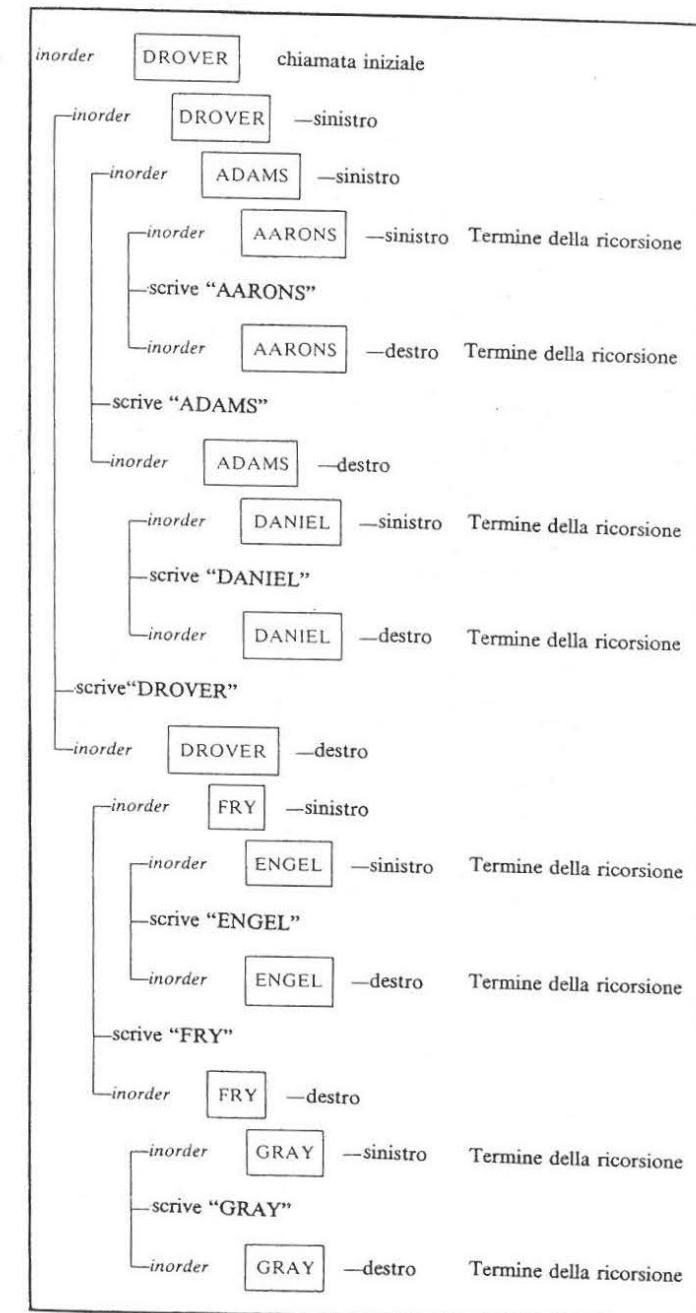
Nell'implementare algoritmi ricorsivi dovremo sempre fare attenzione alla potenziale profondità della ricorsione del nostro problema. Se è possibile che essa sia grande relativamente alla dimensione del problema, dovremo, o cercare una soluzione iterativa, o cambiare il limite superiore della profondità della ricorsione. Il motivo è che con problemi di grandi dimensioni può esserci un costoso incremento di memoria necessaria. Per il nostro problema attuale, a patto che l'albero sia stato costruito accuratamente, la profondità della ricorsione sarebbe dell'ordine di $\log n$ per un albero con n nodi.

Prima di fornire la descrizione completa del metodo simmetrico di visita è necessario considerare i metodi *anticipato* e *differito*. Gli altri due metodi di visita di un albero binario che vogliamo considerare differiscono per l'ordine in cui visitano il nodo radice ed i suoi sottoalberi sinistro e destro.

Per la visita *anticipata*, l'ordine in cui vengono attraversati il nodo radice ed i sottoalberi sinistro e destro è:

- Visita la radice e stampa il nome.
- Visita il sottoalbero sinistro.
- Visita il sottoalbero destro.

Fig. 8.4.
Traccia della visita in ordine simmetrico di un albero.



Come per l'attraversamento simmetrico, questo processo e quello differito descritto qui di seguito, sono ricorsivi.

Per l'attraversamento *differito*, l'ordine in cui sono visitati il nodo radice ed i due sottoalberi sinistro e destro è:

1. Visita il sottoalbero sinistro.
2. Visita il sottoalbero destro.
3. Visita la radice e stampa il nome.

Le visite in ordine *anticipato* e *differito* sono entrambe importanti in relazione alla valutazione tramite il computer di espressioni aritmetiche che si può supporre abbiano la forma di un albero binario. I nodi interni o ramificati dell'albero contengono gli operatori, mentre i nodi foglia contengono le variabili e le costanti.

Attraversando la rappresentazione ad albero binario di un'espressione aritmetica si trova una rappresentazione dell'espressione (detta *notazione inversa polacca o postfissa*), in cui ogni operatore segue la sua coppia di operandi. Quest'ultima rappresentazione è molto conveniente per la valutazione di tali espressioni con l'uso di una pila. Non andremo oltre con lo sviluppo delle procedure anticipata e differita poiché la loro implementazione ricorsiva deriva direttamente dalle definizioni e ha una forma molto simile a quella della procedura simmetrica.

Prima di abbandonare questo argomento, è istruttivo confrontare i tre metodi di visita che abbiamo considerato per assicurarci di avere capito bene le loro differenze. Se riguardiamo l'albero binario dei nomi, vediamo che la visita in ordine simmetrico corrisponde all'ordine in cui incontriamo i nodi quando ci spostiamo da sinistra a destra della pagina. Un altro modo di considerare questo e gli altri metodi di visita è di immaginare l'albero come una struttura «solida e muoversi attorno al suo perimetro partendo dal sottoalbero sinistro come mostrato in Fig.8.5. (Il percorso seguito è contrassegnato da una linea tratteggiata).

Fig. 8.5.
Percorso della visita
di un albero.

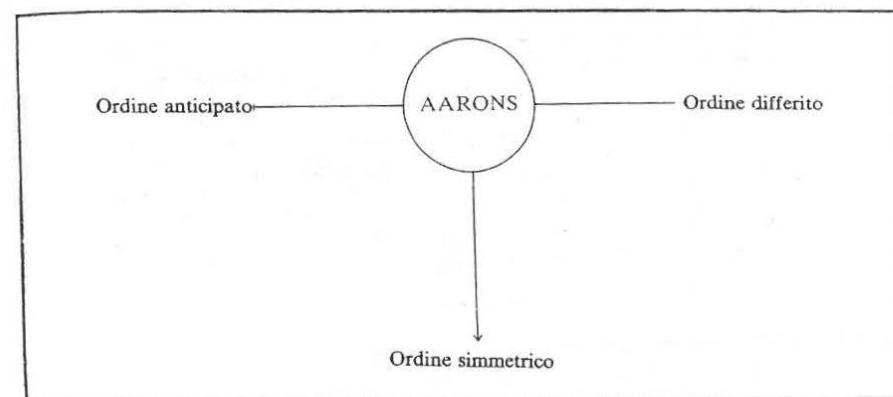
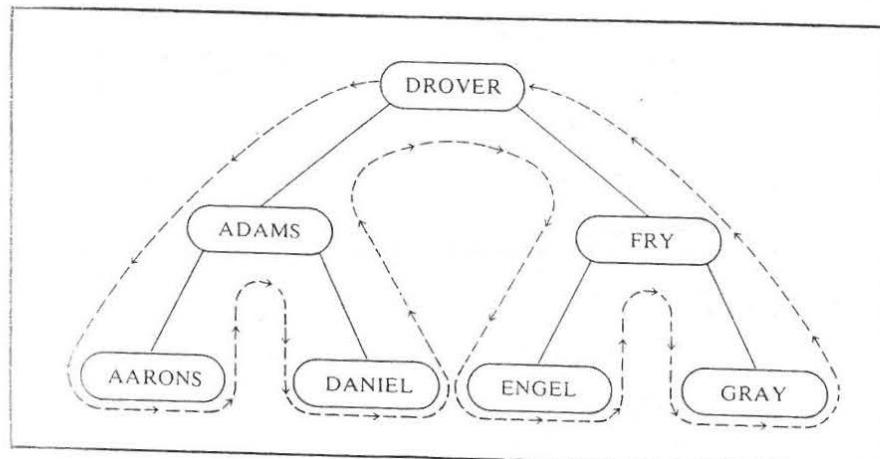


Fig. 8.6.
Schema per la
destinazione tra le
visite in ordine
anticipato,
simmetrico e
differito.

Ora, se immaginiamo che l'albero possa avere tre raggi (due orizzontali ed uno verticale verso il basso) che si dipartono come indicato in Fig.8.6, vedremo molto semplicemente la differenza tra le visite *simmetrica*, *anticipata*, *differita*. L'ordine in cui il nostro percorso tratteggiato attraversa il raggio verticale di ogni nodo corrisponde all'attraversamento *simmetrico*. Analogamente, l'ordine in cui esso attraversa il raggio a *sinistra* di ogni nodo corrisponde all'*anticipato* ed il raggio a *destra* corrisponde alla visita in ordine *differito*.

I tre diversi ordini dell'insieme di nomi del nostro esempio sono illustrati di seguito:

Ordine simmetrico

AARONS, ADAMS, DANIEL, DROVER, ENGEL, FRY, GRAY

Ordine anticipato:

DROVER, ADAMS, AARONS, DANIEL, FRY, ENGEL, GRAY

Ordine differito:

AARONS, DANIEL, ADAMS, ENGEL, GRAY, FRY, DROVER

Descrizione dell'algoritmo [*]

- (1) Procedura per la visita in ordine simmetrico di un albero binario ordinato

[*] Non vengono incluse le descrizioni delle visite in ordine anticipato e differito in quanto differiscono dalla procedura corrente solo dalla disposizione dei punti (a), (b) e (c) del passo 2.

1. Definisci il nodo corrente (inizialmente la radice dell'albero).
2. Se il nodo corrente non è *nil*, allora
 - (a) compi ricorsivamente una visita simmetrica del sottoalbero sinistro del nodo corrente;
 - (b) stampa il nome del nodo corrente;
 - (c) compi una visita in ordine simmetrico del sottoalbero destro del nodo corrente.

Implementazione in Pascal

```

procedure inorder(current: treepointer);
begin
  if current <> nil then
    begin
      inorder(current^.left);
      {assert: names in all nodes in left subtree of current node have
       been printed in lexical order}
      writeln(current^.treename);
      {assert: name in current node printed}
      inorder(current^.right)
      {assert: names in all nodes in right subtree of current node
       have been printed in lexical order}
    end
  end

procedure preorder(current: treepointer);
begin
  if current <> nil then
    begin
      writeln(current^.treename);
      {assert: name in current node printed}
      preorder(current^.left);
      {assert: names in all nodes in left subtree of current node have
       been printed in preorder order}
      preorder(current^.right)
      {assert: names in all nodes in right subtree of current node
       have been printed in preorder order}
    end
  end

procedure postorder(current: treepointer);
begin
  if current <> nil then
    begin
      postorder(current^.left);
      {assert: names in all nodes in left subtree of current node have
       been printed in postorder order}
      writeln(current^.treename);
      {assert: name in current node printed}
      postorder(current^.right)
      {assert: names in all nodes in right subtree of current node
       have been printed in postorder order}
    end
  end

```

```

      been printed in postorder order}
      postorder(current^.right);
      {assert: names in all nodes in right subtree of current node
       have been printed in postorder order}
      writeln(current^.treename)
      {assert: name in current node printed}
    end
end

```

Note di progetto

1. L'esecuzione degli algoritmi di visita in ordine simmetrico, anticipato e differito è direttamente proporzionale al numero di nodi dell'albero. Il numero di chiamate ricorsive effettuate è probabilmente la misura migliore da usare. Per un albero binario ordinato di n nodi, con la presente implementazione ci saranno $(2n + 1)$ chiamate di ognuna delle procedure ricorsive.
2. Bisogna impostare che la procedura *inorder* stampi per primi i nomi contenuti nel sottoalbero sinistro, quindi il nome alla radice e tutti gli elementi nel sottoalbero destro. Per cercare di caratterizzare l'andamento di questa procedura, consideriamo il caso in cui la chiamata corrente di *inorder* non comporti alcuna ulteriore chiamata ricorsiva interna di se stessa. Ciò implica che siamo giunti ad un nodo radice, poiché non ci sarà alcuna ricorsione interna solo per *current^.left* e *current^.right nil*. In questo caso, le due chiamate interne di *inorder* non avranno alcun effetto. Come risultato il nome corrente (cioè il nome nel nodo foglia) verrà stampato come richiesto. Se ora supponiamo che la chiamata interna di *inorder* si risolva nella corretta stampa del nome del sottoalbero sinistro, allora la chiamata corrente di *inorder* si risolverà nella stampa del nome del nodo corrente e quindi di tutti i nomi nel sottoalbero destro. Quindi verrà stampato correttamente anche l'albero che racchiude i due sottoalberi destro e sinistro. Questo argomento può formare le basi per una prova induttiva formale della procedura. Ogni chiamata ricorsiva di *inorder* si risolve in uno spostamento verso un livello più basso dell'albero e poiché noi sappiamo che la ricorsione termina sempre ali nodi foglia, possiamo inesattamente concludere che l'algoritmo terminerà. Le procedure *preorder* e *postorder* possono essere trattate in maniera analoga.
3. Se fossero state fatte implementazioni non ricorsive di questi algoritmi di visita, sarebbe stato necessario salvare esplicitamente su di una pila le informazioni relative ai nodi visitati ma non stampati. Il risultato è in ogni caso un'implementazione apparentemente molto più complicata e meno trasparente.
4. Nelle situazioni in cui è necessaria una pila, di solito è meglio usare un'implementazione ricorsiva del problema.
5. Nella presente implementazione notiamo che in ogni nodo foglia ci sono sempre due chiamate infruttuose di procedura. Ciò può essere costoso, poiché ci sono sempre più nodi foglia che nodi interni. È possibile evitare queste chiamate ridondanti arrestando la ricorsione

un passo prima. Per questo dobbiamo aggiungere un esplicito, ma computazionalmente economico test per i nodi foglia. I cambiamenti saranno:

```
begin  
  notleaf := (current^.left <> nil) or (current^.right <> nil)  
  if notleaf then  
    begin  
      if (current^.left <> nil) then inorder(current^.left);  
      writeln(current^.treename);  
      if (current^.right <> nil) then inorder(current^.right)  
    end  
  else  
    writeln(current^.treename)  
end
```

Se utilizziamo questa implementazione sarà necessario un test separato fuori dalla procedura per assicurare che la procedura non sia chiamata per stampare un albero senza nodi.

6. Per risolvere questo problema abbiamo utilizzato la ricorsione binaria.

Applicazioni

Trattamento di un grande numero di informazioni, valutazione di espressioni aritmetiche.

Problemi supplementari

- 8.1.1 Progettare un algoritmo non ricorsivo per la visita simmetrica di un albero binario ordinato.
- 8.1.2 Progettare una procedura ricorsiva per contare il numero dei nodi di un albero binario ordinato.
- 8.1.3 Progettare una procedura ricorsiva che conti il numero dei nodi foglia di un albero binario ordinato.
- 8.1.4 Progettare ed implementare algoritmi ricorsivi per «l'inserimento e la cancellazione in un albero».
- 8.1.5 Progettare un algoritmo che usi un albero binario ordinato per ordinare un insieme di nomi. Una volta creato l'albero, stampare il risultato ordinato.
- 8.1.6 Un albero binario può essere visitato senza usare direttamente una pila o la ricorsione. Ciò può essere fatto attraversando l'albero in ordine triplo definito dai passi seguenti (ogni nodo è visitato tre volte):

Meccanismo di ordine triplo

1. visita il nodo corrente
2. attraversa il suo sottoalbero sinistro
3. visita il nodo corrente
4. attraversa il suo sottoalbero destro
5. visita il nodo corrente.

Se assicuriamo che ogni puntatore *nil* sia sostituito da un puntatore riflessivo al nodo stesso, per attraversare l'albero in ordine triplo può essere usato il seguente aggiustamento dei puntatori in un semplice schema iterativo. L'algoritmo può terminare quando *current=nil*.

Aggiustamento dei puntatori per una visita in ordine triplo
next: = current^.left;
current^.left: = current^.right;
current^.right: = previous;
previous: = current;
current: = next.

Implementare questo metodo di visita di un albero (Ved. B: Dwyer, "Simple algorithms for traversing a tree without an auxiliary stack", Inf. Proc. Letts., 2, 143-5 (1974)).

ALGORITMO 8.2 QUICKSORT RICORSIVO

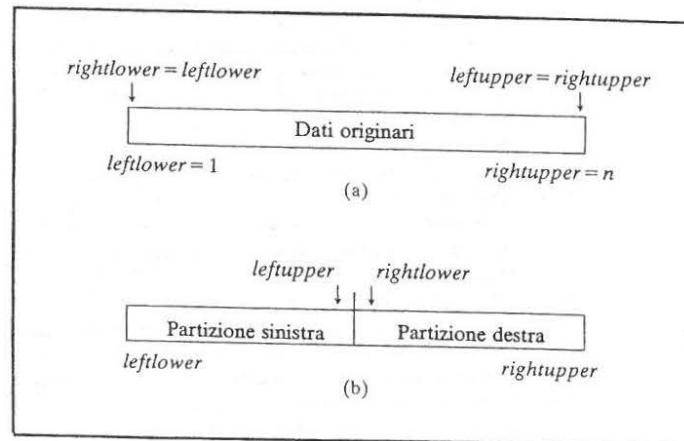
Problema

Progettare ed implementare una versione ricorsiva dell'algoritmo di ordinamento rapido (quicksort).

Sviluppo dell'algoritmo

Abbiamo visto precedentemente che l'algoritmo di quicksort (algoritmo 5.6) usa la strategia di divide et impera. L'insieme originale dei dati viene dapprima diviso in due "partizioni" come mostrato in Fig. 8.7. Dopo la separazione, tutti gli elementi nelle prime (*leftlower* — 1) posizioni (partizione di sinistra) saranno minori o uguali di tutti gli elementi in posizione da (*leftupper* + 1) a *n* (la partizione di destra). Dopo questo primo passo, sarà applicato alle due partizioni più piccole lo stesso meccanismo (cioè ad *a[1..leftupper]* e *a[rightlower..n]*). La partizione di segmenti sempre più piccoli continua finché non troveremo un segmento di un solo elemento. A questo punto, dalla nostra

Fig. 8.7.
(a) Prima della separazione;
(b) Dopo la separazione.



precedente discussione segue che l'array sarà completamente ordinato.

In questo meccanismo si vede che lo stesso processo viene ripetutamente applicato a problemi sempre più piccoli: esso è quindi di natura ricorsiva. Dopo aver fatto queste osservazioni, scriviamo i passi si base che si ripetono e cerchiamo di sviluppare un algoritmo ricorsivo per il problema. I passi sono:

1. Dividi i dati nelle partizioni sinistra e destra purchè nell'insieme ci sia più di un elemento.
2. Ripeti il processo di partizione alla partizione di sinistra.
3. Ripeti il processo di partizione alla partizione di destra.

Per costruire la nostra procedura ricorsiva, dobbiamo perfezionare questi passi e definire i dettagli per la terminazione della ricorsione.

In relazione al meccanismo di partizione, possiamo usare il metodo di partizionamento descritto nell'algoritmo 5.6. Il primo passo del nostro quicksort ricorsivo può quindi essere una chiamata di procedura del tipo:

partition(a, leftlower, rightupper, leftupper, rightlower)

Non discuteremo qui il meccanismo di partizione se non per dire che *rightlower* e *leftupper* definiranno i limiti dei sottoinsiemi dopo la chiamata di *partition*. I valori *leftlower*, *leftupper*, *rightlower* e *rightupper* derivano dal nostro algoritmo originario.

Consideriamo ora i passi (2) e (3). Per ripetere il processo di partizione alla partizione di sinistra, possiamo compiere una chiamata ricorsiva della nostra originale procedura a tre passi. Se la procedura ricorsiva è chiamata *quicksort2*, allora una chiamata del tipo:

quicksort2(a, leftlower, leftupper)

ci permetterà di ordinare la partizione di sinistra. Analogamente, una chiamata ricorsiva del tipo:

quicksort2(a, rightlower, rightupper)

ci permetterà di ordinare la partizione di destra.

Il compito che ci rimane è trovare le condizioni per la terminazione. Ci aspetteremmo che essa sia collegata al segmento più piccolo che deve essere partizionato. Se vengono effettuate chiamate ricorsive di *quicksort2* con partizioni sempre più piccole, verrà sicuramente fatta una chiamata con un sottoinsieme di dimensione unitaria.

Dalla descrizione precedente del partizionamento vediamo che non è significativo cercare di partizionare un insieme di un solo elemento. A questo punto, inoltre, non avrà più senso effettuare altre chiamate ricorsive, poiché non ci saranno più partizioni di destra e sinistra su cui operare. Possiamo quindi concludere che la chiamata di un segmento di un solo elemento comporterà la terminazione del processo ricorsivo. Il corpo di tale processo potrebbe essere sorvegliato da un test per vedere se il sottoinsieme corrente contiene più di un elemento. Dato che nella chiamata di *quicksort2* sono definiti i limiti superiore ed inferiore del sottoinsieme, possiamo usare il test:

se "il segmento contiene più di un elemento" allora
 (a) partiziona in sottoinsiemi sinistro e destro,
 (b) *quicksort2* partizione sinistra
 (b) *quicksort2* partizione di destra.

Dato che *rightupper* sarà maggiore di *leftlower* quando il segmento contiene più di un elemento il test può assumere la forma:

se *leftlower* < *rightupper* allora
 (a) continua con la procedura ricorsiva

Implementazione in Pascal

```

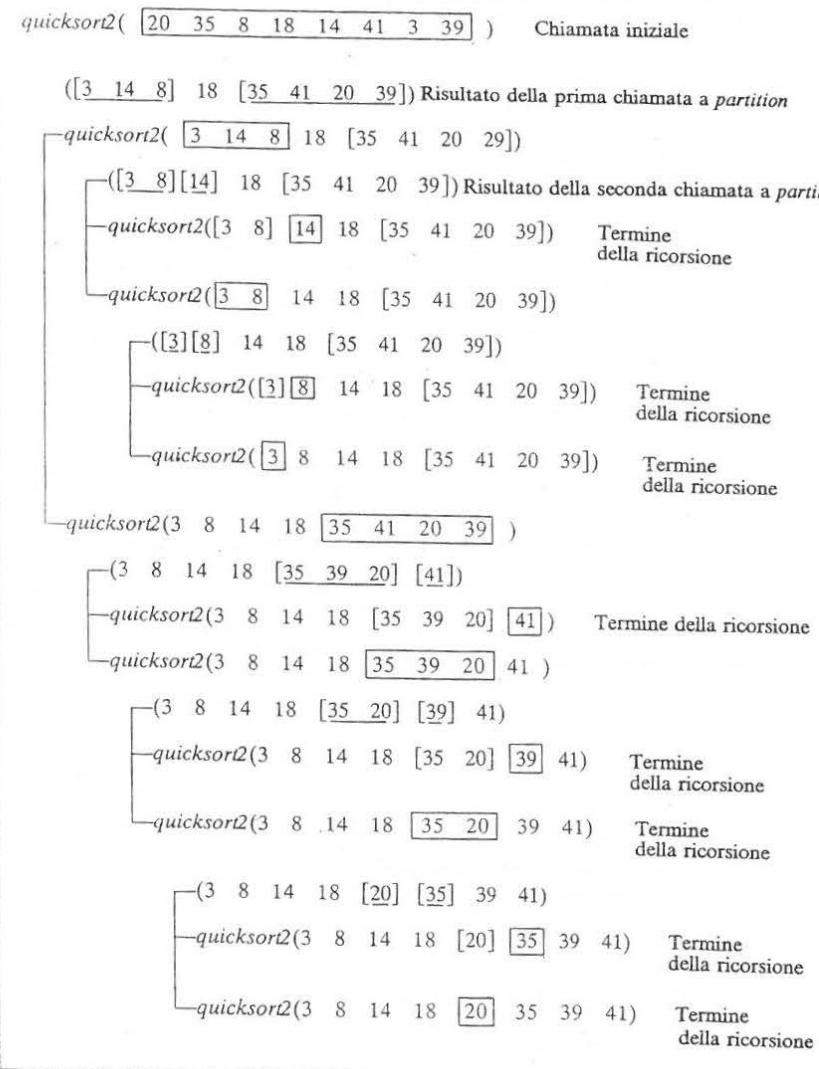
procedure quicksort2(var a:nelements; leftlower,
                      rightupper:integer);
var leftupper {upper limit of left segment},
    rightlower {lower limit of right segment}:integer;
begin
  if leftlower<rightupper then
    begin
      partition(a, leftlower, rightupper, leftupper, rightlower);
      quicksort2(a, leftlower, leftupper);
      quicksort2(a, rightlower, rightupper)
    end
end

```

Fig. 8.8.

Traccia del quicksort ricorsivo.

Ancora una volta abbiamo usato la ricorsione binaria per risolvere il problema. A questo punto è utile fare una traccia di un piccolo esempio, Fig. 8.8, per assicurarsi di aver capito il meccanismo e convincersi della sua correttezza. La partizione effettuata dalla chiamata corrente di *quicksort2* è "incassata". Le due partizioni generate dalla chiamata corrente di *partition* sono sottolineate. Ogni chiamata è posta tra parentesi quadre. Notare che poiché il partizionamento termina



sull'elemento 18, esso può essere escluso in prima istanza, poiché si trova già in posizione corretta.

Da questo esempio vediamo che c'è un numero considerevole di partizioni costruite che devono essere ordinate successivamente. Ci potremo, quindi chiedere quale sia la massima profondità della ricorsione che può risultare da uno spazio di memoria occupata molto grande. Nel caso peggiore possibile, ogni chiamata di *partition* con un sottoinsieme di n elementi, fornirà una partizione sinistra di $(n-1)$ elementi ed una destra di un elemento che sarà salvato e trattato successivamente. Ne segue che la profondità della ricorsione può aumentare fino a $(n-1)$. Questo accadrà nel caso di n molto grande e di un considerevole spazio di memoria utilizzato. Nell'implementazione a pila dell'algoritmo avevamo aggirato questo problema ordinando per prime le partizioni più piccole. La massima dimensione della pila era allora di $\log_2 n$, con n elementi da ordinare. Da ciò deriva che il massimo problema più piccolo ad ogni chiamata successiva è $1/2 n$. Prima che la pila diminuisca ci potranno essere al massimo $\log_2 n$ chiamate. Per limitare a $\log_2 n$ la profondità della ricorsione dovremo essere in grado di trasferire questi concetti nell'implementazione ricorsiva. Ci sono solo due situazioni di cui ci dovremo occupare:

1. La partizione di sinistra è minore di quella di destra, nel cui caso deve essere ordinata per prima.
2. La partizione di destra è minore o uguale a quella di sinistra, nel cui caso deve essere ordinata per prima.

Queste due situazioni possono essere adattate facendo precedere la chiamata ricorsiva da un test che decida l'ordine in cui debbano essere ordinate le due partizioni. Cioè:

- se la partizione di sinistra è minore di quella di destra, allora
- (a) ordina per prima la partizione di sinistra,
 - (b) quindi ordina la partizione di destra,
 - altrimenti
 - (c) ordina per prima la partizione di destra,
 - (d) quindi ordina la partizione di sinistra

Può essere ora data la completa descrizione dell'implementazione ricorsiva di *quicksort2*.

Descrizione dell'algoritmo

1. Definisci l'array degli elementi da ordinare ed i limiti superiore ed inferiore del corrente segmento che deve essere ordinato.
2. Se il segmento corrente contiene più di un elemento, allora
 - (a) partiziona il segmento corrente in due segmenti più piccoli in modo che tutti gli elementi del segmento di sinistra siano minori o uguali di tutti quelli di destra;

- (b) se il segmento di sinistra è minore di quello di destra, allora
- chiama ricorsivamente *quicksort2* per il più piccolo segmento di sinistra,
 - chiama ricorsivamente *quicksort2* per il più piccolo segmento di destra, altrimenti
 - chiama ricorsivamente *quicksort2* per il più piccolo segmento di destra,
 - chiama ricorsivamente *quicksort2* per il più piccolo segmento di sinistra.

Implementazione in Pascal

```

procedure partition{var a: nelements; l,u: integer; var j,i: integer};
var k {index of partitioning element},
    t {temporary variable for exchange},
    x {guess at median - used for partitioning}: integer;
begin {partition segment a[l..u] into segments a[l..j] and a[j..u]
about x}
  {assert: l=<u}
  k := (l+u) div 2;
  x := a[k];
  i := l;
  j := u;
  while a[i]<x do i := i+1;
  while x<a[j] do j := j-1;
  {invariant: l=<i ∧ all a[l..i-1]=<x ∧ j=<u ∧ x=<all a[j+1..u]}
  while i<j-1 do
    begin {exchange wrongly partitioned pair and then extend both
partitions}
      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 x<a[j] do j := j-1
    end;
  if i=<=j then
    begin
      if i<j then
        begin
          t := a[i];
          a[i] := a[j];
          a[j] := t
        end;
      i := i+1;
      j := j-1
    end
  {assert: j<i ∧ all a[l..i-1]=<all a[j+1..u]}
end

```

```

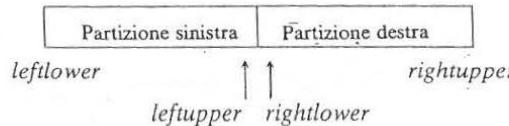
procedure quicksort2{var a: nelements; leftlower, rightupper:
integer};
var leftupper, {upper limit of left segment}
    rightlower {lower limit of right segment}: integer;

begin {proceed with sort if segment contains more than one
element}
  if leftlower<rightupper then
    begin {partition current into two smaller segments}
      partition(a, leftlower, rightupper, leftupper, rightlower);
      {assert: leftlower<rightlower ∧ all a[leftlower .. leftupper]=<all a[rightlower .. rightupper]}
      if (leftupper - leftlower)<(rightupper - rightlower) then
        begin {sort the smaller left segment first}
          quicksort2(a, leftlower, leftupper);
          {assert: a[leftlower .. leftupper] sorted in non-descending
order}
          quicksort2(a, rightlower, rightupper)
          {assert: a[rightlower .. rightupper] sorted in non-descending
order}
        end
      else
        begin {sort smaller right segment first}
          quicksort2(a, rightlower, rightupper);
          {assert: a[rightlower .. rightupper] sorted in
non-descending order}
          quicksort2(a, leftlower, leftupper)
          {assert: a[leftlower .. leftupper] sorted in non-descending
order}
        end
    end
  end
end

```

Note di progetto

- In termini di confronti e scambi, l'esecuzione dell'implementazione ricorsiva di quicksort è la stessa di quella a pila. (Vedere descrizione dell'algoritmo 5.6). Nel caso peggiore, sarà richiesto un numero di confronti dell'ordine di n^2 per l'ordinamento, mentre in media saranno necessari circa $n \log_2 n$ confronti.
- Per la procedura *quicksort2* occorre stabilire che, dopo il completamento dell'esecuzione, gli elementi nell'array saranno in ordine non decrescente (in ordine crescente se sono singoli). L'andamento dell'algoritmo si caratterizza in termini di limitazione dei limiti delle partizioni di sinistra e destra. Se la chiamata corrente di *quicksort2* si risolve in una chiamata di *partition*, allora l'ultima funzione ritnerà due parametri *leftupper* e *rightlower*; permuterà inoltre gli elementi compresi tra *leftlower* e *rightupper* in modo che tutti gli elementi della partizione di sinistra saranno minori o uguali a quelli della partizione di destra.
Cioè:



e quindi si è in situazione di ordinamento parziale. Se ora supponiamo che le due chiamate ricorsive di *quicksort2* funzionino correttamente allora il risultato sarà il corretto ordinamento degli elementi compresi tra *leftlower* e *rightupper*, come richiesto originariamente. L'algoritmo termina perché ad ogni chiamata di *quicksort2* vengono esaminati per l'ordinamento due segmenti, più piccoli. Alla fine tutti i segmenti si ridurranno a dimensione unitaria, situazione in cui la ricorsione ha sempre termine.

3. Con questo problema abbiamo visto l'importanza di fare attenzione alla profondità della ricorsione. Quando un problema viene suddiviso in due problemi più piccoli e si usa la ricorsione, è in genere meglio cominciare da quello più piccolo.
4. L'implementazione ricorsiva del quicksort è concettualmente la più semplice, più naturale e la più facile da programmare. Per applicazioni pratiche, tuttavia, in cui il tempo di ordinamento ha più alta priorità, non si raccomanda l'uso dell'implementazione ricorsiva a causa dell'alto numero di chiamate ricorsive a cui è esposta.
5. La condizione di terminazione della ricorsione deriva dalla considerazione del problema di ordinamento più piccolo.
6. Abbiamo visto ancora una volta come possa essere usata la ricorsione binaria per implementare un importante algoritmo.

Applicazioni

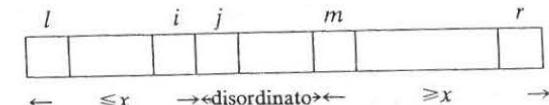
Dimostrazione della natura ricorsiva dell'algoritmo di quicksort.

Problemi supplementari

- 8.2.1 Implementare le due versioni ricorsive del quicksort date e confrontare la massima profondità della ricorsione che richiedono per dati casuali.
- 8.2.2 Confrontare le differenze tra il tempo di esecuzione per le due implementazioni ricorsive per:
 - (a) un insieme di dati che producono il caso peggiore di esecuzione,
 - (b) dati casuali.
- 8.2.3 Confrontare le differenze tra i tempi di esecuzione dell'implementazione ricorsiva migliore e quella a pila del quicksort sul vostro particolare calcolatore.

- 8.2.4 Modificare l'algoritmo di *quicksort2* in modo che generi un istogramma della frequenza di chiamate fatte per segmenti di dimensioni comprese tra 1 ed n. Che conclusione potete trarre da questo profilo?

- 8.2.5 Un quicksort ricorsivo modificato può essere implementato basandolo sul meccanismo di partizione finale usato nell'algoritmo 4.6. Questo meccanismo spezza ad ogni chiamata un segmento in tre parti come mostrato qui di seguito:



La struttura di base dell'algoritmo è:

```
procedure threesort (var a: nelements; l, r: integer);
  :
  if l < r then
    begin
      partition (a, l, r, i, j, m);
      threesort (a, l, i);
      threesort (a, j, m-1);
      threesort (a, m, r);
      threemerge (a, l, i, j, m, r)
    end
  :
  
```

dove la procedura *threemerge* fonde i tre segmenti dopo che sono stati ordinati. Implementare tale algoritmo.

ALGORITMO 8.3 PROBLEMA DELLE TORRI DI HANOI

Problema

Progettare ed implementare un algoritmo ricorsivo per "risolvere il problema delle Torri di Hanoi per uno o più dischi.

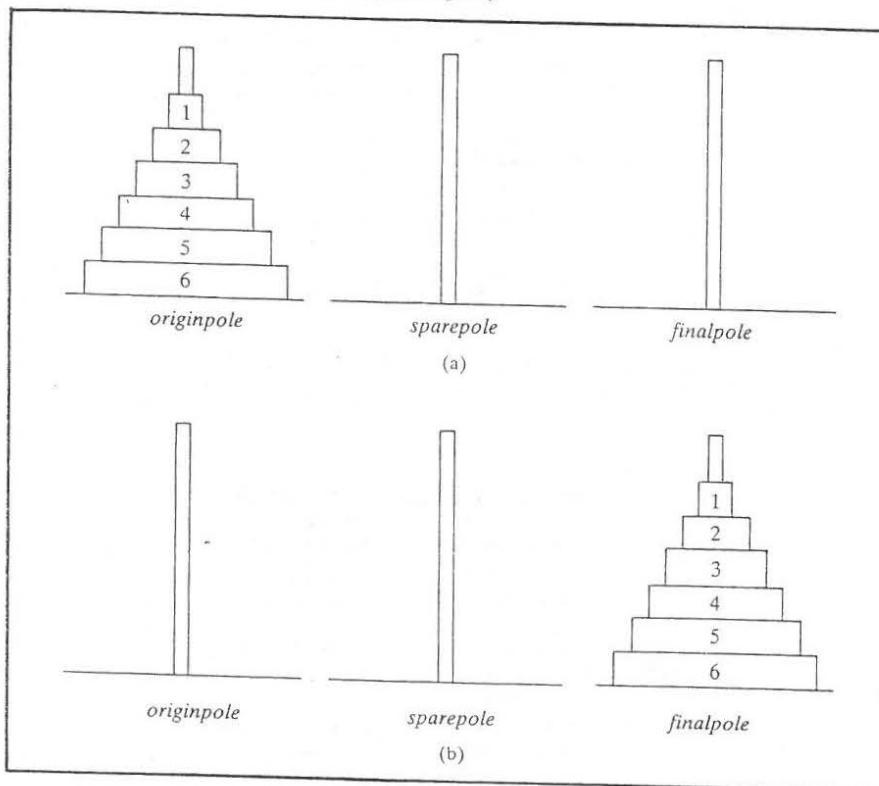
Sviluppo dell'algoritmo

Un'antica fiaba Bramina narra che la vita dell'universo è definita in termini del tempo impiegato da un gruppo di monaci (che lavorano continuamente) per spostare un insieme di 64 dischi d'oro, tutti di diametro diverso, da un palo all'altro. Ci sono regole specifiche su come debba essere fatto il trasferimento. Queste regole rendono il progetto non banale ed anche a "lungo termine". Esse sono:

1. Può essere mosso un solo disco alla volta.
2. Può essere usato un terzo palo per l'immagazzinamento temporaneo dei dischi.
3. Nessun disco deve essere posto sulla cima di uno avente diametro inferiore.
4. I dischi sono inizialmente ordinati sul palo originale con il disco più largo in basso, il secondo in ordine di grandezza sopra il primo e così via. Il disco più piccolo si trova in cima alla pila.

Il problema, come è stato definito, è conosciuto come problema delle Torri di Hanoi (ved. Fig. 8.9).

Fig. 8.9.
Il problema delle
Torri di Hanoi per
sei dischi:
(a) Configurazione
iniziale;
(b) Configurazione
finale richiesta.

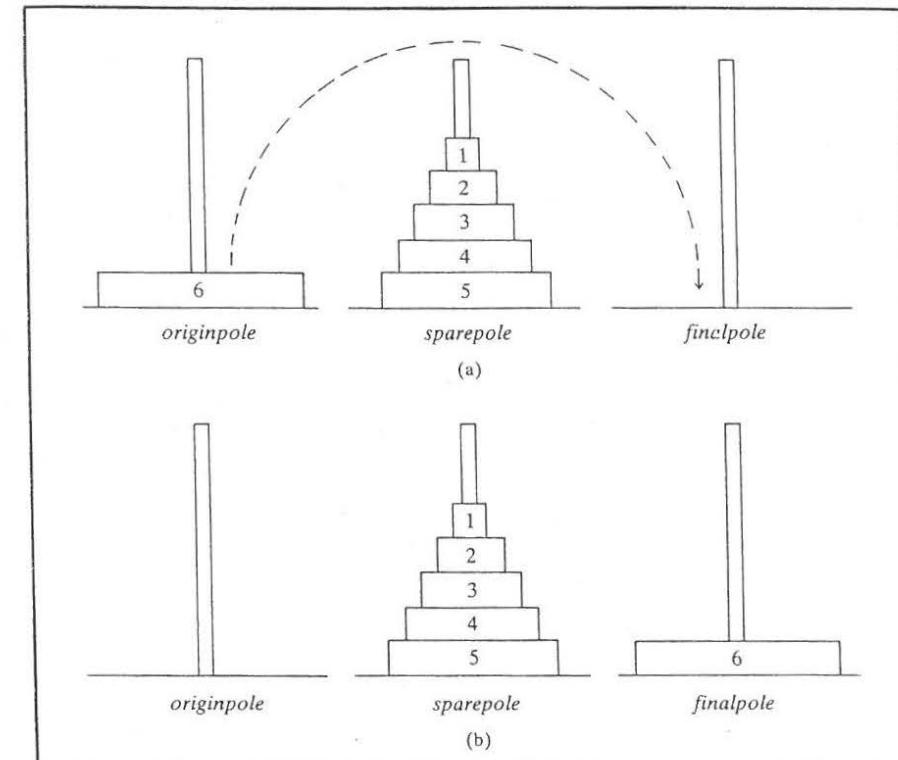


Non appena iniziamo a divertirci con questo problema, ci accorgiamo presto che esso è lontano dall'essere semplice, anche per il caso in cui debbano essere spostati solo 6 dischi da *originpole* a *finalpole* (questo caso, infatti, comporta per lo meno 63 trasferimenti di dischi). Se a questo punto non avete tentato di risolvere il problema anche per soli 4 dischi, è istruttivo farlo per avere un'idea di cosa comporti.

Dopo aver fatto queste considerazioni preliminari sul problema, abbiamo in genere la sensazione che sia difficile da spiegare, per non dire altro. Una soluzione con l'uso del calcolatore del problema sembra essere molto lontana. L'altra risposta che probabilmente avremo è che abbiamo l'impressione che debba esserci un procedimento sistematico per risolverlo. Il nostro compito è quindi cercare di scoprire un metodo sistematico che specifichi come debbano avvenire i trasferimenti dei dischi. Per questo considereremo il caso in cui debbano essere spostati 6 dischi da *originpole* a *finalpole*.

La nostra esperienza precedente ci ha resi consapevoli che è molto facile insabbiarsi nei dettagli e perdere di vista un possibile meccanismo. Con questo in mente, possiamo chiederci cosa potrebbe costituire un reale passo avanti verso l'obiettivo finale. Una risposta potrebbe

Fig. 8.10.
(a) Configurazione
prima del
trasferimento dal
disco più grande;
(b) Configurazione
dopo il trasferimento
dal disco più grande.



essere che avremmo progredito se avessimo trovato una configurazione in cui il disco più grande (il 6, nel nostro esempio), fosse stato spostato a *finalpole*, ma prima di essere in grado di farlo dovremo togliere tutti gli altri dischi! Questo comporta una seconda domanda: dove possono essere messi i dischi che si trovano sopra quello più grande, senza infrangere le regole date? Essi non possono ovviamente essere distribuiti tra *sparepole* e *finalpole* ma neanche tutti in *finalpole*. L'unica posizione conveniente per tutti sarebbe allora *sparepole*.

La corrispondente configurazione è mostrata in Fig.8.10(a). Solo in presenza di questa configurazione potremo trasferire il disco maggiore da *originpole* a *finalpole*. (Fig. 8.10(b)).

Se potessimo trovare un modo di muovere direttamente i cinque dischi più piccoli in *sparepole*, allora saremmo in grado di trasferire il disco maggiore da *originpole* a *finalpole*. Vediamo ora di delineare questa nuova situazione. Abbiamo fatto dei progressi (e delle ipotesi) sul fatto che il disco più grande sia dove vogliamo noi, ma ci sono ancora cinque dischi in posizione errata.

Per fare ulteriori progressi, il secondo disco più grande (il quinto) che attualmente è in fondo a *sparepole* dovrà essere spostato in cima al disco più grande su *finalpole*. Seguendo la stessa linea di ragionamento usata in precedenza, i quattro dischi più piccoli in cima al disco 5

dovranno essere temporaneamente spostati su *originpole*, dopodiché il disco 5 può essere spostato da *sparepole* a *finalpole*.

Anche per il secondo passo vale l'ipotesi che possediamo un mezzo per trasferire i quattro dischi più piccoli da *sparepole* a *originpole*. I passi necessari sono mostrati in Fig.8.11.

Rifacendo di nuovo il punto della situazione, vediamo che ora i due dischi maggiori si trovano nella posizione voluta e gli altri quattro si trovano in *originpole*. Da uno studio più accurato della configurazione più recente, riconosciamo che essa corrisponde essenzialmente a quella del problema originale. Ciò suggerisce che noi possiamo essere in grado di risolvere il problema ricorsivamente. Le ipotesi (che al primo passo potessimo avere i cinque dischi più piccoli in *sparepole*), possono finire "se se ne tiene conto", nella ricorsione.

Nella soluzione ricorsiva di un problema, ci aspettiamo sempre di essere in grado di applicare la stessa procedura ad un problema di *una dimensione minore*. Nella nostra situazione, il problema di lunghezza 4 è simile a quello di lunghezza 6. Ci sembra di avere in qualche modo evitato il problema di lunghezza 5, non è vero? Guardiamolo nuovamente e più da vicino: troviamo che nell'ultimo passo esso è realmente analogo al problema di dimensione 6. L'unica differenza è che sono stati scambiati i ruoli di *sparepole* e *originpole*. Ciò è incoraggiante, perché ora cominciamo a vedere la relazione tra il problema di una dimensione data e quello di dimensione inferiore di una unità. Se vogliamo essere in grado di costruire un algoritmo ricorsivo per il problema, dovremo cercare di capire completamente questa relazione.

A questo punto cerchiamo di applicare ciò che abbiamo imparato per definire i passi di base dell'algoritmo in termini dello stesso problema di dimensione inferiore.

Se, in generale, devono essere trasferiti *ndisks*, allora il problema iniziale è di:

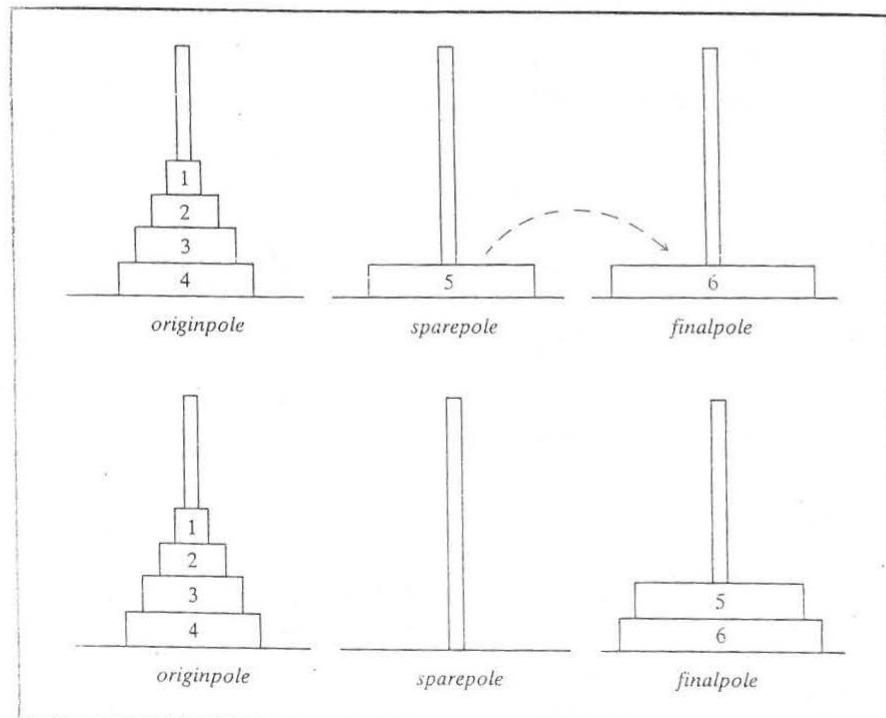
trasferire *ndisks* da *originpole* a *finalpole*

In termini di un problema con (*ndisks* — 1) dischi, il meccanismo comporterà i passi mostrati in Fig.8.12. (Confrontarla con la Fig.8.10 che mette in relazione il problema di lunghezza 6 con quello di lunghezza 5).

Da questa descrizione riconosciamo che entrambi i passi (a) e (c) implicano la soluzione di problemi inferiori di una unità rispetto al problema originale. Potremo chiederci come possano essere risolti questi due problemi più piccoli. La nostra risposta è che non sembra esserci nulla di inerentemente diverso nella risoluzione di ognuno di questi problemi rispetto agli altri due di una dimensione inferiore. È in questo modo che la ricorsione può essere inserita nel meccanismo.

Una volta progredito fin qui, se sono rimaste delle incertezze, ad esempio sul fatto che il meccanismo risolva o meno il problema, possiamo ricorrere ad un diagramma. È infatti sempre buona regola durante lo sviluppo di algoritmi ricorsivi, disegnare un diagramma per spiegare il meccanismo. Esaminando gli esempi precedenti, vediamo

Fig. 8.11.
Trasferimento dal
secondo disco più
grande a finalpole.



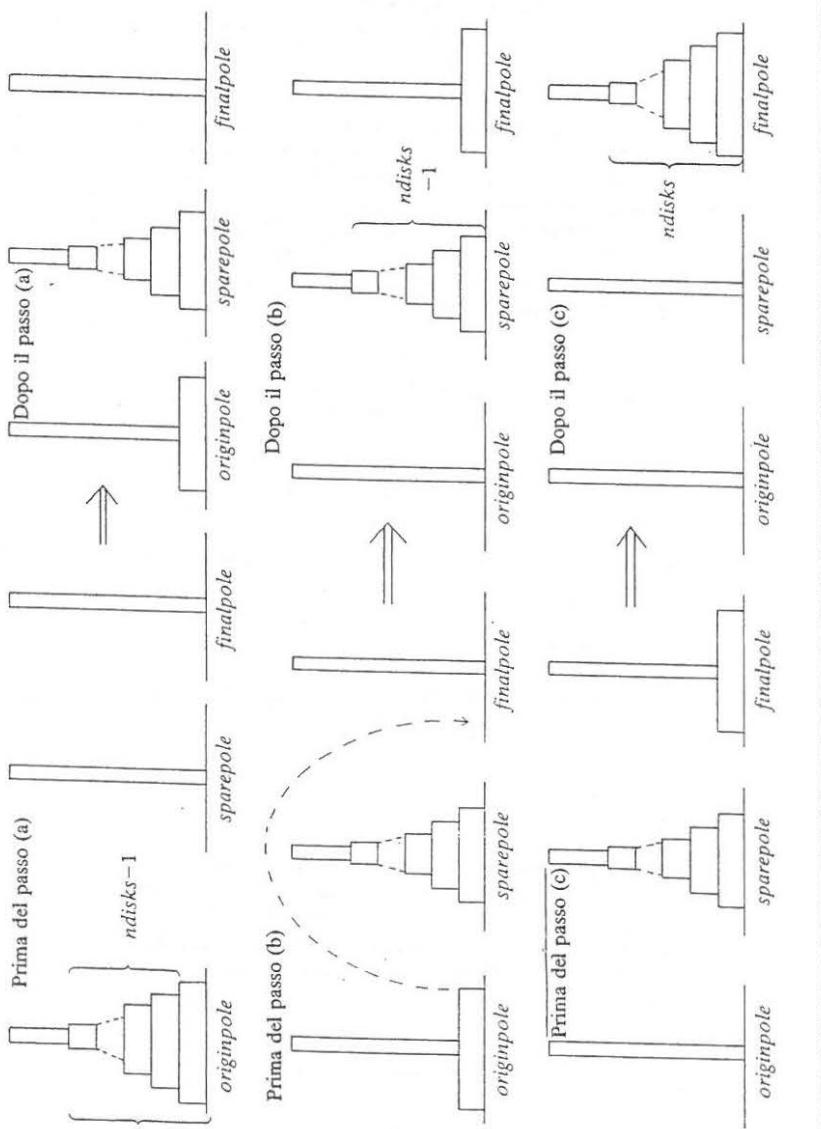


Fig. 8.12.
 (a) Trasferimento di $n_{disks}-1$ da originpole a sparepole;
 (b) Trasferimento del disco rimanente da originpole a finalpole;
 (c) Trasferimento di $n_{disks}-1$ da sparepole a finalpole.

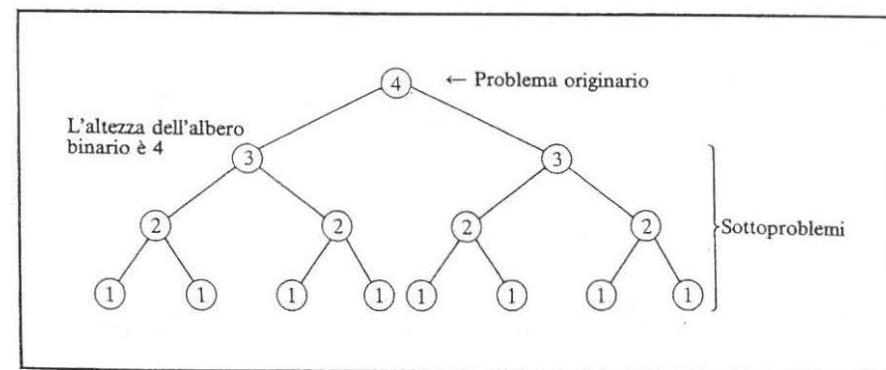
che un albero binario è un modello che descrive naturalmente la nostra procedura ricorsiva: una volta ancora stiamo trattando la ricorsione binaria. Per la risoluzione del problema di lunghezza 4, consideriamo l'albero mostrato in Fig.8.13. Questo problema si divide in due sottoproblemi di grandezza 3, che a loro volta si dividono in due problemi di grandezza 2, e così via.

La radice dell'albero corrisponde al problema originale che deve essere risolto. Possiamo immaginare che venga attuato un trasferimento ad ogni nodo. Per un dato nodo, il *sottoalbero sinistro* rappresenta il trasferimento di tutti i dischi più piccoli sulla cima di esso. Il *sottoalbero destro* rappresenta il ritrasferimento di tutti i dischi più piccoli sulla cima del disco corrente *dopo* il suo trasferimento. (Ad esempio, il sottoalbero sinistro del nodo 4 rappresenta il meccanismo di trasferimento dei dischi 3,2,1 *sul* 4; il sottoalbero destro rappresenta il ritrasferimento degli *stessi* sopra il disco 4 *dopo* che questo è stato spostato su *finalpole*). Le *foglie* dell'albero corrispondono tutte a problemi unitari. Torneremo più avanti sul problema della terminazione.

Dalla precedente discussione vediamo che la ricorsione ci fornisce di un modo di *posticipare* la soluzione di un problema di grandi dimensioni finché non siano stati risolti problemi più piccoli e più semplici. Ad ogni stadio della soluzione, il semplice passo del trasferimento costruisce la soluzione verso l'obiettivo finale. La rete del passaggio dei parametri assicura che ogni problema minore è appropriatamente riferito al problema più grande che deve essere risolto e, per ultimo, al problema originale che ci siamo proposti di risolvere.

Il diagramma ad albero binario ci informa esattamente sull'*ordine* in cui vengono fatti i trasferimenti come dell'ordine in cui sono *proposti*. L'ordine in cui sono fatti i trasferimenti è esattamente lo stesso di quello in cui vengono stampati i nodi nella visita in ordine *simmetrico* di un albero binario (ved. algoritmo 8.1). È infatti più semplice pensare al problema delle Torri di Hanoi come una visita in ordine simmetrico di un albero binario. Una volta definiti tutti i dettagli, i meccanismi di fondo sono gli stessi.

La considerazione del modello ad albero per questo meccanismo ci ha fornito una migliore osservazione della natura del problema ed anche un indizio su come poter trattare la terminazione. Il compito che



rimane è raffinare il meccanismo ricorsivo proposto ed includere le condizioni di terminazione.

Nella costruzione delle chiamate della procedura ricorsiva, è molto importante l'ordine in cui vengono specificati i parametri che individuano i tre pali, poiché all'interno dei sottoproblemi abbiamo visto che *originpole*, *sparepole* e *finalpole* giocano ruoli diversi rispetto a quelli definiti dal problema originale.

Cominciamo dalla definizione delle chiamate della procedura originale. Se *towershanoi* è il nome della procedura ed *ndisks* è il numero di dischi da trasferire, allora la chiamata può assumere la forma:

towershanoi(ndisks, originpole, sparepole, finalpole)

Possiamo ora stabilire le chiamate delle procedure per i sottoproblemi, riferite alla chiamata della procedura originale.

La prima chiamata ricorsiva nel corpo della procedura comporta il trasferimento di (*ndisks* — 1) dischi da *originpole* a *sparepole*. In questo problema è stato scambiato il ruolo tra *finalpole* e *sparepole*.

Considerando questo fatto, la chiamata della procedura per il passo (a) sarà:

towershanoi(ndisks — 1, originpole, finalpole, sparepole)

Il passo (b), che comporta il trasferimento di un solo disco dal corrente *originpole* al corrente *finalpole* può essere trattato come una chiamata ad una procedura *transferdisk*s che riceve in ingresso due parametri, *originpole* e *finalpole*. La chiamata può essere:

transferdisk(originpole, finalpole)

Rimanderemo la discussione di questa procedura finché non sarà stata considerata la seconda chiamata ricorsiva di *Towershanoi*. Tornando ai diagrammi ed alla descrizione del passo (c), osserviamo che ciò che resta da fare per risolvere il problema di dimensione *ndisks* è trasferire (*ndisks* — 1) dischi da *sparepole* a *finalpole*. Questa volta sono scambiati i ruoli di *originpole* e *sparepole*. La chiamata di procedura avrà quindi la forma:

towershanoi(ndisks-1, sparepole, originpole, finalpole)

La procedura *transferdisk* ha due parametri di ingresso, *originpole* e *finalpole* che possono assumere entrambi tre possibili etichette. Il modo più semplice per definire queste etichette è dichiarare un tipo *pole*, ad esempio:

type pole = (origin, spare, final)

La procedura *transferdisk* consisterà allora di due sole istruzioni, una per identificare e stampare l'etichetta *originpole* e l'altra per identifi-

care e stampare l'etichetta *finalpole*. I costrutti possono avere la forma:

```
case originpole of
  origin : write('origin-pole to');
  spare : write('spare-pole to');
  :

```

L'unica considerazione rimasta riguarda la terminazione. Abbiamo visto in precedenza, nella descrizione dei tre modelli per un problema, che la ricorsione procede fino al raggiungimento di un problema di dimensione unitaria che comporta un trasferimento unitario. Se essa andasse oltre, implicherebbe la risoluzione di un problema di zero dischi. Possiamo quindi usare la ricorsione fino all'occorrenza di un problema di lunghezza zero. Il corpo della procedura ricorsiva può essere controllato da un'istruzione del tipo:

```
if ndisks>0 then
  "procedi con la ricorsione"

```

L'implementazione in Pascal può quindi essere:

```
procedure towershanoi(ndisks:integer; originpole, sparepole, finalpole:pole);
begin
  if ndisks>0 then
    begin
      towershanoi(ndisks-1, originpole, finalpole, sparepole);
      transferdisk(originpole, finalpole);
      towershanoi(ndisks-1, sparepole, originpole, finalpole)
    end
  end

```

Quando il problema diventa unitario, notiamo che ci sono due chiamate interne di *towershanoi* senza risultato. Dato che qualsiasi problema delle Torri di Hanoi comporterà un certo numero di sottoproblemi unitari, sarebbe una strategia migliore arrestare la ricorsione un passo prima. Per fare questo, il controllo dovrà essere cambiato in:

```
if ndisks>1 then
  "procedi con la ricorsione"

```

Con questo cambiamento, il trasferimento diretto per il problema unitario dovrà essere trattato separatamente, cioè:

```

if ndisks>1 then
  "procedi con la ricorsione"
else
  transferdisks(originpole, finalpole)

```

Questa modifica elimina le chiamate ridondanti per problemi unari. Può essere ora data una completa descrizione dell'algoritmo.

Descrizione dell'algoritmo

(1) Descrizione della procedura Torri di Hanoi

1. Definisci il numero di dischi da trasferire, *originpole*, *sparepole*, *finalpole*.
2. Se il problema corrente comporta il trasferimento di più di un disco, allora
 - (a) chiama ricorsivamente la funzione per il trasferimento di tutti i dischi tranne quello finale dal corrente *originpole* a *sparepole*,
 - (b) trasferisci il disco finale da *originpole* a *finalpole*,
 - (c) chiama ricorsivamente la funzione per trasferire tutti i dischi da *sparepole* a *finalpole*
 altrimenti
 - (a') trasferisci il disco corrente da *originpole* a *finalpole*.

(2) Procedura di trasferimento dei dischi

1. Definisci le etichette correnti di *originpole* e *finalpole*.
2. Determina l'identità della corrente etichetta di *originpole*.
3. Stampa l'etichetta per il corrente *originpole*.
4. Determina l'identità della corrente etichetta di *finalpole*.
5. Stampa l'etichetta per il corrente *finalpole* indicando che è stato fatto il trasferimento a questo palo.

Implementazione in Pascal

```

procedure towers(ndisks: integer);
type pole = (origin,spare,final);
procedure transferdisk(originpole, finalpole: pole);
begin {simulate transfer of disk from current originpole to current
finalpole}
  case originpole of
    origin: writeln('origin-pole to ');
    spare: writeln('spare-pole to ');
    final: writeln('final-pole to ')
  end;

```

```

  case finalpole of
    origin: writeln('origin-pole');
    spare: writeln('spare-pole');
    final: writeln('final-pole')
  end;
end;

procedure towershanoi(ndisks: integer; originpole, sparepole,
finalpole: pole);
begin {solves the towers of hanoi for ndisks disks}
  if ndisks > 1 then
    begin {solve current problem by first solving next smallest
problem}
      towershanoi(ndisks - 1, originpole, finalpole, sparepole);
      {assert: (ndisks - 1) transferred from originpole to sparepole
according to rules}
      transferdisk(originpole, finalpole);
      {assert: disk transferred from originpole to finalpole}
      towershanoi(ndisks - 1, sparepole, originpole, finalpole);
      {assert: (ndisks - 1) transferred from sparepole to finalpole}
    end
  else
    transferdisk(originpole, finalpole);
    {assert: disk transferred from originpole to finalpole}
  end;
begin {towers}
  towershanoi(ndisks,origin,spare,final)
end

```

Note di progetto

1. L'esecuzione della procedura *towershanoi* può essere misurata in termini del numero di trasferimenti necessari per spostare tutti i dischi da *originpole* a *finalpole*. Il modo più semplice per contare questi trasferimenti è riferirsi al modello ad albero dell'algoritmo. Il numero di trasferimenti necessari per risolvere un dato problema corrisponde al numero di nodi nell'appropriato albero binario ordinato. Un problema con n dischi corrisponde ad un albero di altezza n (dove n è il numero dei livelli dell'albero). Un albero di altezza n ha $(2^n - 1)$ nodi. Ne segue che il numero di trasferimenti richiesti è una funzione esponenziale del numero di dischi da trasferire. È interessante tornare al compito portato a termine dai monaci. Il loro lavoro era trasferire 64 dischi. Il numero di trasferimenti per loro è $(2^{64} - 1)$, ovvero 18'446'774'073'709'551'615, compito che sicuramente richiede molto tempo ad ogni livello di progresso.
2. La caratterizzazione del funzionamento della procedura *towershanoi* può essere fatta in termini molto simili a quelli usati per descrivere l'algoritmo di visita in ordine simmetrico di un albero binario (algoritmo 8.1) perché anch'essa fa uso della ricorsione binaria. L'unica

reale differenza è che anziché stampare i nomi di ogni nodo, come viene fatto per la visita simmetrica, viene effettuato il trasferimento di un disco. Il lettore è quindi rimandato alle note di progetto dell'algoritmo 8.1.

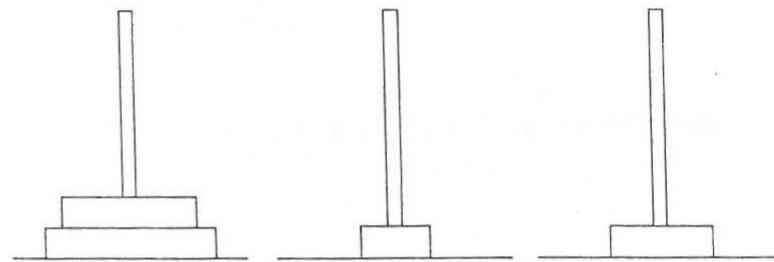
3. Nello sviluppo di questo algoritmo, abbiamo visto come cercando parziali soluzioni di un problema si possa arrivare ad una semplice e potente soluzione generale. È anche utile guardare a cosa potrebbe costituire la più piccola quantità di progressi positivi nella risoluzione di un problema.
4. Nel risolvere questo problema, lo abbiamo scomposto in due problemi più piccoli che sono stati risolti ricorsivamente e ricomposti per fornire la soluzione del problema originale. Questa strategia, che è conosciuta come principio di scomposizione, è largamente usata nella risoluzione di problemi combinatori.
5. Un altro modo di vedere questo problema è affermare che la ricorsione ci ha permesso di *partecipare* la soluzione di un problema di grandi dimensioni finché non fosse stato risolto un gruppo di problemi più piccoli.
6. Abbiamo visto come sia spesso utile costruire un diagramma per modellare il funzionamento dell'algoritmo. In questo esempio ciò ci ha largamente informato sul meccanismo ed anche sulla terminazione dell'algoritmo.
7. Con la ricorsione,abbiamo di solito diverse possibilità di terminazione. Lasciamo come esercizio al lettore il compito di confrontare le due implementazioni Pascal fornite in termini di chiamate effettuate da esse.
8. Questo esempio ci ha aiutati a scoprire le relazioni implicite tra l'algoritmo delle Torri di Hanoi e quello della visita in ordine simmetrico di un albero binario.
9. Questo algoritmo non ha, di per sé stesso, altra applicazione pratica se non quella di misurare la vita dell'universo. Ci fornisce, tuttavia, un'importante illustrazione di come possa essere usata la ricorsione per rendere facile da risolvere un problema apparentemente difficile. (Accade anche che vi sia una semplice soluzione iterativa. Ved. P.Buneman e L.Levy, "The Towers of Hanoi Problem", Inf. Proc. Letts. 10, 243 (1980)).
10. La profondità della ricorsione è al massimo n per la soluzione di un problema che comporta il trasferimento di n dischi.
11. Occorre notare che ogni mossa *alternativa* consiste nel trasferimento del disco più piccolo da un palo all'altro. Se immaginiamo i tre pali disposti a cerchio ed i dischi numerati dal più piccolo al più grande, come 1,2,3,..., n , allora tutti i dischi di numero dispari ruotano in una direzione e quelli di numero pari ruotano nell'altra. Questa affermazione può formare le basi per una soluzione iterativa.

Problemi supplementari

- 8.3.1 Implementare una versione della procedura *towershanoi* che utilizzi tre array ciascuno di n elementi per risolvere il problema di n dischi. Se i dischi sono etichettati da 1 ad n , con n il disco

avente diametro maggiore, il contenuto dei tre array dopo ogni trasferimento rifletterà la corrispondente situazione dei tre pali. Includere una stampa che rifletta lo stato dei pali dopo ogni trasferimento.

- 8.3.2 Inserire nell'algoritmo una routine che disegni un rettangolo (o un disco) in modo che i dischi ed i pali siano appropriatamente disegnati dopo ogni trasferimento. Un tipico disegno può essere:



- 8.3.3 Progettare un algoritmo per la risoluzione del problema delle Torri di Hanoi.
 8.3.3 Progettare un algoritmo per la risoluzione del problema delle Torri di Hanoi che non utilizzi la ricorsione (ved. note 9 e 11).
 8.3.4 Progettare un algoritmo per la risoluzione del problema delle Torri di Hanoi in cui ci siano n dischi e $K = \lceil \sqrt{2n} \rceil$ pali. Con questa configurazione l'algoritmo dovrebbe avere un funzionamento *lineare* anziché esponenziale. Per risolvere questo problema non è necessaria la ricorsione.

ALGORITMO 8.4 GENERAZIONE DI CAMPIONI

Problema

Progettare un algoritmo che generi tutte le permutazioni dei primi n interi, presi r alla volta e permetta ripetizioni senza restrizioni. Questo processo è conosciuto di solito col nome di "generazione di campioni". I valori di n ed r sono maggiori di zero.

Sviluppo dell'algoritmo

Il problema di generare permutazioni senza limite di ripetizioni è probabilmente il più semplice tra i problemi combinatori. Il nostro interesse non sorge per l'implicazione del calcolo combinatorio, bensì perché questo problema non ha una semplice ed ovvia soluzione iterativa. Un esempio chiarirà il concetto. Supponiamo che n siano i primi 5 interi che debbano essere combinati tre alla volta ($r = 3$). In Fig.8.14 sono elencate alcune tra le prime permutazioni e l'ultima di questo insieme.

Come caso a sé stante, val la pena di notare che se gli interi sono limitati a due (0 e 1), allora per un dato r l'insieme delle permutazioni con ripetizioni limitate corrisponderà alle rappresentazioni binarie ad r bit dei primi 2^r numeri interi (incluso lo 0).

Fig. 8.14.
Permutazione senza
limite di ripetizione
per $n = 5$
ed $r = 3$.

colonna 1	colonna 2	colonna 3
1	1	1
1	1	2
1	1	3
1	1	4
1	1	5
1	2	1
1	2	2
1	2	3
:	:	:
5	5	5

$n^r = 5^3 = 125$
differenti permutazioni
in questo set.

Da un accurato studio della Fig.8.14 vediamo che ognuna delle 3 (r) colonne può assumere *tutti* i 5 (n) valori. Inoltre, poiché i valori in terza colonna ripetono l'intero intervallo per ogni cambiamento in seconda colonna, possiamo concludere che questo *particolare* problema può essere risolto con tre cicli annidati del tipo:

```

:
for i := 1 to 5 do
    for j := 1 to 5 do
        for k := 1 to 5 do
            :
            "stampa i valori di i, j, k"

```

Un controllo di questo meccanismo conferma che esso risolverà il nostro particolare problema; tuttavia, se volessimo generare un insieme di permutazioni senza limite di ripetizioni di dimensioni diverse

(cioè con diversi valori di r), saremmo nei guai. È facile vedere che il numero di cicli annidati deve essere uguale al valore di r . Ciò crea un problema se cerchiamo di costruire un algoritmo che funzioni per tutti i valori di r . La ragione del problema è che i nostri linguaggi di programmazione non ci permettono di scrivere facilmente programmi con un numero variabile di cicli annidati. Non è molto promettente eseguire con semplicità un ciclo che ne racchiude un altro. Se vogliamo fare qualche progresso con il problema, dovremo cercare qualche altro meccanismo per generare questo insieme di permutazioni. In qualche modo dovremo porre la nostra attenzione sul calcolo delle variabili del ciclo.

Prima di iniziare un approccio alternativo, esaminiamo più accuratamente la soluzione proposta per il problema discusso in precedenza: essa suggerisce alcune cose. La prima è che tutti i cicli annidati sembrano avere lo stesso ruolo. Tutto ciò che deve fare un ciclo è generare un indice nell'intervallo da 1 a 5 (in generale l'indice sarebbe da 1 a n). In secondo luogo, è possibile un'ulteriore permutazione solo dopo che è stato fatto un cambiamento di indice nel ciclo *più interno*. La prima permutazione, ad esempio, è possibile solo dopo che ci si è spostati dal ciclo più esterno a quello più interno. Ogni ciclo fornisce un *solo* intero alla prima permutazione ed anche a tutte le altre. Vediamo dunque che la soluzione di un problema di grandi dimensioni può essere costruita risolvendo e combinando un insieme di soluzioni di altri problemi più piccoli ognuno dei quali può essere risolto in modo simile.

Per cercare di fare progressi con questo problema, concentriamoci per prima cosa sulla risoluzione del problema *più piccolo*, quello di generare l'insieme di valori della prima colonna. Questo problema può essere risolto semplicemente da un ciclo che si estende da 1 a n . Il primo passo verso una soluzione del problema generale può quindi essere del tipo:

```

for i := 1 to n do
begin
    "risvoli nello stesso modo i rimanenti problemi più piccoli"
end

```

Per generare gli indici della seconda colonna, occorre ancora un ciclo da 1 a n . Per il nostro problema generale, esso deve essere esteso a tutto l'intervallo da 1 a n , per *ogni* valore del primo ciclo. Per ogni iterazione del secondo ciclo dobbiamo quindi risolvere un problema essenzialmente uguale a quello per il primo ciclo. È a questo punto che dovremo riconoscere che il processo descritto è in realtà un processo ricorsivo. Se il processo originale chiama se stesso una volta ad ogni iterazione, noi abbiamo risolto il problema di generare una permutazione per due colonne. Analogamente, se ognuna delle chiamate secondarie introduce un altro insieme di chiamate, avremo risolto il problema per tre colonne. È semplice vedere che questo meccanismo ci permetterà di trattare il caso di n colonne. I passi centrali di questo algoritmo potrebbero quindi essere:

Finché tutti gli indici della colonna corrente non sono stati generati

- (a) genera l'indice successivo per la colonna corrente,
- (b) ripeti lo stesso processo di generazione degli indici per la colonna successiva.

Questi passi sembrano incorporare la maggior parte di quello di cui abbiamo bisogno per risolvere ricorsivamente il problema. Il nostro meccanismo, all'inizio, non contiene dettagli relativi alla terminazione ed all'uscita delle permutazioni.

Nel meccanismo originale, l'output era determinato semplicemente dalla stampa degli indici del ciclo corrente, dopo ogni cambiamento nel ciclo più interno. Nel nostro algoritmo ricorsivo un simile approccio non sembra possibile a causa dell'influenza ristretta delle variabili di ciclo. Quale potrebbe essere, quindi, un'alternativa che ci permetta di passare le informazioni sullo stato della computazione ai valori delle altre colonne? La nostra unica possibilità è memorizzare i valori della colonna in un array. Sorge ora la domanda su come possa essere usato l'array per memorizzare i valori della colonna. La nostra risposta sarà probabilmente che l'unica soluzione pratica è quella di associare ogni elemento dell'array ad una colonna dell'insieme della permutazione. Così ora, anziché generare indici di ciclo, dovremo aggiornare appropriatamente gli elementi dell'array. Per assicurare che l'elemento dell'array associato alla k -esima colonna $column[k]$ assuma l'insieme adatto di valori, possiamo inserire nel ciclo principale del nostro meccanismo la seguente istruzione:

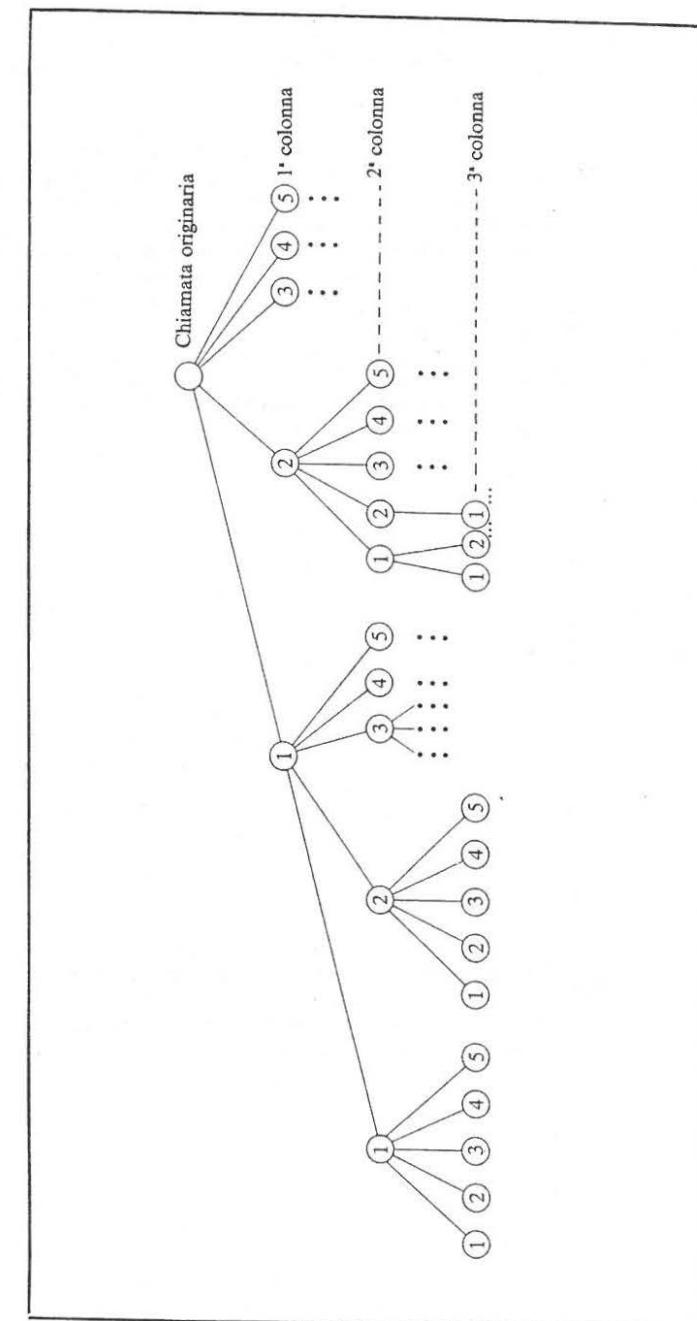
```
column[k] := column[k] + 1
```

Prima di entrare nel ciclo $column[k]$ dovrà essere inizializzata a zero. Se chiamiamo *sample* la procedura ricorsiva, ed aggiungiamo i raffinamenti suggeriti precedentemente avremo:

```
procedure sample( ? )
:
begin
  column[k] := 0;
  while column[k]<n do
    begin
      column[k] := column[k]+1;
      sample( ? )
    :
  end
end
```

Ci rimane ora il compito di decidere quali parametri debbano essere passati alla nostra procedura sia dall'esterno che nella chiamata ricorsiva. Dobbiamo inoltre stabilire il meccanismo per la terminazione ed il punto in cui sia pronta per la stampa una nuova permutazione. Uno studio del meccanismo creato suggerisce che k può essere considerato

Fig. 8.15.
Parte del
meccanismo ad
albero per il
problema tU_5



un parametro perché identifica la colonna corrente che sta per essere aggiornata. La variabile k può quindi essere inserita nella lista dei parametri della procedura. La chiamata ricorsiva interna di *sample* ha lo scopo di occuparsi della colonna *successiva*. In termini di k , questa è la $(k+1)$ -esima colonna; quindi $(k+1)$ può essere un parametro per la chiamata interna. I parametri n ed r non variano, cosicché possono essere inseriti nella procedura chiamante. Sarà inoltre più semplice memorizzare l'array nella procedura chiamante.

Per cercare di definire i dettagli che stabiliscono quando sia possibile una nuova permutazione e quando debba essere forzata la terminazione, torniamo ancora all'algoritmo a cicli annidati. Vediamo che nella soluzione iterativa è possibile una nuova permutazione soltanto dopo che è stato aggiornato il ciclo più a destra (più interno) o l' r -esima colonna della permutazione. Ci possiamo ora chiedere quale sia la soluzione equivalente per il nostro algoritmo ricorsivo. Esso può aggiornare l' r -esima colonna quando:

$$k = r$$

Ciò suggerisce che quando k è uguale ad r saremo in grado di fornire in uscita i primi r elementi dell'array *column* come permutazione corrente. Se abbiamo qualche incertezza, possiamo provare a tracciare un diagramma che cerchi catturare il meccanismo abbozzato. Per questo ci possiamo riferire al problema rU_3 discusso precedentemente (U si riferisce alla permutazione senza limite di ripetizioni). La prima chiamata della procedura per questo problema genera 5 valori della prima colonna. Associato ad ognuno di questi valori esiste un insieme di valori della seconda colonna; e, a sua volta, associato ad ogni valore della seconda colonna c'è un insieme dei valori della terza. Ciò suggerisce l'uso di una struttura ad albero. Parte del meccanismo ad albero del problema rU_3 è mostrato in Fig.8.15. Ognuna delle permutazioni senza limitazione di ripetizioni può essere trovata tracciando un percorso dalla radice attraverso l'albero fino ai nodi foglia. Tale diagramma conferma che, quando arriviamo all' r -esima colonna (in questo caso alla terza colonna o al nodo foglia), sarà possibile una nuova permutazione. Si può immaginare che il nostro algoritmo compia una visita in ordine *simmetrico* dei nodi di quest'albero (ved., algoritmo 8.1).

La nostra discussione suggerisce che tutte le volte in cui $k=r$, saremo in grado di stampare la permutazione successiva. Possiamo quindi inserire le seguenti istruzioni nel ciclo della procedura dopo l'aggiornamento del valore di *column*[k]:

```
if k = r then
begin
  for j := 1 to r do
    write(column[j]);
  writeln
end
```

Fig. 8.16.
Traccia della
generazione
dell'insieme campione
per il problema rU_r

Ci rimane ora solo il problema della terminazione. In generale, dobbiamo solo generare permutazioni di valori per r colonne. Il nostro algoritmo ricorsivo è strutturato in modo tale che, mentre genera i valori della colonna corrente, inizia la chiamata della procedura che

$\text{—sample } (k = 1 \boxed{0 \ 0 \ 0})$	Prima chiamata — livello 1
$(k = 1 \boxed{1 \ 0 \ 0})$	
$\text{—sample } (k = 2 \boxed{1 \ 0 \ 0})$	Prima chiamata — livello 2
$(k = 2 \boxed{1 \ 1 \ 0})$	
$\text{—sample } (k = 3 \boxed{1 \ 1 \ 0})$	Prima chiamata — livello 3
$(k = 3 \boxed{1 \ 1 \ 1})$	1* permutazione stampata
$(k = 3 \boxed{1 \ 1 \ 2})$	2* permutazione stampata
$(k = 3 \boxed{1 \ 1 \ 3})$	3* permutazione stampata
$(k = 3 \boxed{1 \ 1 \ 4})$	4* permutazione stampata
$(k = 3 \boxed{1 \ 1 \ 5})$	5* permutazione stampata
$\text{—sample } (k = 2 \boxed{1 \ 1 \ 5})$	Seconda chiamata — livello 2
$(k = 2 \boxed{1 \ 2 \ 5})$	
$\text{sample } (k = 3 \boxed{1 \ 2 \ 5})$	Seconda chiamata — livello 3
$(k = 3 \boxed{1 \ 2 \ 1})$	6* permutazione stampata
$(k = 3 \boxed{1 \ 2 \ 2})$	7* permutazione stampata
$(k = 3 \boxed{1 \ 2 \ 3})$	8* permutazione stampata
$(k = 3 \boxed{1 \ 2 \ 4})$	9* permutazione stampata
$(k = 3 \boxed{1 \ 2 \ 5})$	10* permutazione stampata
$\text{—sample } (k = 2 \boxed{1 \ 2 \ 5})$	Terza chiamata — livello 2
$(k = 2 \boxed{1 \ 3 \ 5})$	
$\vdots \quad \vdots \quad \vdots \quad \vdots$	\vdots
$\vdots \quad \vdots$	

genera i valori della colonna successiva. Ne segue che, quando ci troviamo in r -esima colonna (cioè quando $k = r$), non ci sono altre colonne da considerare e quindi non ci saranno ulteriori chiamate a livelli più profondi di ricorsione. Questo si collega alla stampa delle permutazioni tutte le volte che $k = r$. In questo caso, infatti, invece di effettuare un'ulteriore chiamata ricorsiva al livello successivo, possiamo semplicemente mandare in uscita la permutazione corrente.

Per cercare di comprendere meglio il nostro algoritmo possiamo esaminare i primi cicli del problema rU_n . È utile collegare questi passi al meccanismo ad albero (ved. figg. 8.15 e 8.16).

La procedura può iniziare con $k = 1$ e tutti gli elementi dell'array $column$ devono essere azzerati. Qui di seguito è fornita una completa descrizione dell'algoritmo.

Descrizione dell'algoritmo

1. Definisci la dimensione positiva e diversa da zero dell'insieme, quella della permutazione e l'array per memorizzare le permutazioni. Definisci inoltre l'indice della colonna corrente dell'insieme delle permutazioni.
2. Inizializza a zero il valore corrente della colonna.
3. Finché l'indice corrente della colonna è minore della dimensione dell'insieme
 - (a) incrementa di uno il valore corrente della colonna,
 - (b) se l'indice corrente della colonna non è alla colonna finale, allora
 - (b.1) chiama ricorsivamente la procedura che genera tutti i valori della colonna successiva associati al valore corrente della colonna
 - altrimenti
 - (b'.1) è disponibile una nuova permutazione nell'array, quindi stampala.

Implementazione in Pascal

```

procedure sampleset(n,r: integer);
var column: array[1..20] of integer;
    k:integer;

procedure sample(k: integer);
var j {index for printing permutations}: integer;
begin {recursively generates all permutations with unrestricted
repetitions for the first n integers taken r at a time. The permutations
are generated in lexical order}
  {assert: n>0 ∧ r>0 ∧ r=<n ∧ 1=<k=<r}
  column[k] := 0;

```

```

{invariant: after ith iteration column[1..k-1]=c1, c2, ..
column[k]=i ∧ columns column[k+1..r] have assumed all possible
values [1..n] with unrestricted repetitions and been printed when
k=r ∧ i=<n ∧ all column values in [1..n] ∧ 1=<k=<r}
while column[k]<n do
  begin {generate next value for current column}
    column[k] := column[k]+1;
    if k<r then
      {generate subsequent column values associated with current
      column value}
      sample(k+1)
    else
      begin {next permutation available - write it out}
        for j := 1 to r do
          write(column[j]);
        writeln
      end
    end
  {assert: all permutations beginning with c1, c2, ... , c(k-1) have
  been generated}
end;

begin {sampleset}
  k := 1;
  sample(k)
end

```

Note di progetto

1. Il numero di volte in cui in un ciclo viene eseguito il test condizionale ci dà una misura della complessità di questo algoritmo (è uguale al numero di nodi del nostro cammino ad albero). Nell'esempio, quando $n = 5$ ed $r = 3$, ci saranno 5 possibilità per la prima colonna. Associata ad ognuna di queste, ci saranno 5 possibilità per la seconda colonna, che diventano n^2 valori per le prime due colonne. In generale, quando ci sono r colonne, il test condizionale sarà eseguito n^r volte. L'algoritmo mostra quindi ad un andamento esponenziale. La profondità della ricorsione è pari ad r .
2. Per caratterizzare il funzionamento di questa procedura che comporta una ricorsione non lineare, consideriamo, per primo, il caso in cui viene chiamata $sample(k+1)$. Se supponiamo che questa chiamata non modifichi gli elementi delle prime k colonne e che genera tutte le permutazioni richieste per le colonne da $column[k+1]$ ad r , allora possiamo considerare dettagliatamente il caso in cui venga chiamata $sample(k)$. Quando la chiamata corrente di $sample$ non comporta nessuna chiamata ricorsiva interna, la nostra ipotesi deve essere corretta. Nel caso generale, quando viene chiamata $sample(k)$, ammettendo l'ipotesi associata alla chiamata di $sample(k+1)$, la presente chiamata lascerà intatti gli elementi in $column[1..k-1]$ perché ad essi non si fa riferimento nella procedura. La presente chiamata, inoltre, assicura che vengano prodotte

correttamente tutte le permutazioni di $column[k..n]$ poichè sono state considerate correttamente tutte le possibilità di $column[k]$. Dopo ogni scelta, viene effettuata una chiamata ricorsiva di $sample(k+1)$. Questo ragionamento pone le basi di una “prova formale induttiva” della procedura. Ogni chiamata ricorsiva di $sample$ si risolve in uno spostamento verso una colonna più alta e termina in tutti i casi per $k=r$. Possiamo quindi affermare liberamente che l’algoritmo termina.

3. In questo algoritmo bisogna notare come la soluzione di un problema di grandi dimensioni sia costruita risolvendo un insieme di problemi più piccoli e correlati.
4. L’impraticabile soluzione iterativa di un problema specifico è stata utile nel guidarci alla soluzione ricorsiva generale del problema. Per comprendere meglio cosa sia necessario per una soluzione ricorsiva, è spesso utile, come passo preliminare, considerare una soluzione iterativa impossibile. Da tali considerazioni è spesso possibile scoprire i dettagli sulla terminazione, i limiti del ciclo e la profondità della ricorsione richiesta.
5. Quando la soluzione di un problema sembra richiedere un numero variabile di cicli annidati, è in genere consigliabile vedere se è possibile risolvere il problema con l’uso della ricorsione non lineare. Usando l’approccio descritto, un insieme di r cicli annidati è stato sostituito da una procedura ricorsiva che si richiama r volte. In questo modo possono essere risolti molti problemi.
6. Nella procedura $sample$, abbiamo scelto di inserire solo la variabile k come parametro per ragioni di efficienza. Per trattare i dettagli dell’inizializzazione, $sample$ è stata inclusa in un’altra procedura. Ciò permette una chiamata di procedura più semplice da parte dell’utente. Questa pratica è stata adottata per i restanti algoritmi del capitolo che presentano lo stesso problema.

Applicazioni

Combinazioni, generazione di sequenze di numeri di una data base, simulazioni.

Problemi supplementari

- 8.4.1 Implementare un algoritmo ricorsivo che generi le permutazioni U , senza limite di ripetizioni, in ordine lessicale inverso.
- 8.4.2 Come accade in questo particolare problema, è possibile costruire una semplice soluzione iterativa se si adotta il meccanismo di convertire i numeri dalla base 10 alla base r . Cercare di implementare questa semplice soluzione del problema.
- 8.4.3 Un altro semplice (ma non così efficiente) modo di generare queste permutazioni in ordine lessicale è di iniziare con le permutazioni $(1,1,1\dots,1)$. La permutazione successiva in ogni caso è generata scandendo quella corrente da destra a sinistra finché si

incontra un elemento che *non* ha raggiunto il suo valore massimo. Questo elemento viene incrementato di un’unità ed a tutti gli elementi alla sua destra vengono riassegnati i valori più bassi permessi, quindi il processo si ripete. Implementare questo algoritmo.

- 8.4.4 Progettare un algoritmo che riceva in ingresso una data permutazione dell’insieme U , e restituisca in uscita
 - (a) la permutazione che la precede direttamente,
 - (b) la permutazione che la segue direttamente in ordine lessicale.
- 8.4.5 Modificare l’algoritmo di generazione di permutazioni in modo tale che stampi solo quelle che sommate danno come risultato 5. Non deve essere usato un ciclo separato per fare la somma.
- 8.4.6 Quando il nostro algoritmo è usato per generare i primi 2^n interi binari, vediamo che la maggior parte del tempo, più che per una cifra binaria, deve essere speso nel passaggio da un intero al successivo. Un altro approccio per generare questo insieme (non in ordine lessicale) è di ordinare la computazione in modo che non debba essere cambiata più di una cifra nel passaggio da una configurazione ad un’altra. Per questo la cifra che deve essere cambiata per generare la k -esima rappresentazione binaria dalla $(k-1)$ -esima, è uguale ad uno più la potenza della più grande potenza di 2 che divide esattamente k . L’algoritmo deve iniziare con tutte le cifre binarie uguali a zero. Implementare questo algoritmo. Tali rappresentazioni sono chiamate codice gray.

ALGORITMO 8.5 GENERAZIONE DI COMBINAZIONI

Problema

Progettare un algoritmo che generi tutte le combinazioni dei primi n interi presi r alla volta, con n ed r maggiori o uguali ad 1.

Sviluppo dell’algoritmo

Ci sono molte occasioni in scienza dell’informazione e nelle applicazioni di Teoria della probabilità in cui vogliamo conoscere il numero di modi possibili (chiamati C) di scegliere un sottoinsieme di r oggetti

da un insieme di n oggetti. In altre parole, vogliamo identificare esplicitamente ognuna di queste combinazioni. Avremo a che fare con quest'ultimo problema. L'algoritmo che svilupperemo è utile nella risoluzione di un certo numero di altri problemi. Per semplificare il discorso, concentriamoci sullo sviluppo di un algoritmo che genererà tutte le combinazioni dei primi n interi presi r alla volta.

Per avere idea di cosa comporti ciò, consideriamo un esempio specifico, come quello di generare tutte le combinazioni di lunghezza 3 usando i primi 5 interi. Quando ci occupiamo di combinazioni (come opposte alle permutazioni) di oggetti, bisogna notare che non è importante l'ordine. Ad esempio, le sequenze $(1,2,3)$, $(2,1,3)$ e $(3,1,2)$ rappresentano tutte un'unica combinazione. Per evitare calcoli non necessari, sarà altamente desiderabile generare un'unica rappresentazione di ogni combinazione. Non è molto difficile mostrare che le 10 combinazioni dei primi 5 interi presi 3 alla volta sono:

1	1	1	1	1	1	2	2	2	3
2	2	2	3	3	4	3	3	4	4
3	4	5	4	5	5	4	5	5	5

Da un accurato studio di questo esempio possiamo concludere che l'insieme di combinazioni può essere generato usando tre cicli annidati. Il più esterno tratterà la prima colonna di interi, quello centrale la colonna centrale ed il più interno la colonna di destra. Possiamo trovare i limiti dei tre cicli osservando l'intero iniziale e quello finale di ogni colonna.

La parte centrale del nostro algoritmo di generazione di combinazioni di lunghezza 3 dei primi 5 interi può quindi essere del tipo:

```
for a := 1 to 3 do
  for b := 2 to 4 do
    for c := 3 to 5 do
      writeln(a,b,c);
```

Da un controllo di questo meccanismo troviamo che i limiti *inferiori* di ciclo non sono corretti (infatti una delle combinazioni generate sarebbe $(3,2,3)$). Dopo aver studiato ancor più accuratamente l'esempio, concludiamo che il ciclo b deve sempre iniziare ad un valore che supera di *un'unità* il corrente valore a . Analogamente, il ciclo c deve sempre iniziare con un valore superiore di *un'unità* rispetto al corrente valore b . Con questi cambiamenti troviamo:

```
for a := 1 to 3 do
  for b := a+1 to 4 do
    for c := b+1 to 5 do
      writeln(a,b,c);
```

Un controllo di questo meccanismo conferma che esso risolve il problema 3C_3 . Il meccanismo creato, sfortunatamente, risolve solo un

particolare problema. Ad esempio, per risolvere il problema 4C_3 , saranno necessari 4 cicli annidati (uno per ogni colonna) ed un diverso insieme di limiti di ciclo. L'implementazione di un programma con un numero *variabile* di cicli annidati sembrerebbe inattuabile, come abbiamo visto nell'algoritmo precedente. Possiamo quindi chiederci se esista un'altra via da percorrere.

Prima di cercare una diversa direzione, esaminiamo il meccanismo proposto; notiamo che viene scritta una combinazione solo dopo un cambiamento nel ciclo più interno. Per ogni valore di a le variabili b e c assumono un insieme di valori.

Questo meccanismo suggerisce che una soluzione per un problema di grandi dimensioni viene costruita risolvendo una successione di problemi più piccoli in modo che la soluzione di ognuno di essi dipenda dal suo predecessore. Nella ricerca di un algoritmo più pratico per la generazione di combinazioni, concentriamoci per prima cosa sul problema di dimensione minore, quello di generare gli elementi della prima colonna. Per prima cosa occorre stabilire l'intervallo dei valori della prima colonna. Dall'esempio precedente e da altri semplici esempi, possiamo stabilire che in generale la prima colonna può assumere $(n - r + 1)$ valori. Possiamo scrivere un ciclo per generare questi valori come prima. I cicli interni sono stati sostituiti da un'istruzione generale:

```
for a := 1 to n-r+1 do
  begin
    "risovi i rimanenti problemi più piccoli in modo simile"
  end.
```

La domanda che ora sorge è come potremo risolvere "i rimanenti problemi più piccoli in modo simile". Per generare l'intervallo di valori della seconda colonna, sarà necessario un ciclo che generi interi nell'intervallo da 2 a $n - r + 2$. Come abbiamo scoperto nell'esempio 3C_3 , non è semplicemente questione di generare degli interi in questo intervallo. La chiave per risolvere il problema deve trovarsi nel modo in cui i numeri in seconda colonna sono collegati a quelli in prima. Studiando l'esempio precedente, vediamo che per ogni valore della prima colonna i numeri della seconda iniziano sempre con un valore superiore di *un'unità* rispetto a questo valore e li assumono tutti fino al limite superiore. Da ciò possiamo concludere che i valori in seconda colonna potrebbero essere generati da un ciclo simile a quello originario, ma col limite superiore incrementato di *un'unità* e quello inferiore che inizia sempre con il valore corrente della prima colonna "più uno". La combinazione dei risultati della prima e della seconda colonna ci permetterà di compiere un ulteriore passo verso la soluzione del problema generale. È questa analogia tra i problemi più piccoli che suggerisce l'utilità dell'uso della ricorsione.

Inizialmente abbiamo proposto che ogni nuovo ciclo annidato si sarebbe occupato della generazione di un'altra colonna di valori richiesti per l'insieme combinazione. Considerando il problema come ricorsivo, potremmo anticipare che una chiamata ricorsiva nel corpo del

ciclo principale raggiungerebbe lo stesso effetto. Se la procedura originale è chiamata *combinations*, allora potremo avere:

```
procedure combinations(lowervalue, upperlimit)
begin
  for a := lowervalue to upperlimit do
    combinations(a+1, upperlimit+1)
  :
end.
```

La struttura principale della procedura ricorsiva sembra incorporare la maggior parte di ciò di cui abbiamo bisogno per risolvere ricorsivamente il problema. Non ci indirizza, tuttavia, verso i problemi della terminazione e dell'output delle combinazioni. Nel meccanismo originale abbiamo usato gli indici di ciclo che ci hanno fornito le combinazioni cercate. Nell'algoritmo ricorsivo ciò non sembrerebbe possibile a causa degli effetti locali delle variabili di ciclo. La nostra unica possibilità è quindi memorizzare i valori di colonna in un array come abbiamo fatto nell'algoritmo precedente. A questi valori si può allora accedere ad *ogni* chiamata ricorsiva. Possiamo allora associare ad ogni elemento dell'array *una* colonna dell'insieme delle combinazioni. Se adottiamo questa strategia, il nostro compito diviene allora quello di assicurare che ad ogni passo della computazione, tutti gli elementi dell'array siano assegnati ed aggiornati appropriatamente. A patto che questo avvenga correttamente, non appena sarà stata generata una nuova combinazione, sarà solo questione di stampare gli elementi dell'array.

A questo punto, come nell'introduzione di un array, dovremo esprimere il nostro algoritmo in termini dei valori di n ed r usati per definire l'insieme combinazione che stiamo cercando.

Da una nuova stima della situazione, il limite inferiore del ciclo sarà ora determinato dal *valore corrente* dell'elemento dell'array corrispondente alla *colonna precedente*. Se l'array *column* è usato per memorizzare i valori colonna dell'insieme, allora il valore iniziale della k -esima colonna nella situazione corrente sarà:

$$\text{column}[k] := \text{column}[k-1] + 1$$

Dopo aver stabilito e messo in relazione il limite inferiore di $\text{column}[k]$ con il corrente valore della colonna precedente, siamo liberi di incrementare $\text{column}[k]$ fino al suo limite superiore. La precedente discussione ha mostrato che il limite superiore della seconda colonna sarà $n - r + 2$, quello della terza $n - r + 3$ e così via. In generale ci aspetteremo che il limite superiore della k -esima colonna sia:

$$n - r + k$$

Abbiamo ora definito i limiti superiore ed inferiore per la k -esima colonna. Il limite inferiore è stato appropriatamente riferito al valore corrente della precedente colonna ed il limite superiore è stato riferito a

n ed r . Il nostro prossimo compito è cercare di inserire queste idee nella struttura ricorsiva di base.

Il ruolo del meccanismo originario a cicli era assicurare che il valore della colonna corrente (cioè l'indice di ciclo), assumesse l'appropriato intervallo di valori. Nell'attuale contesto ciò si traduce nel fatto che l'elemento corrente dell'array assuma un appropriato insieme di valori. Per incrementare il valore della k -esima colonna possiamo usare:

$$\text{column}[k] := \text{column}[k] + 1$$

Usando questa idea e mettendo in relazione il limite inferiore con la colonna precedente, il ciclo scritto per il caso generale avrà la forma:

```
procedure combinations( ? )
:
begin
  column[k] := column[k-1];
  while column[k] < n - r + k do
    begin
      column[k] := column[k] + 1;
      combinations( ? )
    :
  end
end
```

Dobbiamo ora decidere quali parametri debbano essere passati alla nostra procedura. Studiando il testo appare chiaro che la variabile k deve giocare una parte importante in quanto identifica la colonna su cui stiamo attualmente lavorando, k , e la prossima su cui opereremo, cioè la colonna $k + 1$. Ciò suggerisce che la chiamata della procedura originale debba essere fatta in termini di k e la chiamata della procedura ricorsiva implichi $k + 1$. Le altre variabili, n , r e l'array *column*, possono essere riferiti alla procedura chiamante.

Dopo aver stabilito la maggior parte dei dettagli degli aspetti ricorsivi del meccanismo, ci restano i problemi della terminazione e dell'output delle combinazioni.

Usando una linea di ragionamento simile a quella dell'algoritmo precedente, vediamo che il nostro algoritmo ricorsivo compie solo un aggiornamento dell' r -esima colonna quando:

$$k = r$$

Ciò suggerisce che quando k è uguale ad r noi potremo produrre in uscita i primi r elementi dell'array *column* come combinazione corrente. Con riferimento al problema 3C , discusso in precedenza, la prima chiamata alla procedura per questo problema genera *tre* valori della prima colonna. Associato ad *ognuno* di questi tre valori c'è un insieme di valori della seconda colonna e, a sua volta, associato ad ogni valore della seconda colonna c'è un insieme di valori della terza colonna.

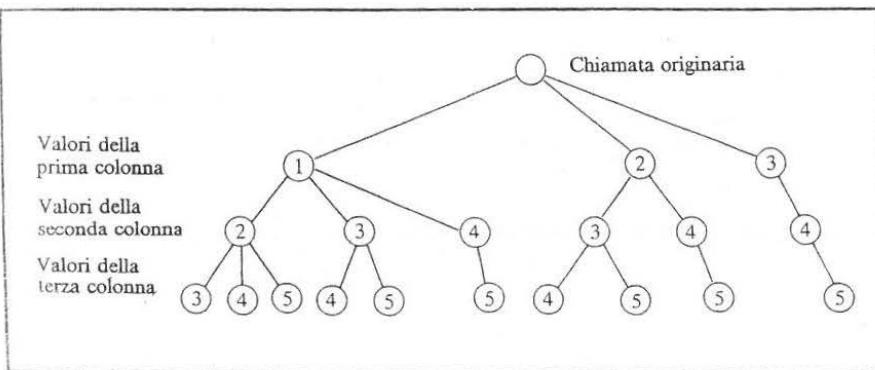


Fig. 8.17.
Meccanismo ad
albero per il
problema 'C'.

Questo suggerisce l'uso di una struttura ad albero. Seguendo questo ragionamento, il meccanismo ad albero del problema 'C', avrà la forma mostrata in Fig. 8.17. Ognuna delle combinazioni può essere trovata tracciando un percorso dalla radice attraverso l'albero fino ai nodi foglia. Questo diagramma conferma che ogni volta che ci troviamo in r -esima colonna (in questo caso la terza o i nodi foglia), sarà possibile una nuova combinazione. Il nostro algoritmo compie inizialmente una visita in ordine simmetrico dell'albero (per la spiegazione della visita in ordine simmetrico ved. Algoritmo 8.1).

La discussione precedente ci ha portati alla conclusione che ogni volta che $k = r$, sarà possibile la stampa di una nuova combinazione. Per stampare la combinazione corrente, possiamo usare:

```
if k = r then
begin
  for j := 1 to r do
    write(column[j]);
  writeln
end
```

Questi passi possono essere inseriti direttamente dopo il punto in cui viene aggiornata la colonna k .

L'ultima cosa da fare è trattare il problema della terminazione. Nel caso generale, vogliamo solo generare combinazioni di valori per r colonne. Il nostro algoritmo ricorsivo è strutturato in modo tale che mentre genera i valori della colonna corrente, inizia la chiamata alla procedura che genera i corrispondenti valori per la colonna successiva. Ne segue che quando ci troviamo in r -esima colonna (cioè quando $k = r$), non ci sono altre colonne da generare e quindi non ci saranno ulteriori chiamate a livelli più profondi di ricorsione. Ciò si lega all'output delle combinazioni quando $k = r$. In questo caso, infatti, invece di fare un'ulteriore chiamata ricorsiva al livello successivo, mandiamo semplicemente in uscita la corrente combinazione. Confrontare questo algoritmo con quello precedente.

L'intero processo ricorsivo può iniziare con una chiamata alla procedura con $k = 1$: questo caso si riferisce all'elemento $column[0]$ che deve essere inizializzato a zero. Qui di seguito è data la completa descrizione dell'algoritmo.

Descrizione dell'algoritmo

1. Definisci la dimensione positiva e diversa da zero dell'insieme, la dimensione della combinazione e l'array per memorizzare le combinazioni.
2. Inizializza il valore corrente della colonna al valore corrente della colonna precedente.
3. Finché il corrente valore di colonna è minore del suo limite superiore
 - (a) incrementa di uno il valore corrente della colonna;
 - (b) se la colonna corrente non è quella finale
 - (b.1) chiama ricorsivamente la procedura per generare tutti i valori della colonna successiva associati al valore della colonna corrente
 - altrimenti
 - (b.1') nell'array è disponibile una nuova combinazione quindi stampala.

Implementazione in Pascal

```
procedure combinationset(n,r: integer);
var column: array[0 .. 20] of integer;
  k: integer;
procedure combinations(k: integer);
var j {index for printing combinations}: integer;
begin {recursively generates all combinations of first n integers r at a
time in lexical order}
  {assert: n>0 ∧ r>0 ∧ r=<n ∧ 1=<k=<r ∧ column[0]=0}
  column[k] := column[k-1];
  {invariant: after the ith iteration column[1 .. k-1]=c1, c2, ...,
  ck-1 ∧ c1<...<ck & column[k]=column[k-1]+i ∧
  column[k]=<n-r+k ∧ column[k+1 .. r] have assumed all
  possible values in [column[k]+1 .. n] such that column[k]<
  column[k+1]<.. column[r]}
  while column[k]<n-r+k do
    begin {generate next value for the current column}
    column[k] := column[k]+1;
    if k<r then
      {generate subsequent column values associated with the
      current column}
      combinations(k+1)
```

```

else
begin {next combination available – write it out}
  for j := 1 to r do
    write{column[j]};
    writeln
  end
end
{assert: all combinations beginning with c1, c2, ..., ck – 1 have
been generated}
end;

begin {combinationset}
column[0] := 0;
k := 1;
combinations(k)
end

```

Note di progetto

- Il numero di volte in cui nel ciclo viene eseguito il test condizionale ci fornisce una misura della complessità in tempo di questo algoritmo (tale numero è pari al numero di nodi del meccanismo ad albero). Riferendoci ai nodi per il nostro esempio 3C_3 , vediamo che ci sono 3C_3 nodi foglia, 3C_2 nodi al livello superiore e solo 3C_1 al livello più alto. In questo caso ci sono quindi ${}^3C_3 + {}^3C_2 + {}^3C_1 = 19$ nodi. In generale, ci saranno:

$$\sum_{i=1}^r {}^{n-i+1}C_{r-i+1} \text{ nodi.}$$

La profondità della ricorsione è r .

- Per considerare il funzionamento di questa procedura ricorsiva non lineare, consideriamo per primo il caso in cui la chiamata sia fatta a *combinations*($k+1$). Se supponiamo che questa chiamata non modifichi gli elementi delle prime k colonne e che generi tutte le adeguate combinazioni delle colonne dalla $(k+1)$ all' r -esima, allora possiamo considerare il caso in cui venga effettuata la chiamata di *combinations*(k). Quando la chiamata corrente di *combinations*(k) non genera nessun'altra chiamata ricorsiva interna, la nostra ipotesi deve essere corretta. Nel caso generale dove viene fatta la chiamata di *combinations*(k), ammettendo le ipotesi associate alla chiamata di *combinations*($k+1$), la presente chiamata lascerà invariati gli elementi di *column*[$0..k-1$], poiché essi non sono contenuti nel corpo della procedura. La presente chiamata, inoltre, assicura che tutte le combinazioni di *column*[$k..n$] siano prodotte correttamente poiché sono state fatte tutte le possibili scelte di *column*[k]. Dopo ogni scelta viene effettuata una chiamata ricorsiva di *combinations*($k+1$). Questa argomentazione forma le basi di una prova formale induktiva della procedura. Ogni chiamata ricorsiva di *combinations* produce uno spostamento verso una colonna più alta ed in

ogni caso termina per $k = r$. Possiamo inesattamente concludere che l'algoritmo termina.

- In questo algoritmo come nel precedente si può notare come la soluzione di un problema di grandi dimensioni sia costruita risolvendo una successione di problemi più piccoli collegati tra loro.
- La non pratica soluzione iterativa di uno specifico problema è stata utile nel guidarci ad una soluzione generale ricorsiva del problema.
- Un insieme di n cicli annidati può essere sostituito da una procedura ricorsiva che si richiama r volte. In questo modo possono essere risolti molti problemi.
- Nell'implementazione abbiamo scelto di includere solo la variabile k come parametro per ragioni di efficienza.

Applicazioni

Combinazioni, applicazioni di teoria della probabilità.

Problemi supplementari

- Modificare l'algoritmo dato in modo che stampi tutte le combinazioni *fino* ad r in una sola volta in ordine lessicale.
- Implementare un algoritmo ricorsivo che generi le combinazioni nC_r in ordine lessicale inverso.
- Progettare un algoritmo di generazione di combinazioni che riceva in ingresso un insieme di n caratteri e produca in uscita tutte le combinazioni di lunghezza r di questi caratteri.
- Un altro semplice modo per generare combinazioni in ordine lessicale è di iniziare dalla combinazione $(1,2,3,\dots,r)$. La combinazione successiva viene generata in questo caso scandendo la combinazione corrente da destra a sinistra finché non si incontra un elemento che non ha raggiunto il suo valore massimo. Questo elemento viene incrementato di un'unità e tutti gli elementi a destra vengono posti ai rispettivi valori minimi possibili; quindi il processo viene ripetuto. Implementare questo algoritmo e confrontare le sua esecuzione con il nostro algoritmo originario.
- Progettare un algoritmo che accetti in ingresso una data combinazione dell'insieme nC_r e produca in uscita la combinazione successiva in ordine lessicale.
- In molti casi l'algoritmo che abbiamo elaborato deve cambiare più di un elemento nel passaggio da una combinazione alla successiva. Cercare di progettare un algoritmo di generazione che faccia un solo cambiamento nel passare da una combinazione alla successiva. (Rif. P.J.Chase, "Combinations of m out of n objects". Comm. ACM, 13, 368 (1970)).

ALGORITMO 8.6 GENERAZIONE DI PERMUTAZIONI

Problema

Progettare un algoritmo che generi tutte le permutazioni dei primi n interi, presi r alla volta, con n ed r maggiori o uguali ad 1.

Sviluppo dell'algoritmo

Come potremo vedere, il problema di generare permutazioni condivide alcune delle caratteristiche degli ultimi due problemi. Possiede anche altri aspetti che ci possono aiutare ad approfondire la conoscenza della ricorsione. Come con i precedenti due problemi, ci concentreremo sul comportamento del problema espresso in termini di interi. Problemi relativi ad elementi diversi possono essere trattati in modo simile.

Per iniziare lo studio, possiamo considerare un esempio specifico che comporta la generazione di tutte le permutazioni dei primi 5 interi presi 3 alla volta. Diversamente dal caso delle combinazioni, l'ordine è importante e quindi sequenze come (1,2,3), (2,1,3) e (3,1,2) rappresentano permutazioni distinte. Alcune delle permutazioni dei primi 5 interi presi tre alla volta sono:

1	1	1	1	1	1	...	5	5
2	2	2	3	3	3	...	4	4
3	4	5	2	4	5	...	2	3

Studiando più dettagliatamente l'esempio, vediamo che ognuna delle 3 (r) colonne può assumere tutti i 5 (n) valori, come nell'algoritmo 8.4. All'interno di una data permutazione, tuttavia, nessun valore può ricorrere in più di una colonna (cioè non sono messe ripetizioni). Anche nel problema di generare combinazioni, le ripetizioni non erano messe. Vedremo quindi se in questo caso possa essere usato un tale approccio.

Sfortunatamente, quando cerchiamo di risolvere il caso della generazione di permutazioni nello stesso modo di quello di generazione delle combinazioni sorgono dei problemi, poiché gli interi all'interno di una data permutazione non devono essere ordinati. (Ad esempio (1,3,2) è una permutazione perfettamente accettabile).

Può valere la pena cercare di porre il problema in termini di una semplice ma non pratica struttura a cicli annidati. Una tale visione può

fornirci gli indizi necessari alla risoluzione del problema. A questo punto abbiamo scoperto tre proprietà degli insiemi di permutazioni. Per prima cosa, ogni colonna deve essere in grado di contenere i primi n valori interi. Secondariamente, gli interi in una permutazione non devono essere ordinati e, in terzo luogo, non devono esserci ripetizioni all'interno di una data permutazione. Nell'algoritmo 8.4, per assicurare che ogni colonna assumesse tutti i valori dell'intervallo da 1 ad n , bastava una semplice struttura a cicli annidati. Per risolvere il problema ' U ', erano necessari tre cicli. Ci potremmo quindi aspettare di dover usare tre cicli per risolvere il problema ' P_3 '. Per questo sarebbe necessario un metodo per decidere che nessun intero sia ripetuto in ogni permutazione stampata. Applicando questa limitazione, l'algoritmo necessario per risolvere il problema ' P_3 ' dovrebbe essere:

```
:  
for i := 1 to 5 do  
  for j := 1 to 5 do  
    for k := 1 to 5 do  
      if "non ci sono ripetizioni tra i, j, e k" then "stampa i, j, e k come  
      prossima permutazione"
```

La nostra esperienza con gli ultimi due algoritmi ci ha mostrato che spesso un insieme di r cicli annidati può essere sostituito da una procedura ricorsiva che chiama se stessa r volte. Potremmo quindi essere incoraggiati e cercare di formulare una soluzione ricorsiva del presente problema. Le deduzioni fatte finora suggeriscono che possiamo descrivere sperimentalmente l'algoritmo richiesto come:

```
for i := 1 to n do  
  begin  
    "risolvi i rimanenti problemi più piccoli assicurandoti che non ci siano ripeti-  
    zioni".  
  end
```

Questo meccanismo tratterà la generazione dei valori della prima colonna. Per generare l'insieme dei valori della seconda colonna, abbiamo bisogno ancora una volta di un ciclo che generi interi nell'intervallo da 1 ad n . La generazione di interi della seconda colonna è accompagnata da una *limitazione*. Nessun valore della seconda colonna può essere uguale al corrispondente valore della prima. Estendendo questa discussione alla terza colonna vediamo che sebbene tutti gli interi dell'intervallo da 1 ad n sono potenziali candidati, nessun valore della terza colonna può essere uguale ai valori in prima e seconda colonna della corrente permutazione. Lo stesso discorso si applica alle colonne

successive. Dobbiamo ora affrontare il problema della generazione di questi valori limitati della colonna. Potremmo risolverlo usando una serie di istruzioni *if* o un ciclo, ma questi metodi appaiono assai pesanti. Dobbiamo quindi cercare un metodo semplice per *ricordare* quale sia la corrente situazione nelle colonne che precedono quella corrente. Sarebbe preferibile un *singolo* test per vedere se l'intero corrente possa essere inserito nella colonna corrente. È però possibile costruire un test ad un solo passo per decidere l'eleggibilità del corrente intero nella permutazione che sta per essere generata? Poiché il test non può comportare una serie di confronti, in quale altro modo è possibile *ricordare* quali altri interi sono già stati usati? L'unica alternativa sembra essere l'uso di un array per memorizzare questi interi usati. La domanda che ora sorge è se un array possa essere compatibile con un semplice test di eleggibilità. La nostra risposta è affermativa, purchè noi facciamo il test per *indirizzo*. Questo comporterà diverse cose. Quando l'intero i è stato impiegato in una colonna, noi dovremo marcare appropriatamente l' i -esima posizione dell'array. In ogni preciso momento potremo allora controllare se un dato intero k sia già stato usato guardando se la locazione k dell'array è stata marcata. A questo scopo può essere usato un array di valori Booleani.

Dopo aver superato il problema di decidere interi gli eleggibili per una particolare colonna, vediamo che il problema che resta da risolvere è pressochè lo stesso di quello di generare permutazioni senza limite di ripetizioni (algoritmo 8.4). Possiamo quindi riassumere i progressi nel seguente schema di un algoritmo di generazione di permutazioni:

per $i := 1$ a n
 se l'intero i è eleggibile per la colonna corrente k allora
 (a) marca l' i -esimo intero in modo che non possa essere inserito nelle colonne successive,
 (b) salva i come corrente valore di colonna,
 (c) genera ricorsivamente i valori delle restanti colonne della corrente permutazione.

Studiando più accuratamente questo meccanismo, vediamo che ha una falla, poichè appena generata la prima permutazione, *tutti* i primi n interi risultano marcati come inutilizzabili. Per cercare di correggerla, consideriamo un esempio specifico. Supponiamo che la prima permutazione generata per $n = 4$ sia:

1 2 3 4

La successiva permutazione sarà:

1 2 4 3

Per derivare la seconda permutazione dalla prima, devono essere cambiati i valori della terza e quarta colonna, e quello in terza colonna deve essere cambiato per *primo*. In questo modo il 4 può essere selezionato come valore della terza colonna. Ciò può accadere se il 4 è stato precedentemente *dall'insieme dei valori non eleggibili. Dobbiamo*

ora stabilire come poter incorporare questa operazione di rimozione nel nostro algoritmo. Da un suo studio, vediamo che il valore dell'indice di ciclo è marcato come "non eleggibile" *prima* che sia effettuata la chiamata ricorsiva, poichè questa informazione deve essere trasportata attraverso le chiamate *successive* dalla permutazione *corrente*. Ne segue che dopo il termine del passo ricorsivo, l'indice corrente di ciclo non è più necessario nell'insieme dei "non eleggibili". A questo punto il valore dell'indice deve essere *rimosso*, in modo che possa essere usato in una colonna successiva (o precedente) della permutazione successiva. Con questa aggiunta il nostro algoritmo funzionerà correttamente. Il passo da aggiungere al ciclo è:

(d) rimuovi l'indice corrente i dall'insieme dei "non eleggibili"

I valori Booleani *true* e *false* possono essere usati per marcare e rimuovere i valori dall'insieme dei "non eleggibili".

A questo punto siamo in grado di descrivere l'algoritmo. Forniremo l'implementazione Pascal nonostante alcune riserve sull'efficienza dell'algoritmo prodotto, poichè ad ogni chiamata ricorsiva deve essere considerato l'intero intervallo da 1 a n .

```
procedure lperm(var k:integer);
var i:integer;
begin
  for i := 1 to n do
    if available[i] then
      begin
        column[k] := i;
        available[i] := false;
        if k < r then
          lperm(k+1)
        else
          "stampa la permutazione corrente"
        available[i] := true
      end
    end
end
```

Le permutazioni sono memorizzate nell'array *column* e l'array booleano *available* è usato per marcare quali interi non sono più disponibili. Essi devono però essere tutti disponibili all'inizio dell'esecuzione. L'algoritmo sviluppato produce l'insieme delle permutazioni in ordine lessicale. Studiando più accuratamente questo meccanismo, osserviamo che l'istruzione dominante è il test per vedere se l'intero i è attualmente disponibile per la selezione. Il test dovrà essere fatto n^r volte, valore che è di solito molto maggiore del numero di permutazioni dell'insieme, " P_r ".

Dalle osservazioni appena fatte, ci chiediamo come poter migliorare il funzionamento del nostro algoritmo. Per questo dobbiamo trovare un modo di ridurre il costo per generare una nuova permutazione. Per

cercare un nuovo metodo, concentriamoci nuovamente sui passi per passare dalla prima alla seconda permutazione con $n = 4$.

1	2	3	4
1	2	4	3

(prima permutazione)
(seconda permutazione)

Queste due permutazioni differiscono solo nei loro ultimi due elementi. Il nostro algoritmo deve eseguire il test:

`if available[i] then ...`

quattro volte per passare dalla prima alla seconda permutazione.

Ciò sembra eccessivo, poiché le due permutazioni si differenziano solo per l'ordine dei loro ultimi elementi. Potremmo quindi chiederci se esista un meccanismo più semplice che ci permetta di passare dalla prima alla seconda permutazione. La risposta è affermativa: tutto ciò che dobbiamo fare è *scambiare* gli ultimi due elementi della prima permutazione.

Dall'esempio, vediamo anche che scambiando ogni coppia di elementi dell'insieme troveremo una nuova permutazione. Ciò suggerisce che possiamo costruire un algoritmo di generazione di permutazioni più efficiente basato sullo scambio di coppie di elementi. Se basiamo l'algoritmo sullo scambio di coppie di elementi, dovremo farlo sistematicamente per evitare sia la perdita di qualche permutazione, sia la generazione di doppi.

La domanda a cui dobbiamo rispondere è ora: come possiamo generare sistematicamente permutazioni usando gli scambi? La nostra precedente esperienza con problemi attinenti suggerisce che potremmo procedere costruendo la soluzione colonna per colonna. Potremmo quindi costruire un algoritmo con almeno parte della struttura del nostro proposito originario. Sappiamo, ad esempio, che la prima colonna deve assumere tutti i valori dell'intervallo da 1 a n . Una volta scelto il primo valore ci rimangono $n - 1$ scelte per il secondo valore della colonna. Inoltre, una volta scelti il primo ed il secondo valore della colonna ci rimangono solo $n - 2$ scelte per la terza colonna e così via.

Ad esempio, per il caso 3P_3 , quando abbiamo le prime due colonne come segue:

3	1
---	---

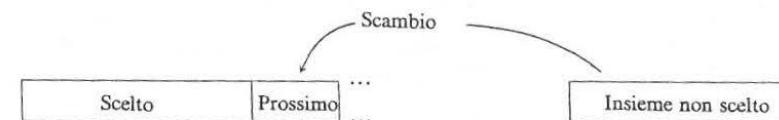
allora le scelte possibili per la terza colonna sono:

{ 2, 4, 5 }

Potremo cioè avere:

3	1	2
3	1	4
3	1	5

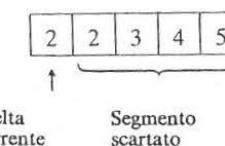
Questo esempio suggerisce che in ogni dato momento i nostri n interi sono divisi in *due* insiemi: uno che è già stato *scelto* e l'insieme corrente *non scelto*. Per distinguere gli insiemi scelto e non scelto, il nostro precedente algoritmo usava un array booleano. Ora invece cerchiamo un modo più efficiente per distinguere questi insiemi; se potessimo trovarlo potremmo sviluppare un algoritmo più efficiente. Una riflessione rivela che una possibilità sarebbe mantenere i primi k valori nelle prime k locazioni dell'array e le restanti $n - k$ nelle $n - k$ locazioni. Il nostro compito, allora, per generare una nuova permutazione, comporterà uno scambio di valori avanti e indietro tra gli insiemi scelto e non scelto, cioè, schematicamente il nostro meccanismo comporterà:



Cerchiamo ora di scrivere la semplice, ma inattuabile struttura a ciclo per il problema 3P_3 e quindi cerchiamo di fare dei progressi. Lo studio precedente suggerisce che saranno necessari tre cicli annidati per generare un insieme di permutazioni a tre colonne. Il modo migliore di scegliere sistematicamente i valori per una data colonna è probabilmente sceglierli *consecutivamente* dall'insieme non scelto. Dopo aver stabilito queste scelte, potremmo proporre la seguente struttura a ciclo:

```
for i := 1 to 5 do
    for j := 2 to 5 do
        for k := 3 to 5 do
            begin
                column[1] := column [i];
                column[2] := column [j];
                column[3] := column [k];
            end
```

Studiando questo meccanismo troviamo che mentre al primo sguardo sembra fare quello che vogliamo, ha alcune falde ed inefficienze. Mentre *column[1]* può assumere tutti i valori, *column[2]* e *column[3]* non li possono assumere tutti. Ad esempio, supponiamo di iniziare con 1, 2, ..., 2 nelle prime cinque locazioni dell'array, quando *column[k]* assume il valore 2 il nostro meccanismo *non* pone 1 nell'insieme non scelto. A questo punto il nostro meccanismo produce:



che significa che $column[2]$ non può assumere il valore 1. A questo punto, invece, vogliamo la seguente configurazione:

2	1	3	4	5
---	---	---	---	---

Come possiamo raggiungere questa configurazione? Un semplice approccio è scambiare il valore della prima colonna con quello della i -esima colonna. Estendendo questa idea, vediamo che il valore in seconda colonna dovrebbe essere scambiato con quello in j -esima colonna e così via. Inoltre, dato che i non varia nei due cicli più interni, lo scambio tra $column[1]$ e $column[i]$ dovrebbe avvenire al di fuori di questi due cicli.

Nell'implementazione precedente, abbiamo riscontrato la necessità di invertire il processo di eleggibilità. Ci potremmo aspettare di applicare una simile situazione al nostro nuovo algoritmo. Una volta che un valore è stato selezionato e inserito in una data colonna, deve essere riportato nella sua posizione originale in modo che possa essere selezionato per altre colonne. Ne segue che il processo di scambio deve essere invertito. La soluzione a cicli annidati del problema 3P , avrà quindi la forma:

- per $i := 1$ a 5
 - (a) scambia i valori in prima ed i -esima colonna,
 - (b) per $j := 2$ a 5
 - (b.1) scambia i valori in seconda e j -esima colonna,
 - (b.2) per $k := 3$ a 5
 - (2.a) scambia i valori in terza e k -esima colonna,
 - (2.b) stampa la permutazione corrente,
 - (2.c) scambia i valori in k -esima e terza colonna,
 - (b.3) scambia i valori in j -esima e seconda colonna,
 - (c) scambia i valori in i -esima e prima colonna.

Un controllo di questo meccanismo conferma che eseguirà il compito richiesto. Il compito successivo è quindi formularlo come algoritmo ricorsivo generale. Studiando il meccanismo a cicli annidati, vediamo che in ogni ciclo è ripetuto essenzialmente lo stesso processo. I passi comuni di ogni ciclo sono:

1. Finché non sono stati selezionati tutti i valori disponibili per la corrente colonna
 - (a) scambia il j -esimo valore disponibile con il valore della corrente colonna,
 - (b) ripeti il processo per le colonne restanti,
 - (c) scambia il valore della colonna corrente con il j -esimo valore disponibile.

Se consideriamo il passo (b) di questo meccanismo come ricorsivo, allora questi passi contengono la maggior parte degli elementi necessari per la soluzione ricorsiva del problema. Dobbiamo ora delineare i problemi della terminazione e dell'output delle permutazioni.

Uno studio più approfondito del nostro meccanismo conferma che esso è strutturalmente simile al nostro progetto originale e quindi la terminazione e l'uscita possono essere trattate in maniera simile. La marcatura di eleggibilità e di non eleggibilità di un dato intero è stata sostituita da uno scambio e dal suo inverso. Il test di eleggibilità è stato sostituito da una regolazione dell'indice del ciclo in modo che scelga sempre dal corrente insieme dei *non scelti* o eleggibili.

Con l'aggiunta dei passi di terminazione e di output, è completata la formulazione dell'algoritmo che può quindi essere descritta dettagliatamente.

Descrizione dell'algoritmo

1. Definisci la dimensione dell'insieme e l'array per memorizzare le permutazioni (deve essere uguale in lunghezza alla dimensione dell'insieme, invece che alla dimensione della permutazione). Definisci inoltre la colonna corrente dell'insieme delle permutazioni.
2. Finché non sono stati selezionati tutti i valori disponibili per la colonna corrente
 - (a) scambia il j -esimo valore dell'insieme non scelto con il valore corrente
 - (b) se l'indice corrente di colonna non è alla colonna finale allora
 - (b.1) chiama ricorsivamente la procedura per generare tutti i valori della colonna successiva associati alla rappresentazione corrente di tutte le colonne fino a quella corrente,
 - altrimenti
 - (b'.1) è disponibile una nuova permutazione, quindi stampala,
 - (c) inverti lo scambio fatto al passo (a).

Implementazione in Pascal

```

procedure permutationset( $n, r$ : integer);
var column: array[1..20] of integer;
    k:integer;

procedure permutations( $k$ : integer);
var  $i$  {index for printing permutations},
     $j$  {index for selecting values from the unchosen set},
    temp {temporary variable used in exchange}: integer;
begin {recursively generates all permutations of the first  $n$  integers
taken  $r$  at a time. The permutations are not in lexical order}
  {assert:  $n > 0 \wedge r > 0 \wedge r = < n \wedge 1 = < k = < r \wedge
  column[1..n] = c1, c2, ..., c[n] \wedge c1, c2, ..., all in [1..n]}$ 
```

{invariant: after jth iteration column[1.. k - 1] = c1, c2, ..., c(k - 1)}

```

where  $c_1, c_2, \dots$  uniquely chosen from set  $[1..n]$  leaving  $n-k+1$ 
unchosen values in  $\text{column}[k..n] \wedge$  all permutations of column
 $[k..n]$  generated in which  $\text{column}[k] = c(j) \wedge k = j \leq n$ 
for  $j := k$  to  $n$  do
  begin {select all available values for the current column}
     $\text{temp} := \text{column}[k];$ 
     $\text{column}[k] := \text{column}[j];$ 
     $\text{column}[j] := \text{temp};$ 
    if  $k < r$  then
      {generate subsequent column values}
       $\text{permutations}(k+1)$ 
    else
      begin {next permutation available - write it out}
        for  $i := 1$  to  $r$  do
           $\text{write}(\text{column}[i]);$ 
         $\text{writeln}$ 
      end;
       $\text{temp} := \text{column}[k];$ 
       $\text{column}[k] := \text{column}[j];$ 
       $\text{column}[j] := \text{temp}$ 
    end
    {assert: all permutations beginning with  $c_1, c_2, \dots, c(k-1)$  have
    been generated}
  end;

begin{permutationset}
  for  $k := 1$  to  $n$  do
     $\text{column}[k] := k;$ 
     $k := 1;$ 
     $\text{permutations}(k)$ 
  end

```

Note di progetto

- Dallo studio della rappresentazione ad albero di questo algoritmo vediamo che la sua complessità è proporzionale a $n!/(n-r)!$ L'albero di ' P ', ha la forma mostrata in Fig. 8.18. Il numero di nodi foglia è $4*3*2 = 24$ che corrisponde al numero di permutazioni di 4 elementi presi 3 alla volta.
- Possiamo caratterizzare il funzionamento di questo algoritmo usando considerazioni molto simili a quelle usate nella descrizione dei precedenti due algoritmi. Questo è lasciato come esercizio al lettore.
- Nello studio dell'implementazione finale vediamo che essa contiene alcune inefficienze strutturali. Occorre notare che la prima volta che si attraversa il ciclo, ad ogni livello è $k=j$ e quindi non è necessario uno scambio. È pure ridondante l'assegnamento $\text{temp} := \text{column}[k]$ all'interno del ciclo, poiché ivi k non cambia.

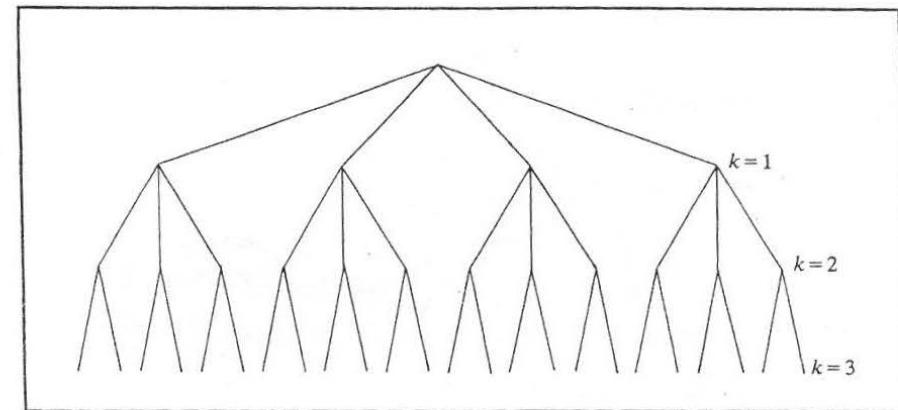


Fig. 8.18.
Mecanismo ad
albero per il
problema ' P '.

- Un problema diffuso riguardante le permutazioni è quello di generare tutte le permutazioni di n oggetti presi n alla volta. Quando viene applicato a questo problema il nostro algoritmo, vediamo che esso compirà alcuni passi non necessari poiché procede fino alla situazione in cui $k = r = n$. Noi dobbiamo proseguire solo fino alla situazione in cui $k = r - 1$ poiché, dopo che sono stati scelti i primi $n-1$ elementi, l' n -esimo elemento è completamente determinato.
- L'implementazione finale, sebbene più efficiente del progetto precedente, è stata solo capace di ottenere questa efficienza sacrificando l'output lessicale delle permutazioni. Possiamo aggiungere che attualmente esistono circa quaranta algoritmi diversi proposti per la generazione di permutazioni. Il nostro progetto non è certamente il più efficiente ma illustra una struttura computazionale generale che può essere applicata ad una vasta gamma di problemi.
- Occorre notare che gli algoritmi per generare permutazioni in ordine lessicale sono in genere meno efficienti di altri poiché usano spesso più di uno scambio per passare da una permutazione alla successiva.
- Nel risolvere questo problema abbiamo visto ancora una volta come la soluzione finale è stata costruita dalla soluzione di problemi più semplici. Partendo dal problema più semplice è sorta in modo naturale l'idea di usare gli insiemi chosen e unchosen. Una volta fatta questa scoperta, bisogna solo costruire i meccanismi per il trattamento di questi insiemi. A questo proposito è stato utile il semplice modello a cicli annidati.
- Nel nostro algoritmo finale ci sono due scambi all'interno di un ciclo. Potremmo sospettare che esista la possibilità di formulare un algoritmo che richieda un solo scambio all'interno del ciclo. Possiamo infatti fare un piccolo cambiamento al nostro algoritmo che produca un'implementazione più efficiente per il calcolo delle $n!$ permutazioni dei primi n interi. Tutto ciò che dobbiamo fare è includere un test prima della chiamata ricorsiva che ci permetta di scambiare il j -esimo con il k -esimo elemento se $(n-k)$ è dispari, o scambiare l' n -esimo con il k -esimo se $(n-k)$ è pari.

Applicazioni

Problemi combinatori, simulazione.

Problemi supplementari

- 8.6.1 Modificare l'algoritmo dato per includere i miglioramenti suggeriti in nota 3.
- 8.6.2 Implementare un algoritmo ricorsivo per il calcolo delle $n!$ permutazioni dei primi n interi. Nella vostra implementazione la chiamata ricorsiva interna deve essere del tipo *permutation*($n - 1$) anzichè *permutation*($k + 1$) come abbiamo usato nel presente algoritmo.
- 8.6.3 Progettare un algoritmo che riceva in ingresso una data permutazione dei dieci interi e fornisca in uscita la permutazione *successiva* dell'insieme in ordine lessicale. Ad esempio, abbiamo:

permutazione corrente	7	9	0	1	6	3	8	5	4	2
permutazione successiva	7	9	0	1	6	4	2	3	5	8

Come suggerimento, notare che gli ultimi quattro numeri della permutazione corrente sono in ordine *descendente*.

- 8.6.4 Progettare un algoritmo ricorsivo per generare permutazioni in ordine lessicale basato sul meccanismo usato per risolvere il problema 8.6.3
- 8.6.5 Implementare i suggerimenti dati in nota 8 e confrontare il numero di scambi di questa nuova implementazione con l'originale ed i risultati del problema 8.6.3.

