Rapporto n. 1 2010
dmsia unibg.it

E. Angelelli, D. Ruggeri
Tecniche algoritmiche di base

Serie Didattica

Dipartimento
di Matematica, Statistica,
Informatica e Applicazioni
"Lorenzo Mascheroni"

UNIVERSITÀ DEGLI STUDI DI BERGAMO

Tecniche algoritmiche di base

Enrico Angelelli, Denis Ruggeri
Università di Brescia – Facoltà di Economia
Dipartimento di Metodi Quantitativi
angele@eco.unibs.it, ruggeri@eco.unibs.it

Indice

Divide et Impera

La tecnica *Divide et Impera*, viene tipicamente applicata per lo sviluppo top-down di un algoritmo scomponendo il problema di partenza in un insieme di problemi di vario tipo, ma di più semplice risoluzione. Se i singoli sottoproblemi non sono abbastanza facili da essere risolti in modo diretto, possono essere ulteriormente scomposti in sottoproblemi e così via fino a sviluppare una struttura ramificata di problemi e sottoproblemi più o meno profonda. Naturalmente la scomposizione si interrompe quando un sottoproblema può essere risolto in modo diretto.

In questo capitolo vedremo alcuni problemi in cui la tecnica viene utilizzata per ricondurre il problema originale ad uno o più problemi dello stesso tipo, ma posti su insiemi di dati che permettano una soluzione più agevole. Per fissare le idee pensiamo ad un problema di ordinamento: intuiamo che è più semplice effettuare l'ordinamento di 100 numeri piuttosto che di 2.000 numeri. Ancor più semplice sarà ordinare 2 numeri piuttosto che 100. Nel prossimo capitolo vedremo che la scomposizione di un problema in un insieme di problemi (più semplici) dello stesso tipo è alla base del principio di ricorsione.

Lo schema base della tecnica *Divide et Impera* è il seguente:

```
SE P è semplice ALLORA
    S ⟵ soluzione diretta di P
ALTRIMENTI
    produci un insieme di sottoproblemi P_i con i=1,...,k
    Per i=1,...,k:
        S_i ⟵ soluzione del problema P_i
```

```
    S ⟵ combinazione delle k soluzioni S_i
FINESE
Restituisci la soluzione S
```

Applicazioni a problemi di ordinamento su vettori

Ordinamento quick-sort

Il metodo di ordinamento noto come quick-sort è una tipica applicazione della tecnica *Divide et Impera* applicata ai dati. Descriveremo una delle numerose varianti esistenti. L'idea è quella di separare gli elementi della sequenza da ordinare `A` in due sottosequenze `A'` e `A"` con le seguenti proprietà:

1. `A'` e `A"` non sono vuote (ciascuna conterrà un numero di elementi inferiore a quello di `A`)
2. tutti gli elementi di `A'` sono minori o uguali di un qualunque elemento in `A"`

Per creare le due sottosequenze si può scegliere un elemento in A (ad esempio il primo) e usarlo come elemento separatore. L'elemento separatore può essere collocato per ultimo nella sottosequenza con minor numero di elementi per garantire che nessuna sottosequenza sia vuota. A questo punto è sufficiente ordinare separatamente `A'` e `A"` e accodare le due sottosequenze ordinate.

Vediamo lo schema di calcolo per l'ordinamento dei dati in un array `A` da una posizione `p` fino ad una posizione `u >= p`:

```
Algoritmo QuickSort:  Ordina gli elementi di A dalla posizione p alla posizione u
-- SE p = u FINE // la sequenza contiene un solo elemento ed è già ordinata
-- ALTRIMENTI:
        Creazione sottoproblemi (Divide):
            dividi gli elementi di A dalla posizione p+1 alla posizione u in due
            sequenze A' e A":
                inserisci in A' gli elementi minori o uguali a A[p]
                inserisci in A" gli elementi maggiori di A[p]
                inserisci A[p] nella sequenza con il minor numero di elementi
        Risoluzione dei sottoproblemi (Impera)
            ordina i dati in A'
            ordina i dati in A"
        Combinazione dei risultati:
            copia la sequenza ordinata A' nell'array A a partire dalla posizione p;
            copia la sequenza ordinata A" nell'array A in coda alla sequenza A'.
```

La fase di creazione dei sottoproblemi richiede una semplice operazione di smistamento dei dati in due distinti insiemi (array). La fase di risoluzione richiede di effettuare ben due ordinamenti, ma su insiemi contenenti un numero di elementi inferiori a quello dell'insieme iniziale. L'operazione di ricombinazione richiede infine una semplice operazione di copiatura delle due sequenze ordinate nell'array originale.

Dal punto di vista implementativo, non è necessario utilizzare due array distinti `A'` e `A"`; infatti, l'operazione di partizione può essere svolta all'interno della porzione di array interessata. In questo modo non sarà nemmeno necessario eseguire un ulteriore ciclo per riportare gli elementi ordinati nell'array originario. Vediamo un esempio.

Dati del problema:
`A={5,8,7,2,4,3}, p=0, u=5`
identificazione dell'elemento di ripartizione:
    `5`
creazione delle due sottosequenze:
    `A'={2,4,3}`

A"={8,7}
aggiunta dell'elemento separatore:
A'={2,4,3}
A"={5,8,7}
ordinamento delle sottosequenze:
A'={2,3,4}
A"={5,7,8}
ricombinazione:
A={A',A"}={2,3,4,5,7,8}

L'ordinamento delle sequenze {2,4,3} e {5,8,7} può essere ottenuto riapplicando lo stesso criterio. Vediamo ad esempio l'ordinamento della sequenza S={5,8,7}

Dati del problema:
S={5,8,7}, p=0, u=2
identificazione dell'elemento di ripartizione:
5
creazione delle due sottosequenze:
S'={}
S"={8,7}
aggiunta dell'elemento separatore:
S'={5}
S"={8,7}
ordinamento delle sottosequenze:
S'={5}
S"={7,8}
ricombinazione:
S={S',S"}={5,7,8}

Ordinamento merge-sort

Il criterio di ordinamento noto come merge-sort utilizza un approccio differente da quello del quick-sort pur mantenendo il principio del *Divide et Impera*. L'idea di base è quella di dividere la sequenza da ordinare in due sottosequenze e ordinarle in modo separato e indipendente. Al termine, le due sottosequenze ordinate vengono fuse in un'unica sequenza complessivamente ordinata.

Formuliamo il problema come segue. Dato un array numerico A, ordinare in modo crescente la sequenza di valori dalla posizione p alla posizione u >= p.

Vediamo lo schema di calcolo per l'ordinamento dei dati in un array A da una posizione p fino ad una posizione u >= p:

```
Algoritmo MergeSort:  Ordina gli elementi di A dalla posizione p alla posizione u
-- SE p = u FINE //la sequenza contiene un solo elemento
-- ALTRIMENTI:
        Creazione sottoproblemi (Divide):
            dividi la sequenza in due sottosequenze:
                A': sequenza degli elementi da posizione p a posizione ⌊(p+u)/2⌋ ;
                A":  sequenza degli elementi da posizione ⌊(p+u)/2⌋+1 a posizione u;
        Risoluzione dei sottoproblemi (Impera)
            ordina i dati in A'
            ordina i dati in A"
        Combinazione dei risultati:
            fondi in modo ordinato le sequenze ordinate A' e A" nell'array A.
```

Il problema di ordinamento viene ancora riproposto, ma su due insiemi contenenti ciascuno un numero di elementi inferiori a quello dell'insieme iniziale. Vediamo un esempio.

Dati del problema:
`A={8,5,7,2,4,3}, p=0, u=5`
creazione delle due sottosequenze:
`A'={8,5,7}`
`A"={2,4,3}`
ordinamento delle sottosequenze:
`A'={5,7,8}`
`A"={2,3,4}`
ricombinazione:
`A={2,3,4,5,7,8}`

L'ordinamento delle sequenze `{8,5,7}` e `{2,4,3}` può essere ottenuto riapplicando lo stesso criterio. Vediamo ad esempio l'ordinamento della sequenza `S={8,5,7}`.

Dati del problema:
`S={8,5,7}, p=0, u=2`
creazione delle due sottosequenze:
`S'={8,5}`
`S"={7}`
ordinamento delle sottosequenze:
`A'={5,8}`
`A"={7}`
ricombinazione:
`A={5,7,8}`

La logica dell'algoritmo può essere implementata come segue:
– Considera l'array come costituito da n sequenze di 1 elemento: $\{a_i\}$ $(i = 0, ..., n-1)$ (ogni sequenza è ordinata in sé);
– fondi in modo ordinato le coppie di sequenze $\{a_0, a_1\}$, $\{a_2, a_3\}$, ... $\{a_{n-2}, a_{n-1}\}$ ottenendo $n/2$ sequenze ordinate di lunghezza 2;
– fondi in modo ordinato le coppie di sequenze $\{a_0 \leftrightarrow a_3\}$, $\{a_4 \leftrightarrow a_7\}$, ... $\{a_{n-4} \leftrightarrow a_{n-1}\}$ ottenendo $n/4$ sequenze ordinate di lunghezza 4;
...
– fondi in modo ordinato le coppie di sequenze $\{a_0 \leftrightarrow a_{\lfloor n/2 \rfloor}\}$, $\{a_{\lfloor n/2 \rfloor + 1} \leftrightarrow a_{n-1}\}$, ottenendo 1 sequenza ordinata di lunghezza n.

Complessità computazionale dell'algoritmo di merge-sort

E' evidente che ad ogni iterazione applichiamo l'algoritmo di fusione ordinata ad ogni coppia da fondere. Il costo di ogni iterazione è pari al numero di elementi coinvolti nelle operazioni di fusione, quindi n. Il numero di iterazioni è pari a $\ln n$ in quanto ad ogni iterazione la lunghezza delle sequenze raddoppia, quindi dopo k iterazioni le sequenze sono lunghe almeno 2^k; il numero di iterazioni necessarie è il più piccolo valore di k tale per cui $2^k \geq n$, ovvero $k \simeq \ln n$.

Ricorsione

La ricorsione è una tecnica di programmazione basata sulla definizione di funzioni (metodi) e tipi di dati (classi) ricorsivi. Una *funzione ricorsiva* è una funzione che, per effettuare il calcolo, utilizza se stessa. Una *classe ricorsiva* (tipo di dato) è una classe che nella sua definizione contiene oggetti del proprio tipo. In questo capitolo consideriamo principalmente le funzioni (metodi) ricorsivi.

Tipi di dato ricorsivi

Immaginiamo di voler definire un dato di tipo `CorpoCeleste` per un programma di astronomia finalizzato alla descrizione del sistema solare. Gli oggetti di nostro interesse sono il Sole, i pianeti e i loro satelliti. Tutti questi oggetti hanno molte caratteristiche in comune: possiedono una massa, un diametro e gravitano intorno ad un altro corpo celeste (unica eccezione il Sole). Secondo questa logica dovremmo descrivere la Terra come un corpo di massa $6 \times 10^{24} Kg$, diametro $12000 Km$ e centro di gravitazione il Sole; la Luna dovrebbe essere descritta come un corpo di massa $7.34 \times 10^{24} Kg$, diametro $3476 Km$ e centro di gravitazione la Terra; ma la Terra è un corpo celeste, quindi per descrivere il corpo celeste Luna, dobbiamo fare riferimento al corpo celeste Terra. Questa relazione fra oggetti della stessa natura innesca la ricorsione quando si passa alla descrizione del tipo di dato (classe). La classe `CorpoCeleste` potrebbe quindi essere definita come segue:

```
class CorpoCeleste {
  double massa;
  double diametro;
  CorpoCeleste centroOrbita;
}
```

La ricorsione entra in gioco per il fatto che per descrivere un oggetto di tipo `CorpoCeleste` facciamo riferimento ad un altro oggetto di tipo `CorpoCeleste`. Naturalmente il Sole deve appartenere alla classe (altrimenti non potremmo indicarlo come centro orbitale della Terra), ma non avendo il Sole un suo centro orbitale, imposteremo a `null` il suo attributo `centroOrbita`.

Alberi gerarchici (o radicati)

Una concreta trasposizione informatica del tipo di dato ricorsivo si può riscontrare negli alberi gerarchici. Un albero gerarchico è una particolare organizzazione logica dei dati i cui elementi sono collegati da una relazione orientata di tipo *padre-figlio*. Gli alberi gerarchici hanno le applicazioni più disparate, i documenti HTML, l'organizzazione dei file su disco e la struttura della distinta base di un prodotto industriale sono solo alcuni esempi di dati organizzati ad albero.

Dato l'ampio campo di applicazione degli alberi gerarchici, non faremo riferimento ad una particolare interfaccia (insieme di metodi definiti sul tipo AlberoGerarchico), ma ci limiteremo ad analizzarne le caratteristiche strutturali logiche ed implementative.

Per nostra comodità possiamo rappresentare gli alberi come un insieme di nodi; all'interno di ciascun nodo può essere depositato un dato (come nelle liste). Il collegamento logico fra i dati è in realtà realizzato mediante un collegamento fra i nodi. In particolare, in un albero gerarchico, i nodi sono collegati fra loro mediante legami logici di *discendenza padre-figlio* che soddisfano le seguenti proprietà:

– esiste un solo nodo che non discende da un padre (*nodo radice*);
– da ogni nodo può discendere un qualunque numero di *nodi figli* $(0, 1, 2, ...)$;
– ogni nodo può discendere da un solo *nodo padre*;

Altri termini utilizzati per classificare i nodi dell'albero sono i seguenti:

– un nodo senza figli è detto *foglia*;
– due nodi che hanno lo stesso padre vengono detti *nodi fratelli*;
– un nodo che non è radice e non è foglia si dice *nodo interno*;

Gli alberi sono *strutture non lineari*, infatti in un albero esiste un solo nodo senza padre (radice), ma possono esistere numerosi nodi terminali (foglie). Una caratteristica evidente di questa struttura dati è che, per ogni dato, esiste un solo percorso di accesso a partire dalla radice.

Notiamo inoltre che la relazione padre-figlio è una relazione non simmetrica per cui possiamo disegnare i collegamenti fra i nodi mediante frecce oppure mettendo la radice in alto e i nodi figli in basso distribuiti su differenti *livelli* in funzione del numero di generazioni che intercorrono fra loro e il nodo radice. Il massimo livello raggiunto da una foglia dell'albero si dice *profondità* dell'albero. La figura 1.1 rappresenta un albero gerarchico e la terminologia adottata.

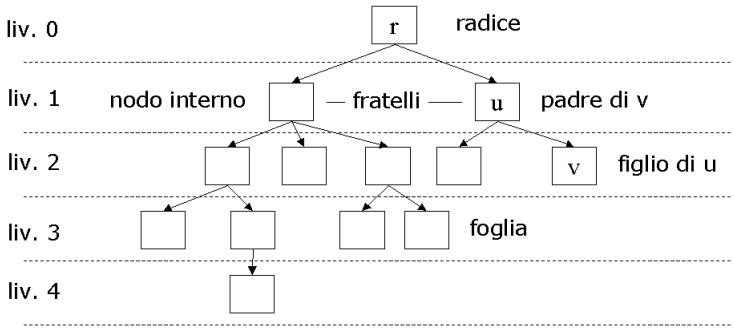

Figura 1.1: Un albero gerarchico di profondità 4.

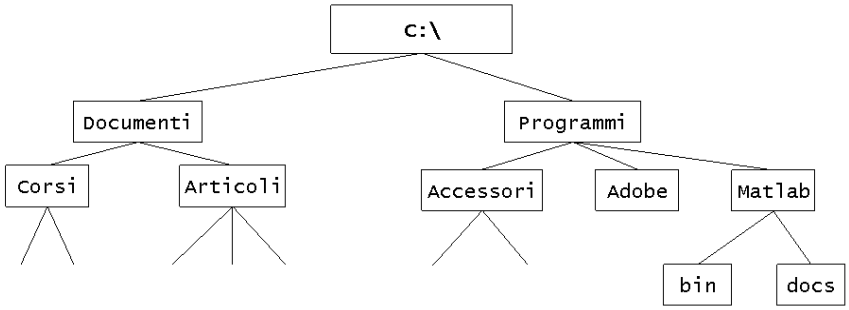

Figura 1.2: Albero delle directory.

Quando i nodi di un albero possono avere al massimo d figli, con $d \geq 2$ fissato, l'albero si dice d-ario. Se le foglie di un albero d-ario sono tutte sullo stesso livello, l'albero si dice *completo*. Nel caso $d = 2$, l'albero si dice *binario*.

Esempi di strutture ad albero

Organizzazione dei file sul disco di un computer (vedi figura 1.2)
– i nodi sono le singole cartelle (directory);
– il dato associato ad ogni cartella è un insieme di file;
– esiste una sola cartella radice e da questa discendono più o meno direttamente tutte le altre dello stesso disco;
– ogni cartella può contenere un qualunque numero di cartelle (nodi figli);

Struttura ricorsiva degli alberi

Osserviamo che un albero può essere costituito da uno o più nodi. Nel primo caso il nodo assume il doppio ruolo di radice e di foglia; se l'albero è costituito da più nodi, allora la radice ha almeno un figlio; ogni figlio della radice preso insieme a tutti i suoi eventuali discendenti conserva la struttura di albero del quale assume il ruolo di radice (figura 1.3);

Un albero è quindi una struttura che può essere definita in modo ricorsivo.
– può essere vuoto;
– Se non è vuoto:
  – contiene un dato (radice);
  – ha un insieme (eventualmente vuoto) di sotto-alberi;

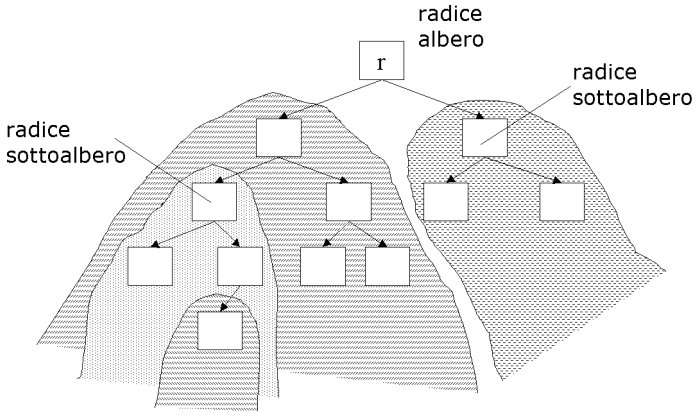

Figura 1.3: Un albero non vuoto contiene un dato e un insieme di (sotto)alberi.

Funzioni ricorsive

Le funzioni ricorsive sono quelle funzioni che contengono sé stesse nella regola di calcolo. Un esempio classico è dato dalla funzione fattoriale $f : N \longrightarrow N,\ f(n) = n!$, che è notoriamente definita come il prodotto di tutti i numeri naturali compresi fra 1 e n; unica eccezione è il fattoriale di 0 che per definizione vale 1. Ad esempio, 5! si calcola come $1 \times 2 \times 3 \times 4 \times 5$ o equivalentemente $5 \times 4 \times 3 \times 2 \times 1$; utilizzando le note proprietà dell'aritmetica possiamo riscrivere il calcolo in questo modo $5! = 5 \times (4 \times 3 \times 2 \times 1)$, ovvero, riutilizzando la definizione di fattoriale: $5! = 5 \times 4!$

In termini pratici questa ultima espressione significa che: possiamo calcolare il fattoriale di 5 se siamo in grado di calcolare il fattoriale di 4 e moltiplicarlo per 5.

La generalizzazione è immediata:

$$n! = \begin{cases} 1, & \text{se } n = 0; \\ n \times (n-1)!, & \text{se } n > 0; \end{cases}$$

La definizione contiene due casi ben distinti. Il caso $n > 0$ è detto *caso ricorsivo* in quanto il calcolo di $n!$ si riconduce al calcolo di $(n-1)!$ (la stessa funzione calcolata su $n-1$) e sarà quindi necessario riapplicare la definizione della funzione (ricorrenza/ricorsione) per risolvere il calcolo richiesto. Il caso $n = 0$ è detto *caso base* in quanto è un caso particolare in cui possiamo fare riferimento alla definizione di $0! = 1$ ottenendo una risposta immediata.

Vediamo come esempio il calcolo di 4! effettuato seguendo la definizione ricorsiva:
– 4 è maggiore di 0, siamo nel caso ricorsivo e perciò $4! = 4 \times 3!$; dobbiamo calcolare 3!
– 3 è maggiore di 0, siamo nel caso ricorsivo e perciò $3! = 3 \times 2!$; dobbiamo calcolare 2!
– 2 è maggiore di 0, siamo nel caso ricorsivo e perciò $2! = 2 \times 1!$; dobbiamo calcolare 1!
– 1 è maggiore di 0, siamo nel caso ricorsivo e perciò $1! = 1 \times 0!$; dobbiamo calcolare 0!
– 0 rientra nel caso base e quindi 0! può essere calcolato in modo diretto, vale 1;

Quindi sostituendo all'indietro otteniamo:
– $0! = 1$;
– $1! = 1 \times 0! = 1 \times 1 = 1$;
– $2! = 2 \times 1! = 2 \times 1 = 2$;
– $3! = 3 \times 2! = 3 \times 2 = 6$.
– $4! = 4 \times 3! = 4 \times 6 = 24$.

Un altro esempio classico di ricorsione, ma meno elementare, è dato dalla successione di Fibonacci $f : N \longrightarrow N$ definita come segue:

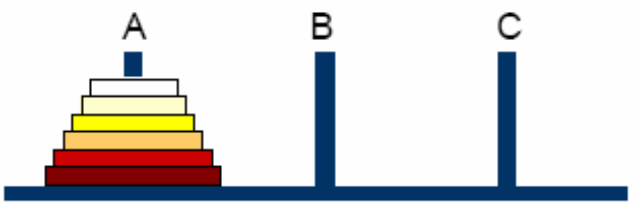

Figura 1.4: La torre di Hanoi in versione ridotta con 6 dischi.

$$f(n) = \begin{cases} 1, & \text{se } n = 0; \\ 1, & \text{se } n = 1; \\ f(n-1) + f(n-2), & \text{se } n > 1; \end{cases}$$

In questo esempio distinguiamo due casi base ($n = 0$ e $n = 1$) ed un caso ricorsivo ($n > 1$). Nel caso ricorsivo la funzione viene riapplicata due volte su valori inferiori a n, vedremo più avanti un'analisi più approfondita delle conseguenze di questo fatto.

Algoritmi ricorsivi

Il problema delle torri di Hanoi è un problema noto ai più. Per chi non lo conoscesse riportiamo la leggenda da cui si dice abbia origine:

> «nel grande tempio di Brahma nella jungla di Hanoi, su di un piatto di ottone, sotto la cupola che segna il centro del mondo, si trovano 64 dischi d'oro puro che i monaci spostano uno alla volta infilandoli in un ago di diamanti, seguendo l'immutabile legge di Brahma: nessun disco può essere posato su un altro più piccolo. All'inizio del mondo tutti i 64 dischi erano infilati in un ago e formavano la Torre di Brahma. Il processo di spostamento dei dischi da un ago all'altro è tuttora in corso. Quando l'ultimo disco sarà finalmente piazzato a formare di nuovo la Torre di Brahma in un ago diverso, allora arriverà la fine del mondo e tutto si trasformerà in polvere»

Il nostro problema consiste nel valutare il tempo necessario per effettuare lo spostamento in questione e decidere se è il caso di prenotare le prossime vacanze al mare. Divideremo il problema in due parti. Per prima cosa cercheremo di capire come può essere svolto il processo di spostamento e in secondo luogo valuteremo il tempo richiesto.

Riformuliamo il problema come segue (vedi figura 1.4). Abbiamo `n` dischi, di diametro tutti diversi pari a $1, 2, ..., n$, impilati in un perno `A` in modo che ogni disco poggi su un disco di diametro superiore. Vogliamo spostare gli `n` dischi in un perno `B` utilizzando come area di sostegno un terzo perno `C` con il vincolo che mai un disco poggi su un altro disco di diametro inferiore.

La soluzione per il caso `n = 1` è ovviamente banale; per `n = 2` è ancora semplice, per `n = 3` comincia a complicarsi (vedi figura 1.5) e per `n = 4` diventa un rompicapo. Il problema ottiene una soluzione molto elegante se proviamo a scomporlo e a ridurlo a problemi di dimensione inferiore.

Osserviamo innanzitutto che, per poter completare la spostamento della torre, il disco più grande e il perno di destinazione devono essere liberi (nessun disco più piccolo nel perno di destinazione o sopra il disco grande) questa situazione è possibile solo se `n=1` oppure tutti gli `n-1` dischi più piccoli sono impilati sul perno di supporto. Quindi, a parte il caso banale di `n=1`, il processo, illustrato in figura 1.6, può essere descritto come segue:

```
-- SE n = 1 ALLORA
    -- sposta il disco da A a B;
```

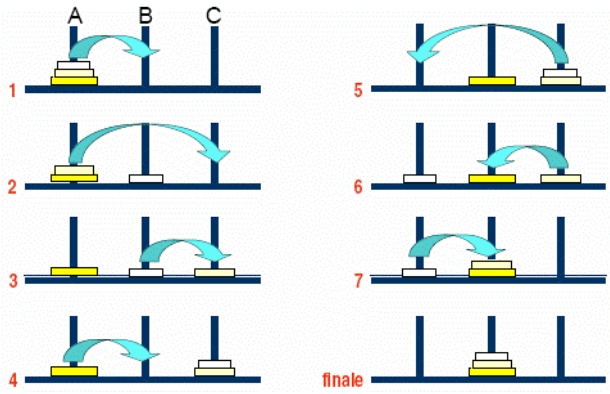

Figura 1.5: Le sette mosse per risolvere il problema con 3 dischi.

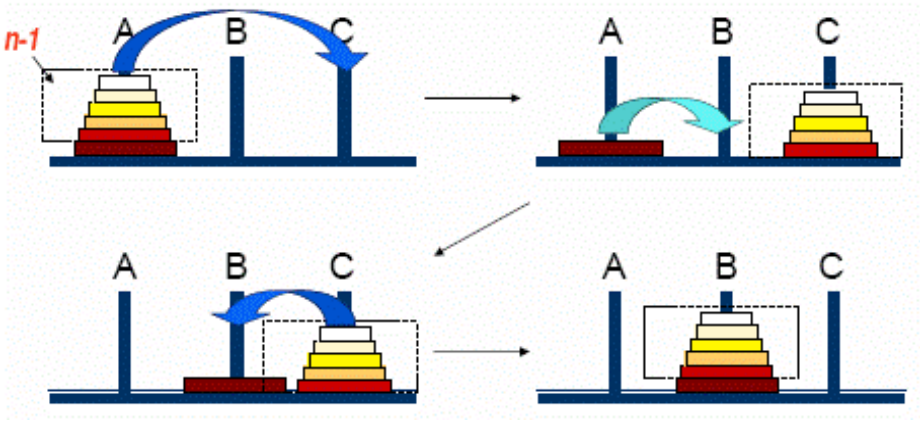

Figura 1.6: Processo risolutivo della Torre con n dischi.

```
-- ALTRIMENTI:
        -- sposta gli n-1 dischi più piccoli da A a C (puoi usare B come supporto);
        -- sposta il disco grande da A a B;
        -- sposta gli n-1 dischi più piccoli da C a B (puoi usare A come supporto).
```

Lo spostamento degli `n-1` dischi più piccoli da un perno `X` ad un altro `Y` usandone un terzo `Z` come supporto è un problema del tutto analogo a quello di partenza, ma la riduzione del numero di dischi lo rende un po' più semplice. Lo spostamento degli `n-1` dischi può allora avvenire seguendo una logica del tutto analoga:

```
-- SE n-1 ALLORA
    -- sposta il disco da X a Y;
-- ALTRIMENTI:
    -- sposta gli n-2 dischi più piccoli da X a Z (puoi usare Y come supporto);
    -- sposta il disco di diametro n-1 da X a Y;
    -- sposta gli n-2 dischi più piccoli da Z a Y (puoi usare X come supporto).
```

Ancora una volta notiamo come il problema si riduce ad un problema analogo posto su un numero inferiore di dischi (quindi più semplice). La somiglianza con il problema del fattoriale dovrebbe a questo punto essere chiara. L'algoritmo generale può allora essere descritto come segue:

```
Sposta(n,X,Y,Z)
    SE n=1 ALLORA
        sposta il disco in cima a X sul perno Y
    ALTRIMENTI
        Sposta(n-1,X,Z,Y)
        sposta il disco in cima a X sul perno Y
        Sposta(n-1,Z,Y,X)
```

Con un breve ragionamento possiamo convincerci che il processo descritto non vìola alcuna regola del gioco: assumendo la regolarità della posizione di partenza notiamo che un disco `k` viene mosso da un perno all'altro solo se tutti i `k-1` dischi più piccoli sono nel frattempo stati spostati sul perno di supporto; nello spostamento di ritorno dei `k-1` dischi più piccoli sopra il disco `k` possono essere coinvolti solo perni con dischi di diametro superiore o uguale a `k` quindi anche in questo caso le regole sugli spostamenti vengono rispettate.

Proviamo ora ad indicare con $f(n)$ il numero di spostamenti elementari necessari per muovere una pila di n dischi da un perno all'altro. Come si deduce facilmente dall'algoritmo, per muovere n dischi occorre effettuare tutti gli spostamenti necessari per muovere due volte una intera pila di $n-1$ dischi più una mossa di spostamento del disco di diametro n.

La funzione $f(n)$ può quindi essere scritta come $f(n) = f(n-1) + 1 + f(n-1) = 2 \cdot f(n-1) + 1 > 2 \cdot f(n-1)$; sfruttando questa relazione otteniamo $f(n) > 2f(n-1) > 4f(n-2) > ... > 2^{n-1}f(1)$. d'altronde sappiamo che $f(1) = 1$ e allora vale $f(n) > 2^{n-1}$. Per essere più precisi, possiamo dimostrare che $f(n) = 2^n - 1$.

Lasciamo come esercizio il compito di valutare il tempo necessario a svolgere lo spostamento dei 64 dischi al ritmo di un disco al secondo e di calcolare quanto tempo rimane a disposizione dell'universo ipotizzando che attualmente abbia circa 10 miliardi di anni.

Definizione di funzioni ricorsive

La ricorsione è una particolare applicazione del principio di *Divide et Impera* ove i sottoproblemi generati durante la scomposizione sono dello stesso tipo del problema originario, ma posti su casi più semplici da risovere. La modalità di scomposizione, il numero di problemi generati ed il modo in cui i risultati vengono ricombinati per ottenere la risposta finale dipendono ovviamente dal problema.

Il principio generale che guida la definizione ricorsiva di una funzione che risolve un problema è il seguente:

Si esaminano gli argomenti della funzione e si distinguono i casi;

- *caso base*: gli argomenti sono tali che è possibile dare una risposta immediata;
- *caso ricorsivo*: il calcolo può essere risolto elaborando le soluzioni dello stesso problema in un insieme di casi più semplici; questi casi daranno origine a loro volta ad insiemi di problemi più semplici e così via fino a raggiungere problemi così semplici da poter essere risolti in modo diretto (casi base).

La definizione ricorsiva di una funzione deve contenere almeno un caso base e almeno un caso ricorsivo. La ricorsione si dice *indiretta* se il caso ricorsivo utilizza una funzione che a sua volta utilizza in modo diretto o indiretto la funzione in questione.

Proprietà delle funzioni ricorsive

Una definizione ricorsiva, per essere ben posta, deve godere delle seguenti proprietà:

- Deve essere *non-ambigua*, nel senso che ogni scelta degli argomenti nel dominio della funzione deve condurre alla selezione di uno e un solo caso, base o ricorsivo. Ovvero, il dominio della funzione viene *partizionato* in un certo numero di sottoinsiemi mutuamente disgiunti e ad ogni sottoinsieme corrisponde un solo caso (ad esempio, per ogni intero n, il calcolo di $n!$ non è mai riconducibile a due casi distinti, tra di loro in contraddizione).

- Deve essere *non-circolare*, nel senso che non deve mai definire il calcolo di una funzione su un argomento mediante il calcolo della stessa funzione sullo stesso argomento (ad esempio, il calcolo di $n!$ non viene mai ricondotto al calcolo di $n!$ per nessun possibile valore di n).

- Deve essere *completa*, nel senso che la catena di ragionamenti che induce deve sempre ricondursi alla fine, a un caso base. Ovvero, ogni sequenza di chiamate prodotta dai casi ricorsivi, deve "convergere" a chiamate della funzione in un caso base (ad esempio, il calcolo di $n!$ non viene mai ricondotto al calcolo di $k!$ senza che la definizione tenga conto di come calcolare $k!$).

Implementazione di metodi ricorsivi in Java

Un metodo ricorsivo è un metodo che, direttamente o indirettamente, può chiamare sé stesso. La funzione fattoriale, la cui definizione ricorsiva è stata mostrata in precedenza, può essere calcolata mediante il metodo ricorsivo `fattoriale` riportato qui sotto:

```
public static int fattoriale(int n) {
  int f;

  if ( n == 0) { f = 1; }
  else { f = n * fattoriale(n-1); }

  return f;
}
```

Il metodo `fattoriale` è ricorsivo perché, se la condizione `n == 0` (caso base) è falsa, allora la sua esecuzione richiede un'altra esecuzione del metodo `fattoriale` stesso.

L'implementazione proposta del metodo `fattoriale` segue una struttura tipica dei metodi ricorsivi:

- il metodo usa la variabile `f` per memorizzare il risultato del calcolo;

- il metodo contiene una istruzione di selezione `if-else` per distinguere i casi (base o ricorsivo) in cui rientrano i dati del problema;

- i blocchi attribuiti ai singoli casi dell'istruzione di selezione calcolano il valore della funzione applicando le regole relative allo specifico caso; il risultato viene poi assegnato alla variabile `f`;

- il metodo termina con la restituzione del valore calcolato e memorizzato in `f`.

Una scrittura più compatta del metodo potrebbe essere la seguente:

```
public static int fattoriale(int n) {
  if ( n == 0) return 1;
  else return n * fattoriale(n-1);
}
```

Ricorsione lineare e non lineare

Fino ad ora abbiamo mostrato solo esempi di *ricorsione lineare*, nel senso che ciascuna chiamata del metodo ricorsivo causa direttamente, al massimo, *una* altra chiamata ricorsiva.

Si parla invece di *ricorsione non lineare* quando una chiamata del metodo ricorsivo può causare direttamente anche due o più chiamate ricorsive.

Possiamo evidenziare la differenza fra le due situazioni tracciando un *diagramma delle chiamate* in cui ad ogni chiamata del metodo corrisponde un blocco rettangolare in cui indichiamo gli argomenti con cui il metodo viene chiamato; il blocco relativo alla chiamata di un metodo viene collegato con una freccia ai blocchi relativi alle chiamate ricorsive effettuate durante la sua esecuzione.

Nel caso di ricorsione lineare, il diagramma delle chiamate forma una catena in cui il blocco iniziale corrisponde alla prima chiamata, l'ultimo blocco rappresenta l'ultima chiamata con i parametri che ricadono nel caso base per cui non ci saranno altre chiamate ricorsive. Si veda, ad esempio, la figura 1.7 che rappresenta il diagramma delle chiamate per l'esecuzione del metodo `fattoriale(3)`.

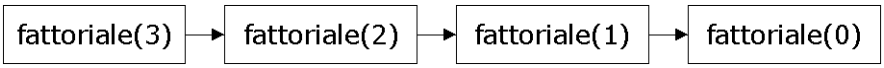

Figura 1.7: Diagramma delle chiamate di `fattoriale(3)`.

Nella ricorsione non lineare il diagramma delle chiamate non corrisponde più ad una catena lineare in quanto da ogni blocco possono seguire due o più blocchi generando una struttura ramificata (struttura ad albero). Esempio classico di questa situazione è il metodo per il calcolo della successione di Fibonacci:

```
public static int fibonacci(int n) {
  int f;
  if ( n == 0) f = 1;
  else if ( n == 1) f = 1;
  else f = fibonacci(n-1) + fibonacci(n-2);
  return f;
}
```

Nella suo caso ricorsivo, il metodo `fibonacci` viene ri-chiamato due volte. Si veda in figura 1.8, ad esempio, il diagramma delle chiamate prodotto dall'esecuzione del metodo `fibonacci` per `n=5`.

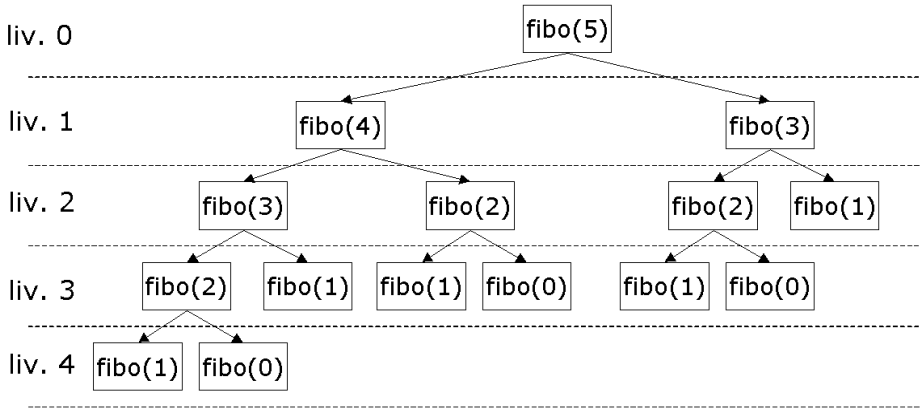

Figura 1.8: Diagramma delle chiamate del metodo `fibonacci` (per motivi di spazio abbreviato in `fibo`).

Se osserviamo la struttura del diagramma, notiamo che molte chiamate del metodo vengono ripetute più volte con gli stessi argomenti; ad esempio, `fibonacci(3)` viene eseguito due volte con tutto il carico di chiamate che ne segue in entrambi i casi. Dal diagramma deduciamo inoltre che `fibonacci(n)` induce `n` livelli di chiamate e, per almeno i primi `n/2` livelli, il numero di blocchi (record di attivazione) raddoppia ad ogni livello indicando un numero minimo di chiamate pari a $2^{n/2} = (\sqrt{2})^n$. Questa non è una buona notizia in quanto indica che il tempo di calcolo richiesto per il calcolo di `fibonacci(n)` è almeno $(\sqrt{2})^n$. L'elevato tempo di calcolo del metodo ricorsivo è dovuta non tanto ad una intrinseca difficoltà del problema, quanto alla inefficiente definizione del metodo stesso. Sappiamo invece che con un semplice algoritmo iterativo può essere calcolato in tempo $O(n)$[1]. Si veda ad esempio il codice seguente che svolge al massimo n iterazioni con un numero fisso di operazioni per ogni iterazione. Con opportune tecniche di memorizzazione dei risultati intermedi, potremmo ottenere la stessa efficienza del metodo iterativo anche con un definizione ricorsiva.

```
public static int fibonacci(int n) {
  int f0 = 1;
  int f1 = 1;
  int fn = 1;
  for (int i = 2; i <= n; i++) {
    fn = f0 + f1;
    f0 = f1;
    f1 = fn;
  }
  return fn;
}
```

Ricerca e ordinamento

Alcuni algoritmi "classici" quali la ricerca binaria ed il merge-sort possono essere implementati in modo molto semplice mediante la ricorsione.

Ricerca binaria

Indicati con `a` l'array in cui cercare un dato `x`, e con `primo` e `ultimo` gli indici che delimitano lo spazio di ricerca, l'algoritmo segue la seguente logica ricorsiva:

- Indichiamo con `centro` la posizione centrale $\left\lfloor \frac{\texttt{primo}+\texttt{ultimo}}{2} \right\rfloor$ dello spazio di ricerca;
- **Caso base 1:** lo spazio di ricerca è vuoto (`primo`>`ultimo`)
  - la ricerca è fallita;
- **Caso base 2:** l'elemento cercato coincide con quello in posizione `centro`
  - la ricerca ha avuto successo la posizione cercata è `centro`;
- **Caso ricorsivo 1**: l'elemento cercato è minore dell'elemento di posizione `centro`
  - la risposta è data dalla ricerca del dato nello spazio dalla posizione `primo` fino alla posizione `centro-1`;
- **Caso ricorsivo 2**: l'elemento cercato è maggiore dell'elemento di posizione `centro`
  - la risposta è data dalla ricerca del dato nello spazio dalla posizione `centro+1` fino alla posizione `ultimo`;

Il seguente codice mostra l'implementazione ricorsiva di questa strategia di ricerca.

```
public static int ricercaBinaria(int[] a, int primo, int ultimo, int x) {
 int posizione;
 int centro = (primo+ultimo)/2;
```

[1] Per completezza aggiungiamo che algoritmi basati sul calcolo di potenze di matrici, calcolano l'ennesimo termine della successione in tempo $O(\lg n)$.

```
 if (primo > ultimo)
              posizione = -1;
 else if ( x == a[centro] )
              posizione = centro;
 else if ( x < a[centro] )
              posizione = ricercaBinaria(a,primo,centro-1,x);
 else
              posizione = ricercaBinaria(a,centro+1,ultimo,x);

 return posizione;
}
```

Si osservi il vantaggio di specificare nel prototipo del metodo gli estremi dello spazio di ricerca. Infatti, una implementazione ricorsiva di un metodo con prototipo

```
int ricercaBinaria(int[] a, int n, int x)
```

in cui si dà per scontato che il primo elemento dello spazio di ricerca occupa la posizione zero dell'array, costringerebbe a copiare in un nuovo array gli elementi su cui proseguire la ricerca.

La ricorsione indotta dalla ricerca binaria ricorsiva è lineare. Il metodo contiene due chiamate distinte, ma al massimo una viene eseguita. Ad ogni chiamata la dimensione dello spazio di ricerca viene dimezzata e un caso base viene raggiunto in non più di $\log_2 n$ chiamate.

Merge-sort

Indicati con `a` l'array da ordinare e con `primo` e `ultimo` gli indici che delimitano lo spazio da ordinare, l'algoritmo segue la seguente logica ricorsiva:

- **Caso base 1:** non ci sono elementi da ordinare (`primo`>`ultimo`)
  - non c'è nessuna operazione da effettuare;
- **Caso base 2:** la porzione di array da ordinare contiene un solo elemento (`primo=ultimo`)
  - non c'è nessuna operazione da effettuare;
- **Caso ricorsivo**: la porzione di array da ordinare contiene almeno due elementi (`primo`<`ultimo`)
  - calcola la posizione centrale dell'array `centro` = $\left\lfloor \frac{\texttt{primo}+\texttt{ultimo}}{2} \right\rfloor$;
  - ordina l'array `a` dalla posizione `primo` alla posizione `centro`;
  - ordina l'array `a` dalla posizione `centro+1` alla posizione `ultimo`;
  - ricombina le due porzioni con una fusione ordinata;

Il seguente codice mostra l'implementazione ricorsiva di questa strategia di ordinamento.

```
public static void mergeSort(int[] a, int primo, int ultimo) {
 int centro = (primo+ultimo)/2;

 if (primo > ultimo) ; // do nothing
 else if (primo == ultimo) ; // do nothing
 else {
   mergeSort(a,primo,centro);
   mergeSort(a,centro+1,ultimo);
   fusioneOrdinata(a,primo,centro,ultimo);
 }
}

public static void fusioneOrdinata(int[] a, int primo, int centro, int ultimo) {
```

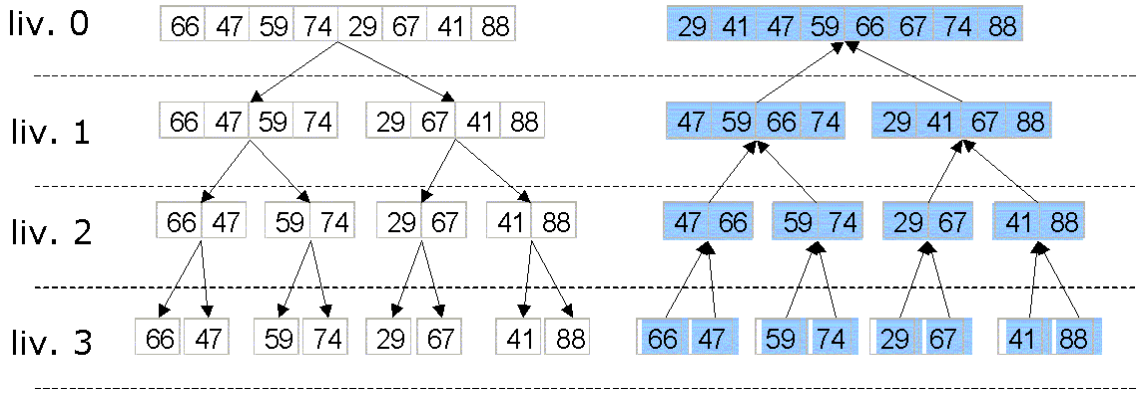

Figura 1.9: Diagramma delle chiamate ricorsive e delle fusioni operate dal merge-sort.

```
//assume a ordinato:
// dalla posizione primo fino a centro
// dalla posizione centro+1 fino a ultimo
  int i1,i2,i3;
  int[] c = new int[ultimo-primo+1];
  i1 = primo;
  i2 = centro+1;
  i3=0;
  while (i1<=centro && i2<=ultimo) {
    if (a[i1] < a[i2]) { c[i3] = a[i1]; i1++; i3++; }
    else { c[i3] = a[i2]; i2++; i3++; }
  }
  while (i1<=centro) { c[i3] = a[i1]; i1++; i3++; }
  while (i2<=ultimo) { c[i3] = a[i2]; i2++; i3++; }
  for (i3=0; i3<c.length; i3++) a[primo+i3] = c[i3];
}
```

La ricorsione consiste nell'ordinare le due porzioni di array con la medesima tecnica di suddivisione-ordinamento-fusione. Poiché ad ogni chiamata ricorsiva viene dimezzato il numero degli elementi da ordinare, siamo sicuri che dopo un numero finito di ricorsioni si arriverà al caso base che consente di completare il calcolo.

La ricorsione è non lineare in quanto nel caso ricorsivo vengono effettuate due chiamate ricorsive. Quindi il diagramma delle chiamate e fatto ad albero (vedere figura 1.9). Tuttavia, questo algoritmo non presenta le trappole della ricorsione non lineare presentata dalla successione di Fibonacci in quanto non viene mai ripetuto un ordinamento sullo stesso insieme di dati. La complessità computazionale dell'algoritmo in forma ricorsiva è la stessa già discussa nel capitolo .

Visita di alberi

La definizione ricorsiva di albero si presta bene a introdurre alcune tecniche di esplorazione in profondità definite in modo ricorsivo. Ricordando che la visita in profondità richiede che, dato un nodo i suoi sottoalberi (nodi discendenti) siano visitati prima dei nodi fratelli (con relativi discendenti), rimane da decidere l'ordine con cui si effettua la visita di un nodo rispetto ai suoi discendenti. Ci sono tre combinazioni diverse che, per semplicità espositiva, discutiamo solo nel caso di alberi binari. Solo una delle tre è specifica degli alberi binari, le altre possono essere facilmente generalizzate al caso di nodi con un numero arbitrario di figli.

– *preordine:* il nodo è visitato prima dei suoi discendenti;
– *inordine (simmetrica):* il nodo è visitato dopo il discendente sinistro, ma prima del discendente destro;
– *postordine:* il nodo è visitato dopo i suoi discendenti;

Visita in preordine

La visita in *preordine* è equivalente alla visita in profondità, come definita in precedenza, e prevede che la radice dell'albero sia visitata per prima, poi il figlio sinistro (con discendenti) e in seguito il figlio destro (con i discendenti). La definizione ricorsiva dell'algoritmo è molto semplice in quanto la ricorsione elimina la necessità di gestire l'insieme dei nodi mediante la pila. Ne vediamo una possibile implementazione:

```
Algoritmo:  visitaPreordine(Albero A)
 SE A è vuoto ALLORA FINE
 ALTRIMENTI
    stampa il dato nella radice dell'albero A
    esegui visitaPreordine(sottoalbero sinistro di A)
    esegui visitaPreordine(sottoalbero destro di A)
 FINE_SE
```

Visita inordine (simmetrica)

La visita *simmetrica* prevede che ogni nodo dell'albero sia effettivamente visitato dopo la visita del sottoalbero sinistro e prima della visita del sottoalbero destro. Questo schema è descritto molto semplicemente dal seguente algoritmo ricorsivo:

```
Algoritmo:  visitaSimmetrica(Albero A)
 SE A è vuoto ALLORA FINE
 ALTRIMENTI
    esegui visitaSimmetrica(sottoalbero sinistro di A)
    stampa il dato nella radice dell'albero A
    esegui visitaSimmetrica(sottoalbero destro di A)
 FINE_SE
```

Visita in postordine

La visita in *postordine* prevede che ogni nodo dell'albero sia effettivamente visitato solo dopo la visita dei figli sinistro e destro:

```
Algoritmo:  visitaPostordine(Albero A)
 SE A è vuoto ALLORA FINE
 ALTRIMENTI
    esegui visitaPostordine(sottoalbero sinistro di A)
    esegui visitaPostordine(sottoalbero destro di A)
    stampa il dato nella radice dell'albero A
 FINE_SE
```

La figura 1.10 mostra le diverse sequenze di visita di uno stesso albero secondo i metodi simmetrico e di postordine.

Riportiamo di seguito due proposte applicative per algoritmi ricorvisi e strutture ad albero:

Si consideri la struttura ad albero delle directory (cartelle) di un disco fisso. Progettare un algoritmo per stampare la lista 'esplosa' delle directory con relative sottodirectory.

Si consideri una struttura ad albero che rappresenti la distinta base di un prodotto. In ogni nodo è riportato l'insieme dei semilavorati necessari per l'assemblaggio e il tempo richiesto per l'operazione da quando i semilavorati sono disponibili. Per le materie prime il tempo indica il tempo di approvvigionamento. Progettare un algoritmo per calcolare il tempo di completamento del prodotto finito.

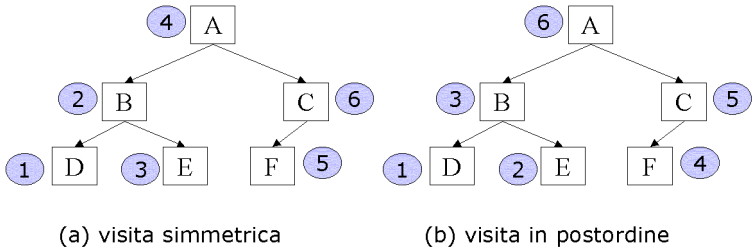

Figura 1.10: Sequenza di visite in profondità (simmetrica e postordine).

Backtracking

Il problema del resto

Supponiamo di essere un progettista di distributori automatici di bevande, merendine e tramezzini e di voler costruire una macchina che sia in grado di fornire il resto agli avventori che richiedono beni di consumo per un importo inferiore a quello inserito. In questo contesto non siamo interessati alla tecnologia con cui la macchina potrà essere realizzata, ma alla logica con la quale essa deciderà come comporre il resto da restituire al cliente. Tale calcolo dovrà essere svolto sulla base delle monete o banconote effettivamente disponibili nella cassa del distributore oltre che dell'importo dovuto.

Immaginiamo per semplicità che il nostro distributore gestisca solo monete di taglio 50, 20, 10 e 5 centesimi e indichiamo con una sequenza di 4 numeri interi non negativi la disponibilità di cassa del distributore. Ad esempio `cassa:{3,4,5,2}` indica la disponibilità di 3 monete da 50, 4 monete da 20, 5 monete da 10 e 2 monete da 5 centesimi. Indichiamo con la stessa notazione ogni possibile prelievo costituito da un sottoinsieme delle monete disponibili in cassa, ad esempio `prelievo:{1,2,0,1}` indica un prelievo dalla cassa di 1 moneta da 50, 2 monete da 20, 0 monete da 10 e 1 moneta da 5 centesimi. E' evidente la distinzione fra i prelievi che costituiscono una soluzione ammissibile del problema e quali no. In linea di principio potremmo risolvere il problema provando a effettuare tutti i possibili prelievi verificando uno per uno se sono ammissibili oppure no. Se durante il processo di generazione ne troviamo uno ammissibile, possiamo fermarci e stampare la soluzione, altrimenti dopo averli provati tutti, potremo affermare che non è possibile produrre l'importo richiesto.

La questione è: quanti prelievi distinti possiamo effettuare? Nel nostro esempio possiamo effettuare $4 \cdot 5 \cdot 6 \cdot 3 = 360$ prelievi distinti (4 quantità distinte di monete da 50 per 5 quantità distinte di monete da 20 etc.). Il numero può sembrare piccolo, ma se consideriamo n tagli di monete e m monete in cassa per ogni taglio, otteniamo $n^{(m+1)}$ che può essere un numero sufficientemente grande da indurci a trovare algoritmi risolutivi più efficienti.

Un algoritmo risolutivo di tipo ingordo (greedy)

La prima idea che ci viene in mente (probabilmente) è quella di raccogliere la moneta di taglio maggiore che risulti minore o uguale all'importo da restituire; tale moneta produce una riduzione del debito, se questo si azzera abbiamo finito altrimenti procederemo con la selezione della più grossa moneta di importo inferiore o uguale al debito rimanente e così via fino al completamento dell'operazione.

Analisi dell'algoritmo Analizziamo i risultati dell'algoritmo su alcuni casi particolari con un importo da produrre di 80 centesimi.

- Se `cassa` ha i valori `{3,4,2,3}`, l'algoritmo produce la seguente soluzione: `{1,1,1,0}`, ma altre soluzioni potrebbero essere `{0,4,0,0}` oppure `{1,0,2,2}` e altre ancora.

- Se `cassa` ha i valori `{2,2,0,1}`, il problema non ha soluzione. L'algoritmo fallisce il proprio compito dopo avere prodotto il seguente prelievo `{1,1,0,1}`. Il fallimento dell'algoritmo comunque non dimostra l'impossibilità di trovare una soluzione (vedere caso seguente).

- Se `cassa` ha i valori `{2,4,0,0}`, l'algoritmo fallisce il proprio compito in quanto preleva subito una moneta da 50 centesimi, poi una da 20, trovandosi poi nell'impossibilità di completare l'importo rimanente con le ultime monete da 20 centesimi. E' facile invece osservare che una soluzione è data dal vettore `{0,4,0,0}` che utilizza le quattro monete da 20 centesimi.

Esercizio. Dimostrare con un esempio che un algoritmo che segue una regola analoga iniziando dalle monete di taglio più piccolo, può ugualmente fallire anche quando il problema ammette una soluzione.

Procedere per correzioni successive (trial-and-error) Le ragioni della fallacità dell'algoritmo sopra descritto stanno nella sua miopia nel costruire la soluzione. L'insieme delle monete è infatti costruito scegliendo una moneta alla volta e senza tenere conto delle conseguenze che tale scelta avrà sulle possibilità di scelta future. Solo quando è impossibile proseguire ci si accorge che una delle scelte precedenti è sbagliata. Per eliminare il dubbio se sia il problema senza soluzione piuttosto che il tentativo errato, diventa essenziale poter "riavvolgere il nastro" (*backtracking*) delle decisioni già prese per considerare scelte alternative. Il termine backtracking può essere tradotto con l'espressione "tornare indietro sui propri passi". Vediamo come si applica il principio al nostro problema.

Processo di costruzione della soluzione (in avanti) Definiamo il processo di costruzione di una soluzione come segue:

1) decidere quante monete da 50 centesimi vanno utilizzate;
2) decidere quante monete da 20 centesimi vanno utilizzate;
3) decidere quante monete da 10 centesimi vanno utilizzate;
4) decidere quante monete da 5 centesimi vanno utilizzate;
5) si verifica la correttezza della soluzione;

Ad ogni passo in avanti possiamo decidere di utilizzare un qualunque numero di monete comprese fra 0 e il massimo quantitativo che ci permette, sommato alle monete già selezionate, di non superare l'importo originale. Ad esempio, se l'importo originale è 85 centesimi e il vettore `nMonete` è `{1,3,0,1}`, allora al passo 1 le scelte disponibili sono 0 oppure 1. Le scelte disponibili al passo 2 dipendono dalle scelte fatte al passo 1: se avessimo scelto 0 monete da 50, allora potremmo scegliere un qualunque numero fra 0 e 3 monete da 20; altrimenti avrebbe senso scegliere solo fra 0 e 1 moneta da 20 centesimi. Notiamo che questo accorgimento ci permette di sfrondare numerosi prelievi certamente non ammissibili.

Processo di backtracking Quando non ci sono più decisioni disponibili (perchè siamo in uno dei primi 4 passi, ma non abbiamo scelte a disposizione oppure siamo al passo 5) dobbiamo arrestare il processo in avanti. Se ci accorgiamo che abbiamo raggiunto una soluzione non accettabile perchè la somma accumulata non corrisponde all'importo richiesto, dobbiamo tornare indietro nel processo di costruzione. Di fatto dobbiamo annullare tutte le decisioni prese fino al passo in cui sono disponibili scelte alternative non ancora prese in considerazione. Da qui possiamo riprendere il processo in avanti.

E' importante, durante il processo in avanti, rappresentare in modo adeguato l'insieme delle scelte disponibili in modo che le decisioni prese possano essere definitivamente eliminate dall'insieme. Questo ci permetterà di evitare di ripetere la stessa scelta durante il processo di backtracking.

Ad esempio, consideriamo il caso `cassa:{1,3,0,1}` con un importo da restituire di 65 centesimi. Per mettere in evidenza quali decisioni sono già state prese e quali sono ancora da prendere rappresenteremo il prelievo con un vettore di lunghezza variabile. I vettori `{}`, `{1}`, `{1,2}`, rappresentano rispettivamente prelievi in cui sono presenti: nessuna moneta, una sola moneta da 50, una moneta da 50 e 2 da 20. Le monete non espressamente indicate sono da considerare in quantità nulla.

La ricerca della soluzione del problema posto può quindi procedere come segue:

inzializzazione: `prelievo={}`
passo **1**: Decisioni disponibili: `{0,1}`; decisione scelta: `1`; Decisioni rimanenti: `{0}`; `prelievo={1}`
passo **2**: Decisioni disponibili: `{0}`; decisione scelta: `0`; Decisioni rimanenti: `{}`; `prelievo={1,0}`
passo **3**: Decisioni disponibili: `{0}`; decisione scelta: `0`; Decisioni rimanenti: `{}`; `prelievo={1,0,0}`
passo **4**: Decisioni disponibili: `{0,1}`; decisione scelta: `1`; Decisioni rimanenti: `{0}`; `prelievo={1,0,0,1}`
passo **5**: il prelievo non è ammissibile, esegui backtracking
backtracking: `prelievo={1,0,0}`;
passo **4**: Decisioni disponibili: `{0}`; decisione scelta: `0`; Decisioni rimanenti: `{}`; `prelievo={1,0,0,0}`;
passo **5**: il prelievo non è ammissibile, esegui backtracking
backtracking: `prelievo={1,0,0}`;
passo **4**: nessuna decisone possibile, esegui backtracking
backtracking: `prelievo={1,0}`;
passo **3**: nessuna decisone possibile, esegui backtracking
backtracking: `prelievo={1}`;
passo **2**: nessuna decisone possibile, esegui backtracking
backtracking: `prelievo={}`;
passo **1**: Decisioni disponibili: `{0}`; decisione scelta: `0`; Decisioni rimanenti: `{}`; `prelievo={0}`;
passo **2**: Decisioni disponibili: `{0,1,2,3}`; decisione scelta: `3`; Decisioni rimanenti: `{0,1,2}`; `prelievo={0,3}`;
passo **3**: Decisioni disponibili: `{0}`; decisione scelta: `0`; Decisioni rimanenti: `{}`; `prelievo={0,3,0}`;
passo **4**: Decisioni disponibili: `{0,1}`; decisione scelta: `1`; Decisioni rimanenti: `{0}`; `prelievo={0,3,0,1}`;
passo **5**: il prelievo corrisponde alla somma richiesta: stampa soluzione e arresto procedura.

Criteri di selezione delle decisioni disponibili

Un criterio di selezione delle decisioni da prendere in considerazione è tanto più efficiente quanto più riesce a escludere le decisioni che non porteranno ad un prelievo accettabile senza escludere quelle che porteranno ad una o più prelievi accettabili. Nella discussione fatta sopra abbiamo adottato il criterio di considerare quantità di monete da 0 al numero massimo che permette di non superare (eventualmente eguagliare) l'importo richiesto. Le ragioni di questa scelta sono evidenti (visto che il processo di costruzione aggiunge monete, se superiamo l'importo richiesto, non potremo certo aspettarci una soluzione corretta alla fine). Il criterio può essere ulteriormente raffinato imponendo un numero minimo di monete di un certo tipo in modo tale che l'importo complessivo formato dalle monete dei tagli che dobbiamo ancora prendere in considerazione sia sufficiente a colmare la somma ancora mancante. Ad esempio, se dobbiamo formare un importo di 90 centesimi con un vettore `nMonete={1,4,1,1}` e al passo 1 abbiamo scelto 1 moneta da 50, allora al passo 2 dobbiamo scegliere almeno 2 monete da 20 centesimi. Se al passo 1 scegliessimo 0 monete da 50, allora al passo 2 dovremmo scegliere almeno 4 monete da 20 centesimi.

Schema generale dell'algoritmo di backtracking

La tecnica di backtracking è una tecnica che viene utilizzata quando un algoritmo deve produrre la soluzione di un problema procedendo per tentativi. Una *soluzione* consiste in una *sequenza di decisioni* (nel nostro esempio le monete da prelevare dalla cassa del distributore automatico) costruita operando una decisione alla volta. Se una soluzione soddisfa tutte le caratteristiche richieste dal problema si dice *ammissibile*, altrimenti se anche un solo requisito non è soddisfatto si dice *inammissibile*. In alcuni casi l'obiettivo è la produzione di almeno una soluzione ammissibile, in altri casi si tratta di generare tutte le possibili soluzioni ammissibili per selezionarne la migliore. Se al termine del processo di generazione non è stata individuata alcuna soluzione ammissibile, allora potremo dire che il problema non ammette soluzioni.

Diamo ora uno schema generale per un algoritmo che applica il backtracking:

```
Algoritmo generale
 inizializza la soluzione S = ∅
 inzializza il contatore dei passi decisionali i ← 0
```

```
D_i ← insieme delle decisioni disponibili al passo i
WHILE i ≥ 0
    SE D_i ≠ ∅ ALLORA // avanza
        estrai una decisione d dall'insieme D_i
        applica d alla soluzione S
        i ← i + 1
        D_i ← insieme delle decisioni disponibili al passo i
    ALTRIMENTI SE Sèè ammissibile
        RETURN S
    ALTRIMENTI // backtrack
        annulla l'ultima decisione nella soluzione S
        i ← i − 1
    FINE_SE
FINE_WHILE
RETURN NULL
```

Backtracking e ricorsione

Gli algoritmi di backtracking si formulano spesso in modo del tutto naturale usando la ricorsione. L'idea è che ogni decisione presa incrementa una soluzione parziale; ad ogni incremento della soluzione il problema può essere riproposto partendo da una situazione più vicina ad una soluzione finale ammissibile oppure ad una soluzione inammissibile. I casi base sono quelli in cui non ci sono più decisioni da prendere e la soluzione non va più incrementata perchè soddisfa i requisiti richiesti (ammissibilità) oppure risulta impossibile proseguire la ricerca di una soluzione ammissibile.

Lo schema generale è il seguente (La prima chiamata dell'algoritmo avviene con $S = \emptyset$.):

```
PROBLEMA(S)
    SE Sèè ammissibile ALLORA
        RETURN S
    ALTRIMENTI
        D ← insieme delle decisioni disponibili
        PER OGNI d ∈ D
            S^1 ← nuova soluzione ottenuta applicando d alla soluzione S
            S^2 ←PROBLEMA(S^1)
            SE S^2èè ammissibile ALLORA: RETURN S^2
        FINE_PER_OGNI
        RETURN NULL
    FINE_SE
```

Il problema delle otto regine

La formulazione del problema è semplice per chi conosce gli scacchi: posizionare otto regine su una scacchiera in modo che nessuna di esse ne "minacci" un'altra. Per chi non ha confidenza con il gioco degli scacchi precisiamo che una scacchiera é una griglia di 8 righe e 8 colonne; una regina è un pezzo che occupa una casella e da lì può muoversi (quindi minacciare) di un numero arbitrario di caselle nelle direzioni orizzontale, verticale e diagonali (vedi Figura 1.11).

Il problema è un classico problema combinatorio che risale a prima del IXX secolo (studiato da Gauss) che può essere formulato su scacchiere di dimensione arbitraria (vedi figura 1.12).

Approccio alla soluzione

Ricerca esaustiva della soluzione: Si può provare a piazzare le regine sulla scacchiera in tutte le combinazioni possibili di 8 caselle fino a che non si trova una soluzione. Questa tecnica si rivela sicuramente inefficiente in quanto produce un numero di soluzioni pari a

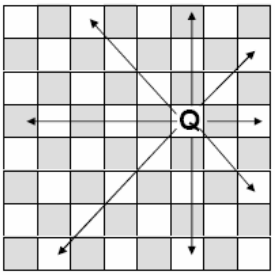

Figura 1.11: Mosse della regina.

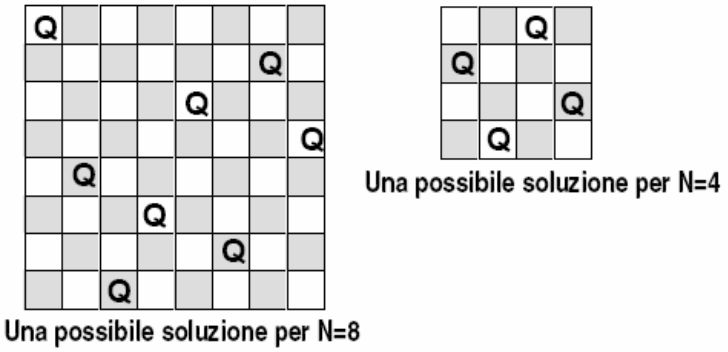

Figura 1.12: Soluzioni per problemi di diversa dimensione.

$$\left(\begin{array}{c} 64 \\ 8 \end{array} \right) = \frac{64!}{8!(64-8)!} = 4,426,165,368 \simeq 4 \times 10^9$$

la maggior parte delle quali sono inammissibili. Lasciamo per esercizio descrivere un algoritmo in grado di generare tutte le combinazioni possibili e di verificare su ogni combinazione la sua ammissibilità.

Restringimento del campo di ricerca Si può ridurre il numero di tentativi generati considerando che, molte delle combinazioni indicate sopra hanno parti comuni dalle quali si può immediatamente dedurre l'inutilità di sviluppare i singoli tentativi. Ad esempio, la considerazione che in una soluzione ammissibile le regine occupano 8 colonne distinte, ci permette di eliminare a priori tutte quelle soluzioni parziali in cui due o più regine occupano una stessa colonna. Il numero di soluzioni alternative cala drasticamente di un fattore 10^2 a

$$8^8 = 16,777,216 \simeq 1 \times 10^7$$

E' altrettanto facile vedere che le 8 regine devono occupare anche righe distinte riducendo ulteriormente il numero di possibilità a

$$8! = 40,320 \simeq 4 \times 10^4$$

Algoritmo risolutivo

Il problema delle otto regine presenta numerose similitudini con il problema del resto visto in precedenza. Costruiamo una soluzione in otto passi; al k-esimo passo (k=1,...,8) tentiamo di piazzare una regina sulla colonna k; si tratta di decidere in che riga piazzarla. L'insieme delle righe disponibili per la regina in colonna k dipende dalle regine già piazzate. Se alla colonna k non abbiamo decisioni disponibili viene

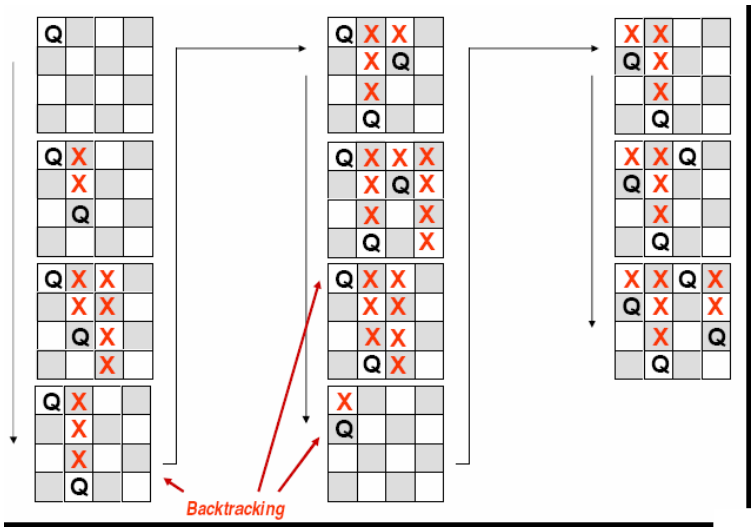

Figura 1.13: Backtracking per un problema di dimensione 4.

effettuato il backtracking: si rinuncia a piazzare la regina nella colonna k, si torna indietro al passo k-1 cercando una posizione alternativa per la regina precedentemente piazzata. Per evitare di ripetere decisioni già prese decidiamo di muovere la regina lungo la colonna solo verso il basso (quando saremo all'ultima riga non avremo altre decisioni a disposizione e dovremo fare backtracking). Completato l'ottavo passo tutte le regine sono posizionate senza interferenze reciproche e la ricerca è terminata. La ricerca termina invece con un fallimento se viene effettuato backtracking anche dalla prima colonna. Si veda in figura 1.13 l'evoluzione della ricerca con il metodo di backtracking di una soluzione per una scacchiera di dimensione 4.

Riformuliamo il problema in questi termini. Avendo una scacchiera S su cui sono piazzate correttamente k-1 regine nelle prime k-1 colonne, è possibile piazzare correttamente le rimanenti regine a partire dalla colonna k (il piazzamento corretto indica che nessuna regina è minacciata dalle altre regine già piazzate). Il problema originale è equivalente a quest'ultimo con k=1.

Il nuovo problema (e quindi anche quello di partenza) può essere risolto dal seguente algoritmo ricorsivo.

```
Algoritmo Regine(S,k)
    caso base:  k >8
        RETURN TRUE // abbiamo ottenuto un piazzamento coerente di tutte le regine
    caso ricorsivo:  k <= 8
        PER OGNI casella C in colonna k non minacciata
            piazza regina su C nella scacchiera S
            SE Regine(S,k+1) RETURN TRUE
            rimuovi regina da casella C nella scacchiera S
        FINE_PER_OGNI
        RETURN FALSE
```

Bibliografia

[1] C. Demetrescu, I. Finocchi, G. F. Italiano, Algoritmi e Strutture di Dati, Mc Graw-Hill, Milano, 2004.

[2] A. Drozdek Algoritmi e strutture dati in Java, Apogeo, Milano, 2001.