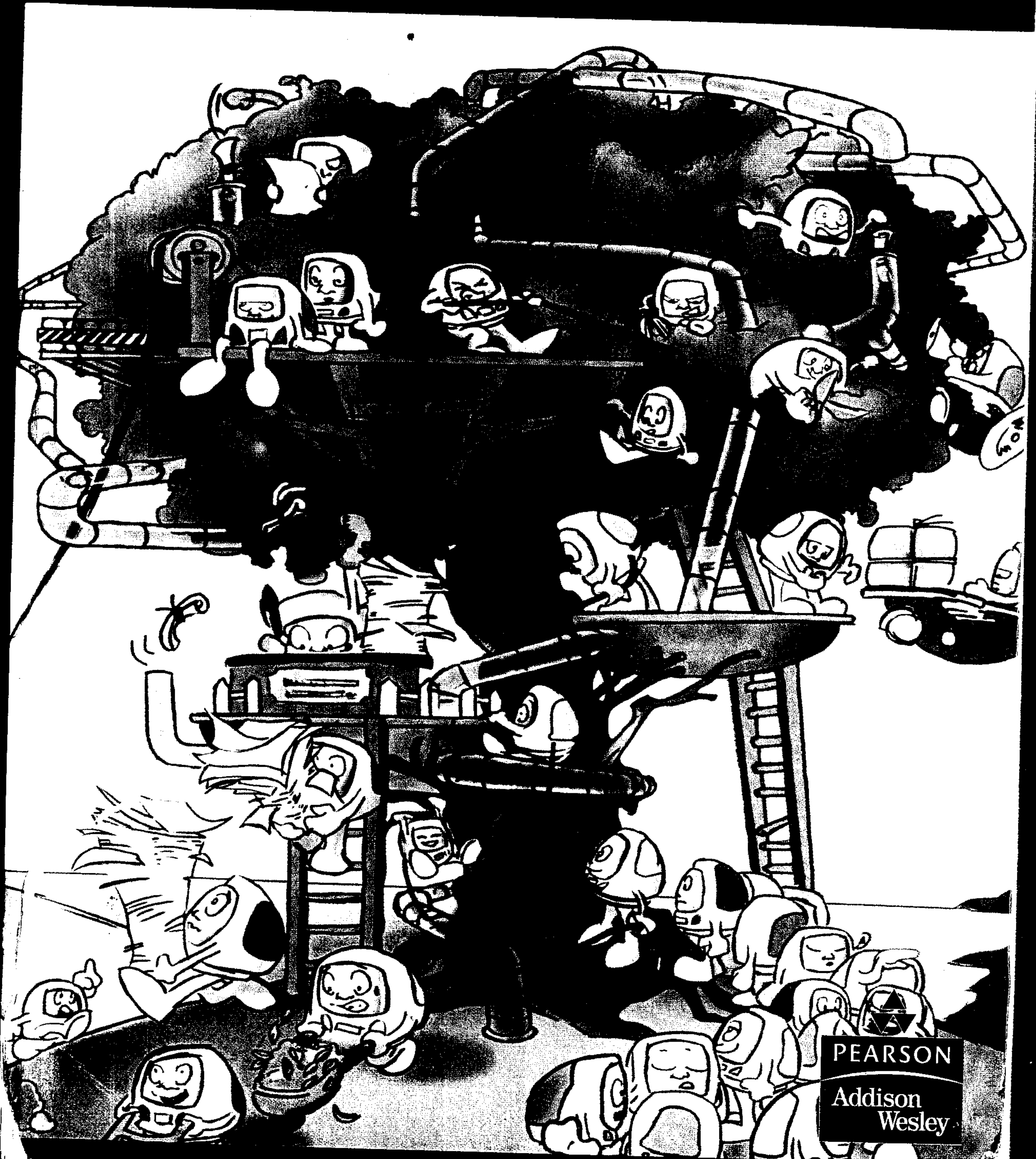
Pierluigi Crescenzi • Giorgio Gambosi • Roberto Grossi

Strutture di dati e algoritmi

Progettazione, analisi e visualizzazione



PEARSON
Addison Wesley

dendo nota (tale accorgimento viene denominato *memoization* in inglese) delle soluzioni già calcolate e prevedendo, in corrispondenza a ogni chiamata ricorsiva, di verificare anzitutto se la soluzione del sotto-problema in questione è già stata calcolata e annotata, così da poterla restituire immediatamente. Ciò comporta l'uso di un array di dimensione opportuna per mantenere le soluzioni dei vari sotto-problemi considerati, prevedendo inoltre che gli elementi di tale array possano assumere un valore "indefinito", che indichiamo con il simbolo $\emptyset$, per segnalare il caso in cui il corrispondente sotto-problema non sia ancora stato risolto.

A titolo di esempio, possiamo sviluppare un algoritmo ricorsivo per il problema della partizione che utilizza il suddetto meccanismo (ricordando comunque che è consigliabile usare la programmazione dinamica). Tale algoritmo inizializza la prima riga dell'array `parti` come nel Codice 2.20 e riempie le righe successive con il valore indefinito $\emptyset$. Successivamente, l'algoritmo invoca la funzione ricorsiva che segue la regola in (2.14).

```
PartizioneMemoization( ):
  FOR (j = 0; j <= s; j = j+1)
    parti[0][j] = FALSE;
  parti[0][0] = TRUE;
  FOR (i = 1; i <= n; i = i+1)
    FOR (j = 0; j <= s; j = j+1) {
      parti[i][j] = ∅;
    }
  RETURN PartizioneRicNota( n, s );

PartizioneRicNota( i, j ):                    ⟨pre: 0 ⩽ i ⩽ n, 0 ⩽ j ⩽ s⟩
  IF (parti[i][j] == ∅) {
    parti[i][j] = PartizioneRicNota( i-1, j );
    IF (!parti[i][j] && (j >= a[i-1])) {
      parti[i][j] = PartizioneRicNota( i-1, j-a[i-1] );
    }
  }
  RETURN parti[i][j];
```

2.7.3 Problema della bisaccia

Mostriamo ora una generalizzazione del problema della partizione a un famoso problema di ottimizzazione: tale problema, denominato problema della **bisaccia** (*zaino* o *knapsack*), può essere definito, in modo pittoresco, come segue.

Supponiamo che un ladro riesca a introdursi, nottetempo, in un museo dove sono esposti una quantità di oggetti preziosi, più o meno di valore e più o meno pesanti. Tenuto conto che il ladro può trasportare una quantità massima di peso, il suo problema è selezionare un insieme di oggetti da trafugare il cui peso complessivo non superi la

possanza che il ladro è in grado di sopportare, massimizzando al tempo stesso il valore complessivo degli oggetti rubati.

Abbiamo quindi un insieme di elementi $A = \{a_0, a_1, \ldots, a_{n-1}\}$ su cui sono definite le due funzioni `valore` e `peso`, le quali associano il valore e il peso a ogni elemento di A (supponiamo che sia il valore che il peso siano numeri interi positivi). Conosciamo inoltre un intero positivo `possanza`, che indica il massimo peso totale che il ladro può portare. Vogliamo determinare un sottoinsieme $A' = \{a_{i_0}, a_{i_1}, \ldots, a_{i_{k-1}}\} \subseteq A$ tale che il peso totale dei suoi elementi rientri nella possanza, ovvero $\sum_{j=0}^{k-1} \mathtt{peso}(a_{i_j}) \leqslant$ `possanza`, e tale che il valore degli oggetti selezionati, ovvero $\sum_{j=0}^{k-1} \mathtt{valore}(a_{i_j})$, sia il massimo possibile.

A=∅ quando i=0

Per applicare il paradigma della programmazione dinamica possiamo definire, come sotto-problema generico, la ricerca della soluzione ottima supponendo una minore possanza e un più ristretto insieme di elementi. In altri termini, per $0 \leqslant i \leqslant n$ e $0 \leqslant j \leqslant$ `possanza`, denotiamo con $P_{i,j}$ il sotto-problema relativo al caso in cui possiamo utilizzare i soli elementi $A = \{a_0, a_1, \ldots, a_{i-1}\}$ con il vincolo di non superare un peso pari a j, e indichiamo con $V(i, j)$ il massimo valore ottenibile in tale situazione.

Per caratterizzare i sotto-problemi elementari, osserviamo che $V(i, 0) = 0$, per $0 \leqslant i \leqslant n$, in quanto il peso trasportabile è nullo e, quindi, il valore complessivo deve essere pari a 0, mentre $V(0, j) = 0$, per $0 \leqslant j \leqslant$ `possanza`, poiché non ci sono elementi disponibili e il valore complessivo è necessariamente 0. Possiamo definire la decomposizione ricorsiva osservando che la soluzione di valore massimo $V(i, j)$ per il sotto-problema $P_{i,j}$ può essere ottenuta a partire da tutte le soluzioni ottime che utilizzano i soli elementi $a_0, a_1, \ldots, a_{i-2}$, in due soli possibili modi:

- il primo modo è che la soluzione ottima di $P_{i,j}$ non includa a_{i-1} e che, in tal caso, abbia lo stesso valore della soluzione ottima del sotto-problema $P_{i-1,j}$, ossia $V(i, j) = V(i-1, j)$;

- il secondo modo è che la soluzione ottima di $P_{i,j}$ includa a_{i-1} e che, pertanto, il suo valore sia dato dalla somma di $\mathtt{valore}(a_{i-1})$ con il valore della soluzione ottima di $P_{i-1,m}$, dove $m = j - \mathtt{peso}(a_{i-1})$: in tal caso, quindi, $V(i, j) = V(i-1, j - \mathtt{peso}(a_{i-1})) + \mathtt{valore}(a_{i-1})$ (se $j \geqslant \mathtt{peso}(a_{i-1})$).

La soluzione ottima di $P_{i,j}$ sarà quella corrispondente alla migliore delle due (sole) possibilità, ovvero $V(i, j) = \max\{V(i-1, j), V(i-1, j - \mathtt{peso}(a_{i-1})) + \mathtt{valore}(a_{i-1})\}$. Per tornare al nostro esempio figurato, supponiamo che il ladro abbia a disposizione gli elementi $a_0, a_1, \ldots, a_{i-1}$ e la possibilità di trasportare un peso massimo j: se egli decide di non prendere l'elemento a_{i-1}, allora il meglio che può ottenere è la scelta ottima tra $a_0, a_1, \ldots, a_{i-2}$, sempre con peso massimo j; se invece decide di prendere a_{i-1}, allora il meglio lo ottiene trovando la scelta migliore tra $a_0, a_1, \ldots, a_{i-2}$ tenendo presente che,

```
Bisaccia( peso, valore, possanza ):
  ⟨pre: peso e valore sono array di n interi positivi, possanza è un valore intero positivo⟩
  FOR (i = 0; i <= n; i = i+1) {
    FOR (j = 0; j <= possanza; j = j+1) {
      V[i][j] = 0;
    }
  }
  FOR (i = 1; i <= n; i = i+1) {
    FOR (j = 1; j <= possanza; j = j+1) {
      V[i][j] = V[i-1][j];
      IF (j >= peso[i-1])) {
        m = V[i-1][j-peso[i-1]] + valore[i-1];
        IF (m > V[i][j]) V[i][j] = m;
      }
    }
  }
  RETURN V[n][possanza];
```

Codice 2.21 Algoritmo iterativo per il problema della partizione.

dato che dovrà trasportare anche a_{i-1} in aggiunta agli elementi scelti, si deve limitare a un peso massimo pari a j decrementato del peso di a_{i-1}. La scelta migliore sarà quella che massimizza il valore.

L'algoritmo iterativo descritto nel Codice 2.21 realizza tale strategia facendo uso di un array bidimensionale V di taglia $(n+1) \times (\texttt{possanza}+1)$, in cui l'elemento V[i][j] contiene il costo $V(i,j)$ della soluzione ottima di $P_{i,j}$. Dato che il numero di sotto-problemi è $O(n \times \texttt{possanza})$ e che derivare il costo di soluzione di un sotto-problema comporta il calcolo del massimo tra i costi di due altri sotto-problemi in tempo costante, ne consegue che il problema della bisaccia può essere risolto mediante il paradigma della programmazione dinamica in tempo $O(n \times \texttt{possanza})$.

ALVIE: problema della bisaccia

Osserva, sperimenta e verifica
Knapsack

2.7.4 Pseudo-polinomialità e programmazione dinamica

Concludiamo il paragrafo sulla programmazione dinamica commentando la complessità computazionale in tempo derivata con tale paradigma. Ricordiamo che la sequenza ottima per la moltiplicazione di n matrici richiede $O(n^3)$ tempo e la determinazione di una sotto-sequenza comune più lunga tra due sequenze di lunghezza n e m richiede $O(nm)$ tempo. Gli algoritmi risultanti sono polinomiali nella dimensione dei dati in ingresso avendo, rispettivamente, un'istanza di n matrici e di $n + m$ elementi nelle due sequenze.

Per il problema della partizione di n interi di somma totale $2s$, abbiamo un costo pari a $O(ns)$, il quale è polinomiale in n e s ma *non* lo è necessariamente nella dimensione dei dati di ingresso, secondo quanto discusso nel Capitolo 1. Pur avendo n interi da partizionare, ciascuno di essi richiede $k = O(\log s)$ bit di rappresentazione. Quindi la dimensione dei dati è nk mentre il costo dell'algoritmo è $O(ns) = O(n2^k)$: tale costo non è polinomiale rispetto alla dimensione dei dati e, per questo motivo, l'algoritmo viene detto **pseudo-polinomiale**, in quanto il suo costo è polinomiale solo se si usano interi piccoli rispetto a n (per esempio, quando $s = O(n^c)$ per una costante $c > 0$). Anche l'algoritmo discusso per il problema della bisaccia è pseudo-polinomiale in quanto richiede $O(n \times \texttt{possanza}) = O(n2^k)$ tempo mentre la dimensione dei dati in ingresso richiede $O(nk)$ bit dove $k = O(\log \texttt{possanza})$. Anche in questo caso, il costo dell'algoritmo non è polinomiale, ma lo diviene nel momento in cui il valore della possanza è polinomiale rispetto al numero di oggetti.

Da ciò consegue che, mentre i primi due algoritmi sono polinomiali a tutti gli effetti, gli ultimi due algoritmi sono solo apparentemente polinomiali perché hanno complessità polinomiale rispetto al numero n di elementi, ma esponenziale rispetto alla lunghezza k della rappresentazione dei valori numerici contenuti nell'istanza del problema. In altri termini, i due algoritmi dati per i problemi della partizione e della bisaccia avrebbero complessità polinomiale se le istanze considerate dei due problemi avessero il vincolo aggiuntivo di non contenere valori numerici "eccessivamente grandi" rispetto a n. La ragione di tale anomalia è che, per valori numerici sufficientemente grandi, il problema della partizione e quello della bisaccia sono NP-completi (da notare, però, che non è vero che ogni problema NP-completo ammetta un algoritmo pseudo-polinomiale, come vedremo nel seguito del libro).

RIEPILOGO

In questo capitolo abbiamo discusso la gestione di una sequenza di elementi, distinguendo tra accesso diretto e accesso sequenziale. Abbiamo trattato approfonditamente la realizzazione dell'accesso diretto mediante array, abbiamo studiato il problema della ricerca e dell'ordinamento mediante confronti e abbiamo mostrato come risolvere ricorsivamente i problemi utilizzando il paradigma del divide et impera e la tecnica della programmazione dinamica, introducendo l'analisi degli algoritmi ricorsivi mediante le equazioni di ricorrenza.