LUIGINO CALVI

STRUTTURE + ALGORITMI = OGGETTI

2022

Una *grande* scoperta risolve un grande problema ma c'è un granello di scoperta nella soluzione di ogni problema.
Il tuo problema può essere modesto; ma se esso sfida la tua curiosità e mette in gioco le tue facoltà inventive, e lo risolvi con i tuoi mezzi, puoi sperimentare la tensione e godere del trionfo della scoperta.
Questa esperienza ad una età suscettibile può creare un gusto per il lavoro mentale e lasciare un'impronta nella mente e un carattere per tutta la vita.

G. Polya, *How to solve it*

Indice

Parte I

STRUTTURE

1

AGGREGARE

Ci fu un tempo, prima che i nostri traguardi diventassero più ambiziosi, in cui trovavamo strano e meraviglioso riuscire a fare una torre o una casa coi blocchetti delle costruzioni. Eppure, benché tutti gli adulti sappiano fare queste cose, nessuno ancora ha capito come s'impari a farle*!*

M. Minsky, *La società della mente*

Nel mondo fisico ogni cosa è composta da elementi fondamentali primitivi non scomponibili (*atomi*) o considerati tali nel contesto in cui ci si pone; similmente in informatica esistono componenti elementari che sono i più piccoli frammenti dotati di significato e non ulteriormente scomponibili: i bit (ai livelli più bassi), numeri, valori logici, stringhe (predisposti dai tradizionali linguaggi di programmazione), oggetti strutturati (ai livelli più alti delle tecniche di programmazione).

Uno dei concetti fondamentali e trasversali a tutta l'informatica, e non solo all'informatica, si fonda sul meccanismo della *aggregazione* di elementi di base, raggruppandoli in un'unica entità che può essere considerata, denominata e manipolata in modo unitario. In fondo, questa non è una scoperta dell'informatica in quanto è la stessa strategia che consente all'artigiano di costruire un *cesto* intrecciando un *fascio di vimini*.

Questo capitolo costituisce un'introduzione ad un grande tema dell'Informatica noto come *Strutture dei Dati*. Questo capitolo pone le premesse per alcuni dei capitoli che seguiranno: il paragrafo sulle pile viene ripreso ed approfondito nel capitolo *Impilare*, il paragrafo sulle sequenze nel capitolo *Sequenziare*.

1.1 Aggregare

Il meccanismo di aggregazione può essere pensato come un'operazione che raggruppa degli elementi, secondo lo schema descritto nella figura 11.2. I singoli elementi componenti vengono generalmente detti *atomi* mentre l'aggregato viene detto anche *struttura* [1].

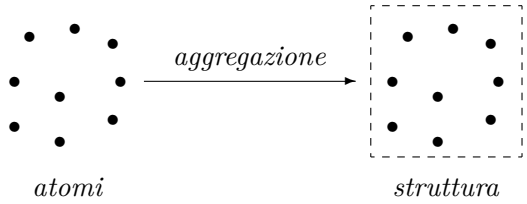


Figura 1.1: Operazione di *aggregazione* di un gruppo di atomi per formare una struttura.

Tipologie di strutture

Gli elementi che costituiscono una struttura possono essere di diversa natura: indistinguibili o differenziati; inoltre, possono essere organizzati in diversi modi. Un'ulteriore caratterizzazione delle strutture è data dalle particolari operazioni ammesse e dalle politiche di gestione delle operazioni di inserimento, eliminazione ed accesso agli elementi della struttura. Si ottengono così tipologie di strutture profondamente diverse e pertanto denominate con nomi diversi. Le principali forme di aggregazione sono elencate a seguire:

- *insiemi*: strutture composte da elementi distinguibili e diversi fra loro
- *pile*: aggregati composti da una sequenza di elementi contigui nella quale è possibile inserire ed accedere agli elementi solo da un'estremità
- *sequenze*: strutture di elementi disposti in fila ed occupanti una specifica posizione individuata da un indice numerico di posizione

Operazioni sulle strutture

Su una generica struttura si possono eseguire diverse tipologie di operazioni: indipendentemente dalla specifica struttura, le operazioni generali sono:

- *identificazione mediante un nome*: siccome opereremo con più strutture e con strutture di diverse tipologie, ci servirà denotare le varie strutture con dei nomi identificativi; utilizzeremo dei nomi che iniziano con un carattere maiuscolo: A, B, C, ..., M, N, ...
- *creazione di una struttura vuota*: a seconda della particolare tipologia (insieme, pila, sequenza), una struttura vuota viene denotata con uno dei seguenti simboli: $\{\}, [\,\rangle, []$; nel seguito indicheremo con $\square$ una generica struttura vuota. Per creare una struttura vuota A utilizzeremo la

[1] Viene generalmente usato il termine *aggregato* quando si vuole evidenziare che gli elementi sono raggruppati fra loro ed il termine *struttura* quando si vuole evidenziare che è sono definite delle ulteriori caratteristiche: operazioni o specificazioni sulla struttura di aggregazione.

tradizionale notazione di assegnazione nella forma

$$A \leftarrow \square$$

- *assegnazione fra strutture*: un'assegnazione della forma

$$A \leftarrow B$$

fra due strutture A e B ha come effetto che "*A diventa un identificatore della struttura B*". Nel caso serva che A diventi un riferimento ad una copia di A, distinta da A, si deve ricorrere alla funzione *clone* che ritorna una copia dell'aggregato indicato nell'argomento alla funzione; l'istruzione si scrive pertanto:

$$A \leftarrow clone(B)$$

Oltre alle operazioni generali sopra descritte, sono predisposte ulteriori operazioni che dipendono e caratterizzano le diverse tipologie di strutture che, a seconda delle operazioni ammesse ed al loro comportamento, assumono nomi particolari (*pile*, *code*, *dizionari*, ...); queste altre operazioni vengono classificate come segue:

- *accesso* agli elementi della struttura
- *inserimento* di elementi nella struttura
- *eliminazione* di elementi dalla struttura
- *modifica* di elementi della struttura

Notazioni delle operazioni

Le operazioni di accesso, inserimento, eliminazione e modifica sopra descritte possono essere realizzate con diverse impostazioni:

- *procedurale*: le operazioni vengono realizzate mediante un comando che modifica la struttura, con la notazione $comando(A, \dots)$; nell'algoritmo che descrive l'operazione viene riportata la precondizione iniziale richiesta (*Require*) e la postcondizione finale (*Ensure*) che viene garantita dall'esecuzione dell'algoritmo
- *funzionale*: adottando l'impostazione della Matematica, l'operazione viene realizzata mediante una funzione che genera e ritorna una nuova struttura, senza modificare la struttura passata come argomento; l'operazione viene denotata nella forma $operazione(A, \dots)$, dove A denota la struttura ed i puntini gli altri eventuali argomenti dell'operazione
- *ad oggetti*: viene posta l'enfasi non sull'operazione ma sulla struttura sulla quale viene eseguita l'operazione; questa impostazione viene denotata come *orientata agli oggetti* e la struttura, che viene detta più propriamente *contenitore*, diventa l'oggetto principale e l'operazione diventa ancillare rispetto ad esso; anche la notazione con la quale viene descritta l'operazione cambia e viene scritta nella forma $A.operazione(\dots)$, dove A denota la struttura ed i puntini gli eventuali altri argomenti dell'operazione

1.2 Strutture iterabili

Una struttura si dice *iterabile* se predispone un meccanismo per accedere in lettura ai suoi elementi. Insiemi e sequenze sono due tipiche forme di strutture iterabili che saranno descritte più avanti in questo capitolo. Per accedere agli elementi di un iterabile è predisposto un controllo ciclico avente la seguente forma:

Algoritmo 1 - elaborazione degli elementi di un iterabile

Input: iterabile A
1: **for** x **in** A **do**
2: elabora l'elemento x
3: **end for**

Un'applicazione di questo schema è descritta nell'esempio che segue.

Esempio 1.2.1 - Per determinare il numero di elementi di una struttura iterabile si può utilizzare il seguente algoritmo:

Algoritmo 2 - numero di elementi di un iterabile

Input: iterabile A
Output: numero di elementi di A
1: $n \leftarrow 0$
2: **for** x **in** A **do**
3: $n \leftarrow n + 1$
4: **end for**
5: **return** n

Oltre allo schema generale descritto nell'algoritmo 1, a seconda della particolare tipologia di struttura, si possono utilizzare altre schemi basati sulle specifiche operazioni della struttura considerata.

1.3 Gli insiemi

La più povera e primitiva forma di aggregazione è costituita da raggruppamenti finiti di elementi distinguibili fra loro, detti *insiemi*. Gli insiemi costituiscono la forma di aggregazione che caratterizza la Matematica e, coerentemente con la definizione che se ne dà in questa disciplina, un *insieme* è un'aggregato di elementi distinguibili uno dall'altro e tutti diversi fra loro [2] . Ad esempio, con la tradizionale notazione matematica $\{2, 3, 5, 7, 11\}$ si denota l'insieme formato dai primi 5 numeri primi. Tale insieme viene spesso denotato come descritto nella figura 1.2.

[2]Nella definizione di insieme si fa talvolta riferimento all'*insieme universale* costituito dagli elementi che si possono inserire nel insieme.

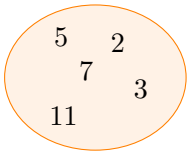


Figura 1.2: Diagramma di Venn che rappresenta l'insieme di numeri $\{2, 3, 5, 7, 11\}$.

Indicheremo con $\varnothing$ o con $\{\}$ l'*insieme vuoto*, ossia l'insieme formato da nessun elemento. Per la creazione di insiemi, per la loro denominazione, per le assegnazioni fra insiemi e per controllare se un insieme è vuoto adotteremo le convenzioni indicate al paragrafo 1.1. Per creare un insieme vuoto, ad esempio A, utilizzeremo la notazione $A \leftarrow \varnothing$.

Osservazione. L'ordine sequenziale con il quale vengono scritti gli elementi di un insieme non è significativo e l'eventuale presenza di elementi duplicati che si precisano è ininfluente ai fini della consistenza dell'insieme; pertanto, ad esempio le scritture $\{2, 3, 5\}$, $\{3, 5, 2\}$ e $\{2, 3, 3, 5, 2\}$ individuano uno stesso insieme.

Operazioni di base sugli insiemi

Sugli insiemi sono definite delle *operazioni insiemistiche* che agiscono su coppie di insiemi e forniscono come risultato un insieme; le principali sono:

$\cup$: unione di due insiemi

$\cap$: intersezione di due insiemi

$\setminus$: differenza fra due insiemi

Adottando un approccio più operativo, di tipo procedurale, in alternativa alle operazione insiemistiche elencate sopra, si possono predisporre delle operazioni di base, di tipo operativo che permettono di *realizzare* le operazioni insiemistiche (ed altre) agendo sui singoli elementi degli insiemi. Traendo spunto dall'analogia delle operazioni che si possono eseguire su un generico contenitore di oggetti, ammetteremo le seguenti operazioni di base:

- $isempty(A)$: test se l'insieme A è vuoto
- $insert(A, x)$: inserisce nell'insieme A l'elemento x; nel caso in cui l'elemento x fosse già presente in A, l'operazione non viene eseguita
- $remove(A, x)$: elimina dall'insieme A l'elemento x; nel caso in cui l'elemento x non sia presente in A, l'operazione non ha effetto
- $extract(A)$: estrae dall'insieme A un elemento e lo ritorna; nel caso in cui l'insieme A sia vuoto, l'operazione non ha effetto e ritorna il *valore nullo*

Le operazioni di base sopra descritte possono essere combinate per realizzare altre operazioni più articolate, come descritto nei seguenti esempi.

Esempio 1.3.1 - L'operazione *elemento*(A), descritta nel seguente algoritmo 3, ritorna un generico elemento dell'insieme A (senza eliminarlo).

Algoritmo 3 - selezione di un elemento - *elemento*(A)

1: $x \leftarrow extract(A)$
2: $insert(A, x)$
3: **return** x

Esempio 1.3.2 - Per svuotare un insieme, ossia per eliminare tutti gli elementi presenti in esso, si può utilizzare il seguente algoritmo 4.

Algoritmo 4 - *svuota*(A)

Require: insieme A
Ensure: l'insieme A è vuoto
1: **while** $\neg$ *isempty*(A) **do**
2: *extract*(A)
3: **end while**

Esempio 1.3.3 - Essendo che un insieme è una struttura iterabile, la funzione *cardinalita*(A) che fornisce numero di elementi dell'insieme A può essere realizzata come segue descritto nell'algoritmo 2.

Elaborazione degli insiemi

Su un insieme, essendo esso un aggregato iterabile, è possibile applicare lo schema generale di elaborazione descritto nell'algoritmo 1. Basandosi su questo schema di elaborazione si può determinare la cardinalità di un insieme.

Basandosi sulle specifiche operazioni predisposte su un insieme, è possibile elaborare un insieme, in modo invasivo, mediante l'algoritmo 5.

Algoritmo 5 - elaborazione invasiva di un insieme A

1: **while** $\neg$ *isempty*(A) **do**
2: $x \leftarrow extract(A)$
3: elabora l'elemento x
4: **end while**

La condizione di controllo di terminazione del ciclo indicata alla linea 1. può essere rafforzata per terminare il ciclo prima di aver scandito tutti gli elementi presenti nell'insieme. Nel caso in cui non avvenga alcuna elaborazione dell'elemento x (linea 3), l'algoritmo 5 produce l'effetto di svuotare l'insieme A.

Problema 1.3.1 Determinare il massimo di un insieme non vuoto.

Soluzione. Il massimo di un insieme può essere determinato passando in rassegna tutti gli elementi dell'insieme. L'inizializzazione del valore massimo

può essere fatta selezionando un elemento che costituirà il valore iniziale di comparazione con tutti gli altri elementi. Il procedimento è dettagliato nel seguente algoritmo 6.

Algoritmo 6 - massimo di un insieme

Input: insieme non vuoto A
Output: massimo valore di A
1: $m \leftarrow elemento(A)$
2: **for** x **in** A **do**
3: **if** $x > m$ **then**
4: $m \leftarrow x$
5: **end if**
6: **end for**
7: **return** m

□

Il crivello di Eratostene per il calcolo dei numeri primi

Un procedimento molto antico per la determinazione dei numeri primi risale a Eratostene di Cirene (276-192 a.C.); il metodo è descritto anche nella *Introductio Aritmetica* di Nicomaco di Gerasa, un matematico greco vissuto nel I sec. d.C.. Il procedimento è il seguente: si scrivono in sequenza i numeri naturali a partire da 2, fino ad un certo numero n; si considera 2 come numero primo e si cancellano tutti i numeri multipli di 2; a seguire, si individua il numero più piccolo fra quelli rimasti, lo si considera come numero primo e si cancellano tutti i suoi multipli; si prosegue così finché non rimane più alcun numero. Il metodo è noto anche come *crivello di Eratostene*, in quanto può essere realizzato praticamente mediante un piano inclinato composto da scanalature, una per ogni numero da controllare; su ogni scanalatura si praticano dei fori a distanza di ciascun numero; facendo scendere delle palline, quelle che arrivano alla fine corrispondono ad un numero primo. Il procedimento può essere realizzato mediante delle operazioni sugli insiemi, come descritto nell'algoritmo 7 che richiede la soluzione dei sottoproblemi P_1, P_2, P_3.

Algoritmo 7 - Determinazione numeri primi mediante crivello di Eratostene

Input: numero naturale n
Output: insieme dei numeri primi minori di n
1: $A \leftarrow \{2, 3, 4, \ldots, n\}$ $[P_1]$
2: $P \leftarrow \varnothing$
3: **while** $\neg\, isempty(A)$ **do**
4: $k \leftarrow$ valore minimo estratto da A $[P_2]$
5: $insert(P, k)$
6: elimina da A tutti i multipli di k $[P_3]$
7: **end while**
8: **return** P

1.4 Le pile

Una *pila* è una particolare sequenza, composta da elementi disposti uno sopra l'altro. La caratterizzazione fisica *uno sopra l'altro* lascia intuire le operazioni che si possono eseguire su una pila: l'inserimento, l'accesso e l'eliminazione di elementi è possibile solo su un estremo (testa della pila); nello sviluppo degli algoritmi ci si può lasciare guidare pensando ad una pila di blocchetti fisici, disposti uno sopra l'altro, come descritto nella figura 1.3.

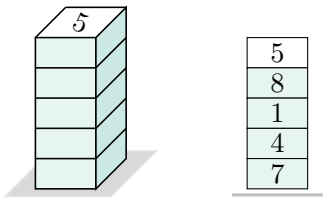


Figura 1.3: Una pila di numeri: a sinistra la versione tridimensionale che lascia vedere solo l'elemento di testa, a destra la versione piatta con l'indicazione degli elementi anche non direttamente accessibili.

Questi particolari vincoli hanno come conseguenza una riduzione del numero di operazioni che si possono utilizzare e, al contempo, stimolano la realizzazione di algoritmi che risultano interessanti proprio per il ridotto repertorio di operazioni utilizzabili.

Le operazioni di base sulle pile

Denoteremo con $[\,\rangle$ una *pila vuota*, ossia una pila composta da nessun elemento, mentre con $[a_1, a_2, \ldots, a_n\rangle$ denoteremo la pila formata dagli elementi $a_1, a_2, \ldots, a_n$; la pila descritta nella figura 1.3 viene indicata con $[7, 4, 1, 8, 5\rangle$.

La creazione e l'inizializzazione di una pila (vuota) avviene mediante l'assegnazione

$$A \leftarrow [\,\rangle$$

Una volta creata, una pila può essere manipolata mediante le seguenti operazioni di base:

- $isempty(A)$: test se la pila A è vuota
- $top(A)$: elemento di testa della pila A
- $push(A, x)$: inserisce in testa alla pila A l'elemento x
- $pop(A)$: elimina l'elemento di testa della pila A

Combinando queste operazioni di base è possibile realizzare delle operazioni elementari fra due pile come descritto negli algoritmi che risolvono i problemi che seguono.

Problema 1.4.1 Svuotare una pila.

Soluzione. La soluzione è immediata, mediante l'algoritmo 8.

Algoritmo 8 - Svuotamento di una pila - $svuota(A)$

Require: pila A
Ensure: pila A vuota
1: **while** $\neg$ $isempty(A)$ **do**
2: $pop(A)$
3: **end while**

□

Problema 1.4.2 Spostare l'elemento di testa da una pila ad un'altra.

Soluzione. Si può operare come descritto nell'algoritmo 9.

Algoritmo 9 - Sposta un elemento da una pila ad un'altra - $sposta(A, B)$

Require: pile A, B di blocchetti, A è non vuota
Ensure: il blocchetto di testa di A è stato spostato in B
1: $push(B, top(A))$
2: $pop(A)$

□

Problema 1.4.3 Scambiare gli elementi di testa delle due pile non vuote.

Soluzione. La semplice soluzione è fornita dall'algoritmo 10 che fa uso di una pila ausiliaria C.

Algoritmo 10 - Scambio degli elementi di testa di due pile - $scambia(A, B)$

Require: pile A, B non vuote
Ensure: pile A, B con gli elementi di testa scambiati
1: $C \leftarrow [\rangle$
2: $sposta(A, C)$
3: $sposta(B, A)$
4: $sposta(C, B)$

□

Problema 1.4.4 Data una pila A ed una pila vuota B, rovesciare gli elementi dalla pila A alla pila B.

Soluzione. Lo spostamento può essere ottenuto mediante l'algoritmo 11:

Algoritmo 11 - Rovesciamento di una pila in un'altra - $rovescia(A, B)$

Require: pila A, pila B
Ensure: pila A vuota, pila B con gli elementi rovesciati
1: **while** $\neg$ $isempty(A)$ **do**
2: $sposta(A, B)$
3: **end while**

□

Problema 1.4.5 Traslare una pila in un'altra, spostandone gli elementi mantenendo la posizione reciproca

Soluzione. È sufficiente combinare due operazioni di rovesciamento, come descritto nell'algoritmo 12.

Algoritmo 12 - Traslazione di una pila in un'altra - $trasla(A, B)$

Require: pile A, B
Ensure: pila A vuota, pila B con in testa gli elementi di A rovesciati
1: $C \leftarrow [\rangle$
2: $rovescia(A, C)$
3: $rovescia(C, B)$

□

Osservazione. Gli algoritmi 8, 9, 10, 11 e 12 costituiscono un ampliamento del repertorio delle operazioni di base previste sulle pile:

- $svuota(A)$: svuota la pila A
- $sposta(A, B)$: sposta l'elemento di testa della pila A in testa alla pila B
- $scambia(A, B)$: scambia gli elementi di testa delle pile A e B
- $rovescia(A, B)$: rovescia la pila A in B
- $trasla(A, B)$: trasla la pila A in B

Queste operazioni, pur non costituendo un potenziamento dell'esecutore, permettono di esprimere gli algoritmi con più facilità ed espressività.

1.5 Le sequenze

Una delle forme più elementari di organizzazione è costituita dalla *sequenziazione* che consiste nel disporre (fisicamente) o considerare (virtualmente) gli elementi in fila, uno dopo l'altro. Con il termine *sequenza* (o *lista*) si denota un aggregato di elementi, disposti in fila, uno dopo l'altro. Ciascun elemento di una sequenza occupa una specifica posizione, individuata da un indice di posizione costituito da un numero naturale; secondo la convenzione adottata da molti linguaggi di programmazione, le posizioni iniziano da 0. Una sequenza viene spesso descritta graficamente mediante uno schema simile a quello riportato nella figura 1.4.

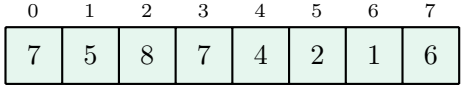


Figura 1.4: Rappresentazione grafica della sequenza $[7, 5, 8, 7, 4, 2, 1, 6]$.

Se $a_0, \ldots, a_n$ sono degli elementi, la scrittura [3]

$$[a_0, \ldots, a_n]$$

denota la sequenza costituita dagli elementi $a_0, \ldots, a_n$. Gli elementi a_i che costituiscono una sequenza possono essere delle generiche espressioni. Con [ ] si denota la *sequenza vuota*, ossia la sequenza composta da nessun elemento. Con $[n \times \square]$ si denota una sequenza composta da n elementi. Una sequenza composta da n elementi viene detta *n-sequenza*; nei casi particolari $n = 2, 3, \ldots$ si usano anche i termini *coppia*, *terna*, ... Data una sequenza $A = [a_0, \ldots, a_n]$ e degli indici i, j tali che $0 \leq i \leq j \leq n$, la sequenza $[a_i, \ldots, a_j]$ dicesi *sottosequenza* di A e viene indicata con la scrittura $A[i..j]$.

Esempio 1.5.1 - Dalla definizione di sequenza, si può immediatamente riconoscere che i seguenti sono degli esempi di sequenze:

$[\,]$
$[7]$
$[3, 2, 5, 2, 7]$
$[(2 + 3) * 5, (4 + 13) > 40, max(4, 7) + 5]$
$[8, \text{"}uno\text{"}, \texttt{TRUE}, 1 + 2, 3 > 4]$

Le operazioni di base sulle sequenze

Una caratteristica importante delle sequenze riguarda le modalità con le quali si definisce e si gestisce la loro dimensione; si possono avere le seguenti due situazioni:

- sequenze *statiche*: vengono create di una determinata dimensione che poi non può più essere modificata; la creazione di una sequenza statica A di dimensione n avviene mediante l'assegnazione

  $$A \leftarrow [n \times \square]$$

  che crea ed assegna ad A una sequenza di n elementi di valore e
- sequenze *dinamiche*: vengono create delle sequenze vuote che successivamente vengono allungate con l'inserimento di nuovi elementi; la creazione e l'inizializzazione di una sequenza (vuota) avviene mediante l'assegnazione

  $$A \leftarrow [\,]$$

  Per inserire elementi in una sequenza dinamica viene utilizzata l'istruzione

  $$append(A, e)$$

[3] La scrittura [...] è sintatticamente compatibile con molti linguaggi di programmazione. Al posto di [...] viene usata anche la scrittura (...): in questo caso si parla di *tuple* che, a differenza delle sequenze, si differenziano per il fatto che, una volta definite, non possono più essere modificate.

che ha l'effetto di inserire alla fine della sequenza A il valore ottenuto dalla valutazione dell'espressione e [4]

Data una generica sequenza A, con

$$len(A)$$

si denota la *lunghezza* di A, ossia il numero di elementi che la compongono.

L'accesso agli elementi di una sequenza sfrutta il fatto che gli elementi componenti occupano una specifica posizione individuata da un indice di posizione; se A denota la sequenza $[a_0, a_1, \ldots, a_n]$ e k una generica espressione numerica naturale, la scrittura

$$A[k]$$

denota il k-esimo elemento della sequenza A, ossia a_k. La modifica di un elemento di una sequenza avviene con il comando di assegnazione

$$A[k] \leftarrow e$$

che ha l'effetto di assegnare alla k-esima componente della sequenza A il valore ottenuto dalla valutazione dell'espressione e.

Esempio 1.5.2 - Con riferimento alla figura 1.4, l'assegnazione

$$A[1] \leftarrow A[2] + 3$$

ha l'effetto di modificare il valore 5 del secondo elemento con il valore 11.

Problema 1.5.1 Scambiare in una sequenza il primo con l'ultimo elemento.

Soluzione. I comandi per eseguire lo scambio sono:

$$\begin{aligned} &t \leftarrow A[0] \\ &A[0] \leftarrow A[len(A) - 1] \\ &A[len(A) - 1] \leftarrow t \end{aligned}$$

□

Elaborazione di sequenze

La forma più elementare di elaborazione di una sequenza consiste nell'esaminare sequenzialmente tutti i suoi elementi. Nel caso in cui gli elementi non debbano essere modificati, la forma più semplice di accesso agli elementi di una sequenza è descritta nell'algoritmo 1, valida per una generica struttura iterabile.

Problema 1.5.2 Determinare la somma degli elementi di una sequenza.

[4] In taluni linguaggi si ammette anche che le sequenze possano essere modificate eliminando l'elemento alla fine oppure troncando la sequenza ad una data dimensione oppure eliminando un elemento all'interno della sequenza.

Soluzione. Il problema può essere risolto mediante il seguente algoritmo 13.

Algoritmo 13 - somma degli elementi di una sequenza numerica

Input: sequenza numerica A
Output: somma degli elementi della sequenza A
1: $s \leftarrow 0$
2: **for** x **in** A **do**
3: $\quad s \leftarrow s + x$
4: **end for**
5: **return** s

□

Nel caso in cui gli elementi della sequenza debbano essere modificati si deve ricorrere ad un ciclo in cui l'accesso agli elementi avviene con la notazione ad indice, come descritto nell'algoritmo 14; in questo modo gli elementi risultano accessibili sia in lettura che in scrittura e possono quindi essere modificati.

Algoritmo 14 - Elaborazione sequenziale di una sequenza A

1: **for** i **from** 0 **to** $len(A)-1$ **do**
2: $\quad$ elabora elemento $A[i]$
3: **end for**

Lo schema generale descritto nell'algoritmo 14 può essere adattato per elaborare (sia in sola lettura che in scrittura) una porzione della sequenza individuata da due indici di posizione; nel caso in cui la porzione di sequenza da elaborare non sia preventivamente individuata, si dovrà utilizzare un adeguato ciclo *while*. I problemi che seguono illustrano alcuni casi fra quelli sopra descritti.

Problema 1.5.3 Azzerare tutti gli elementi di una sequenza numerica.

Soluzione. In questo problema gli elementi devono essere modificati; pertanto bisogna utilizzare l'algoritmo 14. La soluzione del problema è fornita dal seguente algoritmo:

Algoritmo 15 - azzeramento degli elementi di una sequenza

Require: riferimento alla sequenza A
Ensure: gli elementi della sequenza A sono azzerati
1: **for** i **from** 0 **to** $len(A)-1$ **do**
2: $\quad A[i] \leftarrow 0$
3: **end for**

□

Problema 1.5.4 Stabilire se una sequenza è composta da elementi tutti uguali.

Soluzione. Una soluzione è data dal seguente algoritmo.

Algoritmo 16 - Stabilire se una sequenza è formata da elementi uguali

Input: sequenza A da esaminare
Output: `TRUE` se e solo se A è composta da elementi tutti uguali

```
1: uguali ← TRUE
2: i ← 1
3: while uguali ∧ (i < len(A)) do
4:    if A[i] ≠ A[0] then
5:       uguali ← FALSE
6:    else
7:       i ← i + 1
8:    end if
9: end while
10: return uguali
```

□

1.6 Interazione e cooperazione fra strutture

Le strutture, in base alle proprietà e caratteristiche delle loro specifiche operazioni, possono essere utilizzate assieme, in modo cooperativo, come evidenzia la soluzione del seguente problema.

Problema 1.6.1 Stabilire se una sequenza è composta da elementi tutti distinti.

Soluzione. La soluzione descritta nel seguente algoritmo si basa sull'idea di inserire gli elementi della sequenza in un insieme ausiliario, contarne gli elementi e confrontare il numero con la lunghezza della sequenza.

Algoritmo 17 - Stabilire se una sequenza è formata da elementi distinti

Input: sequenza A da esaminare
Output: `TRUE` se e solo se A è composta da elementi tutti distinti

```
1: T ← ∅
2: for x in A do
3:    insert(T, x)
4: end for
5: n ← 0
6: while ¬ isempty(T) do
7:    extract(T)
8:    n ← n + 1
9: end while
10: return n = len(A)
```

□

1.7 Composizione di strutture

Le strutture possono essere formate da elementi che sono a loro volta strutture composte da elementi. Le varie forme di strutture esaminate nei precedenti paragrafi possono essere composte fra loro in tutti i modi, ottenendo una variegata gamma di strutture composte: sequenze di sequenze, insiemi di sequenze, sequenze di pile, insiemi di insiemi, insiemi di sequenze di insiemi, ...

Esempio 1.7.1 - La scomposizione del numero 1200, che è uguale a $2^4 \cdot 3 \cdot 5^2$, può essere descritta mediante l'*insieme di coppie*

$$\{[2,4],[3,1],[5,2]\}$$

Esempio 1.7.2 - La tabella bidimensionale

7	2	1
4	0	6

può essere descritta rappresentata mediante la seguente *sequenza di sequenze* (*coppia di terne*)

$$[[7,2,1],[4,0,6]]$$

Esempio 1.7.3 - La configurazione dei dischi della torre di Hanoi descritta nella figura

nella quale i vari dischi sono identificati mediante un numero naturale (che ne rappresenta la grandezza) può essere descritta dalla seguente *sequenza di pile*:

$$[[2\rangle,[\,\rangle,[4,3,1\rangle]$$

Esempio 1.7.4 - L'insieme $\mathcal{P}(A)$ delle parti dell'insieme $A = \{1,2,3\}$ può essere descritto dall'*insieme di insiemi*

$$\{\{\},\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$$

Le diverse combinazioni delle forme di strutture viste negli esempi precedenti, possono essere elaborate mediante le operazioni specifiche di ciascun aggregato utilizzato nella composizione.

Problema 1.7.1 Determinare l'insieme delle parti di un insieme di elementi.

Soluzione. L'idea di soluzione è la seguente: inizializza un insieme vuoto P che alla fine costituirà il risultato; per ciascun elemento e dell'insieme dato A aggiungi all'insieme risultato P altrettanti insiemi formati dagli insiemi già inseriti in P a ciascuno dei quali viene aggiunto il nuovo elemento e. Questo procedimento si traduce nel seguente algoritmo 18.

Algoritmo 18 - insieme delle parti di un insieme

Input: insieme A
Output: insieme delle parti di A

```
1:  P ← {∅}
2:  for x in A do
3:     Q ← ∅
4:     for T in P do
5:        R ← clone(T)
6:        insert(R, x)
7:        insert(Q, R)
8:     end for
9:     for T in Q do
10:       insert(P, T)
11:    end for
12: end for
13: return P
```

□

1.8 Le matrici

Un caso particolarmente interessante e frequente di composizione di aggregati è costituito dalle *sequenze di sequenze*; nel caso in cui le sequenze componenti siano tutte della stessa lunghezza, questi aggregati sono detti *matrici*. Da questa definizione discende che una matrice è descrivibile mediante una tabella bidimensionale strutturata in *righe* e *colonne*.

Esempio 1.8.1 - La struttura

$$[\,[2,3,7,1],[5,8,4,2],[3,0,2,5]\,]$$

costituisce una *matrice bidimensionale*. Una tale matrice viene usualmente descritta in forma tabellare come segue:

2	3	7	1
5	8	4	2
3	0	2	5

Spesso e compatibilmente con quanto è concesso nei tradizionali linguaggi di programmazione, una matrice viene creata inizialmente con ben specificate dimensioni, precisando il numero di righe e di colonne, che poi non vengono

più modificate. Utilizzando il meccanismo di creazione di una sequenza, una matrice di dimensioni $m \times n$ viene creata con l'espressione $[m[n\,\square]]$. Per creare una matrice M di dimensioni $m \times n$ utilizzeremo la notazione

$$M \leftarrow [m \times [n \times \square]]$$

Se M denota una matrice, coerentemente con la notazione usata per l'operatore di selezione ad indice, la notazione

$$M[i]$$

denota la sequenza costituita dall'i-esima riga della matrice M, mentre

$$M[i][j]$$

denota il j-esimo elemento dell'i-esima riga della matrice M. Con l'istruzione

$$M[i][j] \leftarrow e$$

si ottiene l'effetto di assegnare all'elemento $M[i][j]$ il valore ottenuto dalla valutazione dell'espressione e.

Data una matrice M, si può ricavarne il numero di righe ed il numero di colonne, rispettivamente, mediante le espressioni $len(M)$, $len(M[0])$.

Osservazione. Le operazioni sulle matrici descritte sopra (costruzione, accesso agli elementi, modifica degli elementi) sono realizzate basandosi direttamente sulla particolare forma di aggregazione, considerando le matrici come *sequenze di sequenze*.

Elaborazione di matrici

Gli schemi generali di elaborazione di una matrice si ottengono direttamente mediante combinazione degli schemi di elaborazione delle sequenze già esaminati al paragrafo 1.5.

Nel caso in cui gli elementi di una matrice vengano elaborati in sola lettura, senza essere modificati, si può adottare lo schema descritto nell'algoritmo 19, derivato dal fatto che una matrice è una sequenza di sequenze e le sequenze sono degli iterabili.

Algoritmo 19 - Elaborazione di una matrice

Input: matrice M

```
1: for riga in M do
2:     for x in riga do
3:         elabora l'elemento x
4:     end for
5: end for
```

Nel caso i cui gli elementi della matrice debbano essere modificati si può utilizzare lo schema descritto nell'algoritmo 20.

Algoritmo 20 - Elaborazione di una matrice

Input: matrice M
1: $nr \leftarrow len(M)$
2: $nc \leftarrow len(M[0])$
3: **for** i **from** 0 **to** $nr-1$ **do**
4: **for** j **from** 0 **to** $nc-1$ **do**
5: elabora l'elemento $M[i][j]$
6: **end for**
7: **end for**

Problema 1.8.1 Determinare la somma degli elementi di una matrice numerica.

Soluzione. La somma degli elementi di una matrice numerica può essere calcolata mediante il seguente algoritmo:

Algoritmo 21 - Somma degli elementi di una matrice

Input: matrice numerica M
Output: somma degli elementi di M
1: $s \leftarrow 0$
2: **for** $riga$ **in** M **do**
3: **for** x **in** $riga$ **do**
4: $s \leftarrow s + x$
5: **end for**
6: **end for**
7: **return** s

□

1.9 I multiinsiemi

Un *multiinsieme* [5] è un insieme in cui gli elementi possono essere ripetuti; può essere denotato mediante la tradizionale notazione degli insiemi, ad esempio $\{a, a, b, c, c, c, c, d, d, d\}$. Come per gli insiemi, in un multiinsieme l'ordine degli elementi è ininfluente. Formalmente, un multiinsieme M può essere definito come una coppia (A, m) dove A, detto *insieme di supporto*, è l'insieme degli elementi del multiinsieme e m, detta *funzione di frequenza*, è una funzione $A \to \mathbb{N}$ che ad ogni elemento di A associa il numero di volte che esso compare nel multiinsieme. Basandosi su questa definizione, un multiinsieme può essere rappresentato mediante una combinazione degli aggregati di base, mediante un insieme di coppie; ad esempio il multiinsieme $\{a, a, b, c, c, c, c, d, d, d\}$ viene descritto mediante l'insieme di coppie $\{[a, 2], [b, 1], [c, 4], [d, 3]\}$.

I multiinsiemi, nonostante abbiano avuto un ruolo marginale nel contesto della matematica, a tutto vantaggio del concetto di insieme, si prestano in

[5] Detto anche *sacco* o, con terminologia inglese, *bag*, *multiset*, *mset*.

modo più efficace di descrivere e di elaborate oggetti e situazioni tipiche della matematica. Ne sono testimonianza i seguenti esempi.

Esempio 1.9.1 - La scomposizione in fattori primi del numero 90, ossia $2 \cdot 3^2 \cdot 5$ può essere descritta mediante il multiinsieme $\{[2,1],[3,2],[5,1]\}$.

Esempio 1.9.2 - Il monomio monico x^3y^2 può essere descritto mediante il multiinsieme $\{[x,3],[y,2]\}$.

Esempio 1.9.3 - I caratteri componenti la stringa `calcolo` possono essere descritti mediante il multiinsieme $\{[\texttt{c},2],[\texttt{a},1],[\texttt{l},2],[\texttt{o},2]\}$.

Le operazioni di base predisposte su un multiinsieme sono le stesse già viste per un insieme. L'accesso agli elementi di un multiinsieme può essere realizzato in più modi:

1. considerando un multiinsieme come un iterabile composto da elementi

2. considerando un multiinsieme come un iterabile composto da coppie (e,k), dove e è un elemento presente e k la sua frequenza; ottenuta una generica coppia x, con $x[0]$ si accede al valore e dell'elemento e con $x[1]$ alla sua frequenza k

3. usando una funzione $supporto(M)$ che fornisce l'insieme di supporto del multiinsieme M e con la funzione $frequenza(M,e)$ che ritorna la frequenza con la quale all'interno del multiinsieme M è presente l'elemento e.

1.10 Strutture fisiche ed astratte

Nell'informatica, similmente a quanto avviene in tante altre discipline scientifiche, si cerca di individuare e definire alcuni concetti di base sui quali definire gli altri concetti ed edificare il resto della teoria. Nel contesto dell'informatica e limitatamente alle strutture di aggregazione, se ne sceglie una (insiemi, sequenze, ...) e si cerca di realizzare le altre sulla base di questa. Questa asimmetria fra strutture di base e strutture costruite viene solitamente denominata come strutture fisiche e strutture astratte. L'impostazione è quella di scegliere una (o più) struttura fisica sufficientemente potente per poter realizzare tutte le altre strutture astratte desiderabili. Questa realizzazione avviene dapprima definendo, sulla struttura astratta, le operazioni desiderate ed il loro comportamento e successivamente realizzando le operazioni basandosi sulla struttura fisica e sulle operazioni su di essa predisposte.

Data la sua storica tradizione in seno alla Matematica, il concetto di insieme è stato utilizzato, con buon successo anche se non sempre in modo semplice e lineare, come concetto di base su cui definire gli altri concetti, come evidenziano i seguenti due esempi.

Esempio 1.10.1 - Un numero naturale può essere definito come l'insieme di tutti i numeri che lo precedono. Con questa definizione, intrinsecamente ricorsiva, i numeri vengono definiti in successione a partire dallo zero che coincide con l'insieme vuoto ed i numeri successivi risultano definiti come segue:

$$\begin{aligned} 0 &= \{\} \\ 1 &= \{0\} = \{\{\}\} \\ 2 &= \{0,1\} = \{\{\},\{\{\}\}\} \end{aligned}$$

Una definizione dei numeri naturali, diversa da questa ma più efficace e più utilizzata, è dovuta al matematico e logico italiano Giuseppe Peano (1858-1932).

Esempio 1.10.2 - Il ricorso agli insiemi come struttura di base risulta abbastanza impegnativo specialmente nel caso si debbano rappresentare degli aggregati dove l'ordine sequenziale degli elementi risulti essenziale. Ad esempio, per rappresentare una coppia (x, y) di elementi, dove è essenziale distinguere il primo dal secondo elemento della coppia, si può utilizzare il metodo di Felix Hausdorff (1868-1942):

$$rap((x,y)) = \{\{x,1\},\{y,2\}\}$$

o anche il metodo di Kazimierz Kuratowski (1896-1980):

$$rap((x,y)) = \{\{x\},\{x,y\}\}$$

Osservazione. In Informatica si preferisce scegliere come fondamentale la forma aggregativa delle sequenze e mediante esse realizzare le altre strutture astratte. Questa scelta è coerente con quanto offrono, a livello operativo, i vari linguaggi di programmazione.

Esempio 1.10.3 - Un insieme può essere rappresentato utilizzando una sequenza in cui memorizzarne gli elementi; ad esempio

$$rap(\{2,5,3,1,8,6\}) = [2,5,3,1,8,6]$$

L'operazione di inserimento di un elemento in un insieme viene realizzata mediante un algoritmo che ricerca, nella sequenza che rappresenta l'insieme, se l'elemento da inserire è già presente e, in caso negativo, accoda l'elemento alla sequenza.

ESERCIZI

1.1 Contare gli elementi contenuti in un iterabile.

1.2 Determinare la somma dei numeri contenuti in un iterabile di numeri.

1.3 Determinare la media dei numeri contenuti in un iterabile di numeri.

1.4 Determinare il prodotto dei numeri contenuti in un iterabile di numeri.

1.5 Svuotare un insieme.

1.6 Aumentare di un'unità i valori presenti in un insieme di numeri.

1.7 Determinare l'insieme dei divisori di un dato numero naturale.

1.8 Estrarre il valore minimo contenuto in un insieme di numeri.

1.9 Eliminare da un insieme un dato l'elemento; nel caso in cui l'elemento non sia presente, l'operazione non avrà alcun effetto.

1.10 Eliminare da un insieme un dato elemento e sostituirlo con un altro.

1.11 Calcolare il prodotto cartesiano $A \times B$ fra due insiemi.

1.12 Realizzare, in modalità funzionale ed in modalità procedurale le operazioni insiemistiche di *unione* ($\cup$), *intersezione* ($\cap$) e *differenza* ($\setminus$) fra insiemi.

1.13 Completare l'algoritmo del crivello di Eratostene (Algoritmo 7).

1.14 Realizzare l'algoritmo del crivello di Eratostene memorizzando i numeri in delle sequenze ed elaborando le sequenze.

1.15 Dividere un insieme di numeri in due sottoinsiemi in modo che la differenza fra la somma degli elementi dei due sottoinsiemi sia minima.

1.16 Invertire sul posto una pila di blocchetti.

1.17 Eliminare l'elemento alla base di una pila.

1.18 Eliminare un dato numero di elementi posti in fondo ad una pila.

1.19 Estrarre l'elemento di base di una pila e metterlo in testa alla stessa pila.

1.20 Ordinare una pila di numeri, mettendo l'elemento più grande alla base della pila (e l'elemento più piccolo in cima alla pila). È ammesso usare altre pile ausiliarie.

1.21 Date tre pile contenenti numeri $1, 2, 3$, separare i numeri mettendo tutti gli 1 nella prima pila, i 2 nella seconda ed i 3 nella terza. Non è ammesso usare altre pile ausiliarie.

1.22 Calcolare la media dei valori contenuti in una sequenza di numeri.

1.23 Calcolare la media dei valori contenuti in una sequenza di numeri, escludendo dal calcolo i valori uguali a 0.

1.24 Costruire una sequenza composta dai primi n numeri di Fibonacci.

1.25 Una sequenza è composta da numeri 0 o 1. Determinare il numero naturale corrispondente interpretando i valori della sequenza come cifre della rappresentazione binaria del numero. Ad esempio, data la sequenza $[1, 0, 1, 1]$, si dovrà determinare il numero 11.

1.26 Determinare la sequenza delle cifre binarie di un dato numero naturale. Ad esempio, dato il numero 11, si dovrà determinare la sequenza $[1, 0, 1, 1]$.

1.27 Decidere se una sequenza è composta da elementi tutti uguali fra loro.

1.28 Decidere se una sequenza è composta da elementi tutti distinti fra loro.

1.29 Determinare il numero di elementi distinti presenti in una sequenza.

1.30 Decidere se una sequenza è ordinata (dal valore più piccolo a quello più grande).

1.31 Decidere se in una sequenza è presente un dato valore.

1.32 Determinare quante volte è presente un dato valore in una sequenza.

1.33 Determinare quante volte è presente un dato valore in una sequenza ordinata.

1.34 Eliminare da una sequenza gli elementi duplicati, lasciando una sola occorrenza di ciascun elemento.

1.35 Creare e riempire una matrice con i valori della tavola pitagorica della moltiplicazione.

1.36 Calcolare il prodotto dei numeri contenuti in

1. un insieme
2. una pila
3. una sequenza
4. una matrice

1.37 Ricercare se un dato elemento è presente in

1. un insieme
2. una pila
3. una sequenza
4. una matrice

1.38 Determinare quante volte un dato elemento è presente in

1. un insieme
2. una pila
3. una sequenza
4. una matrice

1.39 Basandosi sulla definizione di numero naturale descritta nell'esempio 1.10.1, scrivere il numero 3 mediante insiemi.

1.40 Basandosi sulle definizioni di coppia descritte nell'esempio 1.10.2, descrivere la terna (a, b, c) mediante insiemi

1.41 Eliminare da un insieme di numeri naturali i multipli di un dato numero. È ammesso utilizzare altre strutture elementari ausiliarie.

1.42 Estrarre da un insieme di numeri il valore più piccolo. Gestire la situazione di insieme vuoto.

1.43 Una matrice 365×24 contiene le temperature registrate in un luogo nelle 24 ore di tutti i giorni dell'anno. Determinare la massima escursione termica verificatasi.

2

IMPILARE

La semplice costruzione di una casa utilizzando una dozzina di blocchetti di legno richiederebbe di vagliare un tale numero di possibilità che a un bambino, per farlo, non basterebbe una vita intera.

Marvin Minsky, *La società della mente*

Un interessante contesto operativo è fornito dal mondo delle *pile di blocchetti*: questo ambiente risulta particolarmente istruttivo in quanto può essere facilmente simulato e sperimentato fisicamente utilizzando delle pile di blocchetti o mazzi di carte da gioco. Le pile di blocchetti hanno la particolare caratteristica di fisicità che permette di pensare di averle lì davanti e ragionare su come manipolare i blocchetti per risolvere il problema considerato; di più, il procedimento può essere simulato agendo su una pila fisica. Dal punto di vista concettuale ed algoritmico ci si potrebbe limitare al singolo termine *pila*; l'aggiunta della qualificazione *di blocchetti* sta proprio ad indicare la specifica caratterizzazione fisica del contesto in cui si opera e sollecita la simulazione pratica dei processi di elaborazione.

2.1 Il problema della torre di Hanoi

Il *problema della torre di Hanoi* è un gioco ideato nel 1883 dal matematico francese Edouard Lucas. Il gioco prende spunto dalla leggenda secondo la quale nel tempio di Benares, in India, i monaci avevano il compito di spostare una piramide di 64 dischi in oro, disposti uno sopra l'altro, dal più grande posto in basso al più piccolo posto in alto, da un piolo di diamante ad un altro, spostando un solo disco alla volta, utilizzando un terzo piolo come appoggio, con il vincolo che ogni disco debba poggiare sopra uno di dimensioni più grandi. Secondo la leggenda il mondo avrà fine quando i monaci avranno finito il proprio lavoro. Sintetizzando quanto tramandato dalla leggenda, il problema è generalmente noto nella seguente formulazione:

> *Sono dati tre pioli e n dischi di diversi diametri. Inizialmente tutti gli n dischi sono impilati su un piolo in modo tale che nessun disco sia disposto sopra di un disco più piccolo. Il problema consiste nello spostare la pila degli n dischi da un piolo ad un altro, muovendo un solo disco alla volta, potendo utilizzare il terzo piolo come supporto ausiliario e rispettando il vincolo che nessun disco sia disposto sopra uno più piccolo.*

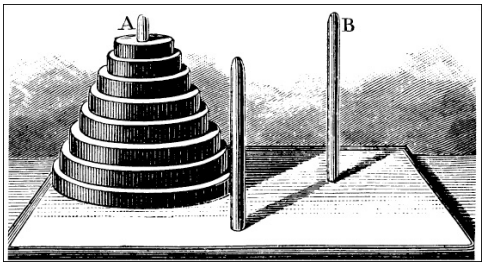


Figura 2.1: Problema della torre di Hanoi con 8 dischi, come proposto nel testo originale di Edouard Lucas.

Il problema suggerisce le seguenti questioni:

1. È possibile risolvere il problema per qualsiasi numero n di dischi?
2. Determinare il numero di spostamenti necessari per spostare n dischi.
3. Determinare la sequenza di spostamenti necessari per spostare n dischi.
4. Determinare e descrivere un procedimento per spostare gli n dischi.

Partiamo dall'ultima questione: individuato un algoritmo che la risolve avremo indirettamente risolto anche le precedenti. Denotiamo con A, B, C le pile in cui impilare i dischi e supponiamo che inizialmente gli n dischi siano impilati in A e, usando B come pila ausiliaria, debbano essere spostati in C. Iniziamo ad affrontare il problema a partire dal caso particolare di 3 dischi (figura 2.2).

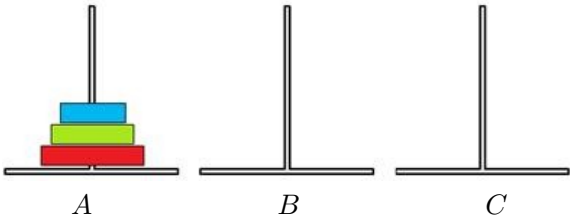


Figura 2.2: Problema della torre di Hanoi con 3 dischi.

In questo caso, e con maggiore evidenza per i casi di 1 e 2 dischi, la soluzione emerge abbastanza chiaramente in quanto è breve la distanza fra la situazione iniziale e l'obiettivo da raggiungere: la mente si orienta senza avere bisogno di una particolare strategia che la guidi. In modo figurato la soluzione è descritta nella figura 2.3.

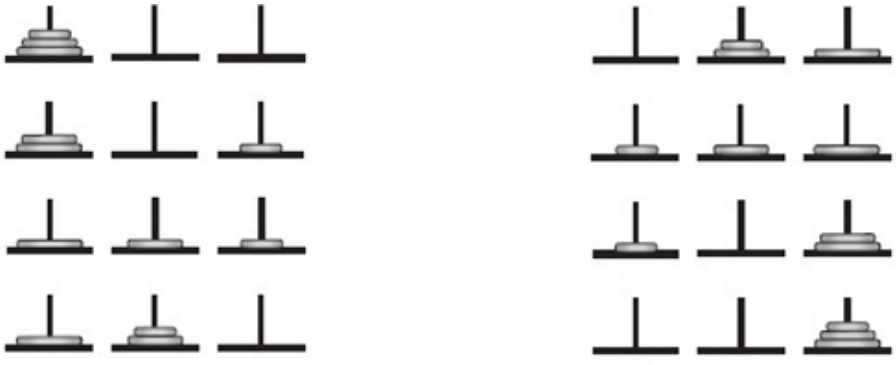


Figura 2.3: Soluzione del problema della torre di Hanoi con 3 dischi.

Per il caso relativo a 3 dischi la soluzione, corrispondente a quella descritta nella figura nella figura 2.3, è data dalla sequenza delle 7 mosse descritte nell'algoritmo 1.

Algoritmo 1 - Spostamento di 3 dischi da A a C usando B

1: $sposta(A, C)$
2: $sposta(A, B)$
3: $sposta(C, A)$
4: $sposta(A, C)$
5: $sposta(B, A)$
6: $sposta(B, C)$
7: $sposta(A, C)$

Concordando di denotare con la coppia (P, Q) una generica azione consistente nello spostamento del disco di testa della pila P alla testa della pila Q, la

soluzione rappresentata dal precedente algoritmo può essere espressa in modo sintetico mediante la seguente sequenza di coppie:

$$[(A,C),(A,B),(C,A),(A,C),(B,A),(B,C),(A,C)]$$

Quella sopra proposta (nelle diverse notazioni) è solo la soluzione della particolare istanza del problema corrispondente a 3 dischi. Per valori maggiori di 3 dischi si rischia i perdere il controllo della situazione.

Una soluzione ricorsiva

Cerchiamo ora di individuare una soluzione del problema per un generico numero n di dischi. Osserviamo che ci dovrà essere una mossa, all'interno della sequenze di mosse del processo risolutivo, in cui si dovrà spostare in C il disco più grande; questo disco dovrà provenire da A o da B; ma non potrà arrivare da B in quanto alla situazione precedente si sarebbe stati alla posizione in cui il disco più grande è in A e B è vuota; ma questo avrebbe allungherebbe la sequenza di mosse (figura 2.4).

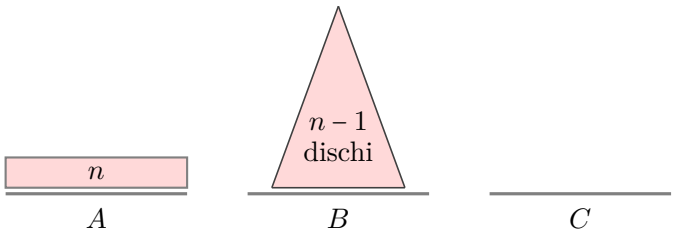


Figura 2.4: Situazione intermedia del problema della torre di Hanoi.

Una volta spostato il disco più grande in C, esso non dovrà più essere rimosso. Questa particolare mossa costituisce un importante punto di taglio della sequenza delle mosse che costituiscono il processo risolutivo e conduce alla soluzione ricorsiva descritta nell'algoritmo 2, che comporta lo spostamento di due pile di $n-1$ dischi.

Algoritmo 2 - Spostamento di n dischi da A a C usando B

Input: numero n dei dischi, pile A, B, C

1: **if** $n > 0$ **then**
2: sposta la torre dei primi $n-1$ dischi da A a B usando C
3: sposta il disco più grande da A a C
4: sposta la torre di $n-1$ dischi da B a C usando A
5: **end if**

Un'altra versione dell'algoritmo 2, più vicina alla codifica tipica di un linguaggio di programmazione, è riportata nell'algoritmo 3. In questo caso, una chiamata del tipo $hanoi(4,A,B,C)$ ha l'effetto di spostare 4 dischi dal piolo A al piolo C usando il piolo B.

Algoritmo 3 - $hanoi(n, A, B, C)$: sposta n dischi da A a C usando B

Input: numero n dei dischi, pile A, B, C

```
1: if n > 0 then
2:     hanoi(n − 1, A, C, B)
3:     spostaDisco(A, C)
4:     hanoi(n − 1, B, A, C)
5: end if
```

Una soluzione iterativa

La soluzione riportata nell'algoritmo 3 può essere utilizzata sia come base per sviluppare l'analisi di complessità del problema che come trampolino di lancio per l'individuazione di altre soluzioni alternative.

È interessante rappresentare l'algoritmo 3 mediante un albero (delle chiamate), come descritto nella figura 2.5.

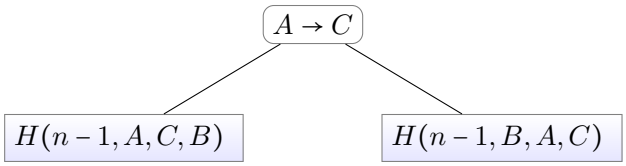


Figura 2.5: Albero delle chiamate ricorsive innescato dalla chiamata $hanoi(n, A, B, C)$: H corrisponde alla chiamata *hanoi* e $A \to C$ corrisponde all'operazione di spostamento di un disco dalla pila A alla pila C.

Le chiamate ricorsive H presenti nell'albero riportato nella figura 2.5 possono essere esplose passo dopo passo fino ad arrivare ad un albero avente tutti i nodi elementari della forma $X \to Y$, corrispondenti all'azione di movimento di un singolo disco. Per l'istanza $n = 3$ si ottiene l'*albero di Hanoi* descritto nella figura 2.6.

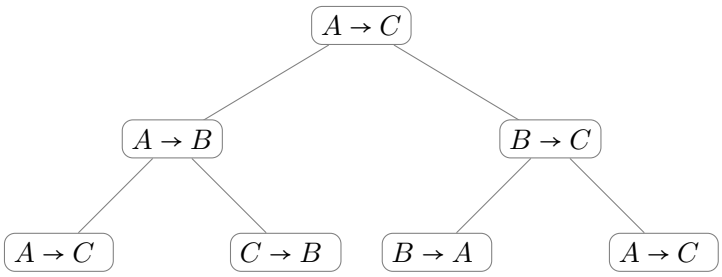
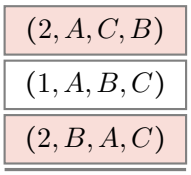


Figura 2.6: Albero di Hanoi corrispondente all'istanza con 3 dischi, ottenuto sviluppando le chiamate a partire dalla radice iniziale $H(3, A, B, C)$.

Una visita in inordine, corrispondente alla sequenza delle chiamate dell'algoritmo 3, dell'albero di Hanoi riportato in fig. 2.6 fornisce l'esatta sequenza delle azioni di spostamento di singoli dischetti per risolvere il problema.

Le precedenti considerazioni sull'albero delle chiamate ci aprono la strada per un algoritmo iterativo per il problema della torre di Hanoi. Utilizziamo una struttura a pila dove inseriamo le varie istanze dei problemi da risolvere; gli elementi della pila saranno delle quaterne della forma (k, X, Y, Z) che avrà il seguente significato: *Sposta* k *dischi dalla pila* X *alla pila* Z *utilizzando la pila* Y *come appoggio temporaneo.* Ad esempio la seguente pila delle chiamate



è indicativa del fatto che il prossimo problema da risolvere è descritto dalla quadrupla $(2, A, C, B)$, che corrisponde a spostare 2 dischi da A a B usando C come appoggio temporaneo; risolto questo, si dovranno risolvere i due sotto problemi corrispondenti alle due sottostanti quadruple. La soluzione di un problema registrato nella pila può esaurirsi in una singola operazione di spostamento di 1 disco oppure nello spostamento di 1 disco fra due chiamate corrispondenti a due istanze di problema con pile composte da un disco in meno; ciascuna di queste istanze dovrà essere a sua volta inserita nella pila. Tutte queste considerazioni permettono di scrivere un algoritmo iterativo (Algoritmo 4) che opera sugli identificatori delle pile ed usa una pila nella quale registrare i sotto problemi di spostamento.

Algoritmo 4 - Soluzione iterativa del problema della torre di Hanoi

Input: numero n dei dischi da spostare, identificatori X, Y, Z delle pile
Output: sequenza degli spostamenti dei dischi

```
1:  s ← [ ⟩
2:  push(s, (n, X, Y, Z))
3:  while ¬ isempty(s) do
4:      (n, X, Y, Z) ← top(s)
5:      pop(s)
6:      if n = 1 then
7:          sposta(X, Z)
8:      else
9:          push(s, (n − 1, Y, X, Z))
10:         push(s, (1, X, Y, Z))
11:         push(s, (n − 1, X, Z, Y))
12:     end if
13: end while
```

□

Osservazione. Quanto visto in questo paragrafo costituisce un interessante intarsio di tre tipici concetti informatici: *iterazione*, *ricorsione* e *strutture dati*; la correlazione fra questi tre concetti può essere sintetizzata mediante una sorta di equazione concettuale della forma *Ricorsione = Iterazione + Pila*.

Analisi della complessità

L'analisi della complessità del problema della torre di Hanoi consiste nel determinare il numero $T(n)$ di spostamenti di dischi necessari per spostare una pila di n dischi dal piolo origine al piolo destinazione. Il calcolo del valore $T(n)$ può essere basato sulla struttura ricorsiva dell'algoritmo 3 dal quale si ricava immediatamente la relazione

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

da cui

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

La soluzione in forma chiusa di questa relazione di ricorrenza può essere ottenuta sviluppando in modo telescopico l'espressione $T(n-1)$ che compare sulla destra:

$$\begin{aligned} T(n) &= T(n-1) + 1 + T(n-1) \\ &= 2(2T(n-2) + 1) + 1 \\ &= 2^2 T(n-2) + 2 + 1 \\ &= \cdots \\ &= 2^k T(n-k) + 2^{k-1} + \cdots + 2^2 + 2 + 1 \\ &= \cdots \\ &= 2^n T(0) + 2^{n-1} + \cdots + 2^2 + 2 + 1 \\ &= 2^{n-1} + \cdots + 2^2 + 2 + 1 \end{aligned}$$

La somma $2^{n-1} + \cdots + 2^2 + 2 + 1$ costituisce una serie geometrica di ragione 2 composta da n termini e vale $2^n - 1$. Pertanto:

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

Altre due soluzioni iterative

Cerchiamo ora una soluzione iterativa al problema. Partiamo dall'osservazione che in ogni istante, nel caso in cui i dischi non si trovino tutti in una sola pila, si possono individuare univocamente i seguenti due dischi:

- *piccolo*: è il disco più piccolo fra quelli in testa alle pile
- *medio*: è il disco secondo in grandezza fra quelli in testa alle pile

Il disco più piccolo può essere mosso in due modi: in senso orario o in senso antiorario (immaginando di avere i tre pioli disposti sui vertici di un triangolo); il secondo disco può essere mosso in un unico modo. Basandosi su queste considerazioni, una soluzione iterativa è espressa nell'algoritmo 5.

Algoritmo 5 - Spostamento di n dischi da A a C usando B

Input: numero n dei dischi, pile A, B, C
1: $S \leftarrow if(n$ è dispari, *antiorario*, *orario*$)$
2: muovi il disco *piccolo* in senso S
3: **while** i dischi non sono tutti in C **do**
4: muovi il disco *medio*
5: muovi il disco *piccolo* in senso S
6: **end while**

Si può dimostrare che l'algoritmo 5 è *corretto* (risolve effettivamente il problema) ed è *ottimale* (esegue $2^n - 1$ movimenti); la dimostrazione di queste proprietà non è facile. Questo algoritmo risulta di facile esecuzione per un esecutore umano; richiede un po' di attenzione per essere implementato in un linguaggio di programmazione tradizionale; in particolare le istruzioni 2., 4. e 5. richiedono che venga mantenuta una rappresentazione della situazione di ciascuna pila.

Descriviamo ora un'altra soluzione iterativa. In ogni istante, considerate due generiche pile X ed Y, non entrambe vuote, esiste un unico spostamento di un disco fra le due pile (in un verso o nell'altro); indichiamo con $muovi(X, Y)$ questa operazione che si incarica di stabilire il verso dello spostamento del disco fra le due pile X ed Y ed eseguirlo. Basandosi su questa operazione si può esprimere una soluzione iterativa al problema della torre di Hanoi mediante l'algoritmo 6, dove il controllo di terminazione del ciclo (tutti i dschi in C) va eseguito dopo ogni operazione *muovi*.

Algoritmo 6 - Spostamento di n dischi da A a C usando B

Input: numero n dei dischi, pile A, B, C
1: **while** i dischi non sono tutti in C **do**
2: **if** n è pari **then**
3: $muovi(A, C)$
4: $muovi(A, B)$
5: $muovi(B, C)$
6: **else**
7: $muovi(A, B)$
8: $muovi(A, C)$
9: $muovi(B, C)$
10: **end if**
11: **end while**

Anche l'algoritmo 6 è *corretto* ed *ottimale*, anche se di non facile dimostrazione. L'esecuzione dell'operazione *muovi* richiede che l'esecutore abbia in ogni momento la rappresentazione della situazione di ciascuna pila per poter stabilire il verso dello spostamento fra le due pile. L'algoritmo risulta di facile esecuzione per una persona ed è di facile implementazione in un linguaggio di programmazione.

2.2 Il mondo delle pile di blocchetti

Un classico contesto in cui si pongono e si risolvono problemi di spostamento è costituito dalle *pile di blocchetti*: ci sono dei blocchetti indistinguibili uno dall'altro oppure caratterizzati da attributi quali colore, grandezza o il numero impresso su di essi; i blocchetti sono disposti su delle pile; è ammesso spostare un solo blocchetto alla volta dalla testa di una pila alla testa di un'altra; è ammesso accedere, senza spostarlo, al solo elemento di testa di una pila; inoltre è ammesso controllare se una pila è vuota. Le varie pile vengono denotate mediante un nome o un numero che identifica la posizione in cui si trova la pila. Una tipica situazione che permette di formulare e risolvere interessanti problemi è costituita da tre pile.

Blocchetti indistinguibili

Le caratteristiche dei blocchetti delle pile influenzano in modo decisivo le tipologie di problemi che si riesce a risolvere. In generale, maggiori e più raffinati sono gli attributi dei blocchetti, più ampie sono le classi di problemi che si riesce a risolvere. Il caso estremo è quando i blocchetti non hanno alcuna caratteristica se non quella di "essere blocchetti". In pratica l'informazione che un blocchetto porta con sé è quella di "unità". In questa situazione una pila di blocchetti è equivalente ad un mucchio di sassolini della stessa cardinalità e la pila perde la sua valenza di *struttura sequenziale*, degradando al rango di *insieme di elementi*. Dal punto di vista dei problemi, affermare che i blocchetti sono indistinguibili equivale ad affermare che è indifferente l'ordine con il quale si troveranno i blocchetti alla fine del processo. Continueremo a chiamarla pila anche se corrisponde al concetto di insieme, e continueremo ad utilizzare le operazioni già descritte per le pile.

Blocchetti colorati

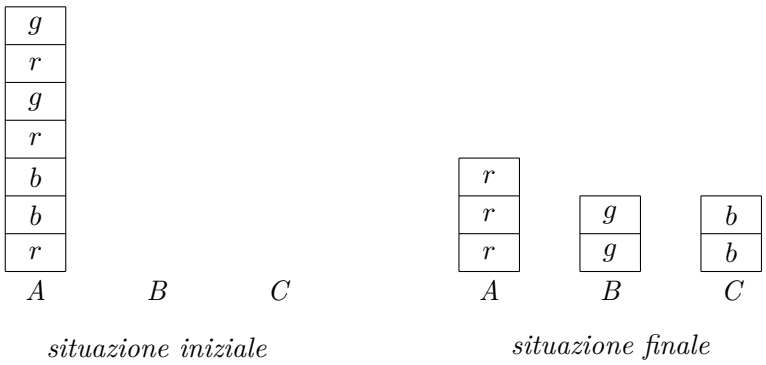
Consideriamo delle pile di blocchetti colorati. L'aggiunta dell'attributo *colore* permette di considerare una più vasta gamma di problemi basati sulla comparazione dei blocchetti. In questo nuovo mondo a colori si ammette che l'esecutore sia in grado di distinguere i colori dei blocchetti e di compararli $(=, \neq)$ in base al loro colore [1].

Ovviamente, lavorando su pile, risultano confrontabili solamente i blocchetti di testa delle due pile. Questa nuova tipologia dei blocchetti apre un vasto repertorio di interessanti situazioni. Nel seguito utilizzeremo il termine "blocchetti colorati" come equivalente a "blocchetti comparabili".

Problema 2.2.1 Nella pila A ci sono blocchetti di tre colori e le pile B e C sono vuote. Separare i blocchetti nelle tre pile in base al colore, in modo che in ogni pila ci siano blocchetti dello stesso colore.

[1] Il fatto di avere dei blocchetti colorati comporta il vantaggio di non indurci ad utilizzare operazioni di confronto d'ordine o operazioni aritmetiche come avrebbe potuto succedere se avessimo utilizzato dei numeri per discriminare i diversi blocchetti.

Soluzione. Ci appoggeremo alla figura che segue, corrispondente ad un'istanza di problema, in modo che ci possa fare da guida nello sviluppo della soluzione del problema.



situazione iniziale *situazione finale*

Adottando la metodologia top-down, il problema può essere scomposto come si vede nell'algoritmo 7.

Algoritmo 7 - Separazione di una pila di blocchetti di 3 colori

Require: pila A di blocchetti di 3 colori, pile B e C vuote
Ensure: pile A, B e C con i blocchetti suddivisi per colore
1: suddividi da A in B (un colore) e C (gli altri due colori)
2: suddividi da C a A (un colore) e B (l'altro colore)
3: sposta i blocchetti del colore di testa di B a C

Nell'algoritmo 7 si evidenziano i sottoproblemi P_1, P_2 e P_3 corrispondenti alle righe 1, 2 e 3. La soluzione è riportata negli algoritmi 8, 9 e 10.

Algoritmo 8 - Soluzione sottoproblema P_1

Require: pila A di blocchetti di 3 colori, pile B e C vuote
Ensure: pila A vuota, pila B con un colore, pila C con gli altri due colori
1: $sposta(A, B)$
2: **while** $\neg\, isempty(A)$ **do**
3: **if** $top(A) = top(B)$ **then**
4: $sposta(A, B)$
5: **else**
6: $sposta(A, C)$
7: **end if**
8: **end while**

Algoritmo 9 - Soluzione sottoproblema P_2

Require: pila A vuota, pila B con un colore, pila C con gli altri due colori
Ensure: pila A con un colore, pila B con secondo colore in alto, pila C vuota

```
1: sposta(C, A)
2: while ¬ isempty(C) do
3:    if top(C) = top(A) then
4:       sposta(C, A)
5:    else
6:       sposta(C, B)
7:    end if
8: end while
```

Algoritmo 10 - Soluzione sottoproblema P_3

Require: pila A con un colore, pila B con secondo colore in alto, pila C vuota
Ensure: pile A, B e C con i blocchetti suddivisi per colore

```
1: sposta(B, C)
2: while top(B) = top(C) do
3:    sposta(B, C)
4: end while
```

□

Osservazione. Gli algoritmi 7, 8, 9 e 10 agiscono modificando gli stati (configurazioni) delle tre pile A, B e C. In ciascuno dei quattro algoritmi sono riportate le precondizioni assunte dall'algoritmo e le postcondizioni generate dall'algoritmo. Gli algoritmi 8, 9 e 10 sono sequenzialmente relazionati fra loro nel senso che la postcondizione di un algoritmo diventa la precondizione dell'algoritmo che segue.

Problema 2.2.2 Stabilire se due pile di blocchetti colorati sono uguali fra loro. Al termine del controllo le due pile dovranno essere uguali a com'erano all'inizio.

Soluzione. Il problema è risolto mediante l'algoritmo 11. Viene usata una pila ausiliaria C.

Algoritmo 11 - Controllo se due pile di blocchetti colorati sono uguali

Input: pile A e B di blocchetti confrontabili
Output: `TRUE` se e solo se le due pile A e B sono uguali

```
1: C ← pila vuota
2: while ¬ isempty(A) ∧ ¬ isempty(B) ∧ top(A) = top(B) do
3:     sposta(A, C)
4:     sposta(B, C)
5: end while
6: uguali ← isempty(A) ∧ isempty(B)
7: while ¬ isempty(C) do
8:     sposta(C, B)
9:     sposta(C, A)
10: end while
11: return uguali
```

□

Problema 2.2.3 Stabilire se una pila di blocchetti colorati è composta da elementi tutti uguali. È ammesso utilizzare un'altra pila ausiliaria.

Soluzione. Il problema risulta è risolto mediante l'algoritmo 12.

Algoritmo 12 - Controllo se i b. di una pila sono tutti dello stesso colore

Input: pila A di blocchetti confrontabili
Output: `TRUE` se e solo se A è composta da blocchetti tutti uguali

```
1: if ¬ isempty(A) then
2:     B ← [ ⟩
3:     sposta(A, B)
4:     while ¬ isempty(A) ∧ (top(A) = top(B)) do
5:         sposta(A, B)
6:     end while
7:     r ← isempty(A)
8:     rovescia(B, A)
9: else
10:    r ← TRUE
11: end if
12: return r
```

□

Problema 2.2.4 Stabilire se una pila di blocchetti è composta da elementi tutti distinti. È ammesso utilizzare altre due pile ausiliarie.

Soluzione. Questo problema è all'apparenza simile a quello precedente, ma è strutturalmente diverso, ed ammette una soluzione sostanzialmente diversa e di diversa complessità computazionale. Il problema è risolto nell'algoritmo 13.

Algoritmo 13 - Controllo se una pila è composta da blocchetti tutti distinti

Input: pila A di blocchetti confrontabili
Output: `TRUE` se e solo se A è composta da blocchetti tutti distinti

1: $B \leftarrow [\,\rangle$
2: $C \leftarrow [\,\rangle$
3: *distinti* $\leftarrow$ `TRUE`
4: **while** $\neg$ *isempty*$(A) \wedge$ *distinti* **do**
5: *sposta*(A, B)
6: controlla che tutti i blocchetti di A siano distinti da *testa*(B), impilandoli in C ed assegnando *distinti* $\leftarrow$ `TRUE` in caso di diversità
7: *rovescia*(C, A)
8: **end while**
9: *rovescia*(B, A)
10: **return** *distinti*

□

Problema 2.2.5 Eliminare da una pila gli elementi duplicati, lasciando una sola occorrenza di ciascun elemento.

Soluzione. Il problema è risolto nell'algoritmo 14. Rimane da esplicitare la soluzione del sotto problema alla linea 5.

Algoritmo 14 - Eliminare da una pila gli elementi duplicati

Require: pila A di blocchetti confrontabili
Ensure: la pila A è composta da blocchetti tutti distinti

1: $B \leftarrow [\,\rangle$
2: $C \leftarrow [\,\rangle$
3: **while** $\neg$ *isempty*(A) **do**
4: *sposta*(A, B)
5: sposta in C tutti gli elementi di A diversi da *top*(B), eliminando gli uguali a *top*(B)
6: *rovescia*(C, A)
7: **end while**
8: *rovescia*(B, A)

□

Problema 2.2.6 Eliminare da una pila i blocchetti di posizione dispari, contando le posizioni da 1 a partire dalla base della pila. È ammesso usare altre pile ausiliare ma non altre variabili (numeriche, booleane o altro tipo).

Soluzione. Il problema risulta facilmente risolvibile potendo usare una sola pila ausiliaria ed una variabile contatore agendo in questo modo: i blocchetti vengono trasferiti dalla pila originale a quella ausiliaria contando man mano i blocchetti; al termine di questa fase i blocchetti vengono alternativamente

eliminati e trasferiti nella pila originale, iniziando con l'operazione di trasferimento o eliminazione in base al test di parità sul numero dei blocchetti. Una variante di questa strategia consiste nell'usare una variabile booleana per testare la parità del numero dei blocchetti; tale variabile alternerà il proprio valore ad ogni spostamento di un blocchetto dalla pila originale alla pila ausiliaria.

Il vincolo di non poter usare alcuna variabile (né numerica né booleana), come richiesto dal problema, richiede una strategia un pò più articolata; è come disporre di un esecutore senza memoria, che deve espletare il procedimento in base alla situazione attuale che vede e, quindi, è come poter disporre della sola operazione di test per stabilire se una pila è vuota. Il procedimento è il seguente: vengono usate due pile ausiliarie; i blocchetti vengono trasferiti dalla pila originale alternativamente nelle due pile ausiliarie, un blocchetto in una ed un altro nell'altra; quando la pila originale si sarà svuotata, in base alla pila ausiliaria nella quale è stato trasferito l'ultimo blocchetto si desume se il numero di blocchetti è pari o dispari ed a seconda del caso si trasferisce i blocchetti di una delle due pile nella pila originale e l'altra pila ausiliaria viene svuotata eliminando i blocchetti in essa contenuti. In base al principio di complessità della soluzione, c'è da aspettarsi che l'algoritmo risolutivo sia meno breve e meno semplice dei due algoritmi corrispondenti alle due strategie descritte all'inizio. Il procedimento è descritto nell'algoritmo 15.

Algoritmo 15 - Eliminazione da una pila i blocchetti di posizione dispari

Require: pila A di blocchetti
Ensure: pila A senza i blocchetti di posizione dispari

```
 1: if vuota(A) then
 2:    B ← [ ⟩
 3:    C ← [ ⟩
 4:    sposta(A, B)
 5:    while ¬ isempty(A) ∧ ¬ isempty(B) do
 6:       sposta(A, C)
 7:       if isempty(A) then
 8:          rovescia(B, A)
 9:          svuota(C)
10:       else
11:          sposta(A, B)
12:          if isempty(A) then
13:             rovescia(C, A)
14:             svuota(B)
15:          end if
16:       end if
17:    end while
18: end if
```

□

2.3 Blocchetti numerati

Etichettando i blocchetti con dei numeri si apre un mondo di possibilità e di interessanti problemi: questo è dovuto principalmente all'esistenza di una relazione d'ordine fra i numeri.

Problema 2.3.1 È data una pila A costituita da blocchetti numerati ed una pila B vuota. Estrarre dalla pila A il blocchetto con il numero più grande, portandolo nella pila B, e riportare tutti gli altri nella pila A.

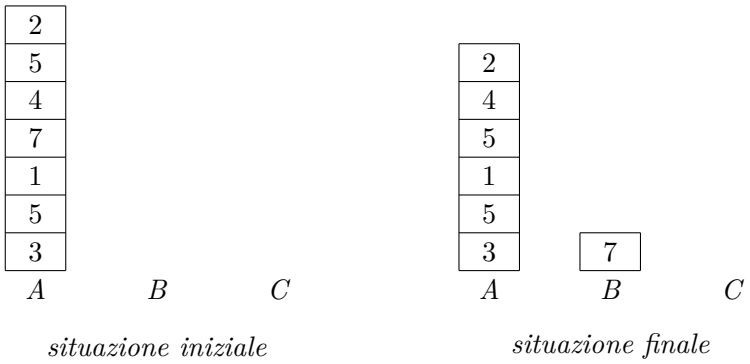


Figura 2.7: Situazione iniziale (a sinistra) e finale (a destra) per l'estrazione dell'elemento più grande.

Soluzione. La soluzione è riportata nell'algoritmo 16.

Algoritmo 16 - Estrazione del valore massimo

Require: pila A non vuota di blocchetti numerati, pila B vuota
Ensure: massimo in B e tutti gli altri in A

```
1:  C ← [ ⟩
2:  if ¬ isempty(A) then
3:      sposta(A, B)
4:  end if
5:  while ¬ isempty(A) do
6:      if top(A) > top(B) then
7:          sposta(B, C)
8:          sposta(A, B)
9:      else
10:         sposta(A, C)
11:     end if
12: end while
13: rovescia(C, A)
```

Le linee 2-4 dell'algoritmo 16 gestiscono anche il caso in cui la pila A sia inizialmente vuota; nel quale caso anche la pila B, alla fine, risulterà vuota. □

Problema 2.3.2 Estrarre da una pila di blocchetti numerati tutti i minimi (uno o più).

Soluzione. La soluzione è riportata nell'algoritmo 17.

Algoritmo 17 - Estrazione dei minimi da una pila di blocchetti numerati

Input: pila A di blocchetti numerati
Output: pila B con i minimi estratti dalla pila A

```
1:  C ← [ ⟩
2:  sposta(A, B)
3:  while ¬ isempty(A) do
4:     if top(A) < top(B) then
5:        rovescia(B, C)
6:        sposta(A, B)
7:     else if top(A) = top(B) then
8:        sposta(A, B)
9:     else
10:       sposta(A, C)
11:    end if
12: end while
13: rovescia(C, A)
```

□

Problema 2.3.3 Sono date 3 posizioni A, B, C. Inizialmente in A è posta una pila di almeno due blocchetti numerati. Usando una pila ausiliaria C, estrarre i due elementi più grandi e metterli in B, rimettendo in A tutti gli altri.

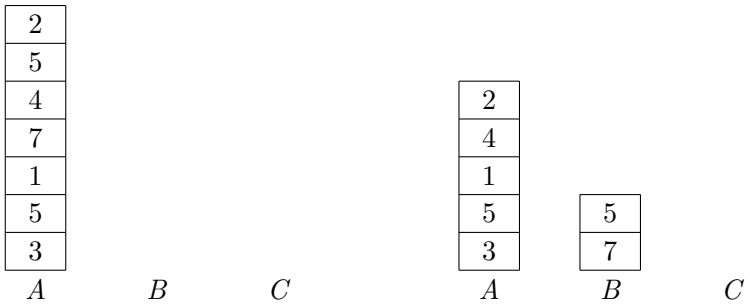


Figura 2.8: Situazione iniziale (a sinistra) e finale (a destra) per l'estrazione dei due elementi più grandi.

Soluzione. Il problema può essere risolto ripetendo 2 volte l'estrazione del valore massimo (da A a B). Per migliorare l'efficienza è possibile considerare una sola volta gli elementi della pila A e determinare contemporaneamente i

due valori più grandi. La soluzione si basa sull'idea di usare una pila ausiliaria C e di spostare gli elementi della pila A mantenendo valido l'invariante di avere in ogni istante i due valori più grandi in testa alle pile B e C con l'ulteriore condizione che sia $top(B) \geq top(C)$.

Il procedimento, dettagliato nell'algoritmo 18, si fonda sulla considerazione che ogni elemento x che viene preso da A dopo i primi due può generare le seguenti situazioni, mutuamente esclusive e complete; con M (testa di B) viene indicato il valore più grande e con m (testa di C) il secondo valore più grande:

C_1 : $x < m$: l'elemento x viene scartato

C_2 : $x \geq M$: vengono aggiornati i valori di m e di M

C_3 : $m \leq x < M$: viene aggiornato il valore di m

Le condizioni C_1, C_2 e C_3 corrispondono alle linee 8., 12. e 15. dell'algoritmo 18.

Algoritmo 18 - Estrazione dei due elementi più grandi

Require: pila A di blocchetti numerati, pila B vuota
Ensure: estrazione in B dei due elementi più grandi

```
 1: C ← [ ⟩
 2: sposta(A, B)
 3: sposta(A, C)
 4: if top(B) > top(C) then
 5:    scambia(B, C)
 6: end if
 7: while ¬ isempty(A) do
 8:    if top(A) < top(C) then
 9:       sposta(C, B)
10:       sposta(A, C)
11:       sposta(B, C)
12:    else if top(A) ≥ top(B) then
13:       sposta(B, C)
14:       sposta(A, B)
15:    else
16:       sposta(A, C)
17:    end if
18: end while
19: sposta(C, B)
20: rovescia(C, A)
```

□

2.4 Fusione di due pile di blocchetti

I procedimenti di spostamento di blocchetti sono logicamente contigui agli algoritmi di ordinamento di blocchetti. Una situazione intermedia è data dai procedimenti di fusione di pile di blocchetti ordinati in modo da ottenere una singola pila di blocchetti ordinati.

Si hanno due pile A e B di blocchetti numerati. Le due pile sono poste inizialmente nelle posizioni A e B. Nelle due pile i blocchetti sono disposti in modo tale che ogni blocchetto (esclusi quelli direttamente appoggiati sul piano) sia appoggiato sopra uno avente un numero maggiore o uguale del suo. Il problema consiste nel fondere [2] le due pile di blocchetti, costruendone una ordinata, composta dai blocchetti delle due pile originali. È ammesso l'uso di una terza pila C per spostare temporaneamente i blocchetti. Alla fine della trasformazione i blocchetti devono trovarsi nella pila A, ordinati dal più grande (appoggiato in basso) al più piccolo (in alto). Sviluppiamo, mediante la metodologia top-down, un algoritmo per risolvere il problema in questione, assumendo l'ipotesi di disporre di un esecutore in grado di spostare un solo blocchetto per volta (uno di quelli posti in testa alle pile) ed in grado di confrontare i numeri impressi sui blocchetti. Si tratta di un problema di ordinamento descrivibile mediante lo schema grafico illustrato nella figura 2.9.

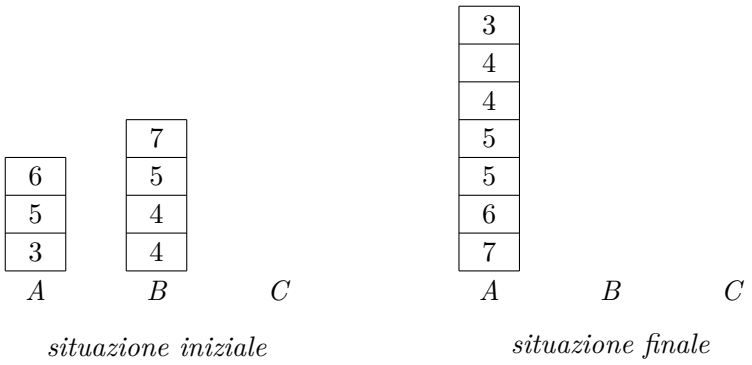


Figura 2.9: Situazione iniziale (a sinistra) e finale (a destra) per un problema di fusione di due pile di blocchetti.

Adottando la metodologia top-down, una prima scomposizione è descritta nell'algoritmo 19.

Algoritmo 19 - Fusione di due pile di blocchetti

Require: pile A e B ordinate crescentemente
Ensure: pila A ordinata crescentemente
1: usa una pila ausiliaria C
2: fondi le due pile A, B mettendo il risultato in C
3: trasla la pila C in A (senza rovesciarla)

[2] *Fondere*: unione di due sequenze ordinate in un'unica sequenza ordinata.

Raffinando l'algoritmo 19 si arriva all'algoritmo 20.

Algoritmo 20 - Fusione di due pile di blocchetti

Require: pile A e B ordinate crescentemente
Ensure: pila A ordinata decrescentemente
1: $C \leftarrow [\,\rangle$
2: **while** ci sono blocchetti in A ed in B **do**
3: **if** $top(A) > top(B)$ **then**
4: $sposta(A, C)$
5: **else**
6: $sposta(B, C)$
7: **end if**
8: **end while**
9: **if** $isempty(A)$ **then**
10: $rovescia(B, C)$
11: **else**
12: $rovescia(A, C)$
13: **end if**
14: $rovescia(C, B)$
15: $rovescia(B, A)$

2.5 Un robot assemblatore

Nell'esempio che segue una pila di blocchetti viene contestualizzata alla situazione in cui lo spostamento dei blocchetti viene eseguito dal braccio meccanico di un robot.

Su un nastro trasportatore arrivano al braccio di un robot assemblatore due tipologie di componenti: *pentole* e *coperchi*. Il braccio prende, uno alla volta, in sequenza come arrivano, le pentole ed i coperchi che arrivano senza un ordine prestabilito, ad esempio con una sequenza del tipo *pentola-coperchio-coperchio-pentola*. L'obiettivo consiste nell'assemblare su un altro nastro il prodotto finito costituito da una coppia $(pentola, coperchio)$. Il braccio può accedere solo alle estremità dei due nastri trasportatori. Il robot può parcheggiare temporaneamente le pentole ed i coperchi in una pila sulla quale è possibile operare, inserendo ed estraendo elementi, solo sulla parte superiore. L'obietttivo consiste nell'assemblare i prodotti finiti composti da una pentola con sopra un coperchio. La situazione è descritta nella figura 2.10.

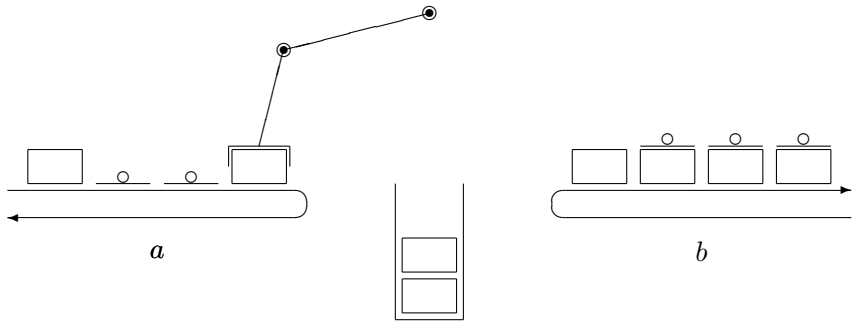


Figura 2.10: Il robot assemblatore.

Un passo preliminare per lo sviluppo della soluzione consiste nell'individuare delle azioni elementari per il robot utilizzando le quali descrivere un algoritmo per gestire il processo di assemblaggio. Adottiamo la strategia di avere la maggiore produttività possibile, ossia di avere in ogni istante il maggior numero di prodotti finiti assemblati. Ad esempio, una adeguata sequenza di azioni a partire dalla situazione descritta nella figura 2.11 è la seguente:

- *inserisci* la pentola dal nastro di ingresso alla pila
- *sposta* il coperchio dal nastro di ingresso al nastro di uscita
- *estrai* la pentola dalla pila e portala sul nastro di uscita
- *sposta* il coperchio dal nastro di ingresso al nastro di uscita

La precedente traccia di esecuzione suggerisce che l'attività del robot può essere composta mediante un'opportuna combinazione delle tre azioni elementari descritte schematicamente nella figura 2.11.

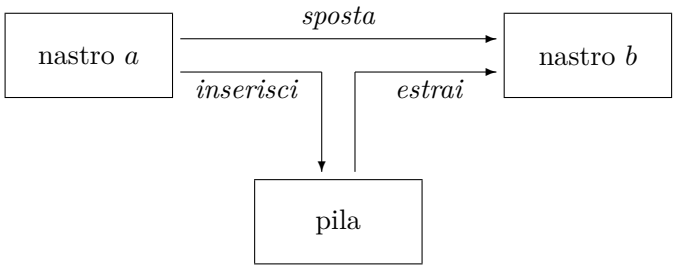


Figura 2.11: Le 3 azioni elementari per il robot assemblatore.

Useremo nel seguito i termini *ingresso*, *uscita* e *pila* rispettivamente per denotare il nastro di ingresso, il nastro di uscita e la pila. La combinazione sequenziale delle tre azioni elementari *sposta*, *inserisci* ed *estrai* dipende dallo *stato* del sistema complessivo il quale è determinato dallo stato dei tre sottosistemi *ingresso*, *uscita* e *pila*; i possibili valori di questi tre sottosostemi è descritto mediante la seguente lista di valori:

$ingresso \in \{pentola, coperchio\}$
$uscita \in \{pentola, coperchio\}$
$pila \in \{vuota, pentola, coperchio\}$

Coerentemente con queste convenzioni, useremo la notazione *pila=vuota* per denotare la condizione che la pila è vuota, *pila=pentola* per denotare che la pila contiene pentole, *ingresso=coperchio* per indicare che il prossimo componente sul nastro di ingresso è un coperchio, *uscita=pentola* per indicare che il prossimo componente richiesto per il nastro di uscita è una pentola, e così via per gli altri casi analoghi.

Una prima soluzione al problema si basa sull'analisi sistematica di tutti i possibili stati nei quali si può trovare il sistema e, in ciascun caso, si determina l'appropriata azione da intraprendere. Combinando fra loro gli *stati delle parti* del sistema si ottengono i possibili *stati del sistema*, descritti nella tabella 2.12, dove, nella colonna *azione* è riportata l'azione da intraprendere nel corrispondente stato.

stato	ingresso	uscita	pila	azione
s_1	*pentola*	*pentola*	*vuota*	*sposta*
s_2	*pentola*	*pentola*	*pentola*	*estrai*
s_3	*pentola*	*pentola*	*coperchio*	*sposta*
s_4	*pentola*	*coperchio*	*vuota*	*inserisci*
s_5	*pentola*	*coperchio*	*pentola*	*inserisci*
s_6	*pentola*	*coperchio*	*coperchio*	*estrai*
s_7	*coperchio*	*pentola*	*vuota*	*inserisci*
s_8	*coperchio*	*pentola*	*pentola*	*estrai*
s_9	*coperchio*	*pentola*	*coperchio*	*inserisci*
s_{10}	*coperchio*	*coperchio*	*vuota*	*sposta*
s_{11}	*coperchio*	*coperchio*	*pentola*	*sposta*
s_{12}	*coperchio*	*coperchio*	*coperchio*	*estrai*

Figura 2.12: Tabella degli stati del robot assemblatore e corrispondenti azioni da eseguire.

Volendo esprimere la soluzione in notazione algoritmica a partire dalla tabella 2.12, si è indotti a scrivere un algoritmo caso-per-caso, mediante una sequenza di costrutti condizionali che fanno perdere la struttura logica del processo di elaborazione.

In alternativa a quanto fatto sopra, si può adottare una strategia di ricerca della soluzione di tipo algoritmico; ciò richiede un pensiero sintetico organizzatore che conduce ad una prima forma di soluzione descritta nell'algoritmo 21. Questa scomposizione evidenzia che i due sottoproblemi P_1 e P_2 sono strutturalmente equivalenti e possono essere espressi mediante un algoritmo parametrico (algoritmo 22). Indicando con $assembla(x)$ l'algoritmo 22, la soluzione complessiva del problema si esprime come riportato nell'algoritmo 23, del tutto equivalente all'algoritmo 21.

Algoritmo 21 - Assemblaggio pentole e coperchi

1: **loop**
2: prendi una *pentola* dalla pila o dal nastro di ingresso e spostala sul nastro di uscita (P_1)
3: prendi un *coperchio* dalla pila o dal nastro di ingresso e spostalo sul nastro di uscita (P_2)
4: **end loop**

Algoritmo 22 - Assemblaggio componente $x \in \{pentola, coperchio\}$

1: **if** $pila = x$ **then**
2: *estrai*
3: **else**
4: **while** $ingresso \neq x$ **do**
5: *inserisci*
6: **end while**
7: *sposta*
8: **end if**

Algoritmo 23 - Assemblaggio pentole e coperchi

1: **loop**
2: *assembla*(*pentola*)
3: *assembla*(*coperchio*)
4: **end loop**

ESERCIZI

2.1 Risolvere il *problema della torre di Hanoi*, determinando la sequenza delle mosse da eseguire.

2.2 Si ha una pila di libri su un tavolo e si dispone di un esecutore capace di spostare un solo libro alla volta. La pila di libri si trova inizialmente sulla posizione A. Il problema consiste nel rovesciare sul posto la pila di libri, usando altre due posizioni ausiliarie di appoggio B e C.

2.3 E' data una pila A di 3 blocchetti numerati ed altre due pile B e C vuote. Distribuire i dischi nelle 3 pile A, B, C in modo che risultino ordinati ($testa(A) \leq testa(B) \leq testa(C)$).

2.4 È data una pila di blocchetti A ed altre due pile vuote B e C. I blocchetti sono sono in numero multiplo di 3. Spostare sulla pila B un terzo dei blocchetti presenti inizialmente in A; alla fine la pila A dovrà contenere il doppio dei blocchetti presenti in B e la pila C dovrà risultare vuota.

2.5 Stabilire se le due pile di blocchetti contengono lo stesso numero di blocchetti. È possibile usare un'altra pila ausiliaria. Alla fine i blocchetti devono trovarsi nelle posizioni originali, nello stesso ordine di come erano all'inizio.

2.6 Estrarre da una pila l'elemento alla base, mettendolo in testa alla pila. È possibile usare altre pile come appoggio.

2.7 Eliminare da una pila tutti i blocchetti presenti, ed esclusione dell'ultimo blocchetto alla base. Gestire anche il caso in cui la pila è vuota.

2.8 Si ha una pila A di blocchetti numerati di colore arancione ed una B di blocchetti numerati di colore blu. Scambiare di posto le due pile di blocchetti, utilizzando come appoggio ausiliario una terza posizione C, facendo in modo che i blocchetti, in ciascuna pila, mantengano la loro reciproca posizione che avevano prima dello spostamento.

2.9 Stabilire se due date pile contengono lo stesso numero di blocchetti. Gestire il caso in cui una o entrambe le pile sono vuote.

2.10 Stabilire se in una pila di blocchetti è presente un dato elemento. Al termine dell'elaborazione la pila deve risultare inalterata.

2.11 Stabilire se una pila di blocchetti numerati è ordinata (con l'elemento più grande alla base). Al termine dell'elaborazione la pila deve risultare inalterata.

2.12 Eliminare da una pila di blocchetti numerati i blocchetti replicati, lasciando nella pila una sola occorrenza di ciascun numero.

2.13 Eliminare da una pila di blocchetti numerati tutti i blocchetti con il valore più piccolo. Gestire la situazione di pila vuota.

2.14 Determinare il minimo di una pila di blocchetti numerati, mettendolo in testa alla pila stessa. È possibile usare altre pile ausiliarie.

2.15 Estrarre da una pila di blocchetti numerati, non necessariamente distinti, i

blocchetti con valore minimo, mettendoli in un'altra pila. È possibile usare altre pile ausiliarie.

2.16 È data una pila di blocchetti numerati. Stabilire se la pila è ordinata, con l'elemento più grande in basso. Al termine dell'elaborazione la pila dovrà risultare inalterata.

2.17 Ordinare una pila di blocchetti numerati, non necessariamante distinti, mettendo il blocchetto con valore più alto in basso e quello con valore più piccolo in alto. È possibile usare altre pile ausiliarie.

2.18 È data una pila di blocchetti numerati, ordinata, con l'elemento più grande in basso. Inserire nella pila un altro blocchetto numerato in modo da mantenere la proprietà di ordinamento della pila. È possibile usare altre pile ausiliarie. Usando il precedente algoritmo, ordinare una pila di blocchetti numerati.

2.19 È data una pila di blocchetti numerati, ordinata, con l'elemento più grande in basso. Eliminare dalla pila tutti gli elementi duplicati, lasciando una sola occorrenza di ciascun elemento. Ad esempio, la pila $[9, 7, 7, 7, 5, 4, 4, 2\rangle$ produce come risultato la pila $[9, 7, 5, 4, 2\rangle$.

2.20 E' data una pila A di blocchetti numerati, ordinata non decrescentemente, con il valore più grande in fondo ed il più piccolo in testa. Eliminare dalla pila A gli elementi duplicati, lasciando la pila A ordinata crescentemente e gli elementi scartati nella pila B. E' possibile utilizzare anche una terza pila C di appoggio. Valutare la complessità dell'algoritmo.

3

SEQUENZIARE

I programmi per computer usualmente operano su tabelle d'informazioni. Nella maggioranza dei casi queste tabelle non sono semplicemente masse amorfe di valori numerici; esse coinvolgono importanti relazioni strutturali fra i dati elementi.

D. Knuth,
The Art of Computer Programming

Una delle forme più frequenti di aggregazione di elementi è costituita dalle *sequenze* in cui gli elementi vengono aggregati in fila, uno dopo l'altro. Le sequenze specializzano il concetto di insieme, in quanto non sono solo un aggregato informe di elementi ma ogni elemento componente la struttura è caratterizzato da una sua specifica posizione all'interno della struttura.

Il meccanismo di sequenziazione fornisce il supporto per lo sviluppo di molti ed importanti algoritmi, come gli algoritmi di ricerca e di ordinamento. Inoltre, le sequenze costituiscono un meccanismo spesso utilizzato per realizzare particolari strutture di dati astratte.

3.1 Le sequenze

Completiamo in questo paragrafo alcune definizioni riguardanti le sequenze, già introdotte nel paragrafo 1.5. Richiamiamo che il termine *sequenza* (o *lista*) denota un aggregato di elementi, detti *atomi*, disposti in fila, uno dopo l'altro e che la scrittura $[a_0, \ldots, a_n]$ denota la sequenza costituita dagli elementi $a_0, \ldots, a_n$, mentre la scrittura $[\,]$ denota la *sequenza vuota*, ossia la sequenza composta da nessun elemento.

Una sequenza composta da elementi dello stesso tipo viene detta *omogenea*; altrimenti è detta *eterogenea*.

Esempio 3.1.1 - Un esempio di sequenza omogenea ed uno di di sequenza eterogenea:

$$[3, 2, 5, 4, 2, 8]$$
$$[255, 74, \texttt{TRUE}, 12, \text{"}cielo\text{"}]$$

Data una sequenza $a = [a_0, \ldots, a_n]$ e degli indici i, j tali che $0 \leq i \leq j \leq n$, la sequenza $[a_i, \ldots, a_j]$ dicesi *sottosequenza* di a e viene indicata con la scrittura $a[i..j]$ [1] . Se $i = 0$ la sottosequenza è detta *prefisso* (di a); se $j = n$ la sottosequenza è detta *suffisso* (di a).

Se h e k sono due numeri naturali con

$$range(h, k)$$

si denota la sequenza di numeri naturali da h a $k-1$ mentre con

$$range(k)$$

si denota la sequenza di numeri naturali da 0 a $k-1$.

In generale, una sequenza può essere formata da una sequenza di espressioni della forma $[e_0, e_1, \ldots, e_n]$. Nel momento in cui una sequenza viene considerata nella parte destra di un'assegnazione oppure come argomento in una chiamata, le varie espressioni che costituiscono i suoi elementi vengono valutate e viene generata una sequenza di valori, come evidenzia l'esempio che segue.

Esempio 3.1.2 - Consideriamo la seguente sequenza di assegnazioni coinvolgenti variabili intere e sequenze:

$$\begin{aligned} x &\leftarrow 10 \\ y &\leftarrow x + 1 \\ x &\leftarrow 20 \\ a &\leftarrow [x, y, x + y] \\ y &\leftarrow 30 \end{aligned}$$

Al termine di queste istruzioni la sequenza a vale $[20, 11, 31]$.

[1] Nel linguaggio Python viene usata la scrittura $a[i : j]$ che denota la sottosequenza $[a_i, \ldots, a_{j-1}]$.

3.2 Operazioni sulle sequenze

La costruzione di una sequenza $[a_0, \ldots, a_n]$ può essere vista come l'applicazione in notazione distribuita dell'operatore [ ] applicato agli operandi $a_0, \ldots, a_n$. Tale operazione viene detta *sequenziazione*. Oltre a questa ed alle operazioni già esaminate nel paragrafo 1.5, sulle sequenze ne sono disponibili molte altre.

L'operazione di *concatenazione* fra due sequenze produce come risultato una sequenza costituita dagli elementi della prima sequenza seguiti dagli elementi della seconda; l'operazione di concatenazione viene solitamente denotata mediante l'operatore binario infisso + come segue [2] :

$$[a_0, \ldots, a_n] \ + \ [b_0, \ldots, b_m] \rightarrow [a_0, \ldots, a_n, b_0, \ldots, b_m]$$

La sequenza vuota [ ] costituisce l'elemento neutro (sinistro e destro) per l'operazione di concatenazione. L'operazione di concatenazione gode della proprietà associativa, ossia, per generiche sequenze a, b, c, vale l'identità $(a + b) + c = a + (b + c)$; l'operazione di concatenazione non gode invece della proprietà commutativa, ossia, in generale, $a + b \neq b + a$.

Le operazioni di *inserimento in testa* ed *aggiunta in coda* di un elemento in una sequenza possono essere espresse combinando l'operatore [ ] di sequenzializzazione e l'operatore + di concatenazione, come illustrato dai seguenti esempi:

$$[x] \ + \ [p, q, r] \rightarrow [x, p, q, r]$$
$$[p, q, r] \ + \ [x] \rightarrow [p, q, r, x]$$

Sulle sequenze si può applicare l'operazione di *ripetizione*: il risultato della ripetizione di una sequenza $[a_0, \ldots, a_n]$ per una costante naturale k è costituito da una sequenza ottenuta concatenando k sequenze a, ossia

$$k * [a_0, \ldots, a_n] \rightarrow [\underbrace{a_0, \ldots, a_n, a_0, \ldots, a_n, \ldots}_{k \text{ volte } a_0, \ldots, a_n}]$$

L'espressione naturale k è detta *ripetitore*. Il valore $k = 1$ costituisce l'elemento neutro per l'operazione di ripetizione di sequenze, ossia $1 * a \rightarrow a$ per ogni sequenza a. Per convenzione si assume che $0 * a = [\,]$. L'operazione di ripetizione può essere espressa anche in notazione postfissa, nella forma $a * k$.

L'operazione di ripetizione di sequenze realizza anche il meccanismo di costruzione e dimensionamento iniziale: l'espressione $n * [x]$ produce una sequenza di n elementi x, ossia

$$k * [x] \rightarrow [\underbrace{x, \ldots, x}_{k \text{ volte } x}]$$

L'operazione di ripetizione di una sequenza per un numero gode di alcune

[2] Qualora non risulti ambiguo, nei testi il segno + dell'operatore di concatenazione viene talvolta omesso e le due sequenze da concatenare vengono semplicemente scritte affiancate; ad esempio, se a e b sono due sequenze, ab denota l'operazione di concatenazione fra le due sequenze a e b.

interessanti proprietà; in particolare, se a è una sequenza e h e k sono due numeri naturali, vale la proprietà

$$h * a + k * a = (h + k) * a$$

Nella precedente espressione si noti il diverso significato dei due operatori +.

Esempio 3.2.1 - A seguire sono riportati alcuni esempi di operazioni e corrispondenti calcoli sulle sequenze:

$$\begin{aligned} [1,2,3] + [4,5] &\rightarrow [1,2,3,4,5] \\ 5 * [0] &\rightarrow [0,0,0,0,0] \\ 4 * [1,2] &\rightarrow [1,2,1,2,1,2,1,2] \\ 2 * [3 * [4]] &\rightarrow [[4,4,4],[4,4,4]] \end{aligned}$$

Esempio 3.2.2 - Le operazioni di aggregazione e ripetizione di elementi possono combinarsi in tutti i modi possibili in modo da costituire un'effettiva algebra con proprie specifiche proprietà; segue qui un esempio di *calcolo* che illustra come si opera in quest'algebra:

$$\begin{aligned} [0,1] + 2 * (2 * [0] + [1]) + [0] &\rightarrow [0,1] + 2 * ([0,0] + [1]) + [0] \rightarrow \\ &\rightarrow [0,1] + 2 * [0,0,1] + [0] \rightarrow \\ &\rightarrow [0,1] + [0,0,1] + [0,0,1] + [0] \rightarrow \\ &\rightarrow [0,1] + [0] + [0,1] + [0] + [0,1] + [0] \rightarrow \\ &\rightarrow [0,1,0] + [0,1,0] + [0,1,0] \rightarrow \\ &\rightarrow 3 * [0,1,0] \end{aligned}$$

3.3 Schemi di elaborazione delle sequenze

La forma più elementare di elaborazione di una sequenza consiste nell'esaminare sequenzialmente tutti i suoi elementi. La modalità più duttile per elaborare gli elementi di una sequenza consiste nell'accedere agli elementi con la notazione ad indice, come descritto nel precedente paragrafo. A questo scopo sono predisposte le due forme di controlli *for* descritte negli algoritmi 1 e 2. Con queste forme di algoritmo gli elementi $a[i]$ risultano accessibili sia in lettura che in scrittura e possono quindi essere modificati.

Algoritmo 1 - Elaborazione sequenziale di una sequenza a

```
1: for i from 0 to len(a)−1 do
2:     elabora elemento a[i]
3: end for
```

Per indicare che gli elementi di una sequenza sono accessibili sequenzialmente si usa affermare che una sequenza è *iterabile*. Per le sequenze (come per ogni altra struttura iterabile (stringhe, insiemi, dizionari, ...)) sono previsti

Algoritmo 2 - Elaborazione sequenziale di una sequenza a

```
1: for i in range(len(a)) do
2:     elabora elemento a[i]
3: end for
```

degli schemi alternativi di accesso che permettono di accedere solo in modalità di lettura e, quindi, non consentono la modifica degli elementi. La forma più semplice di accesso agli elementi di un iterabile è descritta nell'algoritmo 3 e viene denotata come *accesso mediante iteratore implicito*.

Algoritmo 3 - Elaborazione di un iterabile mediante iteratore implicito

Input: iterabile a

```
1: for x in a do
2:     elabora elemento x
3: end for
```

Una forma alternativa equivalente a quella descritta nell'algoritmo 3 consiste nell'usare un iteratore che ispeziona l'iterabile; tale iteratore viene generato dalla sequenza mediante un'apposita funzione *iter*; lo schema complessivo di questa modalità di accesso è descritto nell'algoritmo 4 e viene detto *accesso mediante iteratore esplicito* (_ denota una variabile non utilizzata nel ciclo).

Algoritmo 4 - Elaborazione di un iterabile mediante iteratore esplicito

Input: iterabile a

```
1: k ← iter(a)
2: for _ in range(len(a)) do
3:     x ← next(k)
4:     elabora elemento x
5: end for
```

Osservazione. Gli schemi di elaborazione descritti negli algoritmi 3 e 4 sono indicati quando si deve esaminare tutta la sequenza in *lettura*, ossia senza apportare modifiche ai suoi elementi; gli schemi descritti negli algoritmi 1 e 2 sono più duttili e permettono di esaminare anche una porzione della sequenza (in base alla scelta degli estremi del ciclo *for*) ed inoltre permettono di modificare gli elementi della sequenza.

3.4 Algoritmi elementari sulle sequenze

Gli schemi di elaborazione esaminati nel precedete paragrafo permettono di scrivere la maggior parte degli algoritmi necessari per elaborare le sequenze.

Problema 3.4.1 Determinare la somma degli elementi di una sequenza.

Soluzione. Il problema può essere risolto mediante i seguenti due equivalenti algoritmi.

Algoritmo 5 - $somma(a)$: somma degli elementi di una sequenza numerica

Input: sequenza numerica a
Output: somma degli elementi della sequenza a

```
1: s ← 0
2: for x in a do
3:     s ← s + x
4: end for
5: return s
```

Algoritmo 6 - $somma(a)$: somma degli elementi di una sequenza numerica

Input: sequenza numerica a
Output: somma degli elementi della sequenza a

```
1: s ← 0
2: for i in range(len(a)) do
3:     s ← s + a[i]
4: end for
5: return s
```

Problema 3.4.2 Azzerare tutti gli elementi di una sequenza numerica.

Soluzione. In questo problema gli elementi devono essere modificati; pertanto bisogna utilizzare gli algoritmi 1 e 2. La soluzione del problema è fornita dal seguente algoritmo:

Algoritmo 7 - $azzera(a)$: azzeramento degli elementi di una sequenza

Input: riferimento alla sequenza a
Ensure: gli elementi della sequenza a sono azzerati

```
1: for i in range(len(a)) do
2:     a[i] ← 0
3: end for
```

Problema 3.4.3 Determinare il minimo ed il massimo di una sequenza.

Soluzione. Per questo problema il ciclo più indicato è quello descritto nell'algoritmo 2 in quanto, pur non essendo necessario modificare la sequenza in ingresso, permette di accedere nel ciclo solamente alla porzione di sequenza dalla posizione 1 alla posizione $len(a) - 1$, tralasciando l'elemento di posizione 0, già considerati alla linea 1 dell'algoritmo.

Algoritmo 8 - $minmax(a)$: minimo e massimo della sequenza a

Input: sequenza a
Output: $[minimo(a), massimo(a)]$

```
1: min ← a[0]
2: max ← min
3: for i in range(1, len(a)) do
4:     if a[i] < min then
5:         min ← a[i]
6:     else if a[i] > max then
7:         max ← a[i]
8:     end if
9: end for
10: return [min, max]
```

3.5 Costruzione di sequenze

Molti problemi sono caratterizzati dall'avere delle sequenze come risultato. In questi casi l'algoritmo risolvente il problema costruisce, generalmente mediante un controllo ciclico la cui forma dipende dallo specifico problema, una sequenza che costituirà alla fine il risultato del problema. Tale sequenza risultato viene spesso costruita accodando man mano gli elementi, a partire dalla sequenza iniziale vuota. Il seguente problema illustra questa tecnica.

Problema 3.5.1 Scomporre un numero naturale in fattori primi.

Soluzione. Facciamo riferimento all'usuale metodo di scomposizione descritto dallo schema riportato nello schema sotto, dal quale si ricava $126 = 2 \cdot 3^2 \cdot 7$. In questo caso il risultato può essere rappresentato dalla sequenza $[2, 3, 3, 7]$.

126	2
63	3
21	3
7	7
1	

Seguendo la traccia sopra, si arriva all'algoritmo 9.

Algoritmo 9 - Sequenza dei divisori primi di un numero naturale

Input: numero naturale n
Output: sequenza dei divisori primi di n
1: inizializza la sequenza vuota a dei divisori
2: considera $d = 2$ come primo potenziale divisore
3: **while** $n > 1$ **do**
4: **while** n è divisibile per d **do**
5: accoda d alla sequenza dei divisori
6: dividi n per d
7: **end while**
8: considera il numero primo successivo di d
9: **end while**
10: **return** sequenza a dei divisori primi

Nell'algoritmo 9 rimangono ancora da esplicitare le soluzioni di alcuni sottoproblemi; fra questi, l'unico di una qualche complessità compare alla linea 8. In prima approssimazione (anche se non ottimale) questo problema può essere risolto con l'istruzione "incrementa d di 1". La soluzione completa è riportata nell'algoritmo 10.

Algoritmo 10 - Sequenza dei divisori primi di un numero naturale

Input: numero naturale n
Output: lista dei divisori primi di n
1: $a \leftarrow [\,]$
2: $d \leftarrow 2$
3: **while** $n > 1$ **do**
4: **while** $(n \bmod d) = 0$ **do**
5: $a \leftarrow a + [d]$
6: $n \leftarrow n \operatorname{div} d$
7: **end while**
8: $d \leftarrow d + 1$
9: **end while**
10: **return** a

Osservando che nessun numero pari, escluso 2, è primo, l'algoritmo di scomposizione 10 risulta più efficiente se si gestisce a parte, fuori dal ciclo principale, il caso $d = 2$, si inizializza d con $d \leftarrow 3$ e ad ogni ciclo si incrementa d di 2 unità.

3.6 Generazione di sequenze

In molte situazioni serve generare una sequenza come risultato dell'elaborazione di un'altra sequenza. A questo scopo è predisposto il meccanismo denominato *list comprehension* che permette di generare delle sequenze basandosi su

altre sequenze; la sintassi è

$$[\textit{espressione}\ \textbf{for}\ \textit{valore}\ \textbf{in}\ \textit{iterabile}\ \textbf{if}\ \textit{condizione}\ ]$$

Questa espressione produce una sequenza contenente i valori dell'iterabile soddisfacenti alla condizione sottoposti alla valutazione dell'espressione. La clausola **if** è opzionale.

Esempio 3.6.1 - La sequenza $[1, 3, 5, 7, 9, 11, 13, 15]$ dei primi 8 numeri naturali dispari può essere generata mediante l'espressione

$$[2 * k + 1\ \textbf{for}\ k\ \textbf{in}\ range(8)]$$

Esempio 3.6.2 - La sequenza $[0, 4, 16, 36, 64]$ dei quadrati dei numeri pari minori di 10 può essere generata mediante l'espressione

$$[k * k\ \textbf{for}\ k\ \textbf{in}\ range(10)\ \textbf{if}\ k \bmod 2 = 0]$$

Esempio 3.6.3 - La sotto sequenza dei numeri pari di una sequenza a può essere generata come segue:

$$[k\ \textbf{for}\ k\ \textbf{in}\ a\ \textbf{if}\ k \bmod 2 = 0]$$

3.7 Le fliste

Le *fliste* [3] sono delle particolari sequenze caratterizzate da specifici operatori di accesso agli elementi. Le fliste costituiscono una importante struttura dati, così potente e duttile che si presta a supplire a tutte le altre strutture organizzative. Ne è conferma il fatto che alcuni importanti linguaggi di programmazione (Lisp, Prolog, Logo, Askell ed altri), pur con terminologie, sintassi ed impostazioni diverse, hanno le fliste come unica struttura dati disponibile.

Dal punto di vista organizzativo le fliste sono delle sequenze caratterizzate da specifici operatori per la creazione, per l'accesso agli elementi e per la loro elaborazione. Una flista può essere definita in modo ricorsivo come segue: è la sequenza vuota oppure una coppia (x, a) dove x è un atomo ed a una flista.

Le operazioni sulle fliste

Le operazioni sulle fliste sono definite secondo l'impostazione del paradigma funzionale: il risultato di un'operazione è sempre un atomo o una flista; inoltre le fliste sono gestite come oggetti immutabili in quanto non esistono operazioni per modificare una flista. Derogando un po' dal paradigma funzionale puro si ammette comunque il tradizionale operatore di assegnazione $\leftarrow$ fra fliste. Nell'elenco che segue sono descritti gli operatori (x si denota un atomo e a e b delle generiche fliste):

[3] Leggi: *effe liste*; ll termine *flista* deriva da *functional list.*

- $empty()$: costruttore di una flista vuota:

$$empty() \rightarrow [\,]$$

- $cons(x, a)$: costruttore di una flista avente x come primo elemento e gli elementi della flista a a seguire:

$$cons(x, [a_0, a_1, \ldots, a_n]) \rightarrow [x, a_0, a_1, \ldots, a_n]$$

- $first(a)$: funzione che ritorna come risultato il primo elemento della flista a:

$$first([a_0, a_1, \ldots, a_n]) = a_0$$

  Se $a = [\,]$ o $a =$ `NULL`, allora $first(a) =$ `NULL`.
- $rest(a)$: funzione che ritorna come risultato la flista ottenuta escludendo il primo elemento:

$$rest([a_0, a_1, \ldots, a_n]) = [a_1, \ldots, a_n]$$

  Anche in questo caso, se $a = [\,]$ o $a =$ `NULL`, allora $rest(a) =$ `NULL`.
- $isempty(a)$: predicato che ritorna `TRUE` se a è la flista vuota $[\,]$, `FALSE` altrimenti.
- $isatom(a)$: predicato che ritorna `TRUE` se a è un atomo, `FALSE` altrimenti.

L'insieme di funzioni elencato sopra costituisce un insieme completo di operazioni per elaborare le fliste. Il ricorso al valore nullo `NULL` come risultato delle operazioni inconsistenti fornisce robustezza a questo insieme di operazioni.

Indicando con a una generica flista, le tre operazioni *cons*, $first$ e $rest$ sono fra loro legate dalla proprietà invariante

$$cons(first(a), rest(a)) = a$$

Il meccanismo fondamentale per generare una flista a partire dalla flista vuota consiste nella composizione dell'operatore *cons* secondo il seguente schema:

$$cons(a_0, cons(a_1, cons(a_2, \ldots, cons(a_n, empty()))))$$

Per semplicità di scrittura, tale flista viene scritta nella forma $[a_0, a_1, \ldots, a_n]$. L'operatore *cons* permette di realizzare l'operazione *env* di *imbustamento* (*envelope*) che converte un elemento e in una flista $[e]$ costituita dall'elemento e; può essere realizzato mediante l'operatore *cons*, come segue:

$$cons(e, empty()) \rightarrow [e]$$

Gli operatori $first$ e $rest$ costituiscono un meccanismo alternativo per accedere agli elementi di una flista, consentendone l'elaborazione.

Esempio 3.7.1 - Gli esempi che seguono illustrano le operazioni sulle fliste. Si noti l'uso delle parentesi per forzare l'associatività a destra.

$cons(8, [\,]) \rightarrow [8]$

$cons(1, cons(2, cons(3, [\,]))) \rightarrow [1, 2, 3]$

$cons(3, [7, 2, 5, 4]) \rightarrow [3, 7, 2, 5, 4]$

$first([0, 1, 2, 3, 4]) \rightarrow 0$

$rest([0, 1, 2, 3, 4]) \rightarrow [1, 2, 3, 4]$

$rest(rest(([0, 1, 2, 3, 4])) \rightarrow [2, 3, 4]$

$first(rest([0, 1, 2, 3, 4])) \rightarrow 1$

$isatom([\,]) \rightarrow$ `FALSE`

$isatom([3, 4]) \rightarrow$ `FALSE`

$isatom(7) \rightarrow$ `TRUE`

Elaborare fliste

L'elaborazione di una flista si basa principalmente sull'uso degli operatori $first$ e $rest$ che consentono di accedere agli elementi. Le fliste possono essere elaborate con due meccanismi formalmente diversi ma computazionalmente equivalenti. Una prima modalità consiste nell'elaborazione degli elementi in modalità iterativa, mediante un ciclo, come descritto nell'algoritmo 11.

Algoritmo 11 - Elaborazione di una flista a in modalità *iterativa*

Input: flista a

1: $b \leftarrow a$
2: **while** $\neg\, isempty(b)$ **do**
3: elabora l'elemento $first(b)$
4: $b \leftarrow rest(b)$
5: **end while**

Una modalità di elaborazione alternativa, e più coerente con l'impostazione funzionale, si fonda sulla definizione ricorsiva di flista ed è descritta nell'algoritmo 12.

Algoritmo 12 - Elaborazione di una flista a in modalità *ricorsiva*

Input: flista a

1: **if** $\neg\, isempty(a)$ **then**
2: elabora l'elemento $first(a)$
3: elabora la flista $rest(a)$
4: **end if**

Problema 3.7.1 Determinare il k-esimo elemento di una flista a.

Soluzione. Nella modalità iterativa e ricorsiva i due algoritmi 11 e 12 si scrivono come descritto negli algoritmi 13 e 14.

Algoritmo 13 - $item(a, k)$: k-esimo elemento della flista a

Input: lista a, numero naturale k
Output: k-esimo elemento della flista a

```
1: t ← a
2: for k times
3:     t ← rest(t)
4: end for
5: return first(t)
```

Algoritmo 14 - $item(a, k)$: k-esimo elemento della flista a

Input: flista a, numero naturale k
Output: k-esimo elemento della flista a

```
1: if k = 0 then
2:     return first(a)
3: else
4:     return item(rest(a), k − 1)
5: end if
```

Come si può dedurre dal confronto dei due algoritmi, la versione ricorsiva risulta più concisa e leggibile della corrispondente versione iterativa. □

Osservazione. L'operatore *item*, per questioni di efficienza, viene solitamente prediposto come operatore elementare e viene realizzato direttamente sulla struttura dati fisica che implementa la flista. Il ricorso all'operatore *item*, al di là di ogni considerazione di efficienza computazionale, risulta comunque sconsigliato all'interno di un contesto di programmazione funzionale.

Osservazione. Talvolta, per enfatizzare l'approccio funzionale, il controllo *if-then-else* viene sostituito dall'operatore condizionale ternario *if*, nella notazione funzionale $if(condizione, expTrue, expFalse)$. Con questa notazione l'algoritmo 14 verrebbe espresso con la singola istruzione

$$\textbf{return } if(k = 0, first(a), item(rest(a), k - 1))$$

3.8 Algoritmi sulle fliste

Le fliste, essendo delle strutture che si prestano ad essere definite in modo ricorsivo, suggeriscono, in modo naturale, dei procedimenti ricorsivi per il loro trattamento. A seguire sono riportati degli algoritmi elementari per l'elaborazione delle fliste, adottando l'impostazione ricorsiva.

Problema 3.8.1 Determinare la lunghezza di una flista.

Soluzione. Il problema è risolto dal seguente algoritmo:

Algoritmo 15 - $length(a)$: lunghezza della flista a

Input: flista a
Output: lunghezza della flista a
1: **if** $isempty(a)$ **then**
2: **return** 0
3: **else**
4: **return** $1 + length(rest(a))$
5: **end if**

Problema 3.8.2 Determinare la somma degli elementi di una flista di numeri.

Soluzione. Il problema è risolto dal seguente algoritmo:

Algoritmo 16 - $somma(a)$: somma degli elementi della flista a

Input: flista $a = [a_0, \ldots, a_n]$ di numeri
Output: somma $a_0 + \cdots + a_n$
1: **if** $isempty(a)$ **then**
2: **return** 0
3: **else**
4: **return** $first(a) + somma(rest(a))$
5: **end if**

Problema 3.8.3 Concatenare due fliste.

Soluzione. Il problema è risolto dal seguente algoritmo:

Algoritmo 17 - $conc(a, b)$: concatenazione delle due fliste a e b

Input: fliste a e b
Output: concatenazione fra a e b
1: **if** $isempty(a)$ **then**
2: **return** b
3: **else**
4: **return** $cons(first(a), conc(rest(a), b))$
5: **end if**

Problema 3.8.4 Determinare il massimo elemento in una flista.

Soluzione. Il problema è risolto dal seguente algoritmo:

Algoritmo 18 - $massimo(a)$: massimo della flista semplice a

Input: flista semplice a
Output: massimo della flista semplice a
1: **if** $isempty(a)$ **then**
2: **return** `NULL`
3: **else if** $length(a) = 1$ **then**
4: **return** $first(a)$
5: **else**
6: **return** $max(first(a), massimo(rest(a)))$
7: **end if**

Problema 3.8.5 Determinare la flista costituita dai primi n numeri naturali.

Soluzione. Il problema è risolto dal seguente algoritmo:

Algoritmo 19 - $numeri(n)$: flista $[1, 2, \ldots, n]$

Input: numero naturale n
Output: flista $[1, 2, \ldots, n]$
1: **if** $n = 0$ **then**
2: **return** $empty()$
3: **else**
4: **return** $conc(numeri(n-1), env(n))$
5: **end if**

Problema 3.8.6 Determinare il rovescio di una flista.

Soluzione. Il problema è risolto dal seguente algoritmo:

Algoritmo 20 - $rev(a)$: rovescio della flista a

Input: flista $a = [a_0, \ldots, a_n]$
Output: flista $[a_n, \ldots, a_0]$
1: **if** $isempty(a)$ **then**
2: **return** $empty()$
3: **else**
4: **return** $conc(rev(rest(a)), env(first(a)))$
5: **end if**

Problema 3.8.7 Stabilire se in una flista è presente un dato elemento.

Soluzione. Il problema è risolto dal seguente algoritmo:

Algoritmo 21 - $ricerca(a, x)$: ricerca se nella flista a è presente x

Input: flista a, elemento x
Output: `TRUE` se e solo se x è presente nella flista a

```
1: if isempty(a) then
2:    return FALSE
3: else if first(a) = x then
4:    return TRUE
5: else
6:    return ricerca(rest(a), x)
7: end if
```

Problema 3.8.8 Fondere due fliste ordinate, ottenendo un'unica flista ordinata.

Soluzione. Il problema è risolto dal seguente algoritmo:

Algoritmo 22 - $fusione(a, b)$: fusione di due fliste ordinate a e b

Input: fliste a e b ordinate (crescentemente)
Output: flista ottenuta dalla fusione di a e b

```
1: if isempty(a) then
2:    return b
3: else if isempty(b) then
4:    return a
5: else if first(a) < first(b) then
6:    return conc(env(first(a)), fusione(rest(a), b))
7: else
8:    return conc(env(first(b)), fusione(a, rest(b)))
9: end if
```

3.9 Oggetti di prima classe, funzionali e λ-espressioni

In tanti contesti della programmazione è utile o necessario guardare alle funzioni come ad entità autonome, similmente a quanto fatto per le variabili: possono essere definite, denominate con un nome identificativo, assegnate, passate come argomento ad altre funzioni, ritornate come risultato di funzioni. Un'entità che può essere denotata con un identificatore, memorizzata in delle variabili, passata come argomento a delle funzioni o ritornata come risultato da una funzione viene detta *oggetto di prima classe*.

Funzioni che hanno come argomenti altre funzioni o ritornano come risultato una funzione vengono delle *funzioni di ordine superiore* (*higher-order*

function) o *funzionali*. L'argomento è di specifico dominio della Matematica [4] ma rientra anche in molti situazioni della programmazione.

Le *espressioni lambda* (*λ-espressioni*) permettono di definire delle funzioni senza assegnare loro un nome identificativo. Il formalismo e la teoria di questi argomenti risalgono agli studi del logico americano Alonzo Church che negli anni 30 del secolo scorso sviluppò la teoria del λ-calcolo. Il λ-calcolo è nativo in alcuni linguaggi "storici" quali il Lisp ed in altri più recenti quali Scheme, Haskell, C++, Java e Python. A seguire vengono riportati alcuni brevi frammenti, quel tanto che serve per presentare i meccanismi di elaborazione delle sequenze.

Esempio 3.9.1 - La funzione quadrato che associa ad ogni numero il suo quadrato viene espressa mediante la seguente λ-espressione:

$$\lambda\, x : x * x$$

mentre la funzione che ritorna la somma di due numeri si esprime con [5]

$$\lambda\, x, y : x + y$$

Esempio 3.9.2 - Una λ-espressione può essere applicata a degli argomenti specifici e calcolata producendo un valore. Ad esempio:

$$(\lambda\, x : x * x)(3) \rightarrow 3 * 3 = 9$$

3.10 Elaborare sequenze mediante funzionali

Per elaborare le sequenze sono previsti alcuni meccanismi che combinano l'accesso agli elementi della sequenza con l'elaborazione degli elementi stessi. Questi meccanismi si avvalgono di appositi funzionali della forma $func(f, a)$ aventi funzioni f passate come argomento. Questi argomenti vengono spesso attualizzati al momento della chiamata mediante delle λ-espressioni.

I funzionali più frequentemente utilizzati per l'elaborazione delle sequenze sono *map*, $filter$ e $reduce$; questi funzionali sono predisposti nativamente in alcuni linguaggi di programmazione (ad esempio, Lisp, Python [6] e Java) oppure possono essere creati dall'utente definendo apposite funzioni. Il risultato dell'applicazione di un funzionale può essere una sequenza oppure un singolo valore.

Nelle considerazioni che seguiranno indicheremo con A e B degli insiemi, con $[A]$ una generica sequenza di elementi dell'insieme A e con $A \rightarrow B$ l'insieme delle funzioni da A in B.

[4] La derivata di una funzione è un funzionale $D(f)$ che associa ad una funzione f la sua derivata f'; la derivata n-esima di una funzione è un funzionale $D(f, n)$ che associa alla funzione f ed al numero n la derivata n-esima $f^{(n)}$; l'integrazione indefinita è un funzionale $\int f$ che ad una funzione f associa la funzione primitiva F.

[5] Nella notazione del λ-calcolo le due precedenti funzioni di quadrato e di addizione vengono scritte rispettivamente nei seguenti formati: $\lambda x.x^2$ e $\lambda x.\lambda y.x + y$.

[6] Nel linguaggio Python *map* e *filter* ritornano un iteratore e *list(k)* converte l'iteratore k in una lista.

Il funzionale map

Il funzionale *map* è caratterizzato dal prototipo

$$\begin{array}{rcl} map : (A \to B) \times [A] & \to & [B] \\ (f, a) & \mapsto & b \end{array}$$

applica la funzione f a tutti gli elementi della sequenza a e produce la sequenza b dei risultati. Più precisamente, data una funzione $f : A \to B$ ed una sequenza $a = [a_0, a_1, ..., a_n]$,

$$map(f, a) \to [f(a_0), f(a_1), \ldots, f(a_n)]$$

In una chiamata della forma $map(f, a)$ la funzione f può essere un identificatore di funzione definito precedentemente oppure una funzione definita mediante una λ-espressione.

Il funzionale *map* è esprimibile mediante il meccanismo del list comprehension come segue:

$$map(f, a) \stackrel{\text{def}}{=} [f(x) \textbf{ for } x \textbf{ in } a]$$

Esempio 3.10.1 - Se $f(x) \stackrel{\text{def}}{=} x + 1$ e $a = [4, 7, 2, 1, 8]$, allora

$$map(f, a) \to [5, 8, 3, 2, 9]$$

Equivalentemente, utilizzando una λ-espressione:

$$map(\lambda\, x : x + 1, [4, 7, 2, 1, 8]) \to [5, 8, 3, 2, 9]$$

Problema 3.10.1 Determinare la sequenza dei quadrati dei primi 10 numeri naturali.

Soluzione. Il problema è risolto dalla seguente espressione:

$$map(\lambda\, x : x^2, range(1, 11))$$

Il funzionale filter

Il funzionale $filter$ costituisce un semplice meccanismo per selezionare gli elementi di una sequenza soddisfacenti ad una data proprietà. Lo schema di $filter$ è:

$$\begin{array}{rcl} filter : (A \to \mathbb{B}) \times [A] & \to & [A] \\ (p, a) & \mapsto & b \end{array}$$

ed opera come segue: se $p : A \to \mathbb{B}$ è un predicato unario e $a = [a_0, a_1, ..., a_n]$ è una sequenza, allora

$$filter(p, a)$$

produce la sequenza degli elementi a_i che soddisfano al predicato p.

Il funzionale *filter* è esprimibile mediante il meccanismo del list comprehension come segue:

$$filter(p, a) \stackrel{\text{def}}{=} [x \textbf{ for } x \textbf{ in } a \textbf{ if } p(x)]$$

Esempio 3.10.2 - Se $p(x) \stackrel{\text{def}}{=} (x \bmod 2) = 0$ e $a = [4, 7, 2, 1, 8]$, allora

$$filter(p, a) \to [4, 2, 8]$$

Equivalentemente, utilizzando una λ-espressione:

$$filter(\lambda\, x : (x \bmod 2) = 0, [4, 7, 2, 1, 8]) \to [4, 2, 8]$$

Il funzionale `reduce`

Lo schema del funzionale *reduce* è:

$$\begin{array}{rcl} reduce : (A^2 \to A) \times [A] & \to & A \\ (f, a) & \mapsto & y \end{array}$$

dove $f : A^2 \to A$ è una funzione a due argomenti ed a una sequenza. Se f è una funzione a due argomenti e $a = [a_0, a_1, ..., a_n]$ è una sequenza, allora

$$reduce(f, a) \to f(...f(f(f(a_0, a_1), a_2), a_3)..., a_n)$$

In altri termini, lo sviluppo del calcolo avviene come segue: al primo passo f viene applicata alla coppia (a_0, a_1); successivamente f viene applicata al valore ottenuto al primo passo e ad a_2; si prosegue in questo modo fino ad ottenere un unico valore come risultato.

Esempio 3.10.3 - Se $f(x, y) \stackrel{\text{def}}{=} x + y$ e $a = [4, 7, 2, 1, 8]$, allora

$$reduce(f, a) \to (((4 + 7) + 2) + 1) + 8 = 22$$

Equivalentemente:

$$reduce(\lambda\, x, y : x + y, [4, 7, 2, 1, 8]) \to (((4 + 7) + 2) + 1) + 8 = 22$$

Problema 3.10.2 Determinare il fattoriale di un numero naturale usando il funzionale *reduce*.

Soluzione. Il calcolo del fattoriale di un numero naturale positivo n può essere svolto mediante la seguente espressione:

$$reduce(\lambda\, x, y : x * y, range(1, n + 1))$$

Il pattern mfr

I funzionali *map, filter, reduce* esplicano la loro potenza se utilizzati assieme in modo cooperativo. Il sistema che ne risulta costituisce un pattern di programmazione denotato con il termine *mfr*. Questo design pattern semplifica significativamente molti algoritmi che operano su sequenze di elementi in quanto permette di operare sulle sequenze senza alcun esplicito controllo sul flusso, nascondendo i controlli sulle azioni (*for*, *if*, *return*). Il patterm *mfr* rende spesso il codice più corto e semplice e permette al programmatore di concentrarsi sul cuore del problema, tralasciando i dettagli relativi ai controlli.

Problema 3.10.3 Data la sequenza $a = [a_0, a_1, \ldots, a_n]$ dei coefficienti del polinomio

$$P(x) \equiv a_n\, x^n + a_{n-1}\, x^{n-1} + \cdots + a_0$$

valutare il polinomio per uno specifico valore dell'incognita x.

Soluzione. Un algoritmo per il calcolo del valore $P(x)$ è:

Algoritmo 23 - calcolo di un polinomio

Input: sequenza a dei coefficienti del polinomio, valore x della indeterminata
Output: valore del polinomio in x

1: $p \leftarrow map(\lambda\, k : x^k, range(len(a)))$
2: $q \leftarrow map(\lambda\, x, y : x * y, a, p))$
3: $y \leftarrow reduce(\lambda\, x, y : x + y, q)$
4: **return** y

Le assegnazioni dell'algoritmo 23, secondo la tipica impostazione della programmazione funzionale, possono essere annidate in un'unica espressione che fornisce il valore del polinomio di coefficienti a in corrispondenza del valore x:

$$reduce(\lambda\, x, y : x + y, map(\lambda\, x, y : x * y, a, map(\lambda\, k : x^k, range(len(a)))))$$

ESERCIZI

3.1 Descrivere il significato degli operatori + e $*$ che compaiono nella seguente espressione:

$$((1+2)*[3])+(4*[5*6+7])$$

Valutare l'espressione.

3.2 Valutare le seguenti espressioni:

1. $[1][0]$
2. $[0]+[1]$
3. $2*[1][0]$
4. $(2*[1])[0]$
5. $(3*[4])[2]$
6. $1*[2]+[3]$
7. $(5*[4])[(3*[2])[1]]$
8. $([1]+2*[3]+4*[5])[6]$
9. $[4,5,6,7]\,[3]$
10. $([1,2,3]*2)[1]$
11. $(8*[1])[2]+3$
12. $((3+4)*[1+2])[2]+3$

3.3 Sia x un atomo e m,n due numeri naturali. Discutere se le seguenti espressioni sono equivalenti:

1. $(m*n)*[x]$
2. $m*(n*[x])$
3. $n*(m*[x])$
4. $m*[n*[x]\,]$

3.4 Data la sequenza $a=[1,2,3]$, calcolare le seguenti espressioni:

1. $a+a$
2. $2*a$
3. $[a]+[a]$
4. $2*[a]$
5. $[a+a]$
6. $[2*a]$

3.5 Scrivere delle espressioni che generino le seguenti sequenze:

1. $[1,1,1,1,1,1,1,1,1,1]$
2. $[1,2,3,4,5,6,7,8,9,10]$

3. $[1,2,1,2,1,2,1,2,1,2]$
4. $[1,1,1,1,1,2,2,2,2,2]$
5. $[1,2,3,4,5,1,2,3,4,5]$

3.6 Scrivere delle espressioni che generino le seguenti sequenze di dimensione 16:

1. $[0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]$
2. $[0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1]$
3. $[0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1]$
4. $[0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1]$

3.7 Dato il numero naturale n, generare le seguenti sequenze:

1. $[1,2,3,\ldots,n]$
2. $[n,n-1,\ldots,3,2,1]$
3. $[1,1,2,1,2,3,1,2,3,4,\ldots,1,2,\ldots,n]$
4. $[1,2,2,3,3,3,\ldots,n,n,\ldots,n]$

3.8 Date le sequenze $a = [x]$, $b = [y]$, scrivere un'espressione equivalente alla sequenza $[x,x,\ldots,x,y,y,\ldots,y]$ costituita da k atomi x e da k atomi y.

3.9 Date le sequenze $a = [x]$, $b = [y]$, scrivere un'espressione equivalente alla sequenza $[x,y,x,y,\ldots,x,y]$ costituita da k atomi x e da k atomi y alternati.

3.10 Determinare la sequenza delle cifre (in base 10) di un dato numero naturale; ad esempio, la sequenza delle cifre di 3482 è $[3,4,8,2]$.

3.11 Data una sequenza di cifre binarie, determinare il numero corrispondente; ad esempio, alla sequenza $[1,1,0,1]$ corrisponde il numero 13.

3.12 Dato un numero naturale < 100, determinare la più piccola sequenza di monetine centesimi di taglio $1,2,5,10,20,50$. Ad esempio, dato il numero 96 si deve determinare la sequenza $[50,20,20,5,1]$.

3.13 Dato un numero naturale rappresentante una quantità di denaro espressa in centesimi di Euro, determinare il più piccolo insieme di monete (da 1, 2, 5, 10, 20, 50 centesimi di Euro) equivalente alla data cifra. Esprimere il risultato mediante una sequenza di coppie.

3.14 Data una sequenza di numeri, determinare con un'unica scansione

1. il massimo
2. il minimo ed il massimo
3. i due valori più grandi
4. la somma dei numeri
5. la media dei numeri
6. la media escludendo dal calcolo i numeri uguali a zero

7. la media escludendo dal calcolo i numeri minimo e massimo
8. il prodotto dei numeri
9. il massimo comune divisore
10. il massimo e la sua frequenza
11. il massimo e le sue posizioni
12. la sequenza dei numeri primi
13. la sequenza dei numeri pari e quella dei numeri dispari
14. la sequenza dei numeri di posizione pari e quella dei numeri di posizione dispari

3.15 È data una sequenza ordinata di valori booleani. Determinare, in modo efficiente, il numero di elementi uguali a `TRUE`.

3.16 Determinare il risultato dell'operazione *and* eseguita su tutti gli elementi di una sequenza di valori booleani. Analogamente per l'operatore *or*.

3.17 Data una sequenza di cifre da 0 a 9, determinare il numero naturale corrispondente. Ad esempio, data la sequenza $[3, 7, 6]$ si deve determinare il numero 376.

3.18 Determinare la sequenza numerica di 0 ed 1 corrispondente alla rappresentazione binaria di un numero naturale.

3.19 Determinare i divisori di un dato numero naturale; ad esempio, i divisori del numero 20 sono $1, 2, 4, 5, 10, 20$. Stabilire se un dato numero naturale è *perfetto*, ossia se è uguale alla somma dei suoi divisori propri; ad esempio 28 è perfetto in quanto $1 + 2 + 4 + 7 + 14 = 28$.

3.20 Determinare la media dei valori presenti in una sequenza numerica, escludendo dal calcolo il minimo ed il massimo.

3.21 Determinare il valore dell'elemento più vicino alla media degli elementi di una sequenza di numeri. Ad esempio, l'elemento più vicino alla media 3.2 della sequenza $[2, 7, 1, 4, 2]$ è 4.

3.22 Valutare la media k-filtrata, dei valori di una sequenza numerica di dimensione n ottenuta facendo la media aritmetica degli $n - k$ elementi della sequenza scartando i k elementi più lontani dalla media aritmetica della sequenza originale.

3.23 Una sequenza a contiene i distacchi parziali dei vari concorrenti in una gara, ossia $a[i]$ rappresenta il distacco del concorrente i-esimo dal concorrente $(i - 1)$-esimo ($a[0] = 0$ per il primo concorrente). Determinare la sequenza contenente i distacchi complessivi dal primo concorrente.

3.24 *Rovesciare* una sequenza in modo che il primo elemento venga scambiato con l'ultimo, il secondo con il penultimo e così via.

3.25 *Ruotare* una sequenza $[a_0, a_1, \ldots, a_n]$ di una posizione a sinistra in modo da ottenere la configurazione $[a_1, \ldots, a_n, a_0]$. Analogamente per ottenere delle rotazioni a sinistra ed a destra di k posizioni.

3.26 Utilizzando il meccanismo *list comprehension*

1. generare la sequenza formata dai numeri naturali quadrati e dispari compresi fra 1 e 200
2. generare la sequenza formata dai numeri naturali quadrati presenti in una sequenza di numeri interi.
3. generare la sequenza formata dai numeri naturali dispari compresi fra 1 e 100 che siano divisibili per 3 o per 5.

3.27 Data una sequenza di numeri, usando il meccanismo list comprehension:

1. determinare il numero di numeri dispari presenti nella sequenza
2. determinare la sottosequenza dei numeri pari
3. determinare la sottosequenza dei numeri di posizione pari

Nota. Per ciascuno dei seguenti problemi svolgere i seguenti punti: descrivere un algoritmo risolutivo; stabilire in cosa consistono i casi *ottimo*, *medio*, *pessimo*; valutare la complessità e la complessità asintotica dell'algoritmo nel caso pessimo; stabilire la complessità del problema.

3.28 Decidere se una data sequenza di elementi comparabili è ordinata.

3.29 Determinare l'ampiezza del più piccolo intervallo contenente tutti i numeri di una sequenza.

3.30 Determinare il più piccolo numero naturale non appartenente ad una data sequenza di numeri naturali.

3.31 Decidere se una sequenza di numeri interi è *compatta*, ossia se contiene tutti gli elementi compresi fra il minimo ed il massimo.

3.32 Decidere se una sequenza di elementi è completamente contenuta in un'altra, tenendo conto della frequenza con la quale sono presenti gli elementi nelle due sequenze. Ad esempio, la sequenza $[3, 8, 3, 5]$ è completamente contenuta nella sequenza $[5, 3, 5, 1, 3, 8]$ ma non nella sequenza $[3, 8, 1, 8, 5, 7]$.

3.33 Decidere se due date sequenze di uguale lunghezza sono una permutazione una dell'altra.

3.34 Decidere se due sequenze contengono gli stessi elementi, indipendentemente dalla loro frequenza.

3.35 Determinare il numero di elementi distinti presenti in una sequenza.

3.36 Decidere se una sequenza è costituita da numeri tutti distinti.

3.37 Si consideri il problema P: *Decidere se una sequenza è composta da elementi tutti uguali fra loro.* Spiegare perchè il problema P': *Decidere se una sequenza è composta da elementi tutti diversi fra loro.* è *sostanzialmente* diverso dal problema P.

3.38 Determinare gli elementi che compaiono con la massima frequenza in una sequenza.

3.39 Decidere se una sequenza di numeri è formata da numeri tutti positivi. Risolvere il problema nei seguenti casi:

1. la sequenza non è ordinata
2. la sequenza è ordinata in modo crescente
3. la sequenza è ordinata in modo decrescente

3.40 Decidere se una sequenza è formata da elementi tutti distinti fra loro. Risolvere il problema nei seguenti casi:

1. la sequenza non è ordinata
2. la sequenza è ordinata

3.41 Determinare la massima distanza fra tutte le possibili coppie di numeri di una sequenza.

3.42 Determinare la coppia di elementi di una sequenza di numeri aventi la massima distanza reciproca.

3.43 Valutare le seguenti espressioni:

1. $(rest([3,5,7]))[2]$
2. $[4,5](rest[1,2])$
3. $(rest[1,2,3])[size[4,5]\,]$
4. $first([1,2,3])$
5. $rest([1,2,3])$
6. $first(rest([1,2,3]))$
7. $first([0]) = [0]$
8. $cons(first([1]), rest(first([\,[1,2,3]\,])))$
9. $first([1 > 2]) < first(rest([3 > 4, 5 < 6]))$

3.44 Decidere se una flista è palindroma.

3.45 Decidere se una lista semplice è omogenea, ossia se le foglie sono di uno stesso tipo atomico.

3.46 Determinare il numero di elementi non nulli presenti in una flista piatta di numeri.

3.47 Determinare l'ultimo elemento di una flista.

3.48 Scrivere in modo iterativo ed in modo ricorsivo un algoritmo per stabilire se una data flista è ordinata (non decrescentemente).

3.49 Scrivere in modo iterativo ed in modo ricorsivo un algoritmo per stabilire quante volte un dato elemento è presente in una flista.

3.50 Risolvere la seguente equazione (nell'incognita x):

$$[len(x)] = x$$

3.51 Decidere se una flista è formata da un solo elemento.

3.52 Determinare l'ultimo elemento di una flista.

3.53 Inserire in una flista un dato elemento in una data posizione. Ad esempio, inserendo nella flista $[5, 2, 6, 1, 8]$ l'elemento 7 alla posizione 4 si ottiene la lista $[5, 2, 6, 1, 4, 8]$.

3.54 Data una flista $a = [a_0, \ldots, a_n]$, determinare la flista ottenuta da a mediante una rotazione a destra di una posizione in modo ciclico, ottenendo la flista $a' = [a_n, a_0, \ldots, a_{n-1}]$.

3.55 Data una flista $a = [a_0, \ldots, a_n]$, determinare la flista ottenuta da a mediante una rotazione a sinistra di una posizione in modo ciclico, ottenendo la flista $a' = [a_2, \ldots, a_n, a_0]$.

3.56 Decidere se una flista è composta da elementi tutti uguali.

3.57 Decidere se due fliste sono uguali (basandosi sull'operatore di uguaglianza fra atomi). Le due fliste devono avere la stessa lunghezza ed avere gli elementi uguali nelle stesse posizioni.

3.58 Determinare la flista degli elementi di una flista a maggiori di un dato elemento x.

3.59 Data una flista di numeri naturali, determinare la sottolista dei numeri pari.

3.60 Data una flista, determinare la sottolista degli elementi di posizione pari.

3.61 Date due fliste, determinare la flista degli elementi comuni delle due fliste.

3.62 Una flista di numeri naturali contiene le cifre che rappresentano un numero in notazione decimale. Determinare il numero corrispondente. Ad esempio alla flista $[3, 7, 4]$ corrisponde il numero 374.

3.63 Una flista di numeri naturali contiene le cifre che rappresentano un numero in notazione decimale. Determinare la flista corrispondente al successivo. Ad esempio, data la lista $[3, 5, 9]$ si dovrà determinare la flista $[3, 6, 0]$.

3.64 Decidere se una flista è composta da elementi tutti uguali.

3.65 Decidere se una flista è ordinata.

3.66 Determinare la flista che contiene la prima occorrenza di ciascun elemento ripetuto in una data flista; ad esempio, data la flista $[4, 1, 3, 4, 1, 1, 2, 3, 7, 3]$, si deve ottenere la flista $[4, 1, 3, 2, 7]$.

3.67 Determinare la flista che contiene la prima occorrenza di ciascun elemento ripetuto in una data flista ordinata; ad esempio, data la flista $[3, 4, 4, 4, 6, 8, 9, 9]$, si deve ottenere la flista $[3, 4, 6, 8, 9]$.

3.68 Decidere se una flista è contenuta all'interno di un'altra. Ad esempio la flista $[3, 1, 4]$ è contenuta nella flista $[3, 2, 3, 1, 4]$ ma non è contenuta nella flista $[5, 3, 2, 1, 4]$.

3.69 Data una sequenza di numeri, usando le seguenti modalità:

1. con un ciclo, accedendo agli elementi mediante un indice
2. usando i funzionali *map*, *filter*, *reduce*
3. elaborando la sequenza con gli operatori delle *fliste*

determinare:

1. la somma dei numeri presenti
2. la media dei numeri di una sequenza compresi in un dato intervallo di valori
3. il minimo dei numeri presenti in una sequenza di numeri positivi.
4. la media dei numeri di una sequenza compresi in un dato intervallo di valori

3.70 Data una sequenza numerica a di dimensione n, valutare mediante il pattern *mfr* lo *scarto quadratico medio* di a, definito dalla seguente espressione:

$$\sqrt{\left(\sum_{k=1}^{n}(a_k - m)^2\right)/n}$$

essendo m la media dei valori della sequenza.

3.71 Usando il pattern *mfr* calcolare la *distanza euclidea* fra due punti X ed Y dello spazio n-dimensionale $\mathbb{R}^n$ espressa da

$$d(X,Y) \stackrel{\text{def}}{=} \sqrt{\sum_{k=1}^{n}(x_k - y_k)^2}$$

3.72 Sia a una sequenza di n di numeri reali positivi e w una sequenza di n numeri reali non negativi (detti *pesi*) tali che $\sum_{k=1}^{n} w_k = 1$. Usando, c il pattern *mfr*Calcolare i valori delle seguenti medie pesate:

$$A(a,w) \stackrel{\text{def}}{=} \sum_{k=1}^{n} a_k\, w_k \qquad \text{(media aritmetica)}$$

$$H(a,w) \stackrel{\text{def}}{=} \frac{1}{\sum_{k=1}^{n} \frac{w_k}{a_k}} \qquad \text{(media armonica)}$$

$$G(a,w) \stackrel{\text{def}}{=} \prod_{k=1}^{n} (a_k)^{w_k} \qquad \text{(media geometrica)}$$

$$Q(a,w) \stackrel{\text{def}}{=} \sqrt{\sum_{k=1}^{n} a_k\, w_k} \qquad \text{(media quadratica)}$$

Dimostrare che valgono le disuguaglianze $H \le G \le A \le Q$.

4

STRUTTURARE

I programmi per computer usualmente operano su tabelle d'informazioni. Nella maggioranza dei casi queste tabelle non sono semplicemente masse amorfe di valori numerici; esse coinvolgono importanti relazioni strutturali fra i dati elementi.

D. Knuth,
The Art of Computer Programming

La costruzione di qualsiasi cosa si fonda sulla possibilità di usare elementi di base già esistenti. Non importa quanto articolati e sofisticati essi siano; può trattarsi degli atomi della chimica, dei mattoni usati dal muratore o delle travi in cemento armato usate nella costruzione di un capannone industriale: all'utente questi elementi appaiono come indivisibili e caratterizzati da specifiche proprietà, funzionalità e modalità d'impiego. Riportando in ambito informatico queste considerazioni generali, tutti i linguaggi di programmazione offrono la possibilità di utilizzare delle entità (valori, tipi, controlli, blocchi, ...) già costruite e pronte all'uso e dei meccanismi, forniti mediante dei costrutti sintattici del linguaggio, che l'utente può utilizzare per costruire nuove entità, a seconda delle esigenze che gli si presentano.

Il meccanismo di aggregazione di elementi permette di organizzare dati ed entità generiche in aggregati denominati *strutture*. Le strutture vengono realizzate combinando pochi meccanismi di base; come precisato alla fine del capitolo *Aggregare*, ci si può limitare alle *sequenze*; unitamente a questa forma di aggregazione si ammette la possibilità di creare *sequenze di sequenze*; si ottiene così un sistema completo e produttivo, utilizzabile all'interno della maggior parte dei moderni linguaggi di programmazione, che permette di realizzare qualsiasi forma di strutturazione dei dati.

4.1 Elementi e strutture

Nei vari linguaggi di programmazione le entità vengono spesso classificate nelle seguenti due categorie:

- *elementi*: sono i valori e le istruzioni elementari predisposti dal linguaggio; vengono detti anche *atomi*
- *strutture*: sono le entità composte da oggetti elementari e da altri oggetti composti, mediante appositi meccanismi del linguaggio

La potenza dei linguaggi di programmazione viene esplicata proprio mediante la possibilità di comporre fra loro delle entità di base.

Gli elementi costituenti una struttura possono essere di diversa natura. Al livello di astrazione predisposto dai tradizionali linguaggi di programmazione, gli elementi costituenti una struttura sono numeri, stringhe, istruzioni elementari del linguaggio ed altro ancora; ad un livello più basso, gli atomi sono i byte e le istruzioni in linguaggio macchina; ad un livello più alto, in un sistema complesso, gli atomi sono le componenti stesse del sistema.

4.2 Le strutture

L'operatore di aggregazione $[\dots]$ costituisce il meccanismo fondamentale per costruire le strutture composte da entità elementari già predisposte o entità precedentemente costruite. Con il termine *struttura* si denota una sequenza $[a_0, a_1, \dots, a_n]$ di elementi; ogni elemento a_i può essere un atomo o una struttura. L'estrema generalità di una struttura deriva dalla sua definizione ricorsiva che ammette che le strutture possano essere formate da elementi che sono a loro volta delle strutture.

Le strutture vengono classificate in base agli elementi componenti, come segue:

- *in base al tipo degli elementi componenti*: se tutti gli elementi sono dello stesso tipo le strutture vengono dette *omogenee*, altrimenti vengono dette *eterogenee*
- *in base alla struttura degli elementi componenti*: se tutti gli elementi sono atomici (numeri, valori logici, caratteri, stringhe, ...) le strutture vengono dette *semplici o piatte*, altrimenti, se le strutture sono costituite da elementi che sono a loro volta delle strutture, vengono dette *composte*; una struttura semplice viene detta anche *lista*.

Esempio 4.2.1 - A seguire sono riportate alcune espressioni di strutture semplici eterogenee:

$$[7, \texttt{FALSE}]$$
$$['+', 8, 5, 3, 6, 2]$$
$$[\text{"}Mario\text{"}, \text{"}Rossi\text{"}, 2, \text{"}maggio\text{"}, 2012]$$

Esempio 4.2.2 - Alcuni esempi di strutture composte:

$$[[1],[2],[3]]$$
$$[[7,\texttt{TRUE}],[4,\texttt{FALSE}]]$$
$$[[1],[2,2],[3,3,3],[4,4,4,4]]$$
$$[\,[\,[\,]\,],[\,]\,]$$
$$[3,[4,5],2,[\,[6],[\,]\,]\,]$$
$$[\,[\text{"}Mario\text{"},\text{"}Rossi\text{"}],[2,\text{"}maggio\text{"},2012]\,]$$
$$[\,[\text{"}rosso\text{"},[255,0,0]\,],[\text{"}verde\text{"},[0,255,0]\,]\,]$$

Nella loro massima generalità i dati ed i risultati di un problema sono costituiti da strutture.

Esempio 4.2.3 - Il problema di determinare le coordinate del punto di intersezione fra due rette del piano cartesiano è caratterizzato dai seguenti dati e risultati:

$$\begin{aligned} Dati &: [\,[a_1,b_1,c_1],[a_2,b_2,c_2]\,] \\ Risultati &: [x_0,y_0] \end{aligned}$$

essendo $a_1x+b_1y+c_1=0$, $a_2x+b_2y+c_2=0$ le equazioni delle due rette e (x_0,y_0) le coordinate del punto di intersezione.

Esempio 4.2.4 - Il problema di determinare le coordinate dei punti di intersezione fra una circonferenza del piano cartesiano ed una retta è caratterizzato dai seguenti dati e risultati:

$$\begin{aligned} Dati &: [\,[\,[x_0,y_0],r],[\,[x_A,y_A],[x_B,y_B]\,]\,] \\ Risultati &: [\,[x_1,y_1],[x_2,y_2]\,] \end{aligned}$$

essendo (x_0,y_0) le coordinate del centro ed r il raggio della circonferenza, (x_A,y_A), (x_B,y_B) le coordinate di due punti della retta, (x_1,y_1) e (x_2,y_2) le coordinate dei due punti di intersezione.

4.3 Rappresentazioni delle strutture

In talune considerazioni, specialmente nel caso di sequenze composte, risulta utile descrivere una sequenza mediante uno schema ad *albero*: una generica sequenza $[a_0,\ldots,a_n]$ viene descritta come illustrato nella figura 4.1.

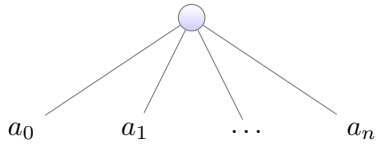


Figura 4.1: Rappresentazione grafica ad albero della sequenza $[a_0,a_1,\ldots,a_n]$.

Nel caso in cui gli elementi della struttura siano non atomici, la rappresentazione indicata nella figura 4.1 deve essere replicata ricorsivamente a tutti gli elementi non atomici, come descritto nell'esempio 4.3.1.

Esempio 4.3.1 - La struttura $[\,[4,[\,],3],6,[8,[5,2],1,[7]\,]\,]$ è descritta dall'albero riportato in figura 4.2. Il simbolo ○ denota l'operatore di sequenzializzazione se si trova in un nodo interno, mentre denota la sequenza vuota se si trova su un nodo foglia.

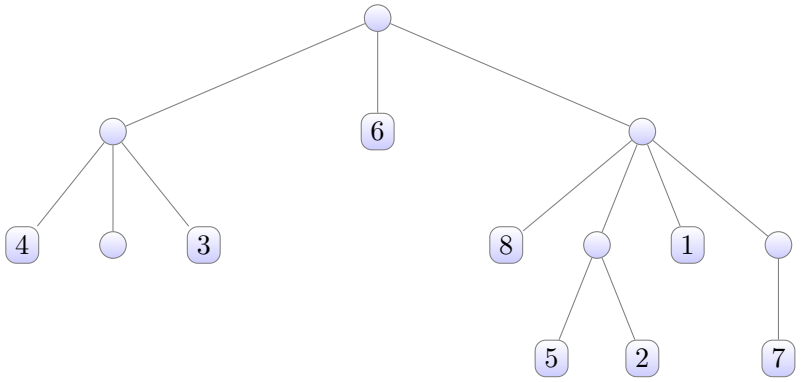


Figura 4.2: Rappresentazione grafica ad albero della struttura $[\,[4,[\,],3],6,[8,[5,2],1,[7]\,]\,]$.

Nel contesto dei linguaggi di programmazione le strutture possono essere denotate mediante degli identificatori e questi identificatori possono essere usati come elementi costituenti altre strutture. Le strutture sono degli oggetti e con un'assegnazione della forma $a \leftarrow [\ldots]$ l'identificatore a diventa un riferimento alla struttura. Un'operazione fra strutture (ad esempio una concatenazione) comporta invece la creazione di una nuova struttura costituita da copie delle due strutture. Questa osservazione diventa fondamentale per interpretare situazioni come quelle descritta nell'esempio che segue.

Esempio 4.3.2 - Con la seguente sequenza di assegnazioni:

$$\begin{aligned} a &\leftarrow [1,2,3] \\ b &\leftarrow a + a \\ c &\leftarrow [b,a] \\ a[2] &\leftarrow a[2] + b[3] \end{aligned}$$

si ottiene la seguente situazione:

$$\begin{aligned} a &= [1,2,4] \\ b &= [1,2,3,1,2,3] \\ c &= [[1,2,3,1,2,3],[1,2,4]] \end{aligned}$$

Nell'uso dei linguaggi di programmazione, quando si fanno operazioni di creazione ed assegnazioni fra strutture, per seguire l'evoluzione dello stato delle strutture, risulta talvolta comodo considerare la situazione in memoria delle varie strutture, unitamente ai riferimenti che le referenziano. Ad esempio, la situazione finale che si crea con le assegnazioni dell'esempio 4.3.2 è descritta nella figura 4.3.

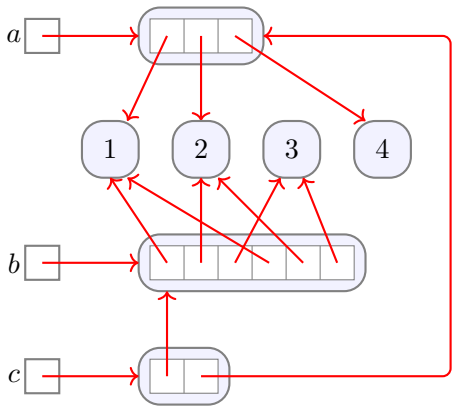


Figura 4.3: Rappresentazione grafica della memoria della situazione finale creata nell'esempio 4.3.2, nell'ipotesi che anche i numeri siano *oggetti*.

4.4 Operazioni sulle strutture

Ogni meccanismo di composizione comporta la definizione di meccanismi di accesso e di manipolazione degli elementi componenti. Per questo motivo sulle strutture sono predisposti degli appositi operatori che sono classificati come segue:

- *costruttori*: sono delle operazioni che generano strutture
- *selettori*: sono delle operazioni che permettono di accedere ad uno specifico elemento della struttura o ad una sua parte
- *manipolatori*: sono operazioni che hanno l'effetto di modificare la struttura, mediante operazioni di inserimento, eliminazione e modifica di elementi della struttura
- *funzioni*: sono operazioni che operano su una o più strutture e danno come risultato un'altra struttura

L'elaborazione delle strutture si fonda sull'esecuzione di operazioni che ricadono nelle seguenti due modalità:

- modalità *funzionale*: l'operazione genera una nuova struttura che costituisce il risultato dell'operazione; una tipica notazione funzionale è:

$$op(a, x_1, \ldots, x_n)$$

dove *op* è l'identificatore dell'operazione, a è l'entità sulla quale si esercita l'operazione e $x_1, \ldots, x_n$ sono gli argomenti ausiliari all'operazione; possono essere utilizzate anche altri operatori in notazione infissa

- modalità *procedurale*: l'operazione modifica la struttura alla quale viene applicata; tale modalità è applicabile solo se l'argomento dell'operazione è un riferimento che identifica una struttura; spesso queste operazioni vengono scritte in notazione puntata, tipica della *programmazione orientata agli oggetti*:

  $$a.op(x_1, \ldots, x_n)$$

  dove a è l'entità sulla quale viene esercitata l'operazione *op* e $x_1, \ldots, x_n$ sono argomenti che specificano l'operazione.

Solitamente i due approcci sono entrambi disponibili nella maggior parte dei linguaggi di programmazione e possono essere utilizzati interscambievolmente. Essi sono, tuttavia, profondamente diversi e rientrano in due diversi paradigmi di programmazione; il primo (paradigma *funzionale*) suggerisce spesso delle soluzioni ricorsive e non richiede l'uso di variabili, riferimenti ed assegnazioni; il secondo approccio (paradigma *procedurale*) si fonda sul fatto che un'operazione modifica direttamente la struttura alla quale viene applicata; in questi casi solitamente si fa ricorso ad algoritmi basati su cicli e si fa uso di variabili, riferimenti e assegnazioni che permettono di modificare selettivamente gli elementi della struttura.

L'operatore di assegnazione $\leftarrow$ svolge un ruolo di intermediazione fra le due modalità funzionale e procedurale, permettendo di modificare un'entità usando operazioni in notazione funzionale:

$$a \leftarrow op(a, x_1, \ldots, x_n)$$

Tenendo conto della possibilità o meno di modifica della struttura, si possono distinguere le seguenti due forme di elaborazione:

- elaborazione *read-only*: la struttura viene solamente esaminata ma non modificata
- elaborazione *read-write*: la struttura viene esaminata e può subire delle modifiche:
  - inserendo un nuovo elemento
  - modificando un elemento presente
  - eliminando un elemento presente

4.5 Costruzione di strutture

Le operazioni di base per costruire ed elaborare una struttura sono le seguenti, derivate direttamente dalle analoghe operazioni già viste per le sequenze:

- $[\ldots]$: sequenzializzazione
- $\alpha + \beta$: concatenazione
- $k * \alpha$: moltiplicazione

Combinando queste operazioni si possono costruire strutture come descritto nell'esempio he segue.

Esempio 4.5.1 - Alcuni esempi di strutture composte:

$$[1, 2, [3, 4]\,]$$
$$2 * [0] + 3 * [1]$$

Nel caso in cui gli elementi non siano noti, la costruzione della struttura avviene a partire da una struttura vuota [ ] e poi accodando sequenzialmente gli elementi.

Esempio 4.5.2 - Condideriamo il problema di scomposizione di un numero naturale n, determinandone i fattori primi con le rispettive molteplicità (vedi tabella 4.4). Una tabella di questo formato può essere rappresentata mediante la sequenza di coppie $[\,[2, 1], [3, 2], [7, 1]\,]$.

fattore	molteplicità
2	1
3	2
7	1

Figura 4.4: Tabella della scomposizione del numero 126.

A seguire è riportato l'algoritmo che genera la tabella riportata nella figura 4.4, in due versioni a diverso livello di dettaglio (algoritmi 1 e 2).

Algoritmo 1 - Sequenza dei divisori primi di un numero naturale

Input: numero naturale n
Output: sequenza dei divisori primi di n

```
1: inizializza la sequenza vuota dei divisori
2: considera d = 2 come primo potenziale divisore
3: while n > 1 do
4:    determina la molteplicità m del divisore d dividendo man mano n per d
5:    if m > 0 then
6:       accoda la coppia [d, m] alla sequenza dei divisori
7:    end if
8:    considera il numero primo successivo di d
9: end while
10: return sequenza dei divisori primi
```

Algoritmo 2 - Sequenza dei divisori primi di un numero naturale

Input: numero naturale n
Output: sequenza dei divisori primi di n

```
1:  a ← [ ]
2:  d ← 2
3:  while n > 1 do
4:     m ← 0
5:     while (n mod d) = 0 do
6:        m ← m + 1
7:        n ← n div d
8:     end while
9:     if m > 0 then
10:       a ← a + [ [d, m] ]
11:    end if
12:    d ← d + 1
13: end while
14: return a
```

4.6 Elaborazione di strutture

Una volta costruita una struttura, per svolgere qualsiasi forma di elaborazione su di essa, risulta necessario accedere ai suoi elementi. Coerentemente con il fatto che le strutture sono delle sequenze, l'espressione

$$a[i]$$

fornisce l'i-esimo elemento della sequenza a. Nel caso $a[i]$ sia un valore non atomico, ossia una struttura a sua volta, per accedere agli elementi atomici costituenti la struttura, è necessario riutilizzare l'aperatore di accesso, ad esempio con notazioni delle forme $a[i][j]$, $a[i][j][k]$, a seconda del livello di annidamento degli elementi atomici.

Per capire quando si è arrivati al livello delle foglie della struttura si può utilizzare l'operatore di *controllo di atomicità*:

$$isatom(x)$$

che fornisce `TRUE` se x è un atomo, `FALSE` altrimenti. Il predicato *isatom* costituisce l'operatore per chiudere la ricorsione nell'elaborazione ricorsiva dei vari livelli di annidamento di una struttura composta.

Nel caso di elaborazione sequenziale di tutti gli elementi della struttura si può adattare il controllo `for` x `in` a già visto per le sequenze ed adattarlo al caso delle strutture composte; lo schema dell'elaborazione è riportato nell'algoritmo 3, impostato in modo iterativo per esaminare gli elementi del primo livello della struttura ed in modo ricorsivo per gestire la possibilità che gli elementi esaminati siano a loro volta delle strutture.

Algoritmo 3 - $elabora(a)$: elaborazione in profondità di una struttura

Input: struttura a
1: **for** x **in** a **do**
2: **if** $isatom(x)$ **then**
3: elabora l'atomo x
4: **else**
5: $elabora(x)$
6: **end if**
7: **end for**

Un modo alternativo per elaborare le strutture consiste nell'usare gli operatori già visti per le fliste: *empty*, *cons*, *first*, *rest*, *isempty* e *isatom*. Queste funzioni orientano verso una modalità di elaborazione di tipo funzionale e ricorsivo basata sullo schema generale di algoritmo 4 corrispondente ed equivalente all'algoritmo 3.

Algoritmo 4 - $elabora(a)$: elaborazione in profondità di una struttura

Input: struttura a
1: **if** $\neg isempty(a)$ **then**
2: **if** $isatom(first(a))$ **then**
3: elabora l'atomo $first(a)$
4: **else**
5: $elabora(first(a))$
6: **end if**
7: $elabora(rest(a))$
8: **end if**

Problema 4.6.1 Calcolare la somma degli atomi di una struttura numerica composta, posti a qualsiasi livello di annidamento.

Soluzione. Il problema può essere risolto mediante l'algoritmo 5.

Algoritmo 5 - $somma(a)$: somma elementi di una struttura

Input: struttura a
Output: somma degli elementi della struttura a
1: $s \leftarrow 0$
2: **for** x **in** a **do**
3: **if** $isatom(x)$ **then**
4: $s \leftarrow s + x$
5: **else**
6: $s \leftarrow s + somma(x)$
7: **end if**
8: **end for**
9: **return** s

Problema 4.6.2 Determinare il massimo degli elementi di una struttura.

Soluzione. È sufficiente adattare l'algoritmo 4.

Algoritmo 6 - $massimo(a)$: massimo della struttura a

Input: lista semplice a
Output: massimo della lista composta a

```
1:  m ← NULL
2:  if ¬isempty(a) then
3:     x ← first(a)
4:     if isatom(x) then
5:        m ← max(m, x)
6:     else
7:        m ← max(m, massimo(x))
8:     end if
9:     m ← max(m, massimo(rest(a)))
10: end if
11: return m
```

□

Problema 4.6.3 Date due n-liste $a = [a_0, ..., a_n]$ e $b = [b_0, ..., b_n]$, determiniamo la lista composta dalle coppie di elementi aventi la stessa posizione nelle due liste:

$$[\,[a_0, b_0], [a_1, b_1], ..., [a_n, b_n]\,]$$

Soluzione. La soluzione di questo problema è riportata nell'algoritmo 7.

Algoritmo 7 - $coppie(a, b)$: lista delle coppie delle due liste a e b

Input: liste a e b
Output: lista delle coppie formate combinando gli elementi di uguale posizione delle due liste a e b

```
1: if isempty(a) ∨ isempty(b) then
2:    return [ ]
3: else
4:    return [ [first(a), first(b)] ] + coppie(rest(a), rest(b))
5: end if
```

□

Problema 4.6.4 Il prodotto cartesiano fra due sequenze è costituito dalla sequenza di tutte le possibili coppie che si possono formare prendendo il primo elemento dalla prima sequenza ed il secondo elemento dalla seconda; ad esempio il prodotto cartesiano fra le due sequenze $[a, b, c]$ e $[x, y]$ produce la sequenza di coppie $[\,[a, x], [a, y], [b, x], [b, y], [c, x], [c, y]\,]$. Calcolare il prodotto cartesiano fra due fliste (operazione *zip*).

Soluzione. Realizziamo dapprima l'algoritmo che esegue il prodotto cartesiano fra un atomo ed una lista (algoritmo 8).

Algoritmo 8 - $prod(x, a)$: prodotto dell'atomo x per la flista a

Input: atomo x, flista $a = [a_0, \ldots, a_n]$
Output: lista delle coppie $[\,[x, a_0], [x, a_1], \ldots, [x, a_n]\,]$

```
1: if isempty(a) then
2:     return [ ]
3: else
4:     return [ [x, first(a)] ] + prod(x, rest(a))
5: end if
```

A questo punto il prodotto cartesiano fra due fliste si esprime mediante il seguente algoritmo 9.

Algoritmo 9 - $prodcart(a, b)$: prodotto cartesiano fra le due liste a e b

Input: liste a e b
Output: lista delle coppie del prodotto cartesiano $a \times b$

```
1: if isempty(a) then
2:     return [ ]
3: else
4:     return prod(first(a), b) + prodcart(rest(a), b)
5: end if
```

□

4.7 Elaborazione con appiattimento

Gli algoritmi per l'elaborazione di strutture composte risultano talvolta di non facile scrittura in quanto si devono gestire parecchie situazioni. Qualora la struttura organizzativa della struttura risulti ininfluente per l'algoritmo, risulta più facile convertire dapprima la struttura composta in una struttura piatta e successivamente applicare l'algoritmo su quest'ultima.

L'operazione di *appiattimento* è presentato nell'esempio che segue.

Esempio 4.7.1 - L'operazione *flat* su una struttura consiste nel determinare la struttura formata dagli atomi della struttura; ad esempio, appiattendo la struttura

$$[\,[1, [2]\,], 3, [\,[\,[4], 5]\,], [\,[\,[6]\,], 7], [8]\,]$$

si genera la struttura semplice

$$[1, 2, 3, 4, 5, 6, 7, 8]$$

Problema 4.7.1 Appiattire una struttura.

Soluzione. Il problema è risolto mediante il seguente algoritmo 10.

Algoritmo 10 - $flat(a)$: appiattimento della struttura a

Input: struttura a
Output: struttura semplice degli atomi di a
1: **if** $isempty(a)$ **then**
2: **return** $empty()$
3: **else if** $isatom(first(a))$ **then**
4: **return** $[first(a)] + flat(rest(a))$
5: **else**
6: **return** $flat(first(a)) + flat(rest(a))$
7: **end if**

□

Problema 4.7.2 Stabilire se in una data struttura gli atomi sono tutti distinti fra loro. Ad esempio, la struttura $[\,[4,[\,],3],6,[8,[5,2],1,[7]\,]\,]$ è formata da atomi tutti distinti fra loro.

Soluzione. Per questo problema risulta comodo svolgere una preelaborazione della struttura, generando da essa una struttura piatta che viene poi analizzata. L'analisi diretta della struttura piatta al fine di testare se è formata da atomi tutti distinti richiede un costo computazionale pari a $O(n^2)$, essendo n il numero di elementi della struttura piatta. Per migliorare l'efficienza di questo approccio conviene preelaborare la struttura piatta ordinandola con un efficiente algoritmo avente la complessità computazionale pari a $O(n \log n)$. A questo punto un'analisi sequenziale della sequenza ordinata richiede un ulteriore costo computazionale pari a $O(n)$. In totale il costo computazionale totale è pari a $O(n \log n) + O(n)$ che è pari a $O(n \log n)$. Le precedenti considerazoni sono riassunte nell'algoritmo 11, impostato in modalità funzionale (*ordinata* è una funzione che ritorna la lista ordinata formata dagli elementi della lista passata come argomento alla funzione).

Algoritmo 11 - $distinti(a)$: test se la struttura è formata da atomi distinti

Input: struttura a
Output: `TRUE` se la struttura a è formata da atomi distinti
1: **return** $distinti(ordinata(flat(a)))$

□

4.8 Le matrici

In molte situazioni i dati vengono organizzati in strutture bidimensionali a forma di tabella rettangolare composta da righe e colonne, dette *matrici*. Dal punto di vista strutturale si tratta di strutture composte da sequenze di sequenze della stessa lunghezza. Da questa definizione discende che una matrice è descrivibile mediante una tabella bidimensionale strutturata in *righe* e *colonne*.

Esempio 4.8.1 - La struttura

$$[\,[2,3,7,1],[5,8,4,2],[3,0,2,5]\,]$$

costituisce una *matrice bidimensionale*. Una tale matrice viene usualmente descritta in forma tabellare come segue:

2	3	7	1
5	8	4	2
3	0	2	5

Spesso una matrice viene creata inizialmente con ben specificate dimensioni che poi non vengono più modificate. Per creare una matrice a di dimensioni $m \times n$ utilizzeremo la notazione

$$a \leftarrow [m \times n]$$

oppure

$$a \leftarrow Matrice(m, n)$$

Se a denota una matrice bidimensionale, coerentemente con la notazione usata per l'operatore di selezione ad indice, con la notazione

$$a[i]$$

si denota la sequenza costituita dall'i-esima riga della matrice a, mentre con

$$a[i][j]$$

si denota il j-esimo elemento dell'i-esima riga della matrice a. Nella scrittura degli algoritmi, per semplicità di scrittura, l'elemento $a[i][j]$ viene spesso indicato con la notazione a_{ij}. Con la scrittura

$$a[i][j] \leftarrow esp$$

si ottiene l'effetto di assegnare all'elemento $a[i][j]$ il valore ottenuto dalla valutazione dell'espressione *esp*. Per indicare una matrice il cui generico elemento è a_{ij} si usa spesso la notazione $[a_{ij}]$. Data una matrice a, si può ricavarne il numero di righe ed il numero di colonne, rispettivamente, mediante le espressioni $len(a)$, $len(a[0])$.

4.9 Elaborazione di matrici

Lo schema generale di elaborazione di una matrice bidimensionale a di n righe ed m colonne è descritto nell'algoritmo 12.

Algoritmo 12 - Elaborazione di una matrice a di dimensioni $n \times m$

Input: matrice a di dimensione $n \times m$

```
1: for i in range(n) do
2:     for j in range(m) do
3:         elabora l'elemento a[i][j]
4:     end for
5: end for
```

Nel caso in cui gli elementi di una matrice vengano elaborati in sola lettura, senza essere modificati, si può adottare lo schema descritto nell'algoritmo 13.

Algoritmo 13 - Elaborazione di una matrice a di dimensioni $n \times m$

Input: matrice a di dimensione $n \times m$

```
1: for riga in a do
2:     for e in riga do
3:         elabora l'elemento e
4:     end for
5: end for
```

Accade spesso di elaborare una matrice accedendo a tutti gli elementi senza tener conto della loro strutturazione bidimensionale in righe e colonne. In queste situazioni si può elaborare una matrice $n \times m$ considerandola come un array unidimensionale di dimensione nm. Lo schema dell'elaborazione è descritto nell'algoritmo 14.

Algoritmo 14 - Elaborazione di una matrice a di dimensioni $n \times m$

Input: matrice a di dimensione $n \times m$

```
1: for k in range(n * m) do
2:     elabora l'elemento a[k ÷ m][k  mod  m]
3: end for
```

4.10 Operazioni sulle matrici

Consideriamo ora delle operazioni fra matrici numeriche, operazioni che si presentano in moltissime applicazioni e per questo frequentemente eseguite sugli elaboratori.

DEFINIZIONE 1. Siano $a = [a_{ij}]$, $b = [b_{ij}]$ due matrici numeriche $n \times m$. Si definisce come *somma* $a + b$ fra le due matrici a e b la matrice $c = [c_{ij}]$ di dimensioni $n \times m$ il cui generico elemento c_{ij} è definito da

$$c_{ij} = a_{ij} + b_{ij}$$

L'operazione di somma fra matrici richiede che le due matrici abbiano le stesse dimensioni. Quando due matrici soddisfano a questo vincolo sulle dimensioni si dice che le due matrici sono *somma-compatibili*.

Adattando a questa definizione lo schema generale dell'algoritmo 12, il calcolo della somma fra due matrici numeriche può essere descritto mediante l'algoritmo 15.

Algoritmo 15 - Somma fra matrici

Input: matrici a e b di dimensione $n \times m$
Output: matrice $c = a + b$

```
1: for i in range(n) do
2:     for j in range(m) do
3:         c_ij ← a_ij + b_ij
4:     end for
5: end for
6: return c = [c_ij]
```

Esempio 4.10.1 - Applicando l'algoritmo 15 si ricava la seguente somma:

3	1	0	4
0	2	5	2
2	3	3	1

+

5	2	5	3
1	2	0	3
4	0	1	4

=

8	3	5	7
1	4	5	5
6	3	4	5

L'operazione di moltiplicazione fra matrici si basa sulla seguente

DEFINIZIONE 2. Sia $a = [a_{ij}]$ una matrice numerica $n \times m$ e $b = [b_{ij}]$ una matrice numerica $m \times p$. Si definisce come *prodotto* $a \times b$ fra le due matrici a e b la matrice $c = [c_{ij}]$ di dimensioni $n \times p$ il cui generico elemento c_{ij} è definito da

$$c_{ij} = \sum_{k=0}^{m-1} a_{ik}\, b_{kj}$$

L'operazione di prodotto richiede che il numero di colonne della prima matrice sia uguale al numero di righe della seconda. Due matrici che soddisfino a questo vincolo sulle dimensioni vengono dette *prodotto-compatibili*. Spesso l'operazione di moltiplicazione fra due matrici a e b viene indicata semplicemente giustapponendo le due matrici, con la notazione ab. Il calcolo del prodotto fra due matrici numeriche, basato direttamente sulla definizione data sopra, è descritto dall'algoritmo 16.

Algoritmo 16 - Prodotto fra matrici

Input: matrici a di dimensione $n \times m$ e b di dimensione $m \times p$
Output: matrice $c = a * b$ (di dimensione $n \times p$)

```
1: for i in range(n) do
2:     for j in range(m) do
3:         c_ij ← 0
4:         for k in range(m) do
5:             c_ij ← c_ij + a_ik b_kj
6:         end for
7:     end for
8: end for
9: return c = [c_ij]
```

Esempio 4.10.2 - Applicando l'algoritmo 16 si ricava il seguente prodotto:

3	1		2	0	1	5		7	3	5	18
0	2	*	1	3	2	3	=	2	6	4	6
2	4							8	12	10	22

4.11 Soluzione di un sistema lineare

Sia A una matrice quadrata di dimensioni $n \times n$ e b un vettore n-dimensionale. A seguire è riportato l'algoritmo per la soluzione di un sistema lineare $A\,x = b$ mediante il *metodo di riduzione di Gauss con pivoting parziale*. Il procedimento si basa sulla trasformazione del sistema $Ax = b$ in un sistema equivalente (avente la stessa soluzione) riducendo la matrice A ad una matrice avente tutti 0 nel triangolo inferiore sinistro. La seconda fase del procedimento consiste nel calcolo a ritroso delle incognite: $x_{n-1} \to x_{n-2} \to \cdots \to x_1 \to x_0$ (algoritmo 17).

Algoritmo 17 - Soluzione del sistema lineare $Ax = b$

Input: matrice A di dimensioni $n \times n$, vettore b di dimensione n
Output: vettore $x = [x_i]$ (di dimensione n) delle incognite

```
1: # riduzione a 0 degli elementi del triangolo inferiore sinistro
2: for j in range(n) do
3:     ricerca della riga p del pivot
4:     scambio delle righe j ↔ p
5:     riduzione a 0 degli elementi della j-esima colonna
6: end for
7: # soluzione del sistema con valutazione a ritroso delle incognite x_i
8: for i in reversed(range(n)) do
9:     calcola x_i a partire dalla conoscenza di x_{i+1}, ..., x_{n-1}
10: end for
11: return vettore x = [x_i]
```

L'algoritmo 17 può essere ulteriormente raffinato come descritto nell'algoritmo 18.

Algoritmo 18 - Soluzione del sistema lineare $Ax = b$

Input: matrice $A = [a_{ij}]$ di dimensione $n \times n$, vettore b di dimensione n
Output: vettore $x = [x_i]$ (di dimensione n) delle incognite

```
1:  # riduzione a 0 degli elementi del triangolo inferiore sinistro
2:  for j in range(n) do
3:      # ricerca della riga p del pivot
4:      p ← j
5:      for k in range(j + 1, n) do
6:          if |a_kj| > |a_pj| then
7:              p ← k
8:          end if
9:      end for
10:     # scambio delle righe j ↔ p
11:     if j ≠ p then
12:         for k in range(n) do
13:             scambia a_jk ↔ a_pk
14:         end for
15:         scambia b_j ↔ b_p
16:     end if
17:     # riduzione a 0 degli elementi della j-esima colonna
18:     for i in range(j + 1, n) do
19:         f ← a_ij / a_jj
20:         for k in range(n) do
21:             a_ik ← a_ik − a_jk * f
22:         end for
23:         b_i ← b_i − b_j * f
24:     end for
25: end for
26: # soluzione del sistema con valutazione a ritroso delle incognite
27: for i in reversed(range(n)) do
28:     s ← 0
29:     for k in range(i + 1, n) do
30:         s ← s + a_ik * x_k
31:     end for
32:     x_i ← (b_i − s) / a_ii
33: end for
34: return vettore x = [x_i]
```

ESERCIZI

4.1 Descrivere la situazione in memoria della struttura descritta nell'esempio 4.3.1.

4.2 Descrivere graficamente la situazione che si crea con le assegnazioni

$$\begin{array}{lcl} a & \leftarrow & [1,2,3,4] \\ b & \leftarrow & [a, a[2]] \end{array}$$

Spiegare come si evolve la situazione con la sequenza di assegnazioni

$$\begin{array}{lcl} a[0] & \leftarrow & a[1] \\ a[1] & \leftarrow & b[1] \\ a[2] & \leftarrow & b \end{array}$$

4.3 Valutare le seguenti espressioni:

1. $[\,[3,4,5],[6,7]\,]\,[2]\,[1]$
2. $(5 * (4 * [1,2,3]))[2]$

4.4 Determinare le coppie di numeri naturali che danno per prodotto un dato numero naturale.

4.5 Costruire una struttura della forma $[[1],[2,2],[3,3,3],[4,4,4,4]]$, composta da un generico numero n di elementi.

4.6 Il *triangolo di Pascal* è costituito da numeri naturali disposti in righe; ogni riga è formata da un numero di numeri uguale al numero di riga e, disponendo le righe in modo centrato, ogni numero è la somma dei due numeri adiacenti della riga superiore. A seguire è riportata una porzione del triangolo.

$$\begin{array}{ccccccccc} & & & & 1 & & & & \\ & & & 1 & & 1 & & & \\ & & 1 & & 2 & & 1 & & \\ & 1 & & 3 & & 3 & & 1 & \\ 1 & & 4 & & 6 & & 4 & & 1 \end{array}$$

1. Determinare il k-esimo numero dell'n-esima riga del triangolo di Pascal (tale numero viene detto *binomiale di n su k*).
2. Determinare l'n-esima riga del triangolo di Pascal.
3. Determinare la sequenza delle prime n righe del triangolo di Pascal $([\,[1],[1,1],[1,2,1],\ldots\,])$

4.7 Determinare tutte le possibili scomposizioni in addendi non nulli di un dato numero naturale (indipendentemente dall'ordine); ad esempio, il numero $n = 4$ può

essere scomposto in addendi come segue:

$$\begin{aligned} 4 &= 4 \\ 4 &= 1+3 \\ 4 &= 2+2 \\ 4 &= 1+1+2 \\ 4 &= 1+1+1+1 \end{aligned}$$

e la soluzione può essere espressa mediante la struttura

$$[\,[4],[1,3],[2,2],[1,1,2],[1,1,1,1]\,]$$

4.8 Decidere se una struttura è semplice, ossia se è formata da elementi atomici.

4.9 Determinare il numero di atomi presenti in una struttura; ad esempio, il numero di atomi della struttura $[1,[2,3],4,[5]\,]$ è 5.

4.10 Determinare la somma degli elementi di una struttura.

4.11 Stabilire se una struttura composta è formata da atomi tutti distinti fra loro.

4.12 Determinare il numero di atomi di una struttura composta.

4.13 Determinare il livello (di annidamento) di una struttura (ossia l'altezza dell'albero che la rappresenta); ad esempio: l'atomo 3 ha livello 0; la struttura semplice $[2,4,6]$ ha livello 1; la struttura $[1,3,[4]\,]$ ha livello 2; la struttura $[1,[3,2,5],[7,[6,7]\,],8]$ ha livello 3.

4.14 Decidere se due strutture sono uguali.

4.15 Decidere se una struttura è contenuta all'interno di un'altra; ad esempio la struttura $[3,4,[5]\,]$ è contenuta nella struttura $[1,[2],[3,4,[5]],[6,7]\,]$.

4.16 Con *schema* di una struttura si intende la struttura ottenuta sostituendo ciascun atomo con un simbolo convenzionale ($*$); ad esempio lo schema della struttura $[4,[9,3],5,[\,[4],7]\,]$ è la struttura $[*,[*,*],*,[\,[*],*]\,]$. Data una struttura, determinarne lo schema. Due strutture aventi lo stesso schema sono dette *isomorfe*; ad esempio sono isomorfe le seguenti due strutture: $[\,[2],4,[\,]\,]$, $[\,[7],2,[\,]\,]$. Decidere se due date strutture sono isomorfe.

4.17 Con *scheletro* di una struttura si intende la struttura in cui vengono eliminati tutti gli atomi e vengono mantenute solo le parentesi di annidamento; ad esempio lo scheletro della lista $[6,[3,5],4,[\,[9],2]\,]$ è la lista $[\,[\,],[\,[\,]\,]\,]$. Data una struttura, determinarne lo scheletro.

4.18 *Comprimere* e *decomprimere* una struttura semplice di numeri naturali, rappresentando una sequenza contigua di numeri mediante la coppia dei due numeri estremi; ad esempio la sequenza

$$[2,3,4,5,7,9,10]$$

viene compressa con la seguente struttura:

$$[\,[2,5],7,[9,10]\,]$$

4.19 *Comprimere* e *decomprimere* una struttura semplice, secondo i seguenti schemi di trasformazione di evidente interpretazione.

1. $[a,b,b,c,d,d,d,d] \leftrightarrow [\,[a,1],[b,2],[c,1],[d,4]\,]$
2. $[a,b,b,c,d,d,d,d] \leftrightarrow [a,1,b,2,c,1,d,4]$
3. $[a,b,b,c,d,d,d,d] \leftrightarrow [a,[b,2],c,[d,4]\,]$

4.20 Comprimere una struttura semplice di 0 e 1, comprimendo le sottosequenze di elementi uguali contigui, ottenendo una sequenza composta, come descritto dal seguente esempio:

$$[1,1,1,0,0,1,1,1,1,1,0,0,0,0,0,0,0] \rightarrow [[3,1],[2,0],[5,1],[7,0]]$$

Seguendo lo stesso criterio, decomprimere una sequenza composta riottenendo la sequenza piatta corrispondente.

4.21 Descrivere graficamente la rappresentazione in memoria della matrice riportata nell'esempio 4.8.1.

4.22 Descrivere graficamente la rappresentazione in memoria corrispondente alla fine delle seguenti assegnazioni, in un contesto in cui ogni valore è un *oggetto*:

$a \leftarrow [3,[4,6],[7]]$
$a[1][1] \leftarrow a[2]$
$a[2] \leftarrow a[1][0]$

Stabilire cosa viene stampato con `print(a)`.

4.23 Costruire una matrice quadrata con i valori 0 e 1, mettendo il valore 1 (e 0 altrove) secondo i seguenti schemi:

1. sulla diagonale principale
2. sulla diagonale secondaria
3. su entrambe le diagonali
4. sui bordi della matrice
5. sul triangolo superiore destro
6. sul triangolo superiore sinistro
7. sul triangolo inferiore destro
8. sul triangolo inferiore sinistro
9. sulle righe di indice pari
10. sulle colonne di indice pari

4.24 Costruire una matrice quadrata ad anelli concentrici, mettendo 1 sull'anello esterno, 2 nel secondo anello e così via.

4.25 Costruire una matrice quadrata $n \times n$ che costituisca la tabella di *addizione modulo n*.

4.26 Costruire delle matrici rettangolari secondo i seguenti schemi:

0	0	0	0	0	0
0	1	1	1	1	0
0	1	1	1	1	0
0	1	1	1	1	0
0	0	0	0	0	0

0	1	0	1	0	1
1	0	1	0	1	0
0	1	0	1	0	1
1	0	1	0	1	0
0	1	0	1	0	1

4.27 Costruire delle matrici di numeri secondo i seguenti formati, riferiti a matrici di dimensioni $m = 5$, $n = 6$:

11	12	13	14	15	16
21	22	23	24	25	26
31	32	33	34	35	36
41	42	43	44	45	46
51	52	53	54	55	56

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30

4.28 Costruire una matrice quadrata di ordine n con gli n^2 numeri naturali $1, 2, 3, \ldots, n^2$, secondo gli schemi sotto riportati:

1	2	6	7	15
3	5	8	14	16
4	9	13	17	22
10	12	18	21	23
11	19	20	24	25

21	22	23	24	25
20	7	8	9	10
19	6	1	2	11
18	5	4	3	12
17	16	15	14	13

4.29 Costruire una matrice quadrata $n \times n$ con i numeri naturali costituenti la tavola pitagorica della moltiplicazione.

4.30 Costruire una matrice quadrata di ordine n dispari con i numeri con i numeri naturali $1, 2, ..., n^2$ posizionando il numero 1 al centro e poi proseguendo a spirale in senso antiorario.

4.31 Costruire una matrice quadrata di numeri naturali con i numeri $1, 2, ..., n^2$ in modo da formare un quadrato magico.

4.32 Costruire una matrice 5×10 con numeri casuali secondo le regole delle estrazioni del lotto.

4.33 Una matrice rettangolare contiene numeri naturali distinti estratti da un'urna; il valore 0 rappresenta la non presenza di un valore *non estratto*. Decidere se due dati numeri formano un ambo, ossia se sono entrambi presenti in una stessa riga della matrice.

4.34 Stabilire l'effetto della seguente porzione di algoritmo agente su una matrice quadrata a di dimensione n:

```
1: for i in range(n) do
2:     for j in range(n) do
3:         a_ij ← a_ji
4:     end for
5: end for
```

Analogamente, spiegare l'effetto sostituendo l'istruzione $a_{ij} \leftarrow a_{ji}$ con l'istruzione di scambio $a_{ij} \leftrightarrow a_{ji}$.

4.35 Determinare l'indice di riga di una matrice numerica per il quale la somma degli elementi è massima.

4.36 Determinare in una matrice bidimensionale binaria il rettangolo di maggior superficie composto da tutti 1.

4.37 Determinare il valore minimo ed il valore massimo fra quelli presenti in una matrice numerica.

4.38 Determinare il minimo ed il massimo e la loro frequenza, in una matrice numerica.

4.39 Decidere se all'interno di una matrice è presente un dato elemento.

4.40 Ricercare se e dove è presente un dato valore in una matrice.

4.41 Determinare quante volte un dato elemento è presente all'interno di una matrice.

4.42 Decidere se una matrice è composta da elementi tutti uguali.

4.43 Determinare il numero di elementi distinti presenti in una matrice.

4.44 Decidere se in una data matrice esistono due righe uguali.

4.45 Determinare il prodotto degli elementi di una matrice di numeri. Ottimizzare il procedimento, terminando il calcolo quando si incontra un elemento di valore zero.

4.46 Determinare la frequenza degli elementi che compaiono con maggiore frequenza in una matrice.

4.47 Determinare gli elementi che compaiono con maggiore frequenza in una matrice.

4.48 Contare il numero di numeri primi presenti in una matrice.

4.49 Determinare l'elemento che compare con maggiore frequenza in una matrice.

4.50 Ricercare se in una matrice di caratteri è presente (orizzontalmente o verticalmente) una data parola.

4.51 Decidere se una data matrice quadrata costituisce un quadrato magico.

4.52 Decidere se una data matrice numerica 9×9 è corretta in base alle regole del gioco del sudoku.

4.53 Realizzare un risolutore del gioco del sudoku.

4.54 Ruotare una matrice quadrata di 90^o a destra rispetto al suo centro.

4.55 Data una matrice di caratteri composta dai soli caratteri ' ' (spazio) e '*' (muro), decidere se esiste un percorso che attraversa la matrice da una casella del bordo ad un'altra casella del bordo (eventualmente anche sullo stesso lato).

4.56 Data una matrice di caratteri ' ' (spazio)' e '*', determinare il numero di isole presenti, interpretando il carattere '*' come terra ed il carattere ' ' come acqua.

4.57 Sia a una sequenza di coppie $[a_i, b_i]$ di numeri, con $a_i \leq b_i$, ciascuna delle quali individua un intervallo della retta.

1. Determinare il più piccolo intervallo contenente tutti gli intervalli dell'array a.
2. Determinare la lunghezza della porzione di retta ricoperta dall'unione degli intervalli di a.
3. Determinare gli intervalli che si ottengono dall'unione degli intervalli di a.
4. Decidere se esistono sovrapposizioni fra gli intervalli di a.
5. Decidere se un punto x è contenuto fra negli intervalli di a.
6. Semplificare la sequenza a, eliminando gli intervalli che sono completamente contenuti in altri.

4.58 In una matrice a un elemento a_{ij} dicesi *punto di sella* se esso è maggiore di tutti gli elementi dell'i-esima riga e minore di tutti gli elementi della j-esima colonna, oppure, simmetricamente, minore di tutti gli elementi dell'i-esima riga e maggiore di tutti gli elementi della j-esima colonna. Determinare i punti di sella di una matrice numerica.

4.59 In un'aula rettangolare si devono predisporre in file allineate delle postazioni per svolgere un esame. Per evitare possibili copiature vengono predisposte 4 prove distinte, individuate mediante i numeri $1, 2, 3, 4$. Per rappresentare la disposizione dell'aula ed il numero della prova corrispondente a ciascuna postazione viene utilizzata una matrice. Riempire una matrice rettangolare con i numeri $1, 2, 3, 4$ in modo che due candidati con lo stesso numero di prova abbiano una distanza, secondo la metrica Manhattan, la maggiore possibile.

4.60 Data una matrice $3 \times n$ contenenti le dimensioni (larghezza, altezza, profondità di n scatole a forma di parallelepipedo, determinare le dimensioni della più piccola scatola in grado di contenerle tutte (una alla volta), mantenendo fra loro parallele le facce della scatola esterna e della scatola interna.

4.61 Una struttura è formata dalle coppie di elementi che sono in una relazione di equivalenza. Determinare la struttura formata dalle sequenze di elementi della partizione indotta dalla relazione di equivalenza.

4.62 Risolvere con il metodo di Cramer un sistema lineare della forma $A\,x = b$, essendo A la matrice quadrata dei coefficienti e b il vettore dei termini noti. Comparare i tempi di esecuzione dell'algoritmo di Cramer con quelli dell'algoritmo di riduzione di Gauss.

5

RAPPRESENTARE

[...] l'intelligenza dipende in modo cruciale dalla capacità di costruire descrizioni di livello superiore di certi raggruppamenti complessi [...].

D. R. Hofstadter, *Gödel, Escher,Bach*

In questo capitolo vengono analizzate le modalità con cui rappresentare gli oggetti mediante la composizione di entità elementari; viene posto l'accento sulla struttura organizzativa degli oggetti composti, formati da aggregazioni di oggetti più elementari.

Gli argomenti che seguono vanno sotto il nome di *strutture* (*organizzative*) *dei dati*. Il discorso affrontato in questo capitolo verrà completato quando, nei capitoli successivi, alle strutture dei dati si affiancheranno le corrispondenti operazioni che saranno dettate non tanto dalla particolare struttura dati utilizzata, ma dalla semantica dell'oggetto rappresentato; anzi, sono proprio le operazioni che si desiderano applicare agli oggetti che orientano su quale sia la struttura dati più idonea per rappresentare l'oggetto. Questa più matura e più produttiva impostazione è caratteristica della programmazione orientata agli oggetti, che sarà affrontata più avanti.

5.1 Rappresentare

Rappresentare significa stabilire una corrispondenza fra gli elementi di due ambienti diversi. La funzione inversa viene detta *interpretazione*. La rappresentazione costituisce un meccanismo trasversale di tutta la Scienza ed in particolare dell'Informatica e della Programmazione.

Esempio 5.1.1 - Un esempio di rappresentazione tratto dalla quotidianità è dato dalla mappa dei percorsi dei servizi pubblici di una città.

Le mappe dei trasporti pubblici di Londra, benché fossero disponibili già dal 1908, erano oggetti di estrema complessità che ne comprometteva un loro semplice uso. La mappa della metropolitana di Londra (riportata in figura 5.1) venne progettata da Harry Beck nel 1931. Viene considerata come un classico del design ed ha ispirato le rappresentazioni dei sistemi di trasporto urbano di tutto il mondo.

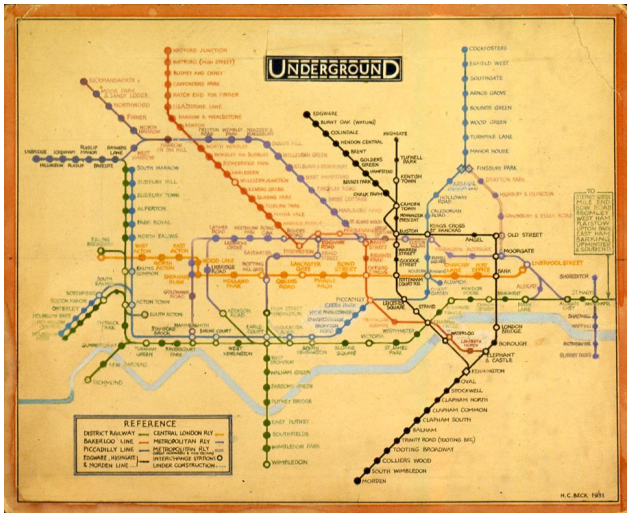


Figura 5.1: La mappa della metropolitana di Londra.

Una mappa di questo tipo è finalizzata a rappresentare in modo facilmente interpretabile solo alcuni aspetti, quali ad esempio la sequenzialità delle stazioni di passaggio, sacrificandone altri, quali gli attributi di tipo metrico: non troveremmo mai sulla mappa una didascalia del tipo $SCALA\,1{:}10000$.

5.2 Rappresentare ed interpretare

Dato un insieme $\mathcal{A}$ ed un insieme $\mathcal{B}$, una *rappresentazione* è una corrispondenza

$$rap : \mathcal{A} \to \mathcal{B}$$

Questa corrispondenza deve essere, preferibilmente ma non necessariamente, univoca, nel senso che ad un elemento di $\mathcal{A}$ possono corrispondere più elementi di $\mathcal{B}$. La corrispondenza inversa di una rappresentazione viene detta

interpretazione ed ha la seguente forma

$$int : \mathcal{B} \rightarrow \mathcal{A}$$

La potenza del meccanismo di rappresentazione risiede e dipende dalla ricchezza e duttilità della struttura dell'insieme $\mathcal{B}$. Nell'informatica, nel contesto della soluzione dei problemi, l'insieme $\mathcal{A}$ viene detto *spazio dei problemi* e rappresenta il contesto in cui sorgono e si analizzano i problemi, mentre l'insieme $\mathcal{B}$ viene detto *spazio delle soluzioni* e costituisce l'ambiente in cui viene descritta la soluzione del problema e viene svolta la fase di elaborazione. Il meccanismo della rappresentazione ed interpretazione sta a fondamento del tradizionale processo di risoluzione di un problema, eseguendo una trasformazione dallo spazio dei problemi allo spazio delle soluzioni dove avviene la fase di elaborazione; successivamente avviene la fase di interpretazione dei risultati ottenuti, come schematizzato nella figura 5.2.

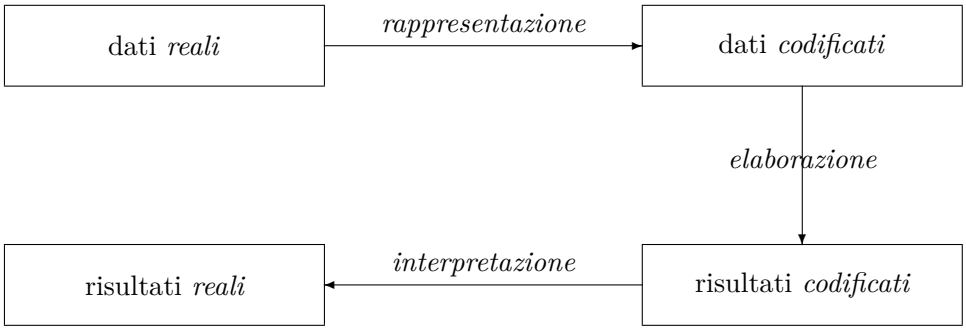


Figura 5.2: Schema delle fasi della rappresentazione dei dati ed interpretazione dei risultati.

5.3 Rappresentare mediante strutture

Uno dei più classici esempi di rappresentazione nell'ambito dell'informatica è costituito dalla codifica dei *numeri naturali* mediante *stringhe binarie*, formate da 0 e 1. Se le uniche forme di elaborazione degli oggetti fossero quelle di lettura da tastiera, visualizzazione di caratteri su video o su un foglio (o, equivalentemente, scrittura e lettura da file), trasmissione e ricezione lungo un canale di comunicazione si potrebbe rappresentare qualsiasi oggetto mediante una stringa. La rappresentazione mediante una stringa sarebbe potenzialmente in grado di rappresentare qualsiasi oggetto (un numero, una figura geometrica, un'immagine, una molecola, ...) ma risulterebbe pesantemente inadeguata nel momento in cui si passa ad eseguire delle operazioni. In tutti questi casi si utilizzano delle rappresentazioni mediante strutture.

Le strutture dati, grazie alla possibilità di essere composte da elementi costituiti a loro volta da strutture, risultano uno strumento molto versatile che si presta a rappresentare qualsiasi oggetto, comunque articolato e comunque

complesso. Come avviene per ogni forma di rappresentazione, individuato un insieme $\mathcal{A}$ di oggetti da rappresentare, si tratta di definire una funzione

$$rap : \mathcal{A} \to \mathit{Struttura}$$

che ad ogni elemento dell'insieme $\mathcal{A}$ associa una struttura che ne è la rappresentazione.

Problema 5.3.1 Rappresentare il seguente scontrino fiscale:

```
Scontrino fiscale N.27
Data: 16 gennaio 2022

2 caffè:
  2 x 1.20 Euro  2.40 Euro
1 toast:
  1 x 2.50 Euro  2.50 Euro

        TOTALE:  4.90 Euro
```

Soluzione. Nello scontrino compaiono valori che costituiscono sia i dati che i risultati. In fase di rappresentazione si considerano solamente i dati e si lascia il calcolo dei risultati ad eventuali fasi successive. Lo scontrino può essere rappresentato mediante la seguente struttura:

$$[[27, [16, \texttt{'gennaio'}, 2022]], [[2, \texttt{'caffè'}, 120], [1, \texttt{'toast'}, 250]]]$$

5.4 Rappresentare oggetti numerici

I numeri costituiscono gli oggetti più caratteristici della matematica; vengono classificati in diverse famiglie; ciascuna famiglia viene definita come ampliamento di una famiglia già esistente, dotandola di caratteristiche che permettono l'esecuzione di operazioni aggiuntive. Alla radice di queste diverse famiglie di numeri si collocano i numeri naturali che, in questo paragrafo, consideriamo come oggetti elementari già realizzati. Questa ipotesi è allineata con lo sviluppo storico del pensiero matematico e con quanto si trova nei tradizionali linguaggi di programmazione.

Esempio 5.4.1 - I *numeri interi* sono numeri della forma 7, $^{-}3$, 0, 48. Supponiamo di avere a disposizione solo i numeri naturali e ci proponiamo di rappresentare i numeri interi. La rappresentazione dovrà supportare le tradizionali operazioni di addizione, sottrazione e moltiplicazione sui numeri interi. Come si fa in Matematica, si rappresenta un numero intero k mediante una coppia $[m, n]$ di numeri naturali definita come segue:

$$rap(k) \stackrel{\text{def}}{=} \begin{cases} [k, 0] & \text{se } k \geq 0 \\ [0, k] & \text{se } k < 0 \end{cases}$$

Ad esempio, il numero 7 è rappresentato dalla coppia $[7, 0]$ ed il numero $^{-}3$ dalla coppia $[0, 3]$.

Esempio 5.4.2 - I *numeri decimali*, sono numeri della forma $27.125, 473., 0.0315$. Supponendo di avere a disposizione i numeri naturali si può rappresentare un numero decimale mediante una coppia di numeri naturali costituita dalla parte intera e dalla mantissa del numero; ad esempio, il numero 27.125 viene rappresentato dalla coppia $[27, 125]$.

Esempio 5.4.3 - Una *frazione* o *numero razionale* è un numero della forma

$$\frac{n}{d}$$

dove n e d sono due numeri naturali detti, rispettivamente, *numeratore* e *denominatore*. Per semplicità ed univocità di scrittura si considera che la frazione sia ridotta ai minimi termini, ossia che numeratore e denominatore siano fra loro coprimi, ossia che non abbiano fattori comuni. Il modo più naturale per rappresentare una frazione consiste nell'usare una coppia di numeri corrispondenti al numeratore ed al denominatore:

$$[n, d]$$

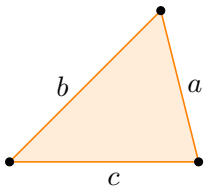
Le frazioni sono oggetti matematici che generalmente non si trovano predisposti nei linguaggi di programmazione. Nei linguaggi di programmazione orientati agli oggetti le frazioni vengono realizzate mediante una classe; ed a quel punto si potranno utilizzare, negli algoritmi, le frazioni come si usano i valori numerici già predisposti nei linguaggi.

Osservazione. I precedenti esempi evidenziano che una stessa struttura può rappresentare diverse tipologie di oggetti (numeri interi, numeri decimali e frazioni). La fase di rappresentazione è solo il primo passo; infatti le diverse tipologie di numeri sono caratterizzate da un ricco repertorio di operazioni; la differenziazione semantica delle strutture avviene proprio definendo le operazioni.

5.5 Rappresentare oggetti geometrici

Nel contesto della geometria del piano si incontrano diverse tipologie di oggetti; alcuni sono descritti negli esempi che seguono.

Esempio 5.5.1 - Un triangolo del piano euclideo risulta univocamente individuato dalle lunghezze dei suoi lati, come descritto nella seguente figura:



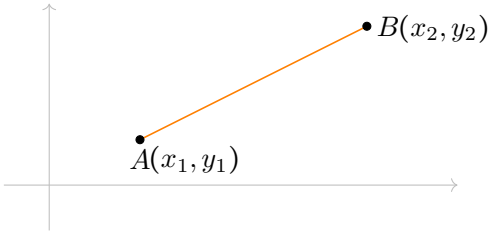
Un tale triangolo può essere rappresentato mediante una semplice struttura:

$$[a, b, c]$$

Questa rappresentazione non è univoca in quanto una qualsiasi permutazione della sequenza rappresenta lo stesso triangolo.

Introducendo un sistema di assi cartesiani, un punto del piano può essere rappresentato mediante la coppia delle sue coordinate; su questa utilizzatissima funzione di rappresentazione si fonda il ramo della Matematica nota con il nome di *geometria analitica*. Questa forma di rappresentazione permette di rappresentare altre entità geometriche più articolate, facendo riferimento alla rappresentazione dei punti.

Esempio 5.5.2 - Un *segmento* di dati estremi, come descritto nella figura che segue:



può essere rappresentato dalla seguente struttura:

$$[[x_1, y_1], [x_2, y_2]]$$

Analogamente, una *spezzata* o un *poligono* possono essere rappresentati dalla sequenza delle coppie di numeri che rappresentano i vertici, ossia mediante una struttura della forma

$$[[x_1, y_1], [x_2, y_2], \ldots, [x_n, y_n]]$$

Esempio 5.5.3 - In un sistema di assi cartesiani una *retta* può essere rappresentata basandosi sulle coordinate di due punti di passaggio, come visto sopra per un segmento, oppure ricorrendo ai coefficienti della sua equazione cartesiana $ax + by + c = 0$:

$$[a, b, c]$$

Esempio 5.5.4 - Una terna $[a, b, c]$ può rappresentare diverse tipologie di oggetti; ad esempio:

- un'equazione di una retta in forma implicita $ax + by + c = 0$
- un trinomio di secondo grado $ax^2 + bx + c$
- un'equazione di secondo grado $ax^2 + bx + c = 0$
- una parabola di equazione $y = ax^2 + bx + c$
- un triangolo di lati a, b, c
- un punto dello spazio $\mathbb{R}^3$

5.6 Rappresentare oggetti algebrici

Il mondo dell'Algebra è popolato da una grande varietà di oggetti: operazioni, espressioni, polinomi, equazioni, funzioni ed altri. La modalità di rappresentazione di questi oggetti viene definita in funzione delle operazioni che si devono svolgere sugli stessi.

Esempio 5.6.1 - Nel caso in cui l'obiettivo sia la determinazione delle radici di un'*equazione di secondo grado*, l'equazione può essere rappresentata mediante la terna dei suoi coefficienti (tralasciando il nome identificativo dell'incognita): l'equazione $ax^2 + bx + c = 0$ viene rappresentata mediante la struttura

$$[a, b, c]$$

Esempio 5.6.2 - Un'*espressione* è un oggetto matematico che viene tradizionalmente scritto, su un testo di Matematica, con una notazione della forma

$$3xy + 7 + (x + y)(x - y)$$

Nel contesto del codice sorgente di un linguaggio di programmazione o nelle operazioni di input/output testuale a carattere, questa espressione può essere scritta come stringa nella forma

```
3*x*y+7+(x+y)*(x-y)
```

Tale rappresentazione in stringa risulta poco comoda ai fini dell'elaborazione dell'espressione (valutazione per specifici valori delle incognite, semplificazione, derivazione, operazioni fra espressioni, ...). Per queste forme di elaborazione risulta preferibile una rappresentazione ad albero come descritto nella figura 5.3.

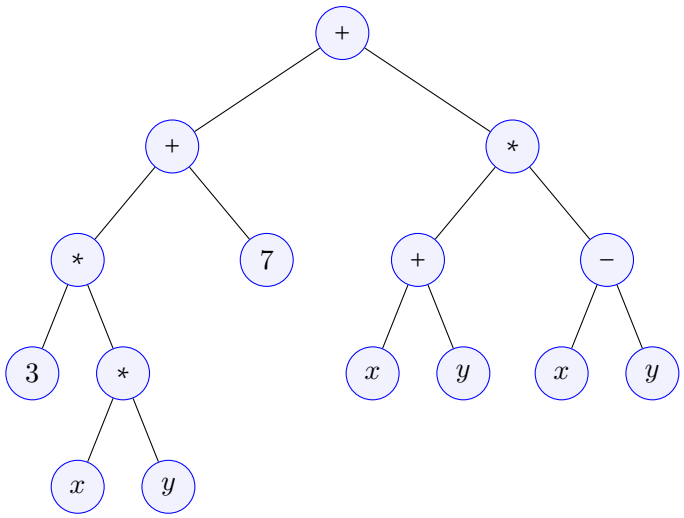


Figura 5.3: Rappresentazione ad albero dell'espressione $3xy + 7 + (x + y)(x - y)$.

L'albero riportato nella figura 5.3 può a sua volta essere descritto mediante una struttura, come segue:

$$[[+,[*,3,[*,x,y]],7],[*,[+,x,y],[-,x,y]]]$$

Esempio 5.6.3 - Una generalizzazione dell'esempio precedente consiste nell'usare operatori di diverse arietà ed utilizzare una struttura composta della forma

$$[\circ, x_1, \ldots, x_n]$$

dove $\circ$ denota un generico operatore e gli argomenti x_i sono delle generiche espressioni. Con questa convenzione, l'espressione $-x + 3y + 4z + 2t$ viene rappresentata mediante la struttura

$$[+,[-,x],[*,3,y],[*,4,z],[*,2,t]]$$

e l'espressione $3xy + 7 + (x + y)(x - y)$ mediante la struttura

$$[+,[*,3,x,y],7,[*,[+,x,y],[-,x,y]]]$$

Esempio 5.6.4 - Un *polinomio* di grado n in una indeterminata x, della forma

$$a_n\, x^n + a_{n-1}\, x^{n-1} + \cdots + a_1\, x + a_0$$

può essere rappresentato mediante una sequenza semplice di dimensione $n + 1$ contenente, in ordine di grado, i coefficienti dei monomi del polinomio:

$$[a_0, a_1, \ldots, a_{n-1}, a_n]$$

Nel caso in cui il polinomio sia *sparso*, ossia molti dei coefficienti siano nulli, si può adottare la rappresentazione che registra le coppie $[a_k, k]$ costituite dai coefficienti a_k (non nulli) ed il grado del corrispondente monomio. Usando queste due modalità di rappresentazione, il polinomio

$$4\, x^7 + 6\, x^5 - 3\, x^2 + 2$$

viene rappresentato, rispettivamente, dalle due strutture

$$[2, 0, {}^{-}3, 0, 0, 6, 0, 4]$$

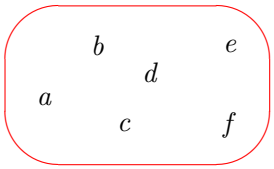
$$[\,[2,0],[{}^{-}3,2],[6,5],[4,7]\,]$$

Il criterio per la scelta della struttura dati si basa sulla semplicità e sull'efficienza delle operazioni che dovranno essere svolte.

5.7 Rappresentare aggregazioni

Le strutture si prestano a rappresentare qualsiasi forma di aggregazione di elementi. Ne sono testimonianza gli esempi che seguono.

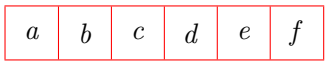
Esempio 5.7.1 - Un *insieme* $\{a, b, c, d, e, f\}$ di elementi può essere rappresentato mediante una struttura come descritto nella figura 5.4. Questa rappresentazione non è biunivoca in quanto ogni permutazione della sequenza $[a, b, c, d, e, f]$ può essere considerata accettabile rappresentazione dell'insieme considerato nella figura 5.4.



$[a, b, c, d, e, f]$

Figura 5.4: Insieme e corrispondente rappresentazione mediante una struttura.

Esempio 5.7.2 - Una *sequenza* di elementi può essere rappresentata mediante una struttura come descritto nella figura 5.5. In questo caso la corrispondenza è biunivoca.



$[a, b, c, d, e, f]$

Figura 5.5: Sequenza e corrispondente rappresentazione mediante una struttura.

Osservazione. I precedenti due esempi evidenziano che una stessa struttura può rappresentare due aggregazioni che dal punto di vista concettuale ed operativo sono completamente diverse (insiemi e sequenze). Il meccanismo della rappresentazione deve essere pertanto completato definendo le operazioni che caratterizzano l'oggetto rappresentato.

Esempio 5.7.3 - Una *matrice* è un particolare oggetto matematico, composto da elementi disposti in una tabella rettangolare di m righe ed n colonne:

$$\begin{bmatrix} a_{11} & \dots & a_{1n} \\ a_{21} & \dots & a_{2n} \\ \dots & \dots & \dots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}$$

Una matrice può essere rappresentata mediante una struttura composta come descritto nella figura 5.6.

a	b
c	d
e	f

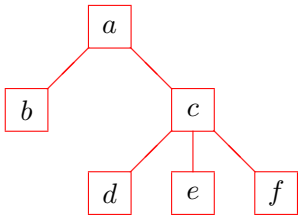
$$[\,[a,b],[c,d],[e,f]\,]$$

Figura 5.6: Matrice e corrispondente rappresentazione mediante una struttura.

Una matrice $m \times n$ può essere rappresentata anche mediante una sequenza di mn elementi, unitamente alla coppia $[m,n]$ delle dimensioni che permette di ricostruire la struttura bidimensionale della matrice. Ad esempio, la matrice descritta nella figura 5.6 può essere rappresentata mediante la struttura

$$[[a,b,c,d,e,f],[3,2]]$$

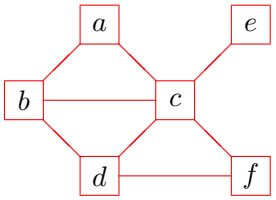
Esempio 5.7.4 - Un *albero* di elementi può essere rappresentato mediante una struttura composta come descritto nella figura 5.7.



$$[a,[b],[c,[d],[e],[f]\,]\,]$$

Figura 5.7: Albero e corrispondente rappresentazione mediante una struttura.

Esempio 5.7.5 - Un *grafo* è un insieme di elementi dello stesso tipo fra loro interconnessi; può essere rappresentato mediante una struttura come descritto nella figura 5.8.



$$[\,[a,b],[a,c],[b,c],[b,d],[c,d],[c,e],[c,f],[d,f]\,]$$

Figura 5.8: Grafo e corrispondente rappresentazione mediante una struttura.

5.8 Rappresentare sistemi

Un *sistema* è un oggetto strutturato composto da oggetti più elementari fra loro connessi.

Esempio 5.8.1 - Una porzione di file system è costituita da una cartella a, che contiene due cartelle b e c, e da una cartella c che contiene una cartella d e due file f e g; può essere descritta mediante lo schema grafico ad albero riportato nella figura 5.9.

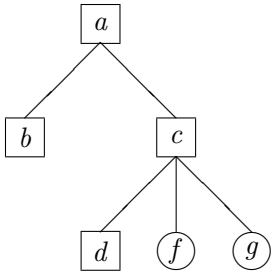


Figura 5.9: Albero di cartelle e file.

L'albero descritto nella figura 5.9 può essere rappresentato mediante la seguente struttura, dove i valori 0 e 1 stanno a specificare se si tratta di una cartella o di un file:

$$[\,[a,0],[\,[b,0]\,],[\,[c,0],[\,[d,0],[f,1],[g,1]\,]\,]\,]$$

C'è da notare che non ogni struttura della forma sopra rappresenta una effettiva porzione di file system, in quanto bisogna imporre il vincolo che un file non contenga cartelle o file. Per prevenire queste situazioni di strutture incoerenti si delega la creazione e la modifica della struttura a delle operazioni, appositamente predisposte, che intervengono sulla struttura in modo controllato.

Problema 5.8.1 Definire una rappresentazione di un sistema a blocchi costituito da blocchi collegati fra loro in serie o in parallelo. Un esempio di questo tipo di sistema a blocchi è riportato nella figura 5.10.

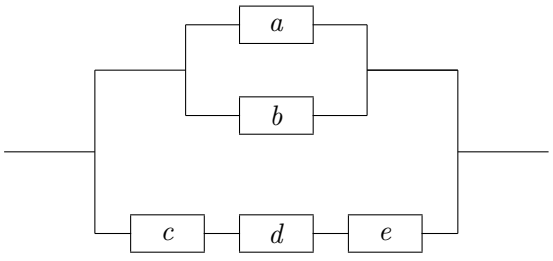


Figura 5.10: Sistema a blocchi.

Soluzione. L'idea sulla quale si basa la rappresentazione di un sistema a blocchi di questo tipo è descritta nella figura 5.11. Analogamente si possono rappresentare gli accoppiamenti in serie ed in parallelo di più di due blocchi.

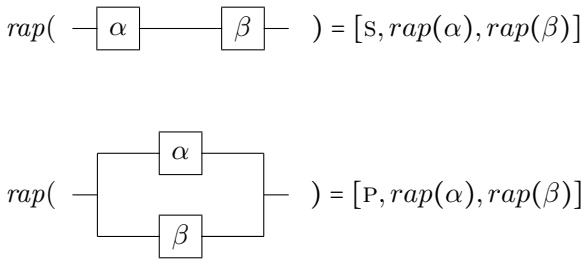


Figura 5.11: Rappresentazione di blocchi in serie ed in parallelo. α e β sono blocchi o elementi atomici e P ed S sono dei valori simbolici che qualificano la tipologia (Serie o Parallelo) dell'accoppiamento fra due blocchi.

Il sistema a blocchi descritto riportato nella figura 5.10 può essere rappresentato mediante l'albero riportato nella figura 5.12 dove i nodi interni indicano se i sottoblocchi rappresentati dai due sottoalberi figli sono connessi in serie o in parallelo.

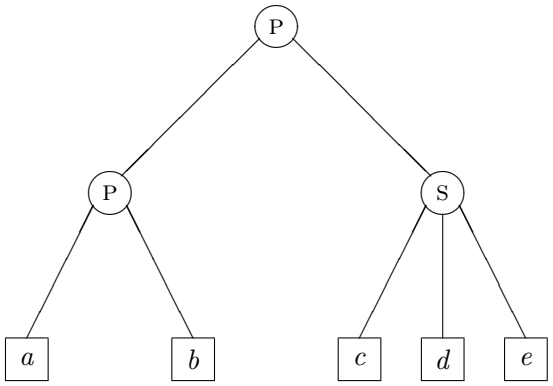


Figura 5.12: Albero che rappresenta il sistema a blocchi di figura 5.10.

Tale albero a sua volta può essere rappresentato mediante la struttura che segue:

$$[\text{P}, [\text{P}, [a, b]\,], [\text{S}, [c, d, e]\,]\,]$$

□

5.9 Composizione di rappresentazioni

La rappresentazione di un oggetto può avvenire attraverso passi successivi in cui ogni livello costituisce la rappresentazione del livello precedente.

Esempio 5.9.1 - Per descrivere i confinamenti fra regioni geografiche si usa solitamente uno schema grafico come quello riportato in figura 5.13, dove vengono trascurate le informazioni di tipo metrico e considerate solo le informazioni topologiche relative ai confinamenti.

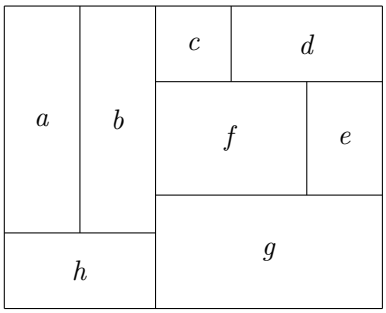


Figura 5.13: Mappa dei confinamenti fra regioni.

Lo schema riportato in figura 5.13 può essere trasformato biunivocamente nel grafo isomorfo riportato nella figura 5.14.

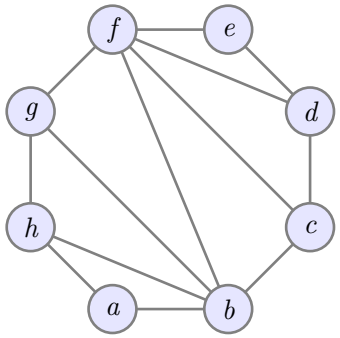


Figura 5.14: Grafo corrispondente alla mappa dei confinamenti fra regioni riportata nella figura 5.13.

Questo grafo risulta più semplicemente traducibile nelle strutture dati predisposte dai linguaggi di programmazione che consentono delle efficienti forme di elaborazione. Ad esempio, il grafo riportato nella figura 5.14 può essere rappresentato mediante una sequenza di coppie, ciascuna delle quali rappresenta un arco che congiunge due nodi:

$$[[a,b],[a,h],[b,c],[b,f],[b,g],[b,h],[c,d],[c,f],[d,e],[d,f],[e,f],[f,g],[g,h]]$$

Il grafo riportato nella figura 5.14 può essere rappresentato anche mediante la sequenza delle etichette dei nodi e da una matrice binaria che rappresenta i collegamenti fra i nodi; le righe e colonne di questa matrice corrispondono ai nodi, come stabilito nella sequenza delle etichette dei nodi (figura 5.15).

0	1	2	3	4	5	6	7
a	*b*	*c*	*d*	*e*	*f*	*g*	*h*

	0	1	2	3	4	5	6	7
0		1						1
1	1		1			1	1	1
2		1		1		1		
3			1		1	1		
4				1		1		
5		1	1	1	1		1	
6		1				1		1
7	1	1					1	

Figura 5.15: Matrice che rappresenta il grafo riportato nella figura 5.14 (per semplicità di lettura non sono indicati i valori uguali a 0).

Le matrici binarie che rappresentano grafi composti da molti vertici risultano spesso costituite da un'alta percentuale di valori uguali a 0. Queste matrici, dette *matrici sparse*, vengono rappresentate, in modalità compressa, registrando solamente i valori non nulli; nel caso in cui la matrice sparsa sia binaria, è sufficiente registrare le coordinate delle posizioni degli elementi non nulli all'interno della matrice; nel caso ancora più particolare in cui la matrice sia simmetrica è sufficiente registrare solamente gli elementi presenti in uno dei due triangoli in cui la matrice è suddivisa dalla diagonale secondaria.

Osservazione. La rappresentazione di un oggetto costituisce solo una parte della fase di implementazione in quanto, per poter utilizzare un oggetto, è necessario applicare su di esso delle operazioni. La fase di rappresentazione viene pertanto completata con la fase di definizione delle operazioni che agiscono sulla struttura dati sottostante che rappresenta l'oggetto; queste strutture non vengono percepite dall'utente, ma vengono protetta e nascoste: l'oggetto risulta caratterizzato da un insieme di operazioni che definiscono un livello semantico che caratterizza l'interfaccia e le funzionalità da esso espletate.

ESERCIZI

5.1 Dato uno scontrino fiscale come descritto nell'esempio 5.3.1, contenente un numero generico di prodotti acquistati, calcolare il totale dello scontrino.

5.2 Descrivere una struttura adatta a registrare un *insieme di scontrini*. Calcolare il totale di un insieme di scontrini.

5.3 Studiare il formato XML per la descrizione dei dati. Descrivere secondo il formato XML i dati riportati nello scontrino fiscale descritto nell'esempio 5.3.1.

5.4 L'insieme dei numeri della forma $a + b\sqrt{2}$, $a, b \in \mathbb{Z}$, può essere rappresentato mediante la coppia $[a, b]$. Tale insieme è chiuso rispetto alle quattro operazioni aritmetiche di base. Definire delle operazioni sulle coppie in modo che costituiscano le 4 operazioni sui numeri della forma considerata.

5.5 Rappresentare l'espressione

$$(17 - 4) * (25 - 6 * 3)$$

mediante una struttura in cui le operazioni binarie sono espresse mediante una terna prefissa. Valutare una generica espressione.

5.6 Rappresentare mediante una struttura l'espressione algebrica

$$\frac{1 - a}{b + 2c}$$

5.7 Descrivere mediante una struttura il seguente sistema lineare:

$$\begin{cases} 3x - 2y = 7 \\ 4x + 7y = 12 \end{cases}$$

Scrivere l'intestazione di una funzione per risolvere un generico sistema lineare di ordine 2.

5.8 Un triangolo del piano euclideo è rappresentato mediante una terna di valori corrispondente alle lunghezze dei suoi tre lati. Stabilire se due triangoli sono isometrici.

5.9 Descrivere una struttura per rappresentare un rettangolo immerso nel piano cartesiano. Con riferimento alla struttura definita, determinare l'area di un rettangolo.

5.10 Rappresentare mediante delle strutture le seguenti classi di oggetti del piano cartesiano:

1. triangolo
2. circonferenza
3. poligono

Per ciascuna classe di figure, determinare il perimetro e l'area in base alla rappresentazione proposta.

5.11 Rappresentare mediante delle strutture le seguenti classi di oggetti del piano cartesiano:

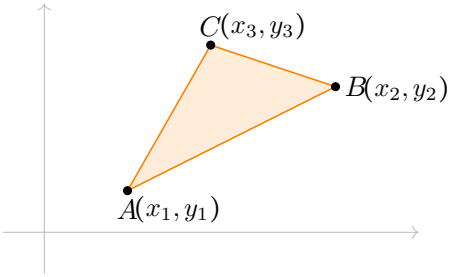
1. punto
2. segmento
3. retta
4. semipiano

5.12 Rappresentare nel piano cartesiano segmenti verticali o orizzontali. Determinare la lunghezza di un segmento. Stabilire se due segmenti si intersecano.

5.13 Descrivere una struttura adatta a rappresentare una *circonferenza* del piano cartesiano. Nell'ipotesi che la struttura venga referenziata da un identificatore c, modificare la circonferenza in modo da traslarla con centro nell'origine degli assi. Stabilire se una data circonferenza interseca gli assi cartesiani.

5.14 Descrivere una struttura adatta a rappresentare un *segmento* del piano cartesiano. Nell'ipotesi che la struttura venga referenziata da un identificatore s, modificare il segmento in modo da traslarlo con centro nell'origine degli assi. Stabilire se un dato segmento interseca gli assi cartesiani.

5.15 Rappresentare mediante una struttura un triangolo del piano cartesiano, come descritto nella seguente figura:



Calcolare l'area di un triangolo.

5.16 Rappresentare un generico poligono del piano cartesiano i cui vertici sono labellati, ossia individuati univocamente mediante un nome identificativo. Con riferimento alla struttura dati scelta, determinare il perimetro e l'area di un poligono.

5.17 Definire dei tipi di dato per rappresentare i punti ed i cerchi del piano cartesiano. Scrivere un sottoprogramma per calcolare la distanza fra due punti. Scrivere un sottoprogramma per decidere se un dato punto si trova all'interno di un dato cerchio. Scrivere una porzione di programma che richiami i sottoprogrammi.

5.18 Data una sequenza $[a_0, a_1, \ldots, a_n]$ che rappresenta una frazione continua, determinarne il valore.

5.19 Rappresentare le *date* dell'anno composte da giorno, mese, anno. Decidere se una data precede un'altra. Determinare i giorni che intercorrono tra due date.

5.20 Rappresentare i *numeri decimali periodici*; ad esempio $4.57\overline{143}$. Convertire un numero così rappresentato in frazione rappresentata mediante la coppia $[numeratore, denominatore]$.

5.21 Data una struttura formata da una sequenza di coppie (n, d) che rappresentano delle frazioni, determinare la frazione che si ottiene moltiplicando fra loro tutte le frazioni della sequenza.

5.22 Determinare la sequenza della rappresentazione in frazione continua di una data frazione. Data la sequenza della rappresentazione in frazione continua di una frazione, determinare la corrispondente frazione.

5.23 Rappresentare delle scatole a forma di parallelepipedo. Con riferimento a questa rappresentazione

1. determinare la superficie di una scatola
2. determinare il volume di una scatola
3. determinare la massima superficie d'appoggio di una scatola
4. decidere se due scatole hanno la stessa forma
5. decidere se una data scatola può contenere al suo interno un'altra data scatola

5.24 Rappresentare mediante delle strutture le seguenti configurazioni di oggetti:

1. gioco della *dama*
2. gioco degli *scacchi*
3. gioco del *tris* (*tic-tac-toe*)
4. gioco del 15
5. gioco del *cubo di Rubik*

Con riferimento a questi giochi, rappresentare una *mossa* ed una *partita*.

5.25 Una *matrice sparsa* è una matrice numerica avente solo pochi elementi diversi da 0, come ad esempio quella di seguito riportata: Per risparmiare memoria,

2	0	0	7	0	0
0	0	0	0	0	0
0	8	0	0	0	0
0	3	0	0	4	0

una matrice sparsa viene rappresentata registrando solamente gli elementi diversi da 0. Proporre una possibile rappresentazione della matrice sopra riportata. Sulla rappresentazione descritta, realizzare le seguenti due tradizionali operazioni sulle matrici:

$$\begin{array}{rcl} a[i][j] & : & \text{accesso all'elemento dell'}i\text{-esima riga e }j\text{-esima colonna} \\ a[i][j] \leftarrow x & : & \text{assegnazione} \end{array}$$

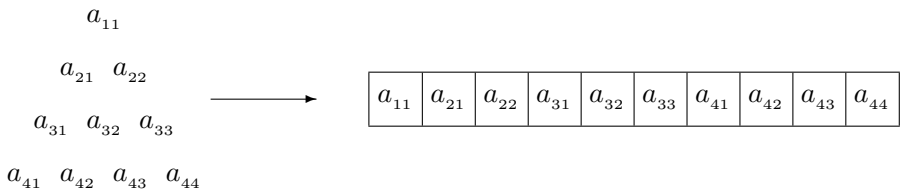
5.26 Descrivere una struttura dati adeguata a rappresentare un generico polinomio in una indeterminata (ad esempio, $4x^3-5x^2+7$). Con riferimento alla struttura dati scelta, realizzare le operazioni di addizione, sottrazione, moltiplicazione e divisione fra polinomi, operazioni per il calcolo di derivate, integrali indefiniti ed integrali definiti sui polinomi.

5.27 Rappresentare mediante delle strutture le seguenti classi di oggetti:

1. insiemi di sequenze di elementi
2. sequenze di insiemi di elementi

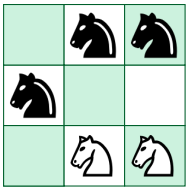
5.28 Rappresentare mediante una struttura una mappa di una metropolitana costituita da varie linee che connettono le varie stazioni, analogamente a quanto descritto nella nella figura 5.1.

5.29 Un triangolo t di numeri composto da k righe viene memorizzato in una sequenza a, secondo lo schema indicato nella figura che segue:



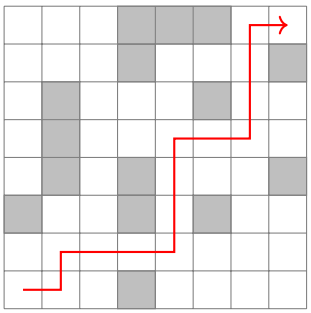
1. Determinare, in funzione del numero k di righe del triangolo t, la dimensione n della sequenza a che lo rappresenta.
2. Definire una funzione $p(i,j)$ che individui la posizione dell'elemento a_{ij}, ossia tale che $a[p(i,j)] = a_{ij}$ fornisca il valore dell'i-esima riga e j-esima colonna del triangolo t.
3. Determinare la massima somma degli elementi partendo la primo elemento e muovendosi verso il basso, raggiungendo un elemento adiacente (ad esempio lungo il percorso $a_{11} \to a_{21} \to a_{32} \to a_{43}$).
4. Determinare tutti i percorsi corrispondenti alla somma massima come descritto al punto precedente.

5.30 Rappresentare mediante una struttura la seguente mini scacchiera con soli cavalli.



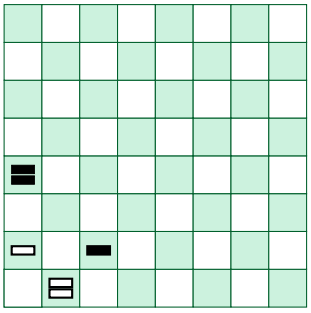
Accedendo alla struttura con modalità diretta ad indice, muovere il cavallo bianco.

5.31 Un *ambiente* è costituito da un reticolo quadrato i cui elementi sono *spazi* o *muri*. Gli elementi estremi in basso a sinistra (*partenza*) ed in alto a destra (*arrivo*) sono spazi. Un *percorso* consiste in una sequenza di *passi* dalla partenza all'arrivo, con il vincolo che ad ogni passo ci si deve muovere verso destra oppure verso l'alto. Nella figura è descritto un ambiente ed un corrispondente percorso.



1. Descrivere una struttura dati adatta a rappresentare un ambiente ed un percorso.
2. Determinare il numero di tutti percorsi.
3. Determinare un percorso.
4. Determinare tutti i percorsi.

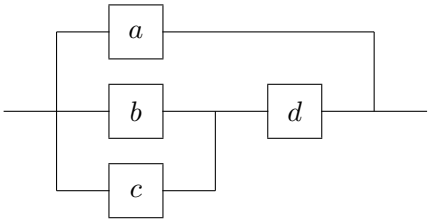
5.32 Descrivere una struttura dati adatta a rappresentare una configurazione del gioco della dama: i pezzi (pedine e dame), di due colori (bianco e nero), sono posizionati nei riquadri scuri e si possono muovere solo sui riquadri adiacenti in diagonale. Descrivere la struttura con riferimento alla seguente configurazione:



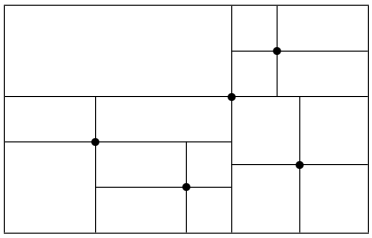
Indicando con a la struttura dati che rappresenta la configurazione riportata in figura, ed ipotizzando di accedere alle componenti con la notazione ad indice $a[i]$, scrivere le istruzioni di modifica della struttura in conseguenza della presa della pedina nera da parte della della dama bianca.

5.33 Rappresentare mediante una struttura un generico circuito elettrico con componenti R, L, C.

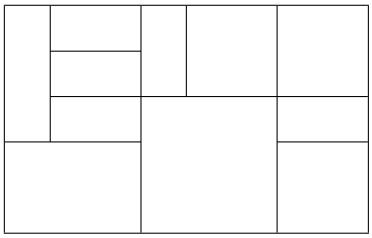
5.34 Rappresentare mediante una struttura il sistema a blocchi riportato nella figura che segue.



5.35 Un *quad-tree* è una partizione di un rettangolo ottenuta mediante le seguenti operazioni di base: si parte da un rettangolo; si procede selezionando un punto interno a qualche rettangolo e si disegnano due segmenti di divisione del dato rettangolo, paralleli ai lati dei rettangoli già costruiti. Rappresentare un quad-tree mediante una struttura.



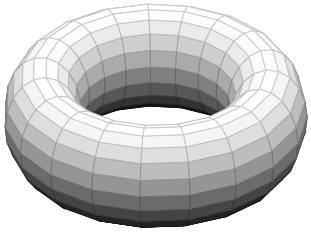
5.36 Un rettangolo può essere suddiviso con il seguente procedimento: si parte da un rettangolo; si procede poi applicando in successione una delle seguenti operazioni: si suddivide un rettangolo con una linea orizzontale o verticale, parallela ai lati.



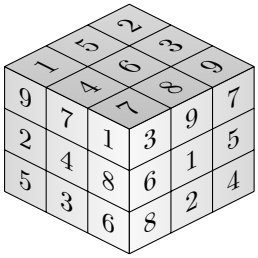
Rappresentare mediante una struttura un rettangolo partizionato nelle seguenti due situazioni:

- serve rappresentare solo la *topologia* della suddivisione, indipendentemente dalla grandezza dei rettangoli; in altri termini serve catturare solo lo scheletro, tenendo conto che le linee divisorie interne possono essere trascinate parallelamente a se stesse
- serve rappresentare la geometria della suddivisione, rappresentando anche la dimensione dei vari rettangoli

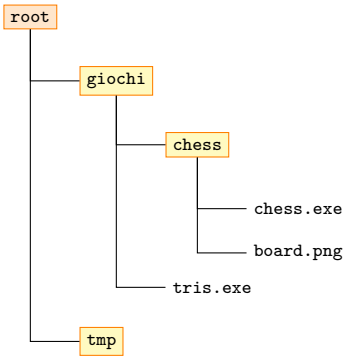
5.37 Rappresentare una superficie toroidale suddivisa in celle rettangolari (vedi figura che segue) contenenti ciascuna un numero; si trascuri la dimensione delle celle e si rappresenti solamente la topologia della superficie.



5.38 Rappresentare un dado a 6 facce ciascuna delle quali è suddivisa in 9 caselle ciascuna contenente un numero, come si vede nella figura che segue.



5.39 Rappresentare la seguente porzione di file system:



Accedendo alla struttura con modalità diretta ad indice, rinominare il file `board.png` in `board.jpg`.

6

ORGANIZZARE

Ogni genere posto ad un nodo alto dell'albero comprende le specie che ne dipendono, ogni specie subordinata a un genere è un genere per la specie che le è subordinata, sino all'estremità inferiore dell'albero, dove sono collocate le specie specialissime *o sostanze seconde, come ad esempio* uomo*.*

U. Eco, *Dall'albero al labirinto*

Organizzare significa dare un assetto organico e funzionale a degli elementi, ossia una *struttura.* In questo capitolo ci si riferirà al caso particolare in cui gli elementi sono dei valori atomici (numeri, caratteri, stringhe) e la modalità di organizzazione è data dal meccanismo di aggregazione mediante struttura [...] già esaminato nei precedenti capitoli. Verranno analizzate due fondamentali modalità di organizzazione: la strutturazione gerarchica ad *albero*, tipica di molteplici situazioni della vita reale e la strutturazione a *grafo*, in cui gli elementi sono associati gli uni agli altri a coppie.

Alberi e grafi sono organizzazioni che possono essere rappresentate mediante strutture organizzative elementari (insiemi, sequenze, matrici, ...) ma a loro volta costituiscono delle forme di rappresentazione astratte di situazioni reali.

6.1 Alberi

Gli alberi costituiscono una modalità di organizzazione dei dati particolarmente importante, per il loro largo uso che ne viene fatto in informatica, per la ricchezza e profondità dei problemi che permettono di descrivere e trattare e per l'eleganza delle soluzioni che suggeriscono.

DEFINIZIONE 3. Un *albero* (*radicato*) è un insieme vuoto oppure un insieme non vuoto di elementi detti *nodi* in cui esiste un nodo speciale detto *radice* e gli altri nodi sono ripartiti in una sequenza ordinata di $n \geq 0$ sottoinsiemi $T_1, \ldots, T_n$, detti *sottoalberi* o *figli* della radice, ciascuno dei quali è un albero. Una sequenza di alberi viene detta *foresta*. Si dicono *foglie* di un albero i nodi dell'albero che non hanno sottoalberi; i nodi che non sono foglie sono detti nodi interni. Si dicono *discendenti* di un nodo tutti gli elementi che appartengono ai sottoalberi del nodo stesso. Un nodo dicesi *padre* dei nodi (*figli*) che costituiscono la radice dei suoi sottoalberi. Tali nomenclature derivate per analogia con i gradi di parentela possono essere estese. Ad esempio, due nodi figli di uno stesso padre vengono detti *fratelli*. Dicesi *cammino* da un nodo p ad un nodo discendente q una successione di nodi $(n_1, \ldots, n_k)$ tali che $n_1 = p$, $n_k = q$, n_i è padre di n_{i+1}, per ogni $i = 1, k-1$. Un *arco* è il cammino che unisce due nodi adiacenti (padre-figlio). La *lunghezza* di un cammino $(n_1, \ldots, n_k)$ è rappresentata dal numero di archi che uniscono il primo nodo all'ultimo nodo, ossia $k-1$. La *profondità* di un nodo è uguale alla lunghezza del cammino dalla radice al nodo stesso. L'*altezza* di un albero è pari alla massima propondità dei suoi nodi foglia. Un *livello* di un albero è costituito dai nodi aventi la stessa profondità. Il numero massimo dei figli di uno stesso nodo viene detto *grado* (dell'albero). Un albero di grado k dicesi *pieno* se ogni nodo interno ha esattamente k figli; dicesi *completo* se è pieno e tutte le foglie hanno la stessa profondità; dicesi *quasi completo* se è completo fino al penultimo livello ed i nodi dell'ultimo livello sono addossati a sinistra. □

Esempio 6.1.1 - Il termine albero è mutuato dalla botanica, anche se usualmente, per questioni di praticità grafica, si usa rappresentare gli alberi con la radice in alto, come si vede nella figura 6.1.

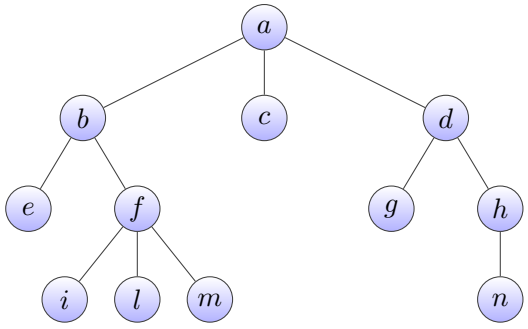


Figura 6.1: Albero di grado 3, altezza 3, costituito da 12 nodi (di cui 5 sono nodi interni e 7 sono foglie), strutturati in 4 livelli.

Esempio 6.1.2 - Nella figura 6.2 sono riportate alcune topologie di alberi descritti nella definizione 3.

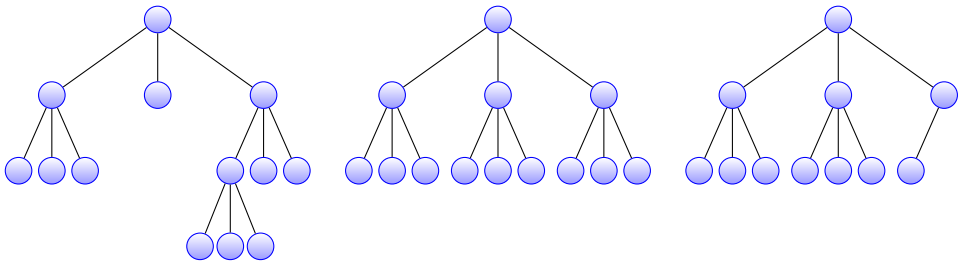


Figura 6.2: Esempi di alberi ternari: *pieno* (a), *completo* (b), *quasi completo* (c).

Osservazione. Gli alberi vengono solitamente definiti in modo ricorsivo, come descritto nella precedente definizione; ciò suggerisce in molti casi la struttura dati (ricorsiva) per la loro rappresentazione e la forma degli algoritmi (ricorsivi) che si adottano per trattare queste strutture.

Rappresentazioni degli alberi

Un metodo per rappresentare un albero T, vuoto oppure costituito dalla sola radice r oppure costituito dalla radice r e dalla sequenza di sottoalberi $[T_1, \ldots, T_n]$, consiste nella rappresentazione a struttura, ossia:

$$rap(T) = \begin{cases} [\,] & \text{se } T \text{ è vuoto} \\ [r] & \text{se } T \text{ è costituito dalla sola radice } r \\ [r, rap(T_1), \ldots, rap(T_n)] & \text{altrimenti} \end{cases}$$

Graficamente un albero non vuoto composto dalla radice r e dalla sequenza di sottoalberi $[T_1, \ldots, T_n]$ può essere rappresentato come descritto nella figura 6.3.

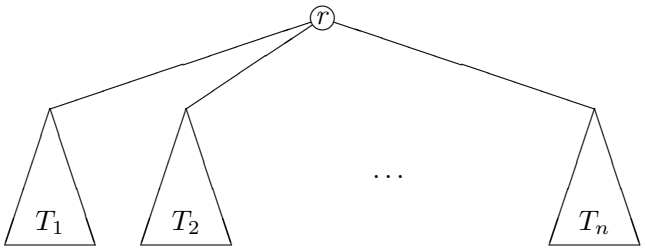
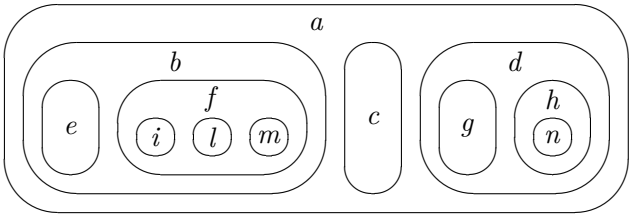


Figura 6.3: Notazione grafica di un albero composto dalla radice r e dalla sequenza di sottoalberi $[T_1, \ldots, T_n]$.

Esempio 6.1.3 - L'albero descritto nella figura 6.1 viene rappresentato mediante la struttura

$$[a, [b, [e], [f, [i], [l], [m]\,]\,], [c], [d, [g], [h, [n]\,]\,]\,]$$

Una rappresentazione alternativa consiste nell'utilizzo dei diagrammi di Venn:



Un'altra rappresentazione mediante parentesi adotta la sintassi del linguaggio Lisp, come descritto come segue:

$$(a(b(e)(f(i)(l)(m)))(c)(d(g)(h(n))))$$

Un albero può essere descritto anche mediante una scrittura indentata dei nodi, come illustrato nello schema che segue:

a
$-\ b$
$-\ -\ e$
$-\ -\ f$
$-\ -\ -\ i$
$-\ -\ -\ l$
$-\ -\ -\ m$
$-\ c$
$-\ d$
$-\ -\ g$
$-\ -\ h$
$-\ -\ -\ n$

Questa notazione risulta utile quando si voglia rappresentare un albero in modo testuale ed in modo che risulti facilmente decifrabile la relazione di gerarchia fra i nodi stessi. La rappresentazione indentata suggerisce un'altra possibilità di rappresentazione: un albero viene rappresentato mediante una sequenza di coppie $[x, n]$ dove x denota l'elemento ed n il grado di indentazione. Questa modalità di rappresentazione è descritta a seguire:

$$[\,[a, 0], [b, 1], [e, 2], [f, 2], [i, 3], [l, 3], [m, 3], [c, 1], [d, 1], [g, 2], [h, 2], [n, 3]\,]$$

Metodi di visita degli alberi

Visitare un albero significa attraversarlo e produrre una sequenza piatta degli elementi memorizzati nei nodi dell'albero. Si possono adottare diverse strategie di visita:

- visita in ordine *anticipato*:
  1. visita la radice
  2. visita i sottoalberi in ordine anticipato
- visita in ordine *differito*:
  1. visita i sottoalberi in ordine differito
  2. visita la radice

Esempio 6.1.4 - Le visite in ordine anticipato e differito dell'albero descritto nella figura 6.1 producono rispettivamente le seguenti sequenze:

$$[a, b, c, f, i, l, m, c, d, g, h, n]$$

$$[e, i, l, m, f, b, c, g, n, h, d, a]$$

6.2 Alberi binari

Una importante e frequentemente utilizzata famiglia di alberi è costituita dagli alberi nei quali ogni nodo ha al più due figli. Questa tipologia di alberi viene più utilmente definita in modo ricorsivo in quanto favorisce l'impostazione ricorsiva tipica della loro elaborazione.

DEFINIZIONE 4. Dicesi *albero binario* un insieme vuoto oppure una terna $[r, T_L, T_R]$ in cui T_L e T_R sono alberi binari detti, rispettivamente, *figlio sinistro* e *figlio destro*.

Esempio 6.2.1 - La figura 6.4 illustra un esempio di albero binario.

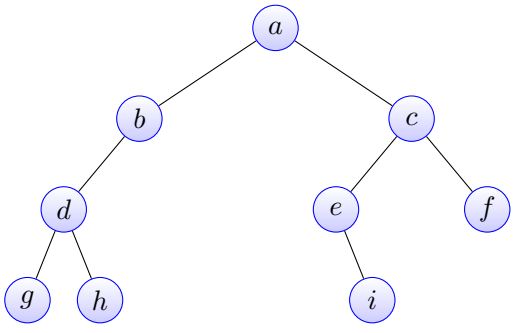


Figura 6.4: Albero binario composto da 9 nodi strutturati in 4 livelli.

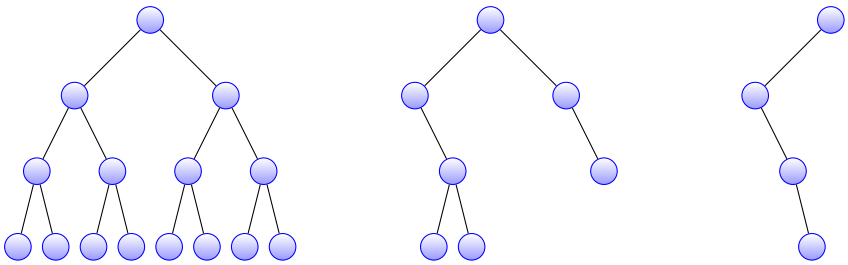
Bilanciamento di un albero binario

Una importante caratteristica strutturale degli alberi binari riguarda la forma dell'albero ed è denominata *bilanciamento*. È definita mediante le seguenti definizioni e descritta graficamente nella figura che segue.

(a) albero *completo*: tutte le foglie si trovano allo stesso livello

(b) albero *sbilanciato*: i nodi occupano più livelli di quelli necessari

(c) albero *degenere*: ogni nodo ha un solo figlio



(a) (b) (c)

Esistono specifiche relazioni funzionali fra il numero n dei nodi, l'altezza h ed il grado di bilanciamento di un albero binario; in particolare:

- in un albero binario di altezza h composto da n si ha

$$h + 1 \leq n \leq 2^{h+1} - 1$$

- in un albero binario completo composto da n di altezza h si ha

$$n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

da cui si ricava

$$h = \log_2(n + 1) - 1$$

Operazioni sugli alberi binari

La costruzione e l'elaborazione degli alberi binari viene svolta basandosi su un insieme limitato di operazioni; in notazione funzionale queste operazioni vengono espresse come segue:

- $B()$: albero binario vuoto
- $B(r)$: albero formato dalla sola radice r
- $B(r, \alpha, \beta)$: albero binario di radice r e figlio sinistro e destro α e β
- $root(\alpha)$: radice dell'albero binario α
- $left(\alpha)$: sottoalbero sinistro dell'albero binario α

- $right(\alpha)$: sottoalbero destro dell'albero binario α
- $isempty(\alpha)$: test se l'albero binario α è vuoto

Utilizzando queste operazioni si possono scrivere moltissimi algoritmi, spesso impostati in modo ricorsivo.

Esempio 6.2.2 - Il numero dei nodi di un albero binario può essere calcolato mediante il seguente algoritmo:

Algoritmo 1 - $nodi(a)$ - numero di nodi di un albero binario

Input: albero binario a
Output: `TRUE` se e solo se x è presente in a

```
1: if isempty(a) then
2:    return 0
3: else
4:    return 1 + nodi(left(a)) + nodi(right(a))
5: end if
```

Esempio 6.2.3 - Il numero di foglie di un albero binario può essere calcolato mediante il seguente algoritmo:

Algoritmo 2 - $foglie(a)$ - numero di foglie di un albero binario

Input: albero binario a
Output: numero di foglie di a

```
1: if isempty(a) then
2:    return 0
3: else if isempty(left(a)) ∧ isempty(right(a)) then
4:    return 1
5: else
6:    return foglie(left(a)) + foglie(right(a))
7: end if
```

Per elaborare gli alberi binari vengono definite anche delle operazioni di *inserimento*, *modifica* e *cancellazione* che permettono di inserire, modificare e cancellare singoli nodi o porzioni di albero. Queste operazioni vengono solitamente impostate in modalità procedurale ed agiscono modificando l'albero sul quale operano. Tutte queste operazioni vengono realizzate direttamente sulla specifica struttura dati utilizzata per rappresentare l'albero ed hanno una complessità computazionale pari a $O(\log n)$ nel caso di alberi bilanciati e $O(n)$ nel caso di alberi degeneri (dove n denota il numero di nodi dell'albero).

Metodi di visita degli alberi binari

Un albero binario può essere visitato secondo le strategie di seguito descritte:

- visita in ordine *preordine*:
  1. visita la radice
  2. visita il figlio sinistro in preordine
  3. visita il figlio destro in preordine
- visita in ordine *inordine* o *in ordine simmetrico*:
  1. visita il figlio sinistro in inordine
  2. visita la radice
  3. visita il figlio destro in inordine
- visita in *postordine*:
  1. visita il figlio sinistro in postordine
  2. visita il figlio destro in postordine
  3. visita la radice

Esempio 6.2.4 - Visitando l'albero riportato nella figura 6.4 mediante una visita in preordine, inordine e postordine si ottengono, rispettivamente, le seguenti sequenze:

$$[a, b, d, g, h, c, e, i, f]$$

$$[g, d, h, b, a, e, i, c, f]$$

$$[g, h, d, b, i, e, f, c, a]$$

Rappresentazioni di alberi binari

Gli alberi binari, essendo dei casi particolari di liste non lineari, possono essere rappresentati derivando il metodo di rappresentazione delle liste non lineari. Poiché questa rappresentazione è poco efficiente, viene usualmente scelta una rappresentazione specifica (in tale modo si perde però, in un linguaggio orientato agli oggetti, la possibilità di derivare la classe degli alberi binari dalla classe delle liste lineari).

Un albero binario vuoto può essere rappresentato graficamente da un simbolo convenzionale, ad esempio •. Un albero binario non vuoto $T = (r, T_L, T_R)$ può essere rappresentato graficamente come descritto nella figura 6.5.

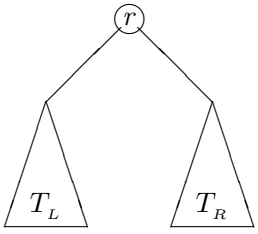


Figura 6.5: Rappresentazione ricorsiva di un albero binario.

Questo schema di rappresentazione può essere immediatamente ricondotto ad una rappresentazione mediante una struttura ricorsiva:

$$[r, rap(T_L), rap(T_R)]$$

chiudendo la ricorsione rappresentando l'albero vuoto con [ ].

Esempio 6.2.5 - L'albero binario riportato nella figura 6.4 viene rappresentato mediante la seguente struttura:

$$[a,[b,[d,[g,[\,],[\,]],[h,[\,],[\,]],[\,]],[c,[e,[\,],[i,[\,],[\,]]],[f,[\,],[\,]]]]$$

Un albero binario può essere efficientemente rappresentato mediante una sequenza come segue: l'albero viene completato in modo da ottenere un albero completo con dei nodi contenenti il valore nullo in corrispondenza dei nodi mancanti; il valore nullo, indicato con •, è costituito da un valore convenzionale non appartenente all'insieme dei valori dei nodi; ad esempio 0 se i nodi sono tutti positivi; successivamente si memorizzano per livelli tutti i nodi dell'albero completo così costruito. I nodi dell'albero completo ottenuto vengono inseriti nella sequenza, a livelli, partendo dal livello della radice.

La rappresentazione sequenziale di un albero di altezza h richiede una sequenza di al più $2^{h+1} - 1$ elementi. Serve inoltre un particolare valore, non appartenente al tipo di base E, per rappresentare l'*elemento nullo*, ossia non esistente.

Esempio 6.2.6 - La figura 6.6 descrive la rappresentazione sequenziale per livelli dell'albero binario riportato nella figura 6.4.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	c	d	•	e	f	g	h	•	•	•	i	•	•

Figura 6.6: Rappresentazione sequenziale dell'albero binario riportato nella figura 6.4

Si ricava facilmente che, dato un elemento di posizione k nella sequenza a che rappresenta l'albero, gli elementi che costituiscono la radice del figlio sinistro, del figlio destro e del padre (se $k > 0$) sono individuati mediante le seguenti relazioni:

$$\begin{aligned} root(left(a[k])) &= a[2k+1] \\ root(right(a[k])) &= a[2k+2] \\ root(father(a[k])) &= a[(k-1)\,\mathsf{div}\,2] \end{aligned}$$

6.3 Alberi binari di ricerca

Consideriamo ora degli alberi binari in cui i nodi contengono valori appartenenti ad un insieme sul quale è definita una relazione d'ordine, che denotiamo col tradizionale operatore infisso $\leq$. In questo contesto viene data la seguente definizione.

DEFINIZIONE 5. Un *albero binario di ricerca* (*ABR*) è un albero binario in cui ogni nodo è maggiore o uguale dei nodi del sottoalbero sinistro e minore o uguale dei nodi del sottoalbero destro.

Notiamo che il quantificatore *ogni* che compare nella predente definizione induce un vincolo d'ordine che coinvolge tutti i nodi dell'albero, a causa della proprietà transitiva di una relazione d'ordine. Questa considerazione porta alla seguente equivalente definizione.

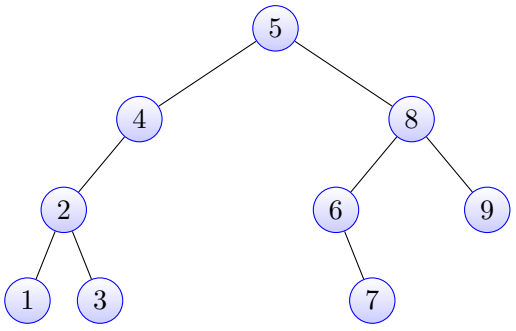
DEFINIZIONE 6. Un *albero binario di ricerca* (*ABR*) è un albero binario in cui per ogni generico nodo x

- il sottoalbero sinistro di x contiene tutti valori minori o uguali di x
- il sottoalbero destro di x contiene tutti valori maggiori o uguali di x
- entrambi i sottoalberi sinistro e destro del nodo x sono alberi binari di ricerca

Per l'elaborazione di un ABR risulta più utile la seguente definizione ricorsiva:

DEFINIZIONE 7. Un *albero binario di ricerca* (*ABR*) è un albero binario vuoto oppure è un albero binario $[r, T_L, T_R]$ dove T_L è un albero binario vuoto oppure è un albero binario di ricerca avente la radice minore o uguale di r e T_R è un albero vuoto oppure è un albero binario di ricerca avente la radice maggiore o uguale di r.

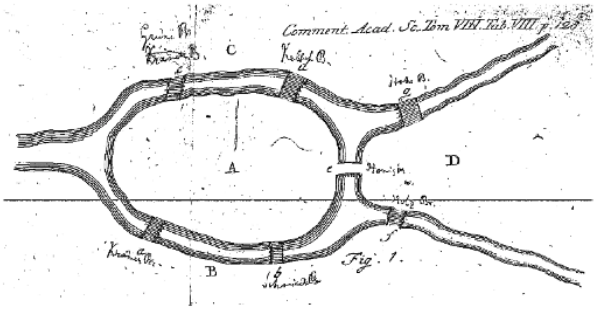
Esempio 6.3.1 - Nella figura che segue è riportato un esempio di albero binario di ricerca.



Gli *ABR* vengono solitamente impiegati come struttura dati sulla quale eseguire efficientemente delle ricerche.

6.4 Grafi

Agli inizi del XVIII secolo gli abitanti di Königsberg, una città della Prussia orientale attualmente situata in territorio russo e nota come Kaliningrad, si chiedevano se fosse possibile fare una passeggiata attraversando una sola volta i sette ponti della cittadina e ritornare al punto di partenza, come schematizzato nella figura che segue. Il problema venne risolto dal grande matematico svizzero Leonard Euler che nel 1736 pubblicò un lavoro da cui si fa storicamente risalire la teoria dei grafi.



I grafi sono strutture che rivestono interesse per un'ampia gamma di campi applicativi. Costituiscono una struttura matematica discreta che può essere studiata dal punto di vista algebrico, indipendentemente dalle specifiche aree applicative. La teoria dei grafi costituisce un'importante parte della combinatoria; i grafi inoltre sono utilizzati in aree come topologia, teoria degli automi, funzioni speciali, geometria dei poliedri, algebre di Lie.

Intuitivamente un *grafo* è una rete di nodi (rappresentati mediante dei punti) e di collegamenti fra di essi (rappresentati mediante delle linee di congiunzione). I grafi vengono utilizzati per descrivere modelli di sistemi e processi studiati nell'informatica (programmi, circuiti, reti di computer, mappe di siti), nell'ingegneria (sistemi fluviali, reti stradali, trasporti), nella chimica, nella biologia molecolare, nella ricerca operativa, nella organizzazione aziendale, nella linguistica strutturale, nella storia (alberi genealogici, filologia dei testi). I seguenti esempi danno un'idea di alcuni campi applicativi dei grafi:

- una *rete stradale* può essere rappresentata mediante un grafo in cui i nodi sono le città e le linee rappresentano i collegamenti fra le città
- le *reti logiche* sono rappresentabili mediante dei grafi in cui i nodi sono le porte logiche *and*, *or* e *not* e le linee corrispondono ai collegamenti fra le porte
- la struttura di un *ipertesto* è rappresentabile mediante un grafo in cui i nodi sono le pagine e le linee sono i collegamenti fra le pagine
- le *molecole* possono essere rappresentate mediante dei grafi in cui i nodi sono gli atomi che le compongono e le linee esprimono i legami fra gli atomi

I termini fondamentali della teoria dei grafi sono riportati nella seguente definizione.

DEFINIZIONE 8. Un *grafo* è costituito da un insieme di elementi detti *vertici* o *nodi* o *punti* e da un insieme di *lati* o *archi* che collegano coppie di vertici [1]. Due vertici u, v connessi da un lato vengono detti *estremi* del lato; un lato risulta identificato dalla coppia formata dai suoi estremi (u, v). Un lato avente due estremi coincidenti si dice *cappio*. Un lato risulta individuato dalla coppia non ordinata dei vertici che collega e graficamente viene denotato mediante una linea che congiunge i due vertici; se la coppia viene considerata ordinata si parla di *arco* e viene denotato graficamente mediante una linea orientata dal primo al secondo vertice. Un grafo composto solo da archi è detto *grafo orientato* (o *diretto*) o *digrafo*. L'*ordine* di un grafo è il numero dei suoi vertici mentre la sua *grandezza* è il numero dei suoi lati.

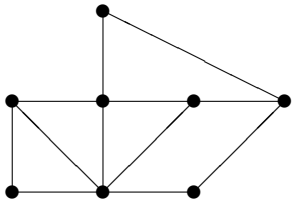


Figura 6.7: Grafo composto da 8 vertici e 12 lati.

Un grafo $\mathcal{G}$ di ordine n e grandezza m viene denotato con $\mathcal{G}(n, m)$. Un *grafo completo* di ordine n, denotato con K_n è un grafo non orientato con n vertici in cui ogni coppia di vertici è connesso da un lato.

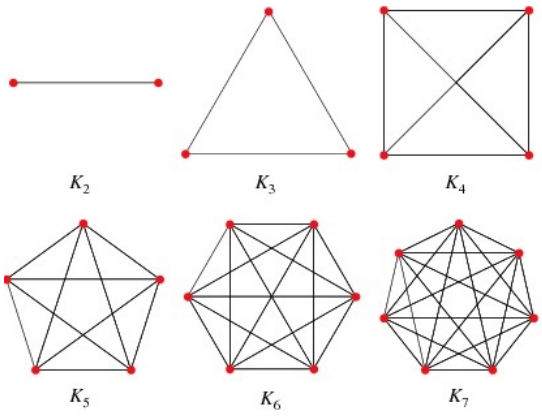


Figura 6.8: Alcuni grafi della famiglia K_n.

[1] Una coppia di vertici può essere unita da più lati o archi; in questi casi si parla di lati (archi) *multipli* o *multilati* (*multiarchi*) ed il grafo viene detto *multigrafo*. In caso contrario si parla di *grafo semplice*. In questo capitolo si considerano solo grafi semplici.

6.5 Operazioni sui grafi

Per individuare e definire le operazioni sui grafi risulta utile riferirsi alla seguente definizione formale.

DEFINIZIONE 9. Un *grafo* è una coppia $\mathcal{G} = (V, E)$ dove

- V è un insieme di elementi detti *vertici*
- E è un insieme di elementi detti *lati* costituiti da coppie di elementi di V, ossia $E \subseteq V \times V$

Il grafo $\mathcal{G} = (\varnothing, \varnothing)$, privo di vertici e di lati, è detto *grafo nullo*.

Le operazioni fondamentali che si possono eseguire su un grafo sono sinteticamente descritte nel seguente elenco:

- creare un grafo vuoto
- aggiungere un vertice u
- aggiungere un lato a congiungente due vertici u e v
- eliminare un dato vertice u (ed i lati ad esso collegati)
- eliminare il lato congiungente due dati vertici u e v
- ottenere la lista dei lati uscenti da un dato vertice u
- ottenere la lista dei lati congiungenti un dato vertice u

6.6 Cammini e percorsi

Un *cammino* da un vertice u ad un vertice v è una successione di vertici distinti $(v_1, \ldots, v_n)$ tali che $v_1 = u$, $v_n = v$ ed inoltre per ogni $i = 1, \ldots, n-1$, (v_i, v_{i+1}) è un lato del grafo. Se $v_1 = v_n$ il cammino viene detto *circuito* o *ciclo*. Il numero dei lati del cammino è detto *lunghezza* del cammino. Se si ammette che i vertici possano essere ripetuti il cammino viene detto *percorso*. Se il cammino passa per tutti i lati una sola volta viene detto *euleriano*. Un grafo contenente almeno un cammino euleriano viene detto *grafo euleriano*. Un circuito è detto *hamiltoniano* se passa una sola volta per tutti i vertici del grafo (escluso il primo). Un grafo contenente almeno un circuito hamiltoniano viene detto *grafo hamiltoniano*. Un grafo si dice *connesso* se esiste un cammino congiungente ogni coppia dei suoi vertici. Nel caso di grafi orientati un cammino viene detto *catena* e risulta composta da una successione di vertici tali che due vertici contigui siano connessi da un arco.

TEOREMA 1. Un grafo $\mathcal{G}$ connesso è euleriano se e solo se ogni vertice di $\mathcal{G}$ è pari.

Il risultato espresso dal teorema precedente fornisce la soluzione (in negativo) al famoso problema dei ponti di Königsberg in quanto, nel grafo che rappresenta il problema, esistono vertici di grado dispari (tutti i vertici hanno grado dispari).

6.7 Tipologie di grafi

In un grafo si possono memorizzare informazioni in diversi modi:

- orientando il lato, ottenendo un *grafo orientato*
- memorizzando delle informazioni nei nodi; tali informazioni vengono dette *label* o *etichette* ed il grafo che si ottiene viene detto *grafo labellato*
- memorizzando delle informazioni nei lati; tale informazioni vengono dette *pesi* e il grafo che si ottiene viene detto *grafo pesato*

La figura 6.9 descrive un grafo orientato, pesato e labellato.

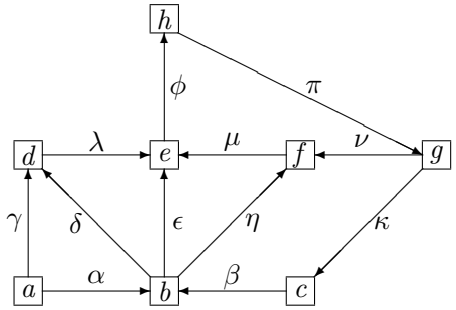


Figura 6.9: Grafo orientato, pesato e labellato.

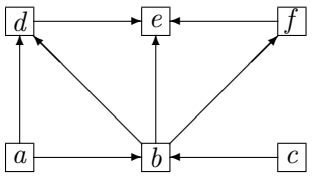
6.8 Rappresentazioni dei grafi

Un grafo viene rappresentato mediante delle strutture dati elementari. Vengono abitualmente utilizzate le strutture predisposte dagli usuali linguaggi di programmazione: array, liste, matrici. La particolare scelta dipende dalla tipologia del grafo e dalle operazioni che si intendono realizzare e, quindi, dal particolare problema da risolvere. A seguire sono descritte alcune rappresentazioni dei grafi:

- rappresentazione mediante *insiemi*:

  basandosi direttamente sulla definizione 9, un grafo può essere rappresentato mediante l'insieme dei vertici e dei lati o degli archi.

  Esempio 6.8.1 - Il grafo orientato e labellato

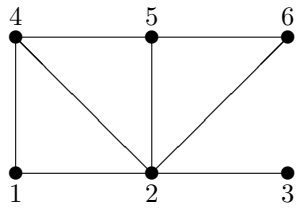
  viene rappresentato mediante la seguente coppia di insiemi:

  $$(\{a,b,c,d,e,f\},\{(a,b),(a,d),(b,d),(b,e),(b,f),(c,b),(d,e),(f,e)\})$$

- rappresentazione mediante *matrice di adiacenza*:

  un grafo di n nodi può essere rappresentato mediante una matrice binaria, identificando ciascun nodo mediante un valore naturale progressivo, $1, 2, 3, \ldots, n$; la matrice risulta composta dai valori 0 ed 1 per indicare, rispettivamente, la non presenza e la presenza dei collegamenti fra i vari nodi.

  Esempio 6.8.2 - Il grafo



  viene rappresentato mediante la matrice binaria

0	1	0	1	0	0
1	0	1	1	1	1
0	1	0	0	0	0
1	1	0	0	1	0
0	1	0	1	0	1
0	1	0	0	1	0

  La rappresentazione di un grafo mediante una matrice di adiacenza è utilizzabile nel caso di grafi orientati e non orientati; in quest'ultimo caso la matrice risulta simmetrica. Una matrice di adiacenza è utilizzabile anche nel caso di grafi pesati nel quale caso al posto dell'1 viene messo il valore del peso dell'arco corrispondente. Spesso, nella matrice di adiacenza vengono indicati solo i valori non nulli e vengano lasciati non definiti i valori corrispondenti allo 0. Nel caso di un grafo labellato, alla matrice binaria bisogna aggiungere un array dove sono registrati le informazioni dei vertici.

- rappresentazione mediante *lista di adiacenza*:

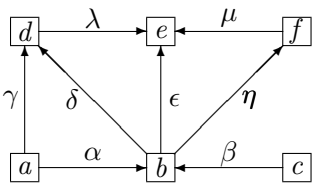
  la rappresentazione mediante una lista di adiacenza si basa sull'idea di rappresentare ciascun nodo con ad esso associata una lista i cui elementi rappresentano gli archi uscenti dal nodo stesso; l'elemento corrispondente ad un generico nodo u ha la struttura

$$(u, \text{insieme dei lati uscenti da } u)$$

  Esempio 6.8.3 - Il grafo orientato, labellato e pesato

  viene rappresentato mediante la struttura composta

$$\{(a, \{(b, \alpha), (d, \gamma)\}), (b, \{(d, \delta), (e, \epsilon), (f, \eta)\}), (c, \{(b, \beta)\}),$$



$$(d, \{(e, \lambda)\}), (f, \{(e, \mu)\})\}$$

6.9 Problemi sui grafi

Esistono molti problemi, della natura più svariata, che possono essere ricondotti, affrontati ed eventualmente risolti all'interno della teoria dei grafi. A seguire sono descritti alcuni di questi problemi.

Problema del cammino minimo

Dato un grafo orientato e pesato determinare un cammino di peso minimo da un nodo di partenza ad un nodo di arrivo, ossia un cammino per il quale è minima la somma dei pesi degli archi che lo compongono. Si tratta di un tipico problema dei trasporti. Si pensi ad esempio al problema di minimizzare i consumi per il trasporto da una città ad un'altra. L'algoritmo risolutivo più noto è attribuito a E. W. Dijkstra ed ha una complessità $O(n^2)$ dove n è il numero dei nodi.

Problema del collegamento minimo

Dato un grafo non orientato pesato che rappresenta una rete di collegamenti fra i vertici, si vuole eliminare i collegamenti superflui e mantenere solo quelli indispensabili in modo da minimizzare il costo totale. Ciò corrisponde a voler trovare un *albero di supporto minimo* di un grafo non orientato connesso. Una della applicazioni più significative è quella del progetto di una rete di comunicazioni in cui i vertici rappresentano delle città. Per questo problema sono noti algoritmi di complessità polinomiale.

Problema del commesso viaggiatore

Il *problema del commesso viaggiatore* fu considerato per la prima volta agli inizi degli anni '30 del secolo scorso, anche se, sotto altra veste, comparve all'interno della teoria dei grafi già nel diciannovesimo secolo. Il problema può essere enunciato come segue: Dato un grafo orientato e pesato, determinare un circuito hamiltoniano (ossia passante una sola volta per tutti i vertici del grafo) di costo minimo. Nella sua formulazione più caratteristica il problema si enuncia come segue: Un commesso viaggiatore deve visitare un dato numero di città. Ogni città è collegata a tutte le altre da una strada di cui si conosce la lunghezza. Determinare il percorso più breve che passa per ogni città una sola volta e ritorna alla città di partenza.

Problema dell'isomorfismo fra grafi

Dati due grafi $\mathcal{G}_1$ e $\mathcal{G}_2$ si dicono *isomorfi* (stessa forma) se sono sostanzialmente lo stesso grafo, cioè se è possibile trovare una corrispondenza biunivoca f fra

i nodi di $\mathcal{G}_1$ e $\mathcal{G}_2$ in modo tale che esiste un arco da u a v in $\mathcal{G}_1$ allora esiste un arco $f(u)$ in $\mathcal{G}_2$ e viceversa. A tutt'oggi, non si conoscono algoritmi non esponenziali rispetto al numero dei nodi; nel caso peggiore si è infatti sempre costretti a provare $n!$ permutazioni possibili dei nodi di uno dei due grafi.

Problema della planarità di un grafo

Un grafo non orientato dicesi *planare* se è possibile disegnarlo su un piano senza sovrapporre i lati. Il problema della planarità si enuncia come segue: Dato un grafo non orientato e connesso, stabilire se è planare o meno, ed eventualmente farlo vedere. Il problema ha interesse pratico nella progettazione di reti stradali, circuiti stampati, ecc. in cui si vuole evitare di sovrapporre dei collegamenti. Già Eulero aveva trovato una semplice condizione necessaria: $m \leq 3n-6$ dove n indica il numero dei nodi ed m il numero dei lati. Nel 1974 è stato scoperto un algoritmo lineare rispetto ad n che risolve questo problema.

ESERCIZI

6.1 Ricostruire graficamente l'albero binario rappresentato dalla seguente struttura:

$$[7,[4,[\,],[8,[3,[\,],[\,]],[5,[\,],[\,]]]],[9,[\,],[6,[\,],[\,]]]]$$

6.2 Il seguente array descrive un albero binario rappresentato a livelli.

5	2	7	•	3	6	8	•	•	1	•	5	•	•	9

1. Stabilire se si tratta di un albero binario di ricerca.
2. Dire come si modifica l'array se vengono eliminate tutte le foglie.

6.3 Stabilire se un dato array costituisce una rappresentazione a livelli di un albero binario.

6.4 Eliminare tutte le foglie da un albero binario rappresentato a livelli mediante un array.

6.5 Eliminare un nodo (e tutti i nodi dei suoi figli) da un albero binario rappresentato a livelli mediante un array.

6.6 Una visita in preordine ed una in inordine di un albero binario producono rispettivamente le seguenti sequenze:

$$a\ b\ c\ d\ e\ f\ g$$

$$c\ b\ e\ d\ f\ a\ g$$

Ricostruire l'albero.

6.7 Dato un albero binario di numeri:

1. determinare il numero di nodi interni
2. determinare la somma dei valori presenti in un albero binario di numeri.
3. determinare il massimo valore presente in un albero binario.
4. determinare il massimo valore delle foglie di un albero binario.
5. determinare l'insieme dei valori delle foglie di un albero binario.
6. determinare il numero di elementi distinti di un albero binario.
7. determinare l'altezza dell'albero
8. determinare la larghezza (massimo numero di nodi su uno stesso livello)
9. decidere se l'albero è completo
10. decidere se l'albero è degenere

6.8 Determinare il numero di livelli necessari per contenere n nodi di un albero binario.

6.9 Determinare il valore minimo ed il valore massimo assumibile dall'altezza di un albero binario costituito da n nodi.

6.10 Definire un *coefficiente di bilanciamento* di un albero binario che esprima il grado di bilanciamento dell'albero.

6.11 Decidere se due alberi binari sono uguali.

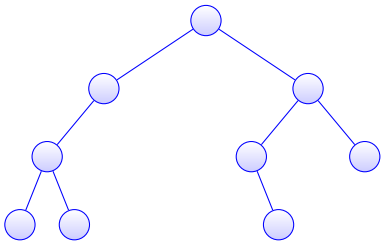
6.12 Scrivere un algoritmo per determinare la somma dei valori presenti nelle foglie di un albero binario.

6.13 Dati due alberi binari:

1. stabilire se i due alberi sono uguali
2. stabilire se i due alberi sono *isomorfi*, ossia se hanno la stessa forma, indipendentemente dai valori dei loro nodi.
3. stabilire se il primo albero è incluso nel secondo

6.14 Un'espressione aritmetica composta da operatori binari infissi espressa nella usuale notazione algebrica con parentesi, può essere rappresentata sotto forma di albero binario (detto *albero sintattico*) dove i nodi non terminali contengono gli operatori e le foglie gli operandi. Descrivere mediante un albero sintattico l'espressione $(7 + (6 - 2)) * 3$. Valutare un'espressione rappresentata mediante un albero sintattico.

6.15 Inserire i numeri $1, 2, 4, 4, 5, 7, 7, 7, 9$ nella seguente struttura di albero binario in modo da ottenere un ABR. Rappresentare l'albero a livelli mediante un array.



6.16 Costruire un ABR a partire da una sequenza di elementi, prendendo il primo elemento della sequenza come radice dell'albero ed inserendo nell'albero gli elementi dal secondo in poi senza spostare gli elementi già inseriti.

6.17 Illustrare, con un esempio, la seguente proprietà: Una visita inordine (in ordine simmetrico) di un ABR produce una sequenza ordinata dei nodi. Utilizzando questa proprietà, costruire un ABR a partire da una generica sequenza di nodi.

6.18 Ordinare una sequenza basandosi sulla proprietà che afferma che la visita preordine di un ABR genera una sequenza ordinata.

6.19 Decidere se un dato albero binario è un ABR.

6.20 Descrivere graficamente un ABR, il più bilanciato possibile, costituito dai 15 numeri $1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5$. Visitare l'albero in postordine.

6.21 Descrivere una situazione pratica in cui risulta idoneo ricorrere ad un grafo pesato, non orientato e non labellato. Fare un esempio di un tale grafo, composto da 5 vertici e 8 lati. Descrivere una struttura dati adeguata a rappresentare il grafo.

6.22 Descrivere graficamente un esempio di *grafo pesato, orientato e non labellato*, composto da 6 nodi e 9 lati. Rappresentare il grafo mediante una matrice. Descrivere come si modifica la matrice aggiungendo un nuovo nodo connesso bidirezionalmente al nodo 3.

6.23 Descrivere un grafo orientato, pesato con pesi positivi, non labellato, non connesso, composto da 6 vertici e 5 lati. Rappresentare il grafo mediante una matrice di adiacenza. Indicando con a la matrice, scrivere delle assegnazioni per connettere un vertice (a scelta) con tutti gli altri.

6.24 Disegnare e stabilire la tipologia del grafo rappresentato dalla seguente matrice di adiacenza:

1	0	1	1
0	0	1	0
1	0	0	1
0	1	0	1

Descrivere la matrice che si ottiene aggiungendo un altro vertice connesso con archi ai primi due vertici e scambiando il verso all'arco congiungente i vertici 2 e 3.

6.25 Stabilire la tipologia di grafo descritto dalla seguente matrice di adiacenza:

0	6	0	5	0
0	0	8	0	0
0	0	0	0	7
0	0	4	2	0
9	0	0	0	0

Disegnare il grafo. Scrivere delle assegnazioni per invertire il verso degli archi.

6.26 Disegnare e stabilire la tipologia del grafo rappresentato dalla seguente matrice di adiacenza:

0	∞	7	∞
6	0	∞	∞
∞	5	2	3
4	∞	∞	0

6.27 Disegnare e rappresentare mediante una matrice il grafo K_5. Stabilire, motivando le affermazioni, se il grafo è planare.

6.28 Dimostrare che il grafo K_4 è planare mentre non lo è K_5.

6.29 Determinare la grandezza del grafo K_n.

6.30 Con $K_{n,n}$ si denota il grafo non orientato, non pesato e non labellato avente $2n$ vertici ed in cui n vertici non sono collegati fra loro e tutti questi vertici sono collegati con tutti gli altri n vertici. Disegnare $K_{3,3}$. Scrivere una porzione di algoritmo per generare la matrice di adiacenza di $K_{n,n}$.

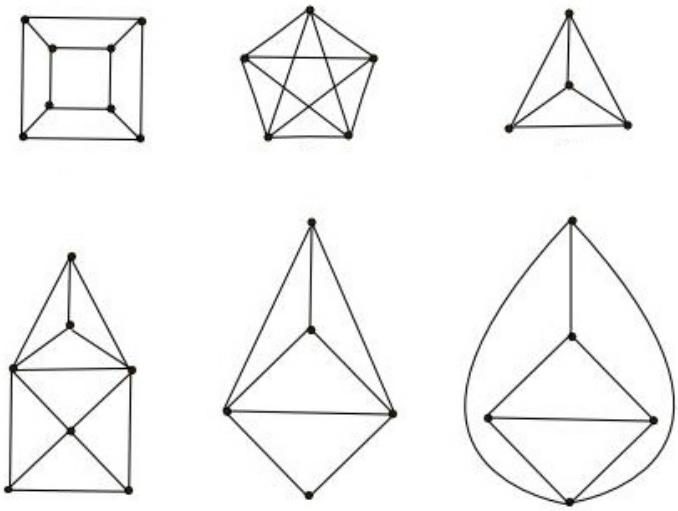
6.31 Descrivere sinteticamente, mediante un esempio, i seguenti problemi sui grafi:

1. *problema del commesso viaggiatore*

2. *problema del minimo albero di supporto*
3. *problema della planarità di un grafo*
4. *problema del cammino minimo.*

6.32 Con K_n si denota il grafo non orientato, non pesato e non labellato avente n vertici ed in cui ogni vertice è collegato con un arco a tutti gli altri vertici, escluso se stesso. Disegnare K_5. Scrivere una porzione di algoritmo per generare la matrice di adiacenza di K_n.

6.33 Determinare quali dei seguenti grafi sono euleriani. Per ciascun grafo euleriano determinarne un ciclo euleriano.



Parte II

ALGORITMI

7

ORDINARE

*Le tecniche di ordinamento forniscono inoltre eccellenti illustrazioni delle idee generali coinvolte nell'*analisi degli algoritmi, *cioè le idee utilizzate per determinare le caratteristiche di efficienza degli algoritmi cosicchè si possa fare una scelta intelligente fra metodi in competizione.*

D. Knuth,
The Art of Computer Programming

Nelle vita quotidiana il problema dell'ordinamento si presenta molto frequentemente. Ad esempio, un giocatore di carte affronta un problema di ordinamento quando deve disporre in mano inizialmente le carte che gli sono state distribuite, al fine di essere agevolato nelle successive fasi del gioco.

L'ordinamento di elementi (numeri o altre tipologie di dati) costituisce una delle più frequenti forme di elaborazione dei dati: si stima che oltre il 25 per cento del tempo di elaborazione dei computer sia impiegato per ordinare.

Il problema dell'ordinamento è spesso accoppiato con il problema della ricerca e, unitamente, costituiscono uno dei capitoli più importanti della teoria dell'elaborazione dei dati. Ne è conferma la monumentale opera *The Art Of Computer Programming* del famoso matematico ed informatico Donald Knuth che ha dedicato un intero e maestoso volume agli algoritmi di ricerca ed ordinamento (vol. II, *Sorting and Searching*).

7.1 Il problema dell'ordinamento

Nella sua forma più generale il *problema dell'ordinamento* viene formulato come segue:

> *Ci sono degli oggetti localizzati in delle posizioni disposte in sequenza e sugli oggetti è definito un criterio d'ordine basato su una relazione d'ordine* $\leq$*. Disporre gli elementi in sequenza, uno dopo l'altro, in modo che ogni elemento preceda gli elementi uguali o più grandi.*

Esistono moltissimi procedimenti per ordinare una sequenza di elementi; questi algoritmi di ordinamento possono essere classificati in diverse famiglie a seconda dell'idea generale sulla quale si basano. Si possono distinguere le seguenti famiglie:

- *ordinamento per scambio*: si eseguono confronti fra coppie di elementi; quando si incontrano due elementi fuori ordine, vengono scambiati
- *ordinamento per selezione*: si individua l'elemento più piccolo; tale elemento viene separato dagli altri; si individua poi il minimo della parte rimanente e si prosegue con la stessa modalità
- *ordinamento per inserzione*: si considerano gli elementi uno alla volta; ciascun elemento viene inserito nella posizione appropriata, rispetto a quelli già considerati, con una tecnica simile a quella che usano i giocatori di carte che le ordinano in mano, raccogliendole una ad una

Esempio 7.1.1 - Due variabili x ed y possono essere ordinate eseguendo un eventuale scambio dei loro valori nel caso esse non rispettino il criterio d'ordine fissato. A seguire è riportato il semplice algoritmo di ordinamento.

Algoritmo 1 - Ordinamento di due variabili

Input: variabili x ed y
Ensure: le variabili sono ordinate, ossia $x \leq y$
1: **if** $x > y$ **then**
2: scambia $x \leftrightarrow y$
3: **end if**

Problema 7.1.1 Il seguente problema, pur riferendosi ad una situazione di vita quotidiana, è interessante dal punto di vista algoritmico.

> *Su un gocciolatoio sono disposti dei piatti piani e delle fondine in modo da occupare tutte le posizioni. Si vuole sistemare ordinatamente i piatti in modo da avere tutti i piatti piani sulla sinistra e le fondine sulla destra. Si suppone di poter eseguire uno scambio fra due piatti, prendendo un piatto alla volta in ciascuna mano e senza la possibilità di appoggiare i piatti fuori dal gocciolatoio.*

Soluzione. Questo problema può essere risolto come descritto nell'algoritmo 2 che si fonda su un'idea sfruttata nell'algoritmo di ordinamento *quicksort* che sarà descritto più avanti.

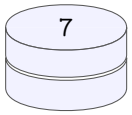
Algoritmo 2 - Algoritmo di ordinamento dei piatti

Require: fila dei piatti da ordinare
Ensure: la fila di piatti è ordinata
1: poni le mani alle estremità della fila dei piatti
2: **while** le mani non si sono incontrate **do**
3: **while** la mano sinistra è su un piatto piano **do**
4: avanza la mano verso destra
5: **end while**
6: **while** la mano destra è su un piatto fondo **do**
7: avanza la mano verso sinistra
8: **end while**
9: **if** i piatti che hai in mano sono fuori ordine **then**
10: scambiali
11: **end if**
12: sposta la mano sinistra di un posto a destra
13: sposta la mano destra di un posto a sinistra
14: **end while**

Osservazione. I precedenti due esempi fanno intuire che gli algoritmi di ordinamento si fondano principalmente sull'operazione di *confronto* fra due elementi e su quella di *spostamento* o *scambio* di elementi. Questa ipotesi è confermata dai vari algoritmi di ordinamento che saranno presentati in questo capitolo. Le specifiche strategie di ordinamento dipenderanno dalla particolare struttura organizzativa degli elementi da ordinare e, quindi, dalle specifiche modalità di accesso agli elementi.

7.2 Ordinare pile

Consideriamo il problena di ordinare una pila di 2 dischi numerati sovrapposti, similmente alla situazione del problema della torre di Hanoi, disponendo i dischi dal valore più grande alla base al più piccolo in testa. Su ciascun disco è impresso sulla parte superiore un numero che risulta non visibile per i blocchetti sotto al primo.



A *B* *C*

Per gestire le operazioni di accesso ai valori impressi sui dischi e le operazioni di spostamento dei dischi utilizziamo le seguenti istruzioni:

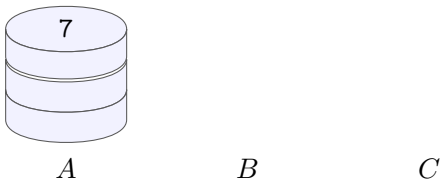
- *testa*(P): numero impresso sul disco di testa della pila P
- *sposta*(P, Q): sposta il disco di testa della pila P alla testa della pila Q

Con queste operazioni il problema è risolto mediante l'algoritmo 3.

Algoritmo 3 - ordinamento di una pila di 2 dischi numerati

```
1: sposta(A, B)
2: if testa(B) ≤ testa(A) then
3:     sposta(B, A)
4: else
5:     sposta(A, C)
6:     sposta(B, A)
7:     sposta(C, A)
8: end if
```

Consideriamo ora, analogamente alla precedente situazione, il problena di ordinare una pila di 3 dischi numerati sovrapposti.



In questo caso (3 dischi) la situazione si complica e non emerge immediatamente un soluzione del problema, data la numerosità delle possibili situazioni che si possono verificare rispetto all'ordine usando 3 dischi. Ci proponiamo allora un obiettivo intermedio, mirando ad ottenere 2 dischi ordinati inversamente (più piccolo alla base e più grande in testa) sulla posizione B, senza spostare il disco alla base della pila A. Si tratta dell'applicazione della metodologia di scomposizione in sottoproblemi, esposta nell'algoritmo 4.

Algoritmo 4 - ordinamento di una pila di 3 dischi numerati

```
1: ordina in B modo inverso i primi due dischi della pila A
2: completa l'ordinamento confrontando le teste delle pile.
```

I sottoproblemi corrispondenti ai punti 1 e 2 dell'algoritmo 4 sono sviluppati nell'algoritmo 5, rispettivamente alle linee 1-8 e 9-22.

Algoritmo 5 - ordinamento di una pila di 3 dischi numerati

```
1: sposta(A, B)
2: if testa(B) ≤ testa(A) then
3:    sposta(A, B)
4: else
5:    sposta(B, C)
6:    sposta(A, B)
7:    sposta(C, B)
8: end if
9: if testa(B) ≤ testa(A) then
10:   sposta(B, A)
11:   sposta(B, A)
12: else
13:   sposta(A, C)
14:   sposta(B, A)
15:   if testa(B) ≤ testa(C) then
16:      sposta(B, A)
17:      sposta(C, A)
18:   else
19:      sposta(C, A)
20:      sposta(B, A)
21:   end if
22: end if
```

I due algoritmi 3 e 5 risolvono due sotto classi di problemi corrispondenti ai casi $n = 2$ dischi e $n = 3$ dischi. Per valori più grandi di n un approccio specifico per ciascun valore di n risulta improduttivo, oltre che difficoltoso. Per gestire un generico numeri di dischi nella pila si può usare un controllo della forma

- *vuota*(P): test se la pila P è vuota

Applicando la metodologia top-down, l'algoritmo risolutivo si può esprimere come segue:

Algoritmo 6 - Ordinamento di una pila di blocchetti numerati

Require: pila di blocchetti numerati A
Ensure: la pila A è ordinata
1: considera due pile ausiliarie B ed C vuote
2: **while** la pila A non è vuota **do**
3: estrai da A il minimo e mettilo in B usando C come appoggio riportando in A tutti gli elementi presenti in A ed in C
4: **end while**
5: rovescia la pila B in A

La strategia dell'algoritmo 6 è condensata alla linea 3. La soluzione di questo sottoproblema è descritta alle linee 4-13 dell'algoritmo 7 che segue.

Algoritmo 7 - Ordinamento di una pila di blocchetti numerati

Require: pila A
Ensure: la pila A è ordinata
1: $B \leftarrow$ *pila vuota*
2: $C \leftarrow$ *pila vuota*
3: **while** $\neg$ *vuota*(A) **do**
4: *sposta*(A, B)
5: **while** $\neg$ *vuota*(A) **do**
6: **if** *testa*$(A) <$ *testa*(B) **then**
7: *sposta*(B, C)
8: *sposta*(A, B)
9: **else**
10: *sposta*(A, C)
11: **end if**
12: **end while**
13: *rovescia*(C, A)
14: **end while**
15: *rovescia*(B, A)

7.3 Ordinamento di sequenze

Frequentemente il problema dell'ordinamento si presenta nella forma di *ordinamento di sequenze* in cui sono ammesse le seguenti operazioni:

- accesso ad un generico elemento della sequenza mediante il corrispondente indice di posizione
- assegnazione di un valore ad un generico elemento della sequenza

Esempio 7.3.1 - Ordinare la sequenza di numeri

27	32	15	11	54	78	15	23

significa operare una permutazione degli elementi in modo da ottenere la sequenza

11	15	15	23	27	32	54	78

Il problema dell'ordinamento viene risolto adottando due possibili impostazioni: *funzionale e procedurale.*

Modalità funzionale

Adottando la *modalità funzionale* la sequenza da ordinare viene esaminata ma non viene modificata; viene generata un'altra sequenza che costituisce il risultato del processo di ordinamento.

In modo più formalizzato e più utile per impostare gli algoritmi di ordinamento, il problema dell'ordinamento, nella sua formulazione generale, si esprime come segue:

> *Data una sequenza di elementi* $[a_0, a_1, \ldots, a_{n-1}]$ *dove ciascun* a_i *è confrontabile con gli altri, determinare una sequenza* $a' = [a'_0, a'_1, \ldots, a'_{n-1}]$ *che sia una permutazione della sequenza* a *e che sia ordinata, ossia* $a'_0 \leq a'_1 \leq \cdots \leq a'_{n-1}$.

Modalità procedurale

Adottando la *modalità procedurale* la sequenza da ordinare viene modificata (mediante assegnazioni e scambi di elementi) ed alla fine risulta ordinata.

Spesso il problema dell'ordinamento viene definito in un contesto imperativo in cui la sequenza a è modificabile; in questo caso si tratta di determinare una permutazione p dell'insieme $\{0, 1, \ldots, n-1\}$ in modo che la sequenza $[a_{p(0)}, a_{p(1)}, \ldots, a_{p(n-1)}]$ risulti ordinata. La permutazione p viene costruita modificando direttamente la sequenza a; il problema si traduce quindi nella seguente formulazione.

> *Data una sequenza* $a = [a_0, a_1, \ldots, a_{n-1}]$*, permutarne gli elementi in modo tale che* $a_0 \leq a_1 \leq \cdots \leq a_{n-1}$

7.4 Ordinamento a bolle

La strategia dell'*ordinamento a bolle* (*bubble-sort*) si basa sull'idea di far salire, per passi successivi, verso la fine della sequenza gli elementi maggiori, attraverso degli scambi fra elementi adiacenti:

> *Si confrontano i primi due: se sono fuori ordine vengono scambiati; si ripete il procedimento con il secondo ed il terzo e così via, fino alla fine della sequenza. Al termine del primo ciclo l'elemento maggiore è sicuramente posizionato alla fine della sequenza. Si inizia allora un altro ciclo per far salire al penultimo posto l'elemento immediatamente minore e così via fino ad aver ordinato tutta la sequenza.*

L'algoritmo che esprime questa strategia può essere descritto come segue:

Algoritmo 8 - Ordinamento a bolle (*bubble-sort*)

Require: sequenza a da ordinare
Ensure: la sequenza a è ordinata
1: **for** i **from** 0 **to** $len(a) - 1$ **do**
2: **for** j **from** 0 **to** $len(a) - i - 2$ **do**
3: **if** $a_j > a_{j+1}$ **then**
4: scambia $a_j \leftrightarrow a_{j+1}$
5: **end if**
6: **end for**
7: **end for**

L'analisi della complessità dell'algoritmo *bubble-sort*, nella versione precedentemente riportata, risulta molto facile in quanto l'algoritmo esegue un uguale numero di confronti indipententemente dai valori presenti nella sequenza. In particolare, indicando con n il numero di elementi della sequenza, alla prima passata vengono svolti $n-1$ confronti, alla seconda $n-2$ e così via per le successive passate, fino ad arrivare a svolgere 1 confronto all'ultima passata quando vengono confrontati gli elementi a_0 e a_1. Vengono complessivamente svolti

$$T(n) = (n-1) + (n-2) + \cdots + 1 = n(n-1)/2$$

confronti e pertanto la complessità asintotica risulta pari a $O(n^2)$. Ciò conferma che le prestazioni dell'algoritmo *bubble-sort* sono particolarmente scadenti in quanto richiede che vengano fatti molti confronti e scambi affinché un elemento raggiunga la propria destinazione finale.

L'algoritmo *bubble-sort* esegue, ad ogni ciclo, una passata su tutta la porzione della sequenza non ancora ordinata. Per questo motivo, se in una passata non avvengono scambi, significa che la sequenza è ordinata. Un'ulteriore ottimizzazione si basa sull'osservazione che ad ogni passata basta spingersi nella sequenza fino alla posizione in cui alla precedente passata era stato eseguito l'ultimo scambio (*bubble-sort ottimizzato*). Come caso estremo, se la sequenza è inizialmente già ordinata è sufficiente un'unica scansione degli elementi per

riconoscere che la sequenza è ordinata e concludere il processo di ordinamento. Questa variante dell'algoritmo, nonostante l'idea di ottimizzazione sulla quale si fonda, ha una complessità asintotica nel caso medio pari a $O(n^2)$. Offre invece buone prestazioni nel caso in cui la sequenza da ordinare sia *quasi ordinato*. Nel caso estremo in cui la sequenza sia già ordinato (caso ottimo per l'algoritmo *bubble-sort*) l'algoritmo esegue $n-1$ confronti ed ha dunque una complessità asintotica pari a $O(n)$.

7.5 Ordinamento per selezione

L'*ordinamento per selezione* (*selection-sort*) si basa su una strategia naturale e spontanea che le persone applicano per ordinare un insieme di elementi. La strategia è descrivibile come segue:

> *Determina il minimo fra gli elementi della sequenza e scambialo con il primo. Determina il minimo fra gli elementi della sequenza, escluso il primo, e scambialo con il secondo. Determina il minimo della sequenza, esclusi i primi due, e scambialo con il terzo e così via, fino a quando la porzione di sequenza da ordinare si è ridotta ad un unico elemento.*

L'algoritmo che esprime questa strategia può essere descritto come segue:

Algoritmo 9 - Ordinamento per selezione (*selection-sort*)

Require: sequenza a da ordinare
Ensure: la sequenza a è ordinata
1: **for** i **from** 0 **to** $len(a)-1$ **do**
2: determina la posizione p del valore minimo della porzione $[a_i, \ldots, a_{n-1}]$
3: **if** $p \neq i$ **then**
4: scambia $a_p \leftrightarrow a_i$
5: **end if**
6: **end for**

Le prestazioni dell'algoritmo *selection-sort* sono indipendenti dai valori della sequenza da ordinare e, per questo, i casi ottimo, medio e pessimo coincidono. Dall'analisi dell'algoritmo emerge che vengono svolte $n-1$ selezioni del minimo; l'i-esima selezione richiede $n-i$ confronti; pertanto il numero complessivo di confronti è pari a

$$T(n) = \sum_{i=1}^{n-1} n - i = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2}$$

La complessità asintotica di questo algoritmo risulta, in tutte le situazioni, pari a $O(n^2)$.

7.6 Ordinamento per inserzione

L'*ordinamento per inserzione* (*insertion-sort*) applica la stessa strategia che usa un giocatore di carte per raccogliere le carte ad una ad una e disporle

in mano in modo ordinato. Il procedimento può essere descritto in modo discorsivo come segue:

> *A partire dal secondo elemento fino all'ultimo, inserisci l'elemento al posto giusto fra quelli compresi fra il primo e quello che si sta inserendo, spostando a destra di una posizione gli elementi maggiori di quello che si sta inserendo.*

L'algoritmo che esprime questa strategia può essere descritto come segue:

Algoritmo 10 - Ordinamento per inserzione (*insertion-sort*)

Require: sequenza a da ordinare
Ensure: la sequenza a è ordinata
1: **for** i **from** 1 **to** $len(a) - 1$ **do**
2: sposta a destra di una posizione gli elementi della porzione già ordinata$[a_0, \ldots, a_{i-1}]$ composta dagli elementi maggiori di a_i
3: memorizza a_i nella locazione lasciata libera dall'ultimo elemento spostato a destra
4: **end for**

Nel caso di ordinamento per inserzione il numero di iterazioni del ciclo *for* più esterno è pari a $n - 1$. Nel caso pessimo l'elemento i-esimo che viene inserito richiede che esso venga confrontato con tutti gli elementi già inseriti e quindi richiede che vengano svolti $i - 1$ confronti. Il numero complessivo di confronti richiesto è dunque pari a

$$T(n) = \sum_{i=2}^{n} i - 1 = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

La complessità asintotica risulta dunque pari a $O(n^2)$.

7.7 Ordinamento veloce

L'algoritmo di ordinamento *quick-sort* appartiene alla categoria di algoritmi di ordinamento veloce ed è uno dei più utilizzati. In molti linguaggi è già implementato nativamente nel linguaggio. Le varie implementazioni dell'algoritmo si basano su un'idea, risalente al 1960, dell'informatico C.A.R. Hoare. L'algoritmo si basa sulla seguente strategia:

> *Scegliere un valore a caso presente nella sequenza (perno); ripartire la sequenza in due porzioni: nella prima mettere tutti gli elementi minori o uguali al perno, nella seconda elementi maggiori o uguali al perno. Ripetere poi ricorsivamente l'ordinamento su entrambe le porzioni della sequenza. La chiusura della ricorsione è costituita dalla condizione di avere una sequenza di un solo elemento.*

La partizione rispetto al perno avviene ponendo inizialmente due indici alle estremità della porzione di sequenza da ordinare, avvicinandoli verso il centro, scambiando gli elementi quando si arriva ad elementi da scambiare, in quanto fuori ordine rispetto al perno.

In notazione algoritmica, la sopra descritta strategia si esprime come segue:

Algoritmo 11 - Ordinamento veloce (*quick-sort*)

Require: sequenza a da ordinare
Ensure: la sequenza a è ordinata
1: **if** c'è più di un elemento da ordinare **then**
2: poni due indici i e j agli estremi della sequenza da ordinare
3: scegli un elemento a caso (perno)
4: ripartisci la sequenza in due porzioni, la prima con valori minori o uguali al perno e la seconda con valori maggiori o uguali al perno, avanzando verso il centro i due indici i e j e scambiando fra loro i due elementi quando sono fuori ordine rispetto al perno
5: applica ricorsivamente il procedimento alle due porzioni della sequenza
6: **end if**

Per l'algoritmo *quick-sort* il caso ottimo si ha quando la sequenza viene ripartita rispetto al perno, ad ogni passo, in due parti perfettamente bilanciate (il perno va a collocarsi sempre al centro della sequenza). In questo caso la complessità asintotica risulta pari a $O(n \log n)$. Il caso pessimo, che corrisponde al caso in cui le ripartizioni rispetto al perno risultino sempre completamente sbilanciate (viene scelto come perno il valore minimo o massimo fra gli elementi della sequenza), ha una complessità asintotica pari a $O(n^2)$.

7.8 Reti di ordinamento

Il problema dell'ordinamento può essere affrontato anche con un approccio *sistemistico* realizzando delle strutture di blocchi che ordinano una sequenza di ingressi. Le *reti di ordinamento* sono dei sistemi combinatori in grado di fornire in uscita, ordinati, gli ingressi.

Esempio 7.8.1 - Vogliamo costruire una rete combinatoria *sort* per ordinare due elementi x_1, x_2, sui quali sia definita una relazione d'ordine. Si tratta di realizzare un blocco soddisfacente alla specifica funzionale descritta nella figura 7.1.

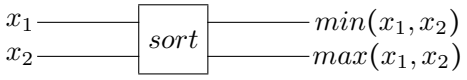


Figura 7.1: Specifica del blocco *sort* che ordina due ingressi.

Supponendo di poter disporre dei due operatori *min* e *max* (che determinano rispettivamente il minimo ed il massimo fra due elementi) il blocco *sort* che

ordina due ingressi può essere descritto nella figura 7.2.

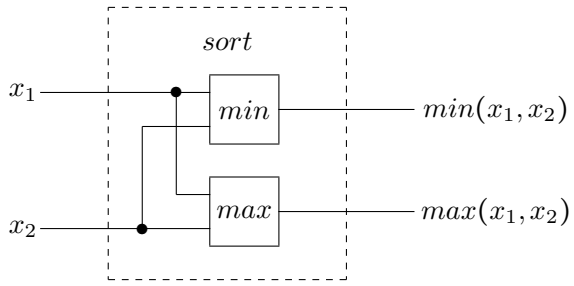


Figura 7.2: Struttura di una rete combinatoria che ordina due ingressi.

In notazione grafica la rete combinatoria che ordina due ingressi viene indicata sinteticamente con lo schema descritto in figura 7.3.

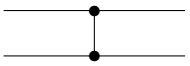


Figura 7.3: Schema di una rete di ordinamento a due ingressi.

Una rete composta da operatori *sort* che produca in uscita una sequenza ordinata viene detta *rete di ordinamento*.

Problema 7.8.1 Componendo degli operatori *sort* che ordinano due elementi, costruire una rete di ordinamento che ordina tre elementi.

Soluzione. Una soluzione è fornita dalla rete descritta nella figura 7.4.

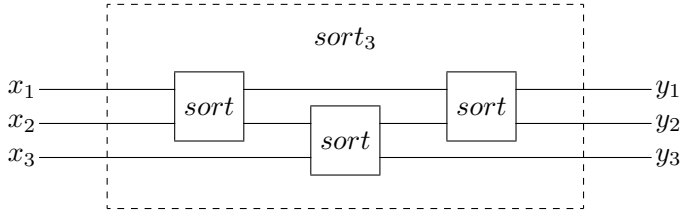


Figura 7.4: Struttura di una rete di ordinamento che ordina tre ingressi.

Una rete di ordinamento come quella descritta figura 7.4 viene solitamente disegnata, in una forma più sintetica, come descritto nella figura 7.5.

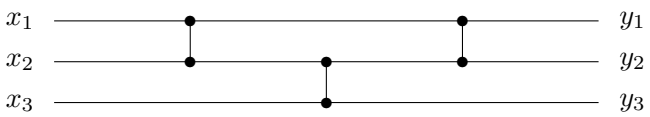


Figura 7.5: Schema di una rete di ordinamanto che ordina tre ingressi.

ESERCIZI

7.1 Ordinare tre variabili x, y, z.

7.2 Due mazzi di carte, nuovi ed uguali, sono stati uniti e mischiati. Separare le carte ricostruendo i due mazzi originali. Determinare la complessità computazionale dell'algoritmo che si è descritto.

7.3 Un grande mazzo di carte è stato ottenuto mischiando vari mazzi di carte, alcuni incompleti. Separare il maggior numero possibile di mazzi completi.

7.4 Discutere la complessità degli algoritmi di ordinamento nel caso in cui la sequenza sia inizialmente già ordinata.

7.5 Discutere la complessità dell'algoritmo *quick-sort* nel caso in cui la sequenza sia inizialmente già ordinata e come perno venga scelto il primo elemento della porzione di sequenza da ordinare.

7.6 Descrivere la situazione della sequenza $[5, 6, 8, 1, 2, 7, 5, 3, 9, 6, 8, 4]$ dopo aver eseguito un ciclo dell'algoritmo *quick-sort* prendendo come perno il primo elemento della sequenza.

7.7 Spiegare cosa significa la frase: *L'algoritmo di ordinamento* quick-sort *ha una complessità asintotica nel caso medio pari a* $O(n \log n)$.

7.8 Fare una tabella che riporti la complessità asintotica degli algoritmi di ordinamento *bubble-sort*, *insertion-sort*, *selection-sort*, *quick-sort*, nei diversi casi ottimo, medio, pessimo.

7.9 Ricercare nella letteratura informatica gli algoritmi di ordinamento *merge-sort* e *heap-sort* e confrontarne la complessità con quella degli algoritmi presentati in questo capitolo.

7.10 Su una rivista si trova la seguente frase: "Gli algoritmi di ordinamento della classe $\mathcal{A}$ hanno, nel caso pessimo, una complessità asintotica pari a $O(n^2)$". Spiegare cosa significa la frase. Dire quali, fra gli algoritmi di ordinamento studiati, appartengono alla classe $\mathcal{A}$ di cui si parla nella rivista citata.

7.11 Su una rivista si legge: "Il prof. Dalla Balla ha scoperto un algoritmo di ordinamento di sequenze avente, nel caso medio, complessità uguale a $n\,log\,(\log n)$". Dire se il prof. Dalla Balla ha fatto una scoperta degna di nota e se è credibile.

7.12 Decidere se una sequenza è ordinata (crescentemente). Decidere se una sequenza è ordinata (crescentemente o decrescentemente).

7.13 Decidere se una data sequenza di numeri interi è ordinata crescentemente ed è composta da valori contigui; ad esempio la sequenza $a = [5, 6, 7, 8, 9]$ è ordinata crescentemente ed ha gli elementi contigui, mentre non lo sono le sequenze $b = [5, 7, 8, 6, 9]$, $c = [4, 6, 7, 8, 9]$.

7.14 Descrivere un algoritmo di complessità lineare per ordinare una sequenza di valori booleani. Motivare perché tale algoritmo non contraddice la teoria che afferma che il problema dell'ordinamento di una sequenza ha complessità asintotica pari a $O(n \log n)$.

7.15 Decidere se due sequenze (di uguali dimensioni) sono una *permutazione* una dell'altra, ossia se contengono gli stessi valori, con la stessa frequenza, indipendentemente dalla posizione. Valutare la complessità dell'algoritmo.

7.16 Decidere se due sequenze contengono gli stessi elementi. Valutare la complessità dell'algoritmo. Dimostrare che la complessità asintotica del problema è non superiore a $O(n \log n)$.

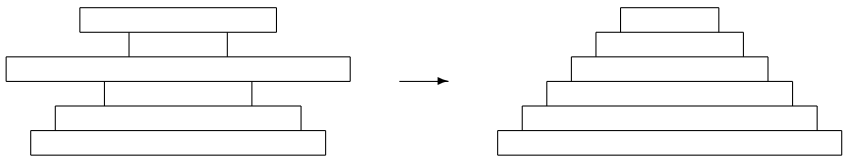
7.17 Dimostrare che il problema *Decidere se due sequenze hanno elementi in comune.* ha una complessità asintotica nel caso pessimo non superiore a $O(n \log n)$. Descrivere un algoritmo avente questa complessità.

7.18 Decidere se una sequenza è composta da elementi tutti distinti. Valutare, nel caso ottimo e nel caso pessimo, la complessità, dell'algoritmo descritto. Motivare perché la complessità asintotica del problema in questione è non superiore a $O(n \log n)$.

7.19 Determinare la frequenza massima con la quale compare l'elemento avente la più alta frequenza all'interno di una sequenza. Valutare la complessità dell'algoritmo. Motivare perché il problema in questione ha una complessità asintotica non superiore a $O(n \log n)$.

7.20 Una formulazione del cosiddetto *problema della bandiera polacca* (composta dai due colori bianco e rosso) si esprime come segue. Su un tavolo sono disposte in fila $n > 1$ pedine, alcune bianche ed alcune rosse. Descrivere un algoritmo per ordinare le pedine, mettendo prima le bianche e poi le rosse. Sono ammesse le operazioni di guardare il colore delle pedine e invertire di posto due pedine. L'algoritmo deve minimizzare il numero di scambi.
Analogamente al problema della bandiera polacca, risolvere il *problema della bandiera olandese* composta dai colori rosso, bianco e blu.
Risolvere il problema della bandiera polacca e della bandiera olandese nell'ipotesi che si possano effettuare scambi solo fra pedine contigue.

7.21 Su un tavolo sono disposti, uno sopra l'altro, n dischi di diverse grandezze. Ordinare la pila di dischi, dal più grande alla base al più piccolo in testa. È ammesso inserire una paletta fra due dischi e rovesciare sul posto la pila di dischi sopra la paletta. Si ammette inoltre di usare l'operazione elementare che consiste nell'individuare il disco più piccolo e quello più grande.

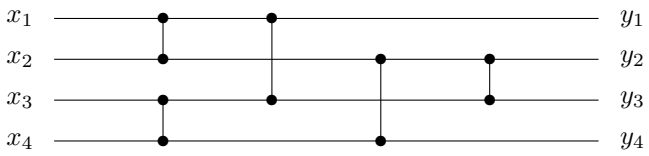


7.22 Descrivere una rete di ordinamento a 4 ingressi, corrispondente all'algoritmo di ordinamento *bubblesort*. Descrivere il processo di ordinamento nel caso di un'istanza di problema avente gli ingressi ordinati in modo decrescente, ossia $x_1 \geq x_2 \geq x_3 \geq x_4$. Determinare il tempo di esecuzione, nel caso sequenziale e nel caso parallelo, assumendo come unitario il tempo di elaborazione di un elemento *sort* che ordina 2 elementi.

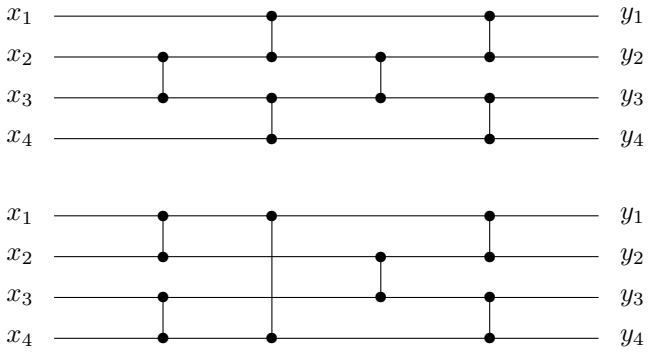
7.23 Descrivere una rete a 4 ingressi e 4 uscite, composta da operatori *sort*, che abbia l'effetto di portare il valore minimo nella prima uscita, ossia in modo che risulti $y_1 = min\{x_1, x_2, x_3, x_4\}$. Descrivere il processo di elaborazione nel caso di un'istanza di problema avente gli ingressi ordinati in modo decrescente, ossia $x_1 \geq x_2 \geq x_3 \geq x_4$. Determinare il tempo di esecuzione, nel caso sequenziale e nel caso parallelo, assumendo come unitario il tempo di elaborazione di un elemento *sort* che ordina 2 elementi.

7.24 Dati quattro ingressi x_1, x_2, x_3, x_4, con la condizione $x_1 \leq x_2$, $x_3 \leq x_4$, descrivere una rete di ordinamento che, usando l'operatore *sort* binario, ordina i quattro ingressi dati. Valutare il tempo di esecuzione nel caso di elaborazione sequenziale ed elaborazione concorrente, assumendo come unitario il tempo di esecuzione di un operatore *sort* binario.

7.25 Componendo l'operatore *sort* si può comporre una rete di ordinamento per ordinare 4 elementi come descritto sotto. Dimostrare che questa rete funziona correttamente. Evidenziare il processo di ordinamento dei numeri 7, 4, 9 e 3, scrivendo i numeri in corrispondenza di ciascun ramo di uscita. Determinare il tempo di esecuzione, nel caso sequenziale e nel caso parallelo, assumendo come unitario il tempo di elaborazione di un blocco *sort* che ordina 2 elementi.



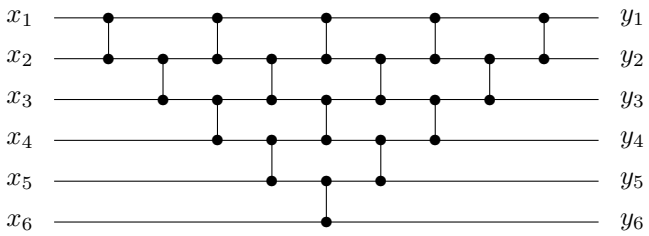
7.26 Dimostrare che le seguenti due reti di ordinamento sono equivalenti (entrambe ordinano i 4 elementi di ingresso).



Stabilire quale delle due è più efficiente nel caso di elaborazione parallela.

7.27 Descrivere una rete di ordinamento per ordinare 5 elementi.

7.28 Dimostrare che la seguente rete di ordinamento ordina correttamente 6 elementi. Individuare l'algoritmo corrispondente alla rete.



Generalizzare al caso di n elementi. Determinare, in funzione di n, quanti sono gli operatori *sort* necessari. Calcolare il tempo di ordinamento in funzione di n (nel caso di esecuzione sequenziale e nel caso di esecuzione parallela).

7.29 Descrivere una rete di ordinamento corrispondente all'algoritmo *bubble-sort*, per ordinare una sequenza composta da 8 elementi.

7.30 Dimostrare che, anche se alquanto inaspettatamente, l'aggiunta di un operatore *sort* ad una rete di ordinamento funzionante, può comprometterne la funzionalità.

7.31 Dimostrare il seguente *Principio Zero-Uno*: Se una rete di ordinamento funziona correttamente con ingressi presi dall'insieme $\{0, 1\}$, allora funziona correttamente anche per elementi di qualsiasi tipo sul quale sia stabilito un criterio di ordine lineare.

7.32 È data una pila di blocchetti numerati, ordinata, con l'elemento più grande in basso. Eliminare dalla pila tutti gli elementi duplicati, lasciando una sola occorrenza di ciascun elemento. Ad esempio, la pila $[9, 7, 7, 7, 5, 4, 4, 2\rangle$ produce come risultato la pila $[9, 7, 5, 4, 2\rangle$.

7.33 Su un array ordinato non decrescentemente è stata eseguita un'assegnazione che può aver falsato la preesistente condizione di ordinamento. Il problema consiste nel ripristinare la condizione di ordinamento, in modo efficiente, mediante un algoritmo a bassa complessità computazionale. Descrivere dapprima l'idea di soluzione a parole e successivamente formalizzarla mediante un algoritmo. Valutare la complessità dell'algoritmo.

8

RICERCARE

Cosa faresti se uno ti desse un grande elenco telefonico e ti chiedesse di trovare il nome della persona il cui numero è 795-6841?

D. Knuth,
The Art of Computer Programming

La ricerca di informazioni rappresenta uno dei problemi più importanti dell'informatica in quanto costituisce la forma più frequente di elaborazione dei dati in diversi ambiti applicativi; di più, questa forma di elaborazione costituisce elemento strategico per diverse attività: si pensi, ad esempio, ai motori di ricerca nel web ed alla ricerca nelle basi di dati; disporre di algoritmi di ricerca veloci costituisce uno dei fattori decisivi di successo di un'applicazione informatica o addirittura di un'intera azienda.

8.1 Il problema generale della ricerca

Il *problema della ricerca*, in generale, può essere formulato come segue:

> *Dato un contenitore A ed una proprietà P, ricavare una desiderata informazione I riguardande gli elementi $x \in A$ che soddisfano alla proprietà P.*

La precedente formulazione generale mette in evidenza i diversi parametri A, P, I del problema della ricerca; questi parametri saranno analizzati nelle seguenti sezioni.

La struttura del contenitore A

Il contenitore A su cui si svolge la ricerca ha una propria struttura organizzativa interna degli elementi di cui è composto; per questo motivo il problema della ricerca è correlato alle strutture dati. I contenitori sono spesso organizzati mediante delle strutture dati più articolate che non la semplice organizzazione sequenziale per rendere più efficienti i procedimenti di ricerca. I casi più frequenti di strutture dei contenitori coinvolti nel problema della ricerca sono indicati nella figura 8.1.

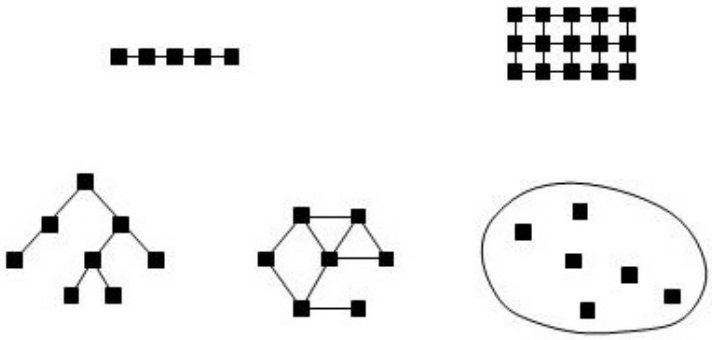


Figura 8.1: Alcune strutture dei contenitori di elementi: sequenza, matrice, albero, grafo, insieme.

Le strategie di ricerca in un contenitore sono strettamente legate alle modalità di accesso agli elementi del contenitore e queste sono, a loro volta, dipendenti dalla specifica struttura organizzativa interna del contenitore (insieme, sequenza, albero, grafo). Su queste tipologie di strutture vengono adottate le seguenti modalità di accesso:

- accesso su una generica struttura iterabile
- accesso in modo sequenziale, ad indice (come negli array)
- accesso in modalità ricorsiva (come negli alberi binari)

La forma della proprietà P

La proprietà P sulla quale si fonda il problema della ricerca è spesso descritta mediante un confronto.

Esempio 8.1.1 - Un contenitore contiene come elementi i dati anagrafici di alcune persone; ciascun dato ha la struttura

$$[CodiceFiscale, Cognome, Nome, DataNascita]$$

In questa situazione si può impostare una ricerca della seguente forma: *Ricercare le persone nate dopo il* 2000. Questa ricerca corrisponde alla proprietà

$$P(x) \equiv DataNascita(x) \geq 1 \text{ gennaio } 2000$$

In molti casi la proprieà P è rappresentata da un confronto di uguaglianza con un valore noto. In questo caso il problema della ricerca si enuncia nel seguente modo:

Dato un aggregato A di elementi ed un elemento x, decidere se $x \in A$.

Spesso gli elementi x registrati nel contenitore A sono composti (non atomici) e la valutazione della proprietà $P(x)$ si basa su una componente k del dato x, detta *chiave*; con $x.k$ si denota l'intero elemento avente valore k della chiave. Un caso estremo frequentemente utilizzato negli esempi descrittivi è quando l'elemento x è un singolo valore numerico e la proprietà $P(x)$ è un predicato della forma $x = x_0$ dove x_0 è il valore da ricercare.

Una caratteristica aggiuntiva riguarda l'esistenza di un criterio d'ordine sugli elementi x; tale relazione viene denotata con il tradizionale operatore di ordine $\leq$; in questi casi la proprietà P può assumere, ad esempio, la seguente forma:

$$P(x) \equiv x \leq x_0$$

essendo x_0 un dato elemento di A.

Grado di affinamento dell'informazione I

I problemi della ricerca possono essere classificati in base al grado di affinamento dell'informazione che si vuole ottenere come risultato. Spesso, l'informazione I ricercata consiste negli stessi elementi x soddisfacenti alla proprietà P, oppure al loro numero oppure alla loro posizione all'interno del contenitore. Su questi criteri si possono differenziare le seguenti forme di ricerca:

R_1: Ricercare *se* esistono elementi soddisfacenti; il risultato è un valore booleano

R_2: Ricercare *quanti* sono gli elementi soddisfacenti; il risultato è un numero naturale

R_3: Ricercare *quali* sono gli elementi soddisfacenti; il risultato è un contenitore costituito dagli elementi individuati

R_4: Ricercare *dove* sono gli elementi soddisfacenti; il risultato è un contenitore di "localizzatori" degli elementi (ossia dei riferimenti agli oggetti o, nel caso di sequenze, degli indici numerici di posizione)

Nonostante l'apparente somiglianza, queste varie forme di ricerca vengono risolte con strategie sostanzialmente diverse; in particolare viene differenziato il primo caso R_1 (si termina il ciclo di ricerca non appena si sia individuato un elemento soddisfacente) dagli altri tre casi (che richiedono l'accesso a tutti gli elementi del contenitore).

8.2 Strategie di ricerca

In generale, il problema della ricerca può essere affrontato e risolto con diverse strategie:

- *strategia analitica*: consiste nel generare in modo algoritmico tutti gli elementi x di A soddisfacenti al predicato P. In questo caso l'algoritmo che genera gli elementi soddisfacenti risulta essere molto vincolato allo specifico problema (in definitiva alla struttura dell'insieme A e del predicato P di validazione).
- *strategia enumerativa*: nel caso in cui l'insieme A sia finito ed esista un metodo-iteratore per passare in rassegna tutti gli elementi di A si può adottare il seguente schema di algoritmo *brute-force*:

  > *Passa in rassegna, uno dopo l'altro, tutti gli elementi di A ed accetta quelli che soddisfano al predicato P.*

  Tale strategia risulta applicabile anche nel caso in cui l'insieme A sia infinito e ci si accontenti di determinare solo un prefissato numero di elementi soddisfacenti.
- *strategia di backtracking*: nei casi in cui gli elementi da ricercare x siano costituiti da una sequenza o insieme di elementi $(x_1, \ldots, x_n)$ si può adottare una particolare strategia, denominata *backtracking*, descrivibile come segue:

  > *Si parte dalla sequenza vuota. Si costruisce la sequenza-risultato aggiungendo un elemento alla volta, usando un criterio per controllare se la sequenza parziale ha la possibilità di successo finale; se ha possibilità di successo si prosegue nella costruzione, altrimenti si abbandona la sequenza parziale e si cambia l'ultimo elemento con un altro; se non esiste una tale alternativa si rimuove l'ultimo elemento sostituendolo con un altro (se ne esiste un altro), oppure si rimuove anche il precedente.*

  Su queste diverse strategie si fondano diversi algoritmi di ricerca.

8.3 Algoritmi di ricerca

Lo schema generale per ricercare *se* in un contenitore A esiste un elemento soddisfacente ad una data proprietà P ha la forma indicata nell'algoritmo 1.

Algoritmo 1 - Ricerca se esiste un elemento soddisfacente

Input: contenitore A in cui cercare, proprietà P da soddisfare
Output: `TRUE` se e solo se esiste un elemento $x \in A$ tale che $P(x)$

```
1: while ¬(trovato elemento) ∧ (lo si può ancora trovare) do
2:     ricerca l'elemento
3: end while
4: return esito della ricerca
```

Questo algoritmo è così generale che può essere considerato un modello di algoritmo adattabile in tanti modi, a seconda della specifica forma del problema di ricerca che si sta considerando, a seconda della struttura del contenitore A ed a seconda della particolare strategia di ricerca attuata; in particolare sono frequentemente adottate le seguenti due strategie:

(a) passare in rassegna sequenzialmente gli elementi di A

(b) restringere ripetutamente il contenitore A ad un suo sottoinsieme

Queste due strategie si traducono negli algoritmi 2 e 3.

Algoritmo 2 - Ricerca di un elemento (strategia (a))

Input: contenitore A in cui cercare, proprietà P da soddisfare
Output: `TRUE` se e solo se esiste un elemento $x \in A$ tale che $P(x)$

```
1: trovato ← FALSE
2: trovabile ← A non è vuoto
3: while (¬ trovato) ∧ trovabile do
4:     considera un elemento x ∈ A non ancora esaminato
5:     if P(x) then
6:         trovato ← TRUE
7:     else if non ci sono più elementi da esaminare then
8:         trovabile ← FALSE
9:     end if
10: end while
11: return trovato
```

Algoritmo 3 - Algoritmo di ricerca (strategia (b))

Input: contenitore A in cui cercare, proprietà P da soddisfare
Output: `TRUE` se e solo se esiste un elemento $x \in A$ tale che $P(x)$

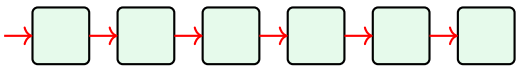
```
1:  trovato ← FALSE
2:  T ← A
3:  while (¬ trovato) ∧ (T ≠ ∅) do
4:      considera un elemento x ∈ T non ancora esaminato
5:      if P(x) then
6:          trovato ← TRUE
7:      else
8:          restringi T ad un suo sottoinsieme proprio
9:      end if
10: end while
11: return trovato
```

La restrizione del contenitore T ad un suo sottoinsieme proprio (istruzione 8 dell'algoritmo 3) può essere realizzata, ad esempio, eliminando l'elemento x dal contenitore T stesso, oppure delimitando, in una sequenza, lo spazio di ricerca; l'istruzione 2 dell'algoritmo 3 serve nel caso in cui si vogliano evitare effetti collaterali del processo di ricerca con la modifica del contenitore A.

8.4 Ricerca su un iterabile

Un *iterabile* è un generico contenitore in cui gli elementi sono accessibili sequenzialmente, uno dopo l'altro, a partire dal primo. Sono esempi di iterabili: le sequenze, le liste, le tuple, gli array, le stringhe. Strutture più articolate, come gli alberi ed i grafi, possono essere inquadrate come iterabili definendo su di essi degli opportuni metodi di accesso agli elementi.

La figura che segue descrive la struttura logica di un iterabile:



Su un iterabile la ricerca viene realizzata passando in rassegna sequenzialmente gli elementi del contenitore, registrando quelli soddisfacenti che costituiranno, alla fine, il risultato della ricerca. Lo schema tipico delle ricerca su tutti gli elementi di un iterabile è riportato nell'algoritmo 4.

Algoritmo 4 - Ricerca su tutti gli elementi dell'iterabile a

```
1: for x in a do
2:     esamina l'elemento x
3: end for
```

Un altro possibile schema di ricerca su un iterabile è riportato nell'algoritmo 5.

Algoritmo 5 - Ricerca in un iterabile

```
1: posizionati sul primo elemento dell'iterabile
2: while ci sono elementi da esaminare do
3:     esamina l'elemento attuale
4:     passa al successivo elemento
5: end while
```

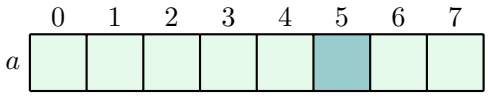
La forma dell'algoritmo 5 risulta più duttile di quella dell'algoritmo 4 in quanto consente una terminazione del ciclo prima di aver terminato l'esame di tutti gli elementi; questa possibilità viene realizzata rafforzando la condizione del ciclo connettendo con l'operatore *and* un'altra condizione che specifica quando il ciclo deve continuare. Ad esempio, l'algoritmo 5, nel caso particolare in cui il problema consista nello stabilire se un dato elemento x è presente nell'iterabile, si declina come descritto nell'algoritmo 6.

Algoritmo 6 - Ricerca in un iterabile

```
1: posizionati sul primo elemento dell'iterabile
2: trovato ← FALSE
3: while ¬trovato ∧ ci sono elementi da esaminare do
4:     if elemento attuale = x then
5:         trovato ← TRUE
6:     else
7:         passa al successivo elemento
8:     end if
9: end while
10: return trovato
```

8.5 Ricerca in una sequenza

Una *sequenza* è un particolare contenitore iterabile in cui gli elementi vengono individuati mediante un indice di posizione; se a è il nome della sequenza, con $a[k]$ si denota il k-esimo elemento. La situazione è descritta nella figura che segue, dove è evidenziato in colore più scuro l'elemento $a[5]$.



Nel caso di una sequenza i precedenti due algoritmi 4 e 5 assumono la forma riportata negli algoritmi 7 e 8.

Algoritmo 7 - Ricerca sequenziale su tutti gli elementi di una sequenza a

```
1: for i in range(len(a)) do
2:     esamina l'elemento a[i]
3: end for
```

Algoritmo 8 - Ricerca in una sequenza a

```
1: i ← 0
2: while (i < len(a)) ∧ serve esaminare altri elementi do
3:     esamina l'elemento a[i]
4:     i ← i + 1
5: end while
```

Osservazione. Oltre agli algoritmi 7 e 8, poiché una sequenza permette un accesso diretto ad un generico elemento di data posizione, sulle sequenze si possono realizzare altri e più efficienti algoritmi di ricerca che sfruttano delle proprietà aggiuntive della sequenza quali, ad esempio, l'ordinamento.

Ricerca in sequenze non ordinate

Nel caso di ricerca se in una sequenza non ordinata a è presente un elemento x, l'algoritmo 2 assume la forma riportata nell'algoritmo 9.

Algoritmo 9 - Ricerca di un elemento in una sequenza (non ordinata)

Input: sequenza a in cui cercare, elemento x da ricercare
Output: `TRUE` se e solo se x è presente in a

```
1: trovato ← FALSE
2: i ← 0
3: while (¬ trovato) ∧ (i < len(a)) do
4:     if a[i] = x then
5:         trovato ← TRUE
6:     else
7:         i ← i + 1
8:     end if
9: end while
10: return trovato
```

Nell'analisi della complessità degli algoritmi di ricerca si assume come dimensione del problema il numero n degli elementi della sequenza e come operazione dominante il confronto fra un elemento della sequenza con l'elemento da ricercare. Per l'algoritmo 9 il caso ottimo corrisponde alla situazione in cui l'elemento cercato si trovi all'inizio della sequenza, cioè alla posizione 0 (complessità asintotica uguale a $O(1)$) mentre il caso pessimo corrisponde al caso in cui l'elemento x non sia presente nella sequenza (oppure sia presente all'ultima posizione); in questo caso la complessità dell'algoritmo è pari a $O(n)$. Nell'ipotesi che gli elementi siano distribuiti in modo uniforme nella

sequenza, anche nel caso medio la complessità risulta pari a $O(n)$ (media delle complessità di tutti i casi possibili).

Ricerca in sequenze ordinate

Gli algoritmi di ricerca ed ordinamento sono fortemente connessi fra loro. Ne è conferma il fatto che nel caso di sequenze ordinate l'algoritmo di ricerca può essere migliorato sfruttando la proprietà di ordinamento degli elementi. La strategia consiste nell'analizzare l'elemento mediano della sequenza; l'esito di questo confronto permette di stabilire se l'elemento considerato sia uguale a quello cercato e, nel caso contrario, stabilire in quale delle due metà l'elemento cercato si possa eventualmente trovare; con un unico confronto viene scartata metà della sequenza.

Algoritmo 10 - Ricerca di un elemento in una sequenza ordinata

Input: sequenza a ordinata in cui cercare, elemento x da ricercare
Output: `TRUE`se e solo se x è presente in a

```
1:  primo ← 0
2:  ultimo ← len(a) − 1
3:  trovato ← FALSE
4:  while ¬ trovato ∧ (primo ≤ ultimo) do
5:     medio ← (primo + ultimo) div 2
6:     if a[medio] = x then
7:        trovato ← TRUE
8:     else if x < a[medio] then
9:        ultimo ← medio − 1
10:    else
11:       primo ← medio + 1
12:    end if
13: end while
14: return trovato
```

Per l'algoritmo 10 il caso ottimo si verifica quando il valore cercato è presente nella posizione centrale della sequenza. In questo caso la complessità asintotica è O(1) come per il caso ottimo dell'algoritmo di ricerca lineare. Nel caso pessimo lo spazio di ricerca viene ripetutamente diviso a metà fino a restare con un unico elemento da confrontare con il valore cercato. Il numero di iterazioni del ciclo risulta pari al numero di dimezzamenti che è uguale a $\lceil \log_2 n \rceil$; pertanto la complessità asintotica è $O(\log n)$.

8.6 Ricerca su matrici

Per ricercare dato elemento è presente in una matrice si può guardare alla matrice come ad un iterabile, scandendo gli elementi riga per riga. Si può adattare l'algoritmo 6 come descritto nell'algoritmo 11; le linee 7 – 12 dell'algoritmo 11 corrispondono alla linea 7 dell'algoritmo 6.

Algoritmo 11 - Ricerca se x è presente nella matrice a di dimensioni $n \times m$

Input: matrice a di dimensione $n \times m$, elemento x
Output: `TRUE` se e solo se x è presente in a

```
1:  (i, j) ← (0, 0)          ▷ posizione iniziale
2:  trovato ← FALSE
3:  while ¬ trovato ∧ (i < n) do
4:      if a[i][j] = x then
5:          trovato ← TRUE
6:      else
7:          if j = m then
8:              i ← i + 1
9:              j ← 0
10:         else
11:             j ← j + 1
12:         end if
13:     end if
14: end while
15: return trovato
```

8.7 Ricerca su alberi

La ricerca sugli alberi, data la loro specifica struttura ricorsiva, viene svolta mediante in modo ricorsivo.

Ricerca su alberi binari generici

L'algoritmo 12 che segue ricerca se in un albero è presente un dato elemento.

Algoritmo 12 - $ricercaAB(a, x)$ - Algoritmo di ricerca in un albero binario

Input: *albero binario* a, elemento x
Output: `TRUE` se e solo se x è presente in a

```
1: if isEmpty(a) then
2:     return FALSE
3: else if root(a) = x then
4:     return TRUE
5: else if ricercaAB(left(a), x) then
6:     return TRUE
7: else
8:     return ricercaAB(right(a), x)
9: end if
```

Ricerca su alberi binari di ricerca

L'obiettivo degli alberi binari di ricerca è quello di permettere degli algoritmi di ricerca con complessità logaritmica, non degradando al contempo la complessità dell'inserimento ordinato di nuovi elementi, mantenendo questa operazione ad una complessità logaritmica (gli array e le liste concatenate non permettono di mantenere entrambe queste performances sia per la ricerca che per l'inserimento). L'algoritmo 13 costituisce la tipica forma della ricerca in un *ABR*.

Algoritmo 13 - $ricercaABR(a, x)$ - Algoritmo di ricerca in un *ABR*

Input: *ABR* a, elemento x
Output: `TRUE` se e solo se x è presente in a

```
1: if isEmpty(a) then
2:     return FALSE
3: else if root(a) = x then
4:     return TRUE
5: else if root(a) > x then
6:     return ricercaABR(left(a), x)
7: else
8:     return ricercaABR(right(a), x)
9: end if
```

Se l'*ABR* a è perfettamente bilanciato, la complessità della ricerca è $O(\log n)$, dove n è il numero dei nodi; nel caso in cui l'*ABR* sia completamente sbilanciato, la complessità della ricerca è lineare, ossia pari a $O(n)$.

8.8 Applicazioni degli algoritmi di ricerca

Gli algoritmi di ricerca presentano spesso delle varianti rispetto ai casi precedentemente analizzati e talvolta compaiono all'interno ed a supporto di altri algoritmi. Ne sono conferma gli esempi che seguono.

Problema 8.8.1 Determinare quante volte un dato elemento è presente in una sequenza.

Soluzione. Il seguente algoritmo risolve il problema:

Algoritmo 14 - *frequenza*(a, x) : frequenza di x in a

Input: sequenza a, elemento x
Output: numero di volte che x è presente a

```
1: for e in a do
2:     if e = x then
3:         k ← k + 1
4:     end if
5: end for
6: return k
```

Problema 8.8.2 Determinare la frequenza massima degli elementi della sequenza, ossia la frequenza dell'elemento che compare più volte nella sequenza.

Soluzione. Il seguente algoritmo risolve il problema:

Algoritmo 15 - Calcolo della frequenza massima in una sequenza

Input: sequenza a
Output: massima frequenza degli elementi di a

```
1: fmax ← 0
2: for i from 0 to len(a) − 1 do
3:     f ← frequenza di a[i]
4:     if f > fmax then
5:         fmax ← f
6:     end if
7: end for
8: return fmax
```

La complessità della funzione *frequenza* è n; essendo che questa funzione viene richiamata n volte all'interno del ciclo *for*, si conclude che l'algoritmo ha complessità asintotica pari a $O(n^2)$. Si lascia per esercizio la valutazione della complessità del problema. □

Problema 8.8.3 Stabilire se in una sequenza un dato valore è presente almeno k volte.

Soluzione. Il seguente algoritmo risolve il problema:

Algoritmo 16 - Decisione se in una sequenza un elemento è presente almeno k volte

Input: sequenza a, valore x, numero naturale k
Output: `TRUE` se e solo x è presente in a almeno k volte

```
1: i ← 0
2: while (k > 0) ∧ (i < len(a)) ∧ (k ≤ (len(a) − i)) do
3:     if a[i] = x then
4:         k ← k − 1
5:     end if
6:     i ← i + 1
7: end while
8: return k = 0
```

□

Problema 8.8.4 Decidere se una sequenza è composta da elementi tutti distinti.

Soluzione. Una soluzione di questo problema può basarsi sulla seguente idea: *Confrontare ciascun elemento con tutti gli elementi che lo precedono.* L'algoritmo 17 esplicita questa idea.

Algoritmo 17 - Decisione se una sequenza è composta da elementi distinti

Input: sequenza a
Output: `TRUE` se e solo se ($a[i] \neq a[j]$ per ogni i, j $(i \neq j)$)

```
1: distinti ← TRUE
2: i ← 1
3: while distinti ∧ (i < len(a)) do
4:     ricerca se a[i] è presente nella porzione di sequenza a[0..i − 1]
5:     if è presente then
6:         distinti ← FALSE
7:     else
8:         i ← i + 1
9:     end if
10: end while
11: return distinti
```

Il problema di ricerca della linea 4 può essere risolto efficientemente adottando una ricerca con sentinella sulla porzione di sequenza $a[0..i]$. Nel caso pessimo (tutti gli elementi sono distinti fra loro) la complessità dell'algoritmo è pari a

$$T(n) = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} \in O(n^2)$$

dove il primo addendo corrisponde al confronto fra $a[1]$ e $a[0]$ e l'ultimo addendo corrisponde al numero di confronti richiesti per controllare che l'ultimo elemento non sia presente nella sottosequenza degli $n-1$ confronti che precedono.
□

ESERCIZI

8.1 Da un mazzo di carte completo viene estratta una carta. Analizzando il mazzo dopo che è stata estratta la carta, individuare la carta che è stata estratta.

8.2 Decidere se in una sequenza è presente un dato elemento nelle posizioni comprese fra due dati indici naturali p e q ($p \leq q$).

8.3 Determinare la posizione in cui compare la prima volta un dato elemento in una sequenza ed in una sequenza ordinata. Valutare la complessità degli algoritmi.

8.4 Determinare le posizioni in cui un dato elemento è presente in una sequenza ed in una sequenza ordinata. Valutare la complessità degli algoritmi.

8.5 Determinare quante volte un dato elemento è presente in una sequenza ordinata. Valutare la complessità dell'algoritmo.

8.6 Data una sequenza ordinata di elementi comparabili, determinare gli elementi compresi fra due dati valori.

8.7 Date due sequenze, determinare il più piccolo elemento presente in entrambe. Valutare la complessità dell'algoritmo.

8.8 Determinare il valore dell'elemento mediano in una sequenza numerica di dimensione dispari. L'elemento mediano è il valore che in un ipotetico ordinamento della sequenza viene a trovarsi nella posizione mediana.

8.9 Una sequenza di dimensione n contiene tutti i numeri (distinti) compresi fra 0 e 1, ad esclusione di uno. Determinare il numero mancante. Risolvere il problema anche nel caso in cui la sequenza sia ordinata.

8.10 Generalizzare l'algoritmo 16 al caso di un generico iterabile. Ottimizzare l'algoritmo terminando la ricerca quando si riconosce che gli elementi ancora da esaminare non consentono di arrivare alla frequenza k.

8.11 Individuare delle possibili strategie di ottimizzazione dell'algoritmo 15 e tradurle in un algoritmo. Analizzare le difficoltà che possono insorgere generalizzando l'algoritmo al caso di un generico iterabile.

8.12 Spiegare cosa si intende con la frase *L'algoritmo di ricerca binaria in una sequenza ordinata ha complessità asintotica logaritmica.* Motivare perchè l'algoritmo di ricerca binaria su una sequenza ordinata ha complessità asintotica pari a $O(\log n)$.

8.13 Discutere ed evidenziare, avvalendosi anche di opportuni esempi, alcuni nessi logici ed operativi esistenti fra i problemi di ordinamento ed i problemi di ricerca.

8.14 Determinare la posizione dove in una sequenza compare per la prima volta un dato valore (-1 se l'elemento non è presente).

8.15 Determinare tutte le posizioni in cui in una sequenza è presente un dato elemento.

8.16 Stabilire se in una sequenza un dato elemento è presente almeno un dato

numero di volte. Risolvere il problema nei seguenti casi:

1. la sequenza non è ordinata
2. la sequenza è ordinata

Stabilire la complessità degli algoritmi, motivando le affermazioni.

8.17 Decidere se in una struttura è presente un dato elemento.

8.18 Determinare quante volte un dato elemento è presente in una struttura.

8.19 Decidere se un dato valore è presente in una lista. Distinguere i due casi di *lista semplice* e *lista composta*.

8.20 Determinare quante volte è presente un dato valore in una lista. Distinguere i due casi di *lista semplice* e *lista composta*.

8.21 Ricercare se e dove è presente un dato valore in una data matrice.

8.22 Determinare la posizione in cui compare per l'ultima volta, in una scansione per righe, un dato valore in una matrice.

8.23 Determinare il massimo elemento presente in una matrice.

8.24 Determinare quante volte un dato elemento è presente in una matrice.

8.25 Determinare quante volte un dato elemento è presente in un *albero binario*.

8.26 Determinare la posizione in cui compare per la prima volta un dato elemento

1. in una sequenza
2. in una matrice

8.27 Determinare le posizioni in cui compare un dato elemento in

1. in una sequenza
2. in una matrice

8.28 Descrivere un *albero binario di ricerca* contenente i primi 20 numeri naturali. Visitare l'albero in postordine. Rappresentare l'albero mediante un array.

8.29 Decidere se un dato elemento è presente in un albero binario.

8.30 Determinare quante volte è presente un dato elemento in un albero binario.

8.31 Determinare il massimo valore presente in un albero binario non vuoto.

8.32 Determinare il massimo numeri di confronti necessario per determinare se in un ABR perfettamente bilanciato composto da n nodi è presente un dato elemento.

8.33 Descrivere graficamente un ABR i cui nodi sono costituiti dai caratteri della parola `ALGORITMO`. Evidenziare graficamente i cammini di ricerca dei caratteri `T` e `Q`. Rappresentare l'albero mediante un array.

8.34 Decidere se in un ABR è presente un dato elemento.

8.35 Determinare quante volte un dato valore è presente in un ABR.

8.36 Studiare la complessità della ricerca in un ABR in funzione del numero n dei nodi, dell'altezza h dell'albero e del grado di bilanciamento. Distinguere i casi ottimo, medio e pessimo.

8.37 Determinare quante volte in un ABR è presente un dato elemento.

8.38 Determinare l'elemento più grande presente in un ABR.

8.39 Determinare in un ABR il numero di elementi compresi fra due dati estremi.

8.40 Discutere la complessità della ricerca in un ABR, in funzione del numero n dei nodi e del grado di bilanciamento dell'albero.

8.41 Descrivere graficamente un ABR, il più bilanciato possibile, i cui nodi sono costituiti dai primi 10 caratteri del proprio cognome e nome. Evidenziare graficamente il cammino di ricerca del carattere `Q`. Rappresentare l'albero mediante un array. Visitare l'albero in preordine.

8.42 Sfruttando le seguenti proprietà su un ABR:

- Il minimo di un ABR è l'ultimo nodo del cammino più a sinistra che parte dalla radice.
- Il massimo di un ABR è l'ultimo nodo del cammino più a destra che parte dalla radice
- Il successore di un nodo x, se esiste, è il minimo del sottoalbero di destra del nodo x, se tale albero non è vuoto; altrimenti è il più basso antenato di x il cui figlio sinistro è un antenato di x.
- Il predecessore di un nodo x, se esiste, è il massimo del sottoalbero di sinistra del nodo x, se tale albero non è vuoto; altrimenti è il più basso antenato di x il cui figlio destro è un antenato di x.

determinare il minimo, il massimo, il successore ed il successore di un valore x, in un albero binario di ricerca.

Parte III

OGGETTI

9

CLASSIFICARE

La classificazione è lo strumento mediante il quale noi organizziamo la conoscenza.
G.Booch,
Object-Oriented Analysis and Design

In un mondo popolato da moltissime e variegate entità, un primo passo per mettere ordine ed agevolare l'analisi e la comprensione consiste nel suddividere queste entità in classi costituite da elementi che hanno delle caratteristiche comuni. Ad esempio, considerando le automobili che vediamo circolare sulle strade, perveniamo al concetto di *automobile*; considerando gli insiemi che possono essere messi in corrispondenza biunivoca fra loro, perveniamo al concetto di *numero naturale*.

Classificare significa operare un atto di astrazione che individua delle qualità comuni degli elementi presi in considerazione. Ad esempio, nell'insieme delle figure geometriche del piano, si può evidenziare la classe dei *triangoli* costituita dalle figure aventi la caratteristica comune di *avere tre lati*. Il processo di classificazione può essere iterato evidenziando delle ulteriori classi più raffinate, quali ad esempio i *triangoli equilateri* aventi la caratterizzazione aggiuntiva di avere *tutti i lati uguali*.

Questo capitolo è incentrato attorno al concetto di *classe* e di *oggetto*. Questi concetti costituiscono un potente strumento mentale e risultano trasversali a molte argomentazioni dell'informatica quali, ad esempio, i tipi di dato, le strutture dati, la programmazione orientata agli oggetti, le basi di dati.

9.1 Classi ed oggetti

Un'***entità*** è un elemento (*istanza*) di un dato insieme, distinguibile dagli altri elementi dell'insieme per delle caratteristiche proprie. Un insieme di entità, dotate di caratteristiche omogenee viene detto ***classe***; un elemento di una classe viene detto ***oggetto*** (della classe).

Una classe è un'astrazione che specifica la comune struttura di un insieme di oggetti con proprietà simili e comportamenti comuni; nella programmazione orientata agli oggetti ciò avviene definendo le operazioni (metodi) applicabili agli oggetti, ossia i *metodi* che permettono di interagire con gli oggetti. Una classe definisce anche degli specifici metodi, detti *costruttori*, caratterizzati da un identificativo uguale al nome della classe mediante i quali si generano gli oggetti. Una classe definisce anche la costituzione interna degli oggetti, precisando i dati interni, detti *attributi*, che costituiscono ciascun oggetto.

Un oggetto si dice *mutabile* se, una volta creato, può essere modificato con delle successive istruzioni; altrimenti l'oggetto si dice *immutabile*. Con il termine *ambito di visibilità* (*scope*) di un oggetto si denota la porzione di programma nella quale l'oggetto è visibile ed utilizzabile. Generalmente l'ambito di visibilità di un oggetto è costituito dal blocco nel quale l'oggetto è definito. Gli oggetti visibili possono esibire diversi gradi di accessibilità che dipendono da chi, da dove e da quando si accede:

- *read-only*: l'entità è accessibile, ma non modificabile
- *read-write*: l'entità è accessibile e modificabile
- *blocked*: l'entità non è momentaneamente accessibile (a causa di un contemporaneo accesso da parte di un altro processo; tale situazione si verifica nella gestione di entità condivise da parte di più processi concorrenti)

Con il termine *tempo di vita* (*lifetime*) di un pggetto si denota il tempo che intercorre dal momento della creazione dell'oggetto al momento in cui viene distrutto. Spesso il tempo di vita di un oggetto corrisponde all'intervallo di tempo nel quale viene eseguito il blocco nel quale l'oggetto è definito.

9.2 Identificare gli oggetti

In un contesto formalizzato un oggetto di una classe risulta univocamente individuato, all'interno della classe, dai valori assunti da un insieme di ***attributi***. Per indicare che una classe $\mathcal{C}$ di entità è caratterizzata da una sequenza di n attributi $k_1, \ldots, k_n$, scriveremo

$$\mathcal{C}(k_1, \ldots, k_n)$$

Nell'ambito della programmazione ad oggetti, $\mathcal{C}(k_1, \ldots, k_n)$ è un costruttore che costruisce (in memoria) una generica entità della classe; usando tale costruttore con degli specifici valori si ottiene la costruzione di un'istanza della classe. Avendo a disposizione gli attributi delle entità di una data classe, si hanno a disposizione tutti gli elementi per risolvere i problemi coinvolgenti le entità di quella classe.

Esempio 9.2.1 - Un poligono regolare è univocamente individuato dal numero n dei suoi lati e dalla lunghezza l dei lati. Per denotare la classe dei poligoni regolari, avente come parametri caratteristici il numero n di lati e la lunghezza l del lato, scriveremo

$$Poligono(n, l)$$

Istanziando con degli specifici valori gli attributi si ottiene una specifica istanza della classe. Ad esempio, $Poligono(4, 8)$ denota il quadrato di lato 8.

Esempio 9.2.2 - Una spirale può essere identificata mediante la seguente lista di parametri caratteristici:

- lunghezza l del lato iniziale
- incremento d del lato ad ogni tratto
- angolo interno a fra due tratti consecutivi
- numero n dei lati della spirale

Tale classe di figure può essere denotata con $Spirale(l, d, a, n)$.

L'insieme degli attributi di una classe di oggetti è, in generale, non univoco, come evidenzia il seguente esempio.

Esempio 9.2.3 - Una frazione $\frac{m}{n}$ è individuata dalla coppia (m, n) di numeri naturali formata dal numeratore m e dal numeratore n. In questo caso l'uguaglianza fisica di due coppie non coincide con l'equivalenza numerica delle due frazioni.

Esempio 9.2.4 - Un punto del piano cartesiano può essere univocamente individuato dalla coppia di valori (x, y) delle sue coordinate cartesiane oppure dalla coppia (ρ, θ) delle sue coordinate polari.

Esempio 9.2.5 - Un'equazione di secondo grado $ax^2 + bx + c = 0$ risulta univocamente individuata dalla terna di valori (a, b, c).

Esempio 9.2.6 - Una retta del piano cartesiano può essere identificata da un'equazione della forma $ax + by + c = 0$ e quindi, in definitiva, da una terna di valori (a, b, c), anche se tale corrispondenza non è biunivoca.

Osservazione. Il fatto che, ad esempio, come visto negli esempi precedenti, un oggetto possa essere rappresentato da strutture diverse diverse e che stesse strutture possano rappresentare oggetti diversi e che siano necessari controlli di comparazione ed operazioni sulle strutture che dipendono dall'oggetto che esse rappresentano lascia presagire, come in effetti è, che un contesto caratterizzato dai soli attributi risulta incompleto in quanto mancante della dimensione semantica che viene precisata definendo delle operazioni e le loro proprietà; si perviene, in questo modo, al concetto di *tipo di dato astratto*. Imponendo poi delle regole (assiomi) sulle operazioni si perviene al concetto di *struttura (algebrica)*.

9.3 Programmare ad oggetti

La *programmazione orientata agli oggetti* (*Object Oriented Programming*, OOP) costituisce un approccio industriale allo sviluppo del software. L'idea di questo approccio risale a più di mezzo secolo fa quando, nel 1968, D. McIlroy, in un workshop della NATO sulla crisi del software, nel 1968 diceva:

> *Perché il software non è più simile all'hardware? Perché ogni nuovo sviluppo deve ripartire da zero? Ci dovrebbero essere cataloghi di moduli software, come ci sono cataloghi di dispositivi VLSI: quando si costruisce un sistema nuovo, si dovrebbero ordinare e combinare insieme componenti trovati su questi cataloghi, piuttosto che inventare di nuovo ogni volta la ruota. Si dovrebbe scrivere meno software, e forse lavorare meglio su quello che deve essere comunque sviluppato. Non si risolverebbero così alcuni dei problemi di cui tutti si lamentano, gli alti costi, il mancato rispetto dei tempi, la mancanza di affidabilità? Perché non è così?*

Similmente a molti prodotti industriali lo sviluppo di oggetti software avviene attraverso fasi ben definite:

1. individuazione delle classi di oggetti e delle loro mutue relazioni
2. progettazione delle classi
3. implementazione delle classi
4. utilizzo delle oggetti generati usando le classi

La OOP è una metodologia di programmazione caratterizzata dalla definizione di *classi* mediante le quali si costruiscono *oggetti* che costituiscono *sistemi di oggetti*. Un'*applicazione* può essere vista come un sistema di oggetti che interagiscono fra loro. Le classi costituiscono il concetto base della programmazione orientata agli oggetti; a macro-livello le classi denotano le entità di interesse per il dominio dell'applicazione che si intende sviluppare ed assolvono alle seguenti due funzionalità:

- creare oggetti, manipolabili richiamando su di loro dei *metodi d'istanza*
- offrire servizi, incapsulando dei *metodi di classe* richiamabili sulla classe stessa

Operativamente, nel contesto di un linguaggio di programmazione, avendo a disposizione una classe di oggetti, si sviluppano dei programmi attraverso le seguenti fasi:

1. dichiarazione dei nomi identificativi degli oggetti
2. creazione degli oggetti mediante i costruttori della classe
3. applicazione dei metodi agli oggetti creati

Esempio 9.3.1 - Con riferimento ad una classe `Robot` i cui oggetti sono dei robot virtuali (che si muovono sul video) o reali (che si muovono in uno spazio fisico) le fasi sopra elencate si scrivono come segue:

```
Robot r;         // dichiarazione di un riferimento r
r = new Robot(); // creazione di un oggetto r di classe Robot
r.avanza(10);    // metodo d'istanza rivolto all'oggetto r
```

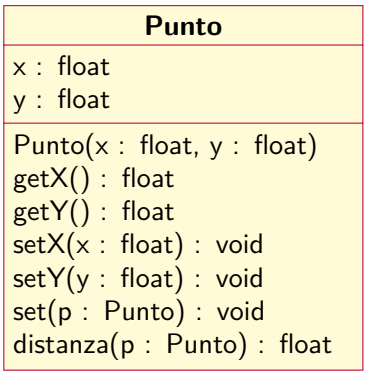
9.4 Descrivere le classi di oggetti: la notazione UML

Una classe di oggetti è definita dalle seguenti caratteristiche:

- *identità* : *nome* identificativo della classe
- *stato* : descritto dai nomi identificativi degli *attributi* di ciascun oggetto e dal tipo dei valori assumibili dagli attributi
- *comportamento* : descritto mediante i *metodi*, che accedono o cambiano il valore degli attributi di ciascun oggetto

Nella notazione UML (*Unified Modeling Language*) una classe viene descritta mediante un diagramma (*diagramma della classe*) che descrive gli oggetti in termini di attributi ed operazioni. Un diagramma della classe è costituito da un rettangolo suddiviso in 3 sezioni: la prima (obbligatoria) denota il nome della classe, la seconda descrive la lista degli attributi di ciascun oggetto, la terza descrive la lista delle operazioni (metodi) applicabili a ciascun oggetto. In questa sezione vengono specificati anche i costruttori degli oggetti. Se non viene alcun costruttore, si suppone che esista comunque un costruttore senza argomenti. Nel caso in cui sia presente solamente la prima e terza sezione si parla anche di *diagramma dell'interfaccia*. Il livello di dettaglio con cui viene descritto un diagramma della classe dipende dal livello delle informazioni che si vogliono descrivere e dallo stadio di avanzamento della progettazione verso l'implementazione. Ad esempio, può essere omessa la lista degli attributi, i metodi costruttori ed il tipo del risultato ritornato dai metodi; nella forma più ristretta è presente solo la sezione che riporta il nome della classe. La sintassi utilizzata può essere orientata verso uno specifico linguaggio di programmazione, in modo da essere aderente alla specifica sintassi di un linguaggio di programmazione ad oggetti.

Esempio 9.4.1 - A seguire è riportato il diagramma della classe dei punti del piano cartesiano individuati dalle loro coordinate cartesiane.



In notazione UML un diagramma della classe può essere commentato mediante un'*annotazione* come la seguente:

x : *coordinata x del punto*
y : *coordinata y del punto*
.....
Punto : *costruttore di un punto di coordinate* (x,y)
getX : *coordinata x del punto*
getY : *coordinata y del punto*
setX : *assegna* x alla coordinata x *del punto*
setY : *assegna* y alla coordinata y *del punto*
set : *assegnazione fra punti*
distanza : *distanza dal punto* p

9.5 Progettare una classe di oggetti

La progettazione di una classe costituisce la fase che nella terminologia dell'impostazione orientata agli oggetti viene definita come *Object Oriented Design* (OOD); consiste nell'individuare il comportamento degli oggetti, specificando i metodi mediante i quali manipolare ed interagire con gli oggetti della classe. Questa lista di metodi viene detta *interfaccia* e costituisce la descrizione esterna delle caratteristiche costitutive e di comportamento degli oggetti di una classe. Nel contesto dei linguaggi di programmazione *definire* un'interfaccia significa descrivere le intestazioni dei metodi degli oggetti di una classe, senza realizzarli.

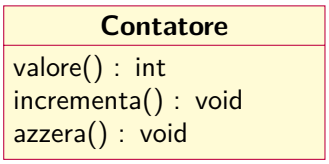
Interfaccia degli oggetti

Il termine *interfaccia* denota l'insieme delle operazioni (metodi) applicabili agli oggetti. Da un punto di vista pratico, un'interfaccia permette di separare la fase di progetto dalla fase dell'implementazione: una modifica all'implementazione non modifica le modalità d'uso degli oggetti, consentendo di modificare l'implementazione senza dover modificare il codice già scritto dove vengono utilizzati gli oggetti: un'applicazione usa un'interfaccia costante per manipolare oggetti con diverse implementazioni. Un'interfaccia specifica non solo le regole sintattiche della chiamata dei metodi, ma precisa anche la loro semantica; non pone però alcun vincolo a come dovranno essere realizzati i metodi.

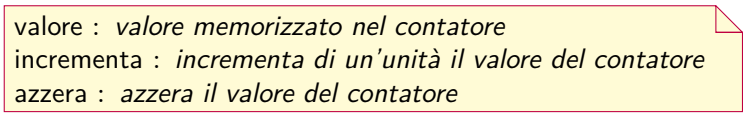
Con la notazione UML già utilizzata per definire un diagramma di una classe, un'interfaccia viene descritta mediante un diagramma (*diagramma dell'interfaccia*) composto da due sezioni: una riporta il nome della classe e l'altra descrive la lista dei metodi.

Osservazione. In molti linguaggi di programmazione (Java ed altri) nelle interfacce non vengono specificati i costruttori in quanto un'interfaccia descrive solamente il comportamento di una categoria di oggetti appartenenti anche a più classi. In questi casi si può pensare di definire, in alternativa, un metodo *init* (da richiamarsi una sola volta all'inizio, dopo la creazione dell'oggetto) che inizializza l'oggetto creato mediante specificati ingressi.

Esempio 9.5.1 - Un *contatore* è un oggetto in grado di memorizzare un numero naturale; è possibile accedere al valore memorizzato e modificarlo incrementandolo di un'unità; inoltre, un contatore può essere azzerato. Un diagramma di un'interfaccia minimale che descrive un contatore è riportato a seguire.



Nel caso in cui il diagramma dell'interfaccia non fosse sufficientemente autoesplicativo, è possibile aggiungere ulteriori informazioni documentative di ciascun metodo mediante un'annotazione, come descritto nella figura che segue.



Codifica delle interfacce

I vari linguaggi di programmazione prevedono specifici meccanismi sintattici per codificare un'interfaccia. In linguaggio Java l'interfaccia `Contatore` precedentemente descritta viene codificata come segue [1] :

Java

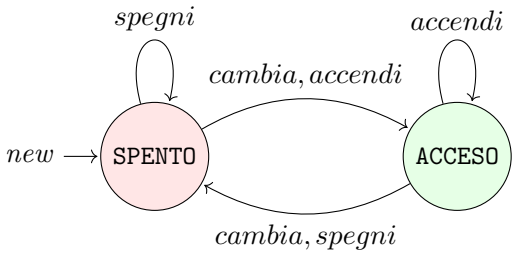
```
interface Contatore_
{
    int valore();
    void incrementa();
    void azzera();
}
```

Esempi di interfacce

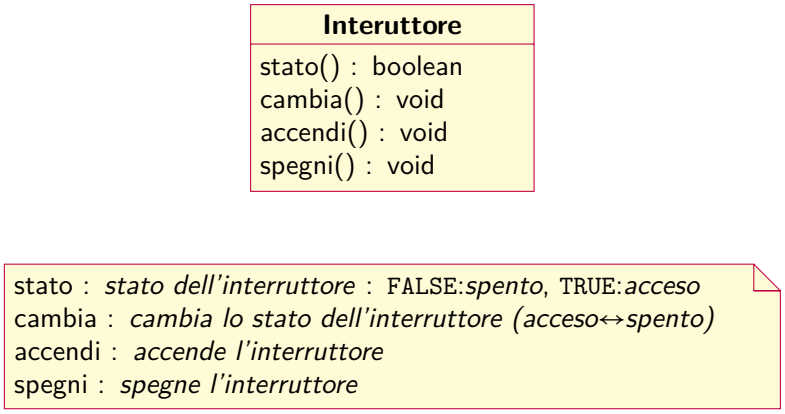
In molti casi la definizione dell'interfaccia degli oggetti è guidata dalle caratteristiche e dal comportamento degli oggetti fisici del mondo reale.

[1] Viene usato l'identificatore `Contatore_` per lasciare libero l'identificatore `Contatore` che verrà utilizzato come nome della classe che implementerà l'interfaccia. Nel linguaggio Java non è permesso specificare i costruttori all'interno di un'interfaccia.

Esempio 9.5.2 - Gli *interruttori* sono degli oggetti dotati ciascuno di un proprio stato interno che può assumere uno dei seguenti valori: *acceso* o *spento*. Gli interruttori sono oggetti osservabili e modificabili. La vita di un interruttore può essere descritta mediante il seguente grafo di transizione di stato di evidente interpretazione:



Tutte le precedenti specifiche possono essere riassunte mediante il seguente diagramma d'interfaccia con annotazione.



Esempio 9.5.3 - Un *dado* è un oggetto che può essere *lanciato* e *guardato* per vedere in numero (casuale fra 1 e 6) uscito. Richiamandosi alla fisicità ed alle modalità d'uso di un dado fisico, il comportamento di un dado può essere descritto mediante la figura 9.1.

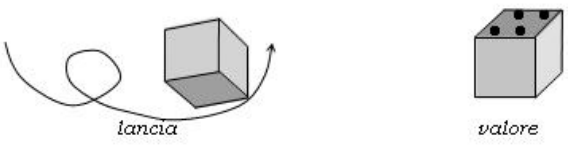


Figura 9.1: Schema di un *dado*.

Dato che si sta progettando un oggetto virtuale, costa poco generalizzare ad un dado ad n facce, con valori da 1 ad n. Un'interfaccia che descrive un dado può essere la seguente:

Dado
Dado(n : int) lancia() : void valore() : int

Dado : *costruttore di un dado con* n facce
lancia : *esegue il lancio del dado*
valore : *numero presente sulla faccia superiore del dado*

Esempio 9.5.4 - Un'*urna* è un contenitore di numeri naturali, da 1 ad un prefissato numero n, che possono essere *estratti* e *reimbussolati*.

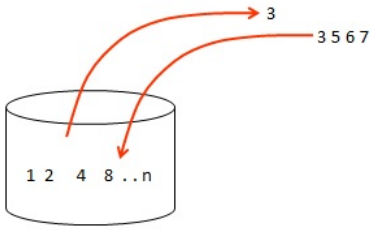


Figura 9.2: Schema di un'urna.

Questi contenitori devono poter essere utilizzati per estrarre numeri per giocare a tombola, per generare un tabellone del lotto, per gestire i premi che vengono vinti ad una sagra paesana, ...". Lasciandoci guidare dalle funzionalità dell'oggetto reale "urna", si può descrivere questo oggetto mediante il seguente diagramma dell'interfaccia e corrispondente annotazione:

Urna
Urna(n : int) isVuota() : boolean estraiNumero() : int reimbussolaUltimo() : void reimbussolaTutti() : void

Urna : *costruttore di un'urna contenente i numeri da* 1 *a* n
isVuota : *controllo se l'urna è vuota*
estraiNumero : *estrae un numero (a caso) fra quelli presenti*
reimbussolaUltimo : *reimbussola l'ultimo numero estratto*
reimbussolaTutti : *reimbussola tutti i numeri e ripristina la situazione iniziale*

Osservazione. La specifica dell'intervallo di numeri che devono essere presenti inizialmente in un'urna è bene che venga fatta dal costruttore dell'urna; una volta creata un'urna con un dato intervallo di numeri, essa conterrà sempre numeri in quel dato intervallo; se servono altri numeri si dovrà creare un'altra urna.

Osservazione. La potenza delle interfacce risiede nel fatto che consentono di generalizzare un problema e di codificare un algoritmo prima che sia stata pensata una implementazione e quindi di scrivere codice che funziona con tutte le possibili implementazioni ed adattarsi a diverse implementazioni contemporaneamente.

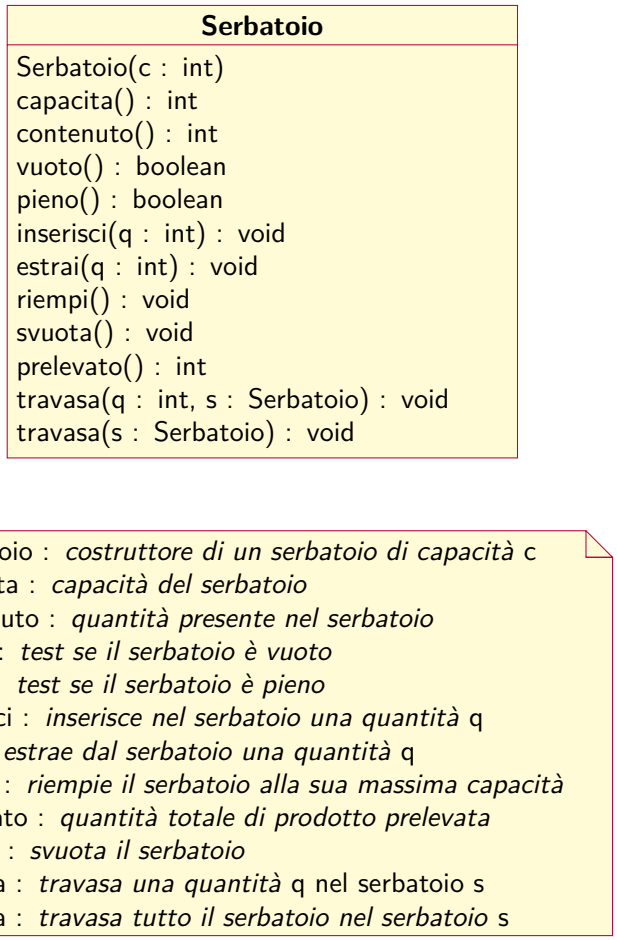
Esempio 9.5.5 - Estrarre da un'urna un numero fino a che si estrae un dato numero k o si svuota l'urna.

Algoritmo 1 - Estrazione di un dato numero k da un'urna

```
1: if isVuota() then
2:     estratto ← FALSE
3: else
4:     estratto ← estraiNumero() = k
5: end if
6: while ¬estratto ∧ ¬isVuota() do
7:     estratto ← estraiNumero() = k
8: end while
9: { è stato estratto k oppure l'urna è vuota }
```

Problema 9.5.1 Descrivere un diagramma di interfaccia che descrive dei *serbatoi* contenenti una certa quantità di prodotto. Ogni serbatoio ha una capienza massima (positiva) fissata al momento della creazione del serbatoio stesso. Da un serbatoio non è possibile estrarre una quantità di prodotto superiore a quella presente in esso, né è possibile inserire una quantità di prodotto tale da superare la massima capienza consentita. L'estrazione e l'inserimento di una data quantità di prodotto modificano la quantità di prodotto presente nel serbatoio. Su un serbatoio si possono eseguire le seguenti operazioni: estrarre una data quantità di prodotto, riempire il serbatoio, svuotare il serbatoio, inserire una data quantità di prodotto, conoscere la quantità di prodotto ancora presente, indagare se il serbatoio è vuoto. Deve essere inoltre possibile eseguire operazioni di spostamento di prodotti fra due serbatoi.

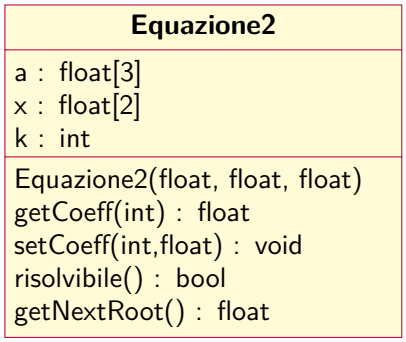
Soluzione. Le precedenti specifiche possono essere sintetizzate mediante il seguente diagramma dell'interfaccia con annotazione:

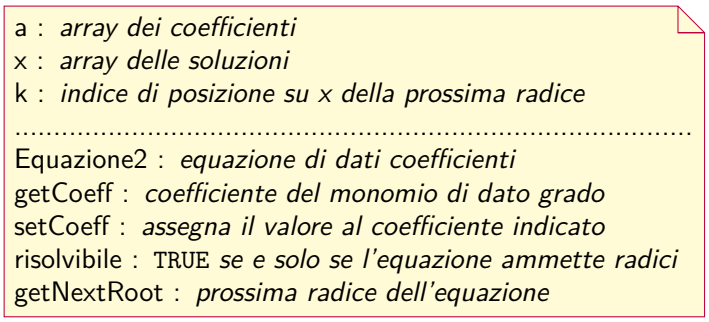


Alcune delle operazioni sopra descritte devono essere svolte compatibilmente con i vincoli di un serbatoio fisico; ad esempio non si può estrarre più contenuto di quelle presente, oppure non si può inserire più contenuto di quanto ammissibile dalla capacità del serbatoio. Tutti questi vincoli non deveno essere controllati da un utente del serbatoio ma devono essere gestiti dai metodi stessi. Per informare l'utente dell'esito, un metodo *setter* può ritornare un valore che informa su come l'operazione è stata eseguita; nel caso specifico di un serbatoio le operazioni *riempi*, *inserisci*, *estrai* e *svuota* ritornano un numero intero per indicare quante unità di contenuto sono state effettivamente inserite o estratte. Similmente, le operazioni *travasa* possono ritornano un numero intero che indica la quantità di prodotto effettivamante travasate □

Problema 9.5.2 Descrivere un diagramma di classe i cui oggetti sono le *equazioni di secondo grado* della forma $ax^2 + bx + c = 0$.

Soluzione. Un'analisi sommaria sulla forma di un'equazione di secondo grado indica che essa è individuata dalla terna (a, b, c) dei valori dei suoi coefficienti (il nome x dell'incognita risulta ininfluente, in quanto si tratta di equazioni ad una sola incognita). Anche se questa osservazione non è vincolante per la scelta degli attributi interni, ci suggerisce che potrebbe risultare utile un costruttore con tre argomenti di tipo `float`; ad esempio il costruttore `Equazione2(4.,0.,-1.)` genera l'equazione $4x^2 - 1 = 0$. Parallelamente ai tre argomenti del costruttore, risulta spontaneo predisporre tre metodi *getter* per accedere a questi tre valori di un'equazione, dopo che essa è stata costruita. Potremmo pensare a tre metodi della forma `getA()`, `getB()`, `getC()`; per non avere una proliferazione di troppi metodi di questo tipo, risulta più sintetico predisporre un unico metodo *getter* della forma `getCoeff(t)` dove l'argomento `t`, al momento non meglio precisato, servirà per precisare a quale dei tre argomenti (a, b, c) dell'equazione si intende accedere; questo argomento `t` di specifica potrebbe essere un carattere (ad esempio, `'a'`, `'b'`, `'c'`) oppure un numero naturale (ad esempio, $1, 2, 3$) per indicare a quale dei tre coefficienti; in vista di una eventuale generalizzazione per passare da equazioni di secondo grado ad equazioni di grado generico, il metodo più indicato per accedere ad un generico coefficiente dell'equazione consiste nel predisporre un unico metodo della forma `getCoeff(g)` dove l'argomento g è uno dei numeri naturali $0, 1, 2$ che indica il grado del monomio di cui si vuole conoscere il coefficiente. Ad esempio: `Equazione2(3.,0.,-1.).getCoeff(2)` fornisce come risultato il numero 3.. Questa modalità di interpretazione dell'argomento di `getCoeff` permette di gestire in modo robusto e trasparente all'utente l'eventuale indicazione di un argomento attuale incompatibile con le equazioni di secondo grado; ad esempio `Equazione2(3.,0.,-1.).getCoeff(5)` darà come risultato il valore 0. in quanto il coefficiente di x^4 di un'equazione di secondo grado vale 0, in quanto tale monomio non esiste. Nel caso in cui si desideri che la classe `Equazione2` si di tipo "mutabile", un discorso del tutto analogo può essere fatto per predisporre dei metodi *setter* per cambiare selettivamente un coefficiente dell'equazione. Tutte le considerazioni svolte fin qui caratterizzano una generico oggetto *terna di valori*; ma un'*equazione di secondo grado* è qualcosa in più, in quanto è caratterizzata dal fatto che è possibile estrarne le radici (ossia i valori numerici che la soddisfano). La parte relativa alle radici può essere gestita mediante un unico metodo `getNextRoot()` che, rivolto ad un'equazione, fornisce la prossima radice dell'equazione, in modo ciclico, come se un'equazione fosse un generatore di radici ed il metodo `getNextRoot` fosse un iteratore che fornisce ciclicamente i valori delle radici dell'equazione. Tutta l'analisi sopra svolta viene condensata nel seguente diagramma di classe.





□

9.6 Implementare

In ambito industriale l'implementazione è il passaggio dal progetto alla realizzazione del prodotto finito. Nella programmazione orientata agli oggetti l'implementazione è una particolare relazione fra due entità: una descrive le responsabilità e l'altra le realizza; corrisponde alla situazione in cui una classe implementa un'interfaccia.

Implementazione di una interfaccia

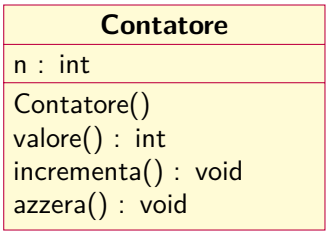
La realizzazione di una classe basata sulla specifica di un'interfaccia avviene implementando l'interfaccia. *Implementare* un'interfaccia significa realizzare concretamente i metodi specificati nell'interfaccia. L'implementazione di una classe consiste nella programmazione (in un linguaggio di programmazione orientato agli oggetti) dei metodi specificati nell'interfaccia. In sostanza si tratta di svolgere le seguenti due fasi:

1. definizione degli attributi di ciascun oggetto
2. codifica dei metodi specificati nell'interfaccia

Alcuni dettagli dell'implementazione possono essere descritti in modo indipendente dal linguaggio di programmazione che verrà utilizzato. In particolare un aspetto che riguarda l'implementazione consiste nell'individuazione degli attributi che serviranno per l'implementazione dei metodi. Queste informazioni possono essere descritte mediante un *diagramma della classe*. Un diagramma

della classe descrive una situazione intermedia fra l'interfaccia e la sua implementazione: definisce infatti la specifica dei metodi, dei costruttori e definisce gli attributi interni a ciascun oggetto.

Esempio 9.6.1 - Ragionando in ottica dell'implementazione risulta immediato convincersi che al suo interno un *Contatore* dovrà avere un attributo $(n : int)$ in cui viene memorizzato il valore del contatore. La creazione di un oggetto di classe *Contatore* viene svolta mediante un costruttore (*Contatore*()) che provvede a inizializzare a zero l'attributo l'attributo n. Tutte queste considerazioni possono essere descritte mediante il seguente diagramma della classe:



Implementazione nei linguaggi di programmazione

A seguire è riportata l'implementazione in Java della classe *Contatore* che implementa l'interfaccia definita nell'esempio 9.5.1.

Java

```
public class Contatore implements Contatore_
{
    private int v;

    Contatore() {
        v = 0;
    } // [c] Contatore

    public int valore() {
        return v;
    } // [m] valore

    public void incrementa() {
        v++;
    } // [m] incrementa

    public void azzera() {
        v = 0;
    } // [m] azzera
} // [class] Contatore
```

Coerentemente al principio di nascondimento delle informazioni (*information hiding*), l'attributo v è stato specificato con il qualificatore di visibilità *private* in modo da prevenire che possa essere modificato in modo anomalo (ad esempio per assegnare dei valori negativi). Il costruttore *Contatore* si incarica di inizializzare (a 0) il valore del contatore al momento della creazione di un oggetto. I metodi *valore*, *incrementa* ed *azzera* sono stati qualificati con *public* in modo da poter essere richiamati. Sono stati inseriti dei commenti a seguire le parentesi graffe per favorire la lettura del codice. Si noti inoltre il particolare passaggio sintattico `Contatore implements Contatore_` che vincola la classe *Contatore* ad implementare tutti i metodi previsti nell'interfaccia *Contatore_*.

Osservazione. Un'implementazione deve farsi carico di alcune scelte riguardanti il comportamento degli oggetti in situazione limite o quando vengono passati ai metodi argomenti errati. Nell'esempio della classe *Serbatoio*, il metodo *inserisci* deve controllare e gestire la situazione che la quantità che si tenta di inserire sia ospitabile nel serbatoio. In situazioni come questa si può adottare una politica minimalista nella gestione della situazione anomala, semplicemente non eseguendo alcuna azione; una politica alternativa, più drastica, consisterebbe nel visualizzare un messaggio di errore ed interrompere l'applicazione; una politica adattiva consisterebbe nell'inserire solamente la quantità possibile (riempendo il serbatoio); per gestire in modo più trasparente questa soluzione si può impostare il metodo *inserisci* in modo che ritorni come risultato la quantità effettivamente inserita nel serbatoio. Nella programmazione orientata agli oggetti queste situazioni anomale vengono gestite mediante dei particolari meccanismi sintattici denominati *gestione delle eccezioni e degli errori*.

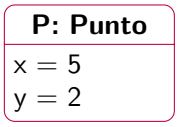
9.7 Usare gli oggetti di una classe

Un *oggetto* è uno specifico elemento (istanza) di una classe. È caratterizzato da propri valori degli attributi, distinti da quelli degli altri oggetti della stessa classe; Un oggetto ha una propria vita all'interno di un'applicazione: viene creato, manipolato e distrutto. Un oggetto è dotato di un proprio *stato* costituito da attributi dotati di uno specifico valore. Un oggetto può avere dei collegamenti ad altri oggetti esterni.

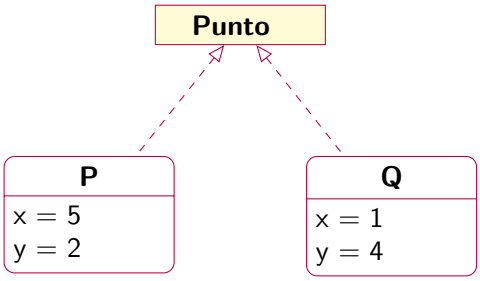
Gli oggetti di una classe sono caratterizzati dai seguenti aspetti:

- *identità* : espressa da un nome identificativo univoco
- *stato* : proprietà specifiche dell'oggetto, costituite dai valori dei suoi attributi
- *comportamento* : descritto mediante i *metodi* comuni, definiti nella classe; i metodi possono modificare lo stato dell'oggetto (metodi *setter*) e ritornare informazioni relative allo stato dell'oggetto (metodi *getter*)

Per denotare un oggetto P di classe *Punto* si usa la seguente notazione:



Per denotare una classe ed alcuni suoi oggetti si usa il seguente *diagramma degli oggetti*:



Le applicazioni

Un'*applicazione* è un programma completo ed autonomo, in grado di essere eseguito e fornire alcune funzionalità. La fase di utilizzo di una classe consiste nella creazione di oggetti della classe e nella chiamata di metodi su di essi. La creazione di un oggetto avviene richiamando un costruttore che crea e ne ritorna l'indirizzo di memoria che viene generalmente assegnato ad un riferimento per poter accedere successivamente all'oggetto creato.

A seguire è riportata una tipica porzione di programma (da inserire nel *main*) basata sull'uso della classe *Contatore*, in linguaggio Java. Nel linguaggio Java un'applicazione viene realizzata definendo il metodo *main* in una classe e, dopo aver compilato, eseguendo il bytecode della classe contenente il *main*. Solitamente nel *main* vengono definiti alcuni riferimenti ad oggetti, vengono creati alcuni oggetti ed assegnati ai riferimenti predisposti e vengono richiamati dei metodi su questi oggetti, dando vita al processo di esecuzione.

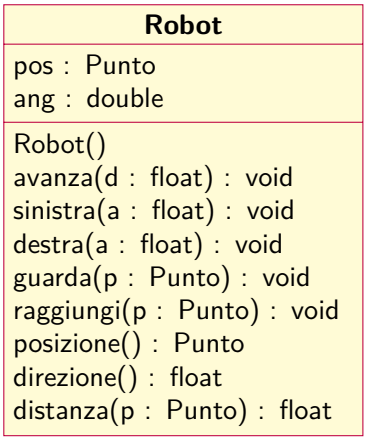
Java

```
Contatore a = new Contatore();
Contatore b = new Contatore();
int x, y;
a.incrementa();
a.azzera();
b.incrementa();
a.incrementa();
b.incrementa();
x = a.valore();
y = b.valore();
```

9.8 Un robot che si muove sul piano

Consideriamo un *robot puntiforme che si muove su un piano*. Il robot è un automa dotato di metodi che ne permettono lo spostamento sul piano; inoltre è dotato di sensori che permettono di interagire con l'ambiente esterno.

Coerentemente con l'ipotesi di robot puntiforme, il robot è caratterizzato dalle sue coordinate che descrivono la sua posizione nel piano. Data la sua caratterizzazione fisica, un robot deve possedere una direzione di avanzamento modificabile. Possiamo lasciarci guidare dall'analogia con la tartaruga della omologa grafica, tralasciando tutti gli aspetti riguardanti la grafica (questi aspetti possono essere aggiunti in un secondo momento, applicando una sorta di metodologia top-down calata in un contesto object-oriented). Il piano su cui si muove il robot è costituito da *punti*. Tutte queste considerazioni possono essere condensate nel seguente diagramma della classe *Robot* e nelle corrispondenti annotazioni.



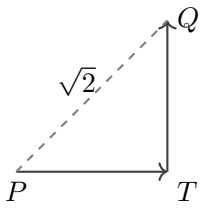
pos : *posizione del robot*
ang : *angolo di avanzamento del robot*
.....
Robot() : *costruttore di un robot posizionato nell'origine degli assi*
avanza : *avanza di un tratto* d
sinistra : *ruota a sinistra di un angolo* a
destra : *ruota a destra di un angolo* a
guarda : *orienta verso il punto* p
raggiungi : *raggiunge il punto* p
posizione : *punto attuale di stazionamento*
direzione : *attuale direzione di avanzamento*
distanza : *distanza da un dato punto*

Osservazione. I metodi descritti nel precedente diagramma di classe non utilizzano alcuni parametri cinematici dei robot reali, quali la velocità di avan-

zamento e di rotazione; questi parametri risultano completamente ininfluenti nel caso si voglia considerare il robot come un oggetto geometrico; nel caso si voglia evidenziare il robot come un oggetto che si muove su un video o, ancor di più, se il robot è un oggetto reale che si muove su un piano fisico, questi parametri sono assolutamente doverosi e potrebbero essere inseriti come un argomento aggiuntivo dei metodi di movimento e rotazione.

Osservazione. I metodi descritti nella precedente interfaccia *Robot* possono essere interpretati come sensori ed attuatori di un robot fisico: il metodo *avanza* costituisce i motori del robot; il metodo *guarda* costituisce un sensore ottico; il metodo *raggiungi* costituisce una combinazione di motori, sensori ottici e sensore di distanza; il metodo *posizione* costituisce una sorta di GPS; il metodo *direzione* simula le funzionalità di un giroscopio, fornendo la direzione di avanzamento; il metodo *distanza* costituisce un sensore di prossimità.

Basandosi sui metodi definiti nel precedente diagramma di classe *Robot* si può calcolare il valore di $\sqrt{2}$ utilizzando un robot: il robot viene spostato dal punto iniziale P, attraverso il punto intermedio T, al punto finale Q, in modo tale che sia $PT = TQ = 1$, $\angle PTQ = 90°$ e quindi $PQ = \sqrt{2}$.



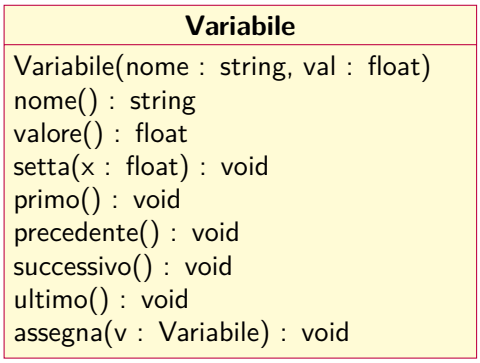
Java

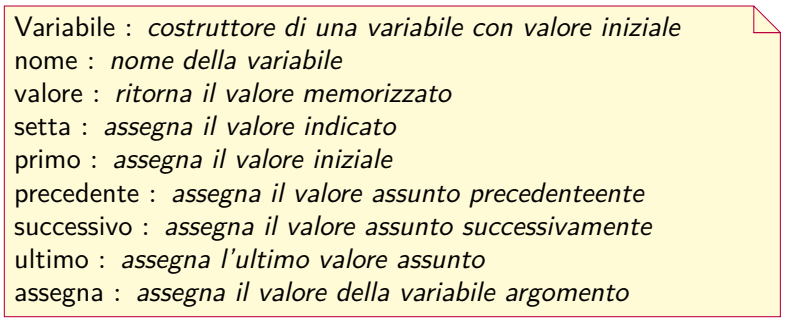
```
// calcolo della radice di 2
Robot r = new Robot();
Punto p = r.posizione();
double rad2;
r.avanza(1);
r.sinistra(90);
r.avanza(1);
rad2 = r.distanza(p);
```

9.9 Una variabile con history

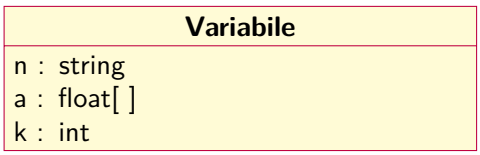
Adottando il punto di vista dell'informatica, una *variabile* è un oggetto che ha un suo nome identificativo e che memorizza un valore; una variabile può essere manipolata mediante specifici metodi che permettono di ottenere e modificare

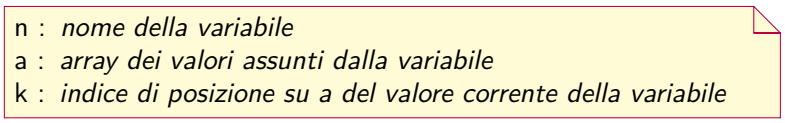
il valore in essa memorizzato. Assumiamo l'ipotesi che il valore memorizzato sia un numero naturale. Queste specificazioni possono essere sintetizzate mediante il seguente diagramma dell'interfaccia.





Ragionando in ottica dell'implementazione risulta immediato convincersi che al suo interno una *Variabile* dovrà avere un attributo $(n : string)$ che memorizza il nome. Inoltre, per gestire la propria storia, una *Variabile* dovrà avere un array che memorizza tutti i valori assunti dalla variabile nel corso della sua vita. La creazione di un oggetto di classe *Variabile* avviene mediante un costruttore (*Variabile*(*id : string, val : float*)). Tutte queste considerazioni possono essere descritte mediante la seguente porzione del diagramma della classe, relativa ai soli attributi.



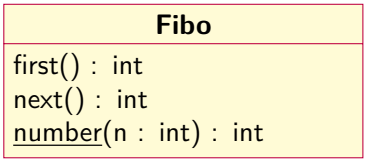


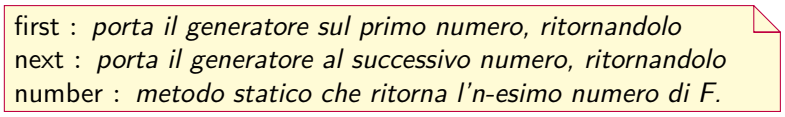
9.10 I numeri di Fibonacci

I *numeri di Fibonacci* sono costituiti da una successione di numeri naturali che inizia con i valori 0 e 1; i numeri successivi sono definiti come somma dei due valori precedenti; quindi i primi numeri della successione sono:

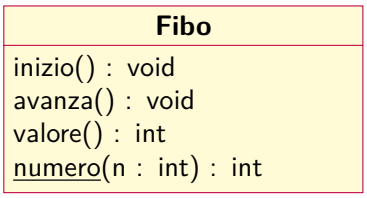
$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad \cdots$$

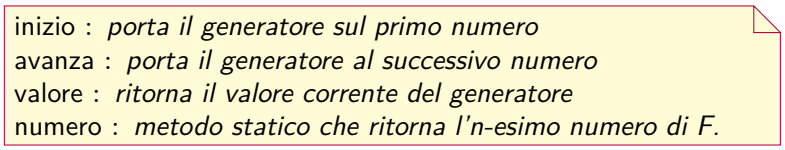
I numeri di Fibonacci possono essere calcolati mediante degli oggetti generatori oppure mediante dei metodi di classe statici; a seguire sono riportate una possibile soluzione.





Una soluzione alternativa consiste nel realizzare dei metodi che muovono una sorta di cursore virtuale sui valori della successione ed un metodo che ritorna il valore attuale del generatore (senza spostare il cursore).





ESERCIZI

9.1 In termini di numeri e di punti, descrivere i parametri caratteristici delle seguenti classi di entità:

1. retta del piano cartesiano
2. circonferenza del piano cartesiano
3. triangolo del piano euclideo
4. numero razionale

9.2 Descrivere dei diagrammi di classi i cui oggetti immutabili sono

1. gli *intervalli chiusi* della retta reale, della forma $[a, b]$
2. le *frazioni* costituite da numeratore e denominatore intero
3. le *scatole* a forma di parallelepipedo.

9.3 Descrivere dei diagrammi di classi che descrivano i seguenti oggetti del piano cartesiano; è possibile fare riferimento alla classe *Punto* già realizzata.

1. i *segmenti del piano cartesiano*
2. le *circonferenze* del piano cartesiano
3. i *triangoli del piano cartesiano*

9.4 Un *misuratore* è un oggetto che registra un valore intero compresi fra un valore minimo a ed un valore massimo b. Ogni (istanza di un) misuratore ha un proprio intervallo $[a, b]$ di misura, specificato al momento della creazione e non più modificabile. Gli estremi a e b dell'intervallo vanno intesi come valori di fondo scala e non vanno oltrepassati. Un misuratore può essere manipolato mediante due metodi che hanno l'effetto di aumentare o diminuire di un'unità il valore del misuratore. Descrivere un diagramma della classe *Misuratore*.

9.5 Con riferimento alla classe *Contatore*, scrivere delle porzioni di programma per:

1. Assegnare ad un *contatore* y il valore del *contatore* x.
2. Scambiare il valore di due *contatori* x ed y.

9.6 Con riferimento alla classe *Robot*, scrivere delle porzioni di programma per:

1. Determinare la distanza fra due robot.
2. Calcolare la radice quadrata di un numero naturale n.
3. Determinare il punto medio fra due dati punti P e Q.
4. Scambiare di posto due robot; compatibilmente con la realtà, nello scambio i due robot non devono scontrarsi, né devono trovarsi nello stesso momento nello stesso punto.
5. Spiegare come dovrebbe essere ampliata la classe *Robot* per disporre di un metodo per sapere il robot più vicino ad un dato robot.

9.7 Descrivere un diagramma della classe `Frazione` i cui oggetti sono le usuali frazioni numeriche. Usando la classe `Frazione` calcolare $\sum_{k=1}^{n} \frac{1}{k}$.

9.8 *Mescolare* un array di n numeri contenente i numeri da 1 ad n.

9.9 La generazione di numeri casuali può essere realizzata mediante il *metodo congruenziale lineare di Lehmer*: fissate tre (opportune) costanti naturali A, B, M, la seguente relazione di ricorrenza:

$$\begin{aligned} x_0 &= \text{valore generico (detto seme)} \\ x_n &= (A * x_{n-1} + B) \mod M, n = 1, 2, ... \end{aligned}$$

genera una successione $x_1, x_2, \ldots$ di valori naturali pseudocasuali compresi fra 0 e $M - 1$. Degli adeguati valori per le costanti sopra indicate sono i seguenti: $A = 27181, B = 13849, M = 65536$. Implementare una classe `Caso` i cui oggetti sono dei generatori di numeri casuali compresi fra due dati valori forniti come parametri al costruttore della classe.

9.10 Realizzare la classe `Urna` memorizzando gli elementi da estrarre in un array che viene inizialmente mescolato e successivamente viene consultato sequenzialmente per accedere agli elementi.

9.11 Realizzare un classe `UrnaBernoulliana` di evidente significato. Realizzare una classe `Urna01` che fornisca numeri casuali uniformemente distribuiti fra 0 e 1. Realizzare una classe `UrnaGaussiana` che fornisce numeri casuali aventi una distribuzione gaussiana.

9.12 Descrivere un *serbatoio* mediante un diagramma della classe. Individuare ed esprimere alcuni metodi dell'interfaccia *Serbatoio* in funzione di altri. Implementare la classe dei *serbatoi*. Definire l'interfaccia di alcuni metodi per travasare da un serbatoio ad un altro e per scambiare la quantità di liquido presente in due serbatoi. Implementare i metodi utilizzando quelli specificati nelle precedente interfaccia.

9.13 Descrivere un *interuttore* mediante un diagramma della classe. Individuare ed esprimere alcuni metodi dell'interfaccia *Interuttore* in funzione di altri. Implementare la classe degli *interuttori*. Modificare opportunamente le classi sugli interruttori in modo che ogni interruttore abbia un numero di serie progressivo identificativo. Allineare allo stesso valore lo stato di 3 *interuttori*, modificando al più lo stato di un interuttore.

9.14 Definire delle interfacce *Tris* e *Sudoku* per gestire i giochi omonimi.

9.15 Analizzare, progettare e realizzare un meccanismo di *lock-unlock* sugli oggetti.

9.16 Progettare e realizzare un meccanismo di *ripristino dello stato* di un oggetto.

9.17 Studiare il comportamento dei metodi *primo*, *precedente*, *successivo*, *ultimo* della classe *Variabile* quando vengono apportate modifiche ad una variabile.

9.18 Integrare il diagramma della classe *Variabile* aggiungendo un metodo statico *valore* che ritorna il valore della variabile avente uno dato nome (argomento del

metodo). Analizzare come si può realizzare questo metodo.

9.19 Individuare gli attributi necessari all'implementazione della classe *Fibo*.

9.20 Descrivere un diagramma di classe i cui oggetti sono le figure geometriche piane costituite dai *rettangoli*. Definire un criterio per stabilire la distanza fra due rettangoli. Codificare, all'interno di una classe *Rettangolo*, questa operazione mediante un metodo *distanza*. Scrivere una porzione di codice che illustri un esempio d'uso del metodo *distanza*.

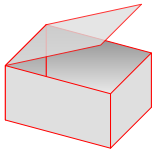
9.21 Descrivere mediante un diagramma di interfaccia i diversi modi con cui si possono realizzare i metodi per *sommare* due frazioni e per determinare il *reciproco* di una frazione (esempio: il reciproco della frazione $\frac{3}{4}$ è la frazione $\frac{4}{3}$). Usando i metodi descritti nel diagramma di interfaccia scrivere una porzione di codice per calcolare $\sum_{k=1}^{n} \frac{1}{k}$. Volendo incapsulare questa porzione di codice in un metodo, descrivere la corrispondente firma da inserire nel diagramma di interfaccia.

9.22 Nell'insieme delle stringhe sull'alfabeto $\mathcal{A} = \{\texttt{0}, \texttt{1}\}$ determinare il cerchio di centro `1101` e di raggio 1 secondo la metrica di edit, ossia l'insieme delle stringhe aventi distanza di edit ≤ 1 dalla stringa `1101`.

9.23 Un *conto* rappresenta un conto bancario contenente i dati relativi all'ammontare di denaro presso una banca. Ogni conto è caratterizzato da un numero identificativo univoco e dal valore del saldo corrispondente (ammontare dei soldi attualmente depositati). Ogni conto è caratterizzato da un *fido* che rappresenta il valore sotto il quale non è possibile andare nei prelievi. Un conto è in *rosso* se il corrispondente saldo è negativo. Su un conto si devono poter fare versamenti e prelievi di denaro (compatibilmente con il fido associato al conto) e transazioni (passaggi di denaro) da un conto ad un altro. Usando i metodi elencati nell'interfaccia *Conto*, scrivere una porzione di algoritmo per codificare i due metodi *transazione* elencati alla fine dell'interfaccia *Conto*. Suggerimento: è sufficiente una sequenza di istruzioni.

9.24 Descrivere in notazione UML un diagramma di interfaccia e corrispondente annotazione che specifichi gli oggetti che sono "robot dotati di motore per muoversi sul piano, dotati di sensori di distanza e di contatto per interagire con l'ambiente esterno".

9.25 Descrivere in notazione UML un diagramma di classe e corrispondente annotazione che specifichi gli oggetti che sono "scatole a forma di parallelepipedo rettangolo, di dimensioni immutabili, dotata di un coperchio che si può aprire o chiudere".



9.26 Descrivere in notazione UML un diagramma di classe e corrispondente annotazione che specifichi gli oggetti che sono "intervalli chiusi della retta reale", tradizionalmente scritti in matematica con la notazione $[a, b]$.

9.27 Scrivere un diagramma di classe che descriva i *punti del piano cartesiano aventi un nome identificativo univoco*. Scrivere una porzione di programma che crei i punti $P(1,2)$, $Q(5,3)$ e determini la loro distanza.

9.28 Scrivere un diagramma di classe che descriva i *punti del piano cartesiano aventi un nome identificativo univoco*. Scrivere una porzione di programma che crei i punti $A(1,2)$, $B(5,3)$, $C(4,6)$ e calcoli il perimetro del triangolo ABC. Spiegare come si può garantire, nell'implementazione dell'interfaccia, che non vengano creati punti aventi lo stesso nome. Discutere se ed eventualmente come è possibile realizzare un metodo che, dato il nome di un punto, fornisce le sue coordinate.

9.29 Definire un'interfaccia *Tartaruga* che descriva degli oggetti per disegnare nel piano secondo la tradizionale grafica della tartaruga. Le tartarughe possono muoversi solo sui punti del piano di coordinate intere e con direzione parallele agli assi coordinati. Implementare una significatica porzione dell'interfaccia, supponendo di poter usare il metodo statico $G.linea(x_1, y_1, x_2, y_2)$ che disegna un segmento congiungente i due punti di coordinate (x_1, y_1) e (x_2, y_2). Scrivere una porzione di *main* per disegnare una spirale quadrata che inizia nel punto di coordinate $(1, 1)$.

9.30 Con riferimento alla seguente interfaccia, scrivere una porzione di programma che abbia l'effetto di scambiare il valore di due variabili.

9.31 Definire un'interfaccia *Riferimento* analoga all'interfaccia *Variabile* adatta a gestire dei riferimenti secondo la semantica dei riferimenti.

9.32 Estendere l'interfaccia *Variabile* in modo che una variabile possa gestire uno storico dei valori memorizzati, con possibilità di ripristinarli come fosse una pila.

9.33 Descrivere un diagramma UML della classe *Bigliettaio* i cui oggetti sono dei generatori (in sequenza) di biglietti della forma: `A00`, `A01`, ..., `A99`, `B00`, `B01`, ..., formati da una serie costituita da un carattere e da una coppia di cifre, come quelli che si trovano al banco dei supermercati o ad un generico servizio, per gestire la coda dei clienti. Riempire un array di dimensione 100 con i biglietti alternativamente con serie `A` e `B`, e con numerazione in progressione; la sequenza dei biglietti memorizzata nell'array dovrà essere pertanto: `A00`, `B01`, `A02`, `B03`, `A04`, ... Un generatore, dopo aver generato l'ultimo biglietto `Z99`, si "esaurisce". Descrivere come può essere gestita questa situazione.

9.34 Descrivere mediante un diagramma UML la classe *Complesso* i cui oggetti sono i numeri complessi. Scrivere un'interfaccia Java corrispondente al diagramma UML. Implementare una significativa porzione dell'interfaccia. Scrivere una porzione di *main* in cui vengono creati e manipolati alcuni numeri complessi.

9.35 Descrivere mediante un diagramma UML la classe *Punto* i cui oggetti sono i punti del piano cartesiano. Scrivere un'interfaccia Java corrispondente al diagramma UML. Implementare una significativa porzione dell'interfaccia. Scrivere una porzione di *main* in cui vengono creati e manipolati alcuni punti.

9.36 Definire un'interfaccia *Tartaruga* che descriva degli oggetti per disegnare nel piano secondo la tradizionale grafica della tartaruga. Le tartarughe si muovono lungo le 4 direzioni $N/E/S/W$. Implementare una significatica porzione dell'interfaccia, supponendo di poter usare il metodo statico $G.linea(x_1, y_1, x_2, y_2)$ che disegna un segmento congiungente i due punti di coordinate (x_1, y_1) e (x_2, y_2).

Scrivere una porzione di *main* in cui viene disegnato un quadrato.

9.37 Descrivere un'interfaccia che descriva delle variabili tipizzate, contenenti un valore ed identificate mediante un nome. Le variabili vengono create, facoltativamente inizializzate e successivamente assegnate. È possibile recuperare i vecchi valori memorizzati in una variabile, mediante un meccanismo di *history* a cursore. Implementare una parte dell'interfaccia.

9.38 Descrivere l'interfaccia di un oggetto *Ascensore*, i cui oggetti sono (le centraline di controllo) degli ascensori che possono essere movimentati e prenotati fra i diversi piani di un edificio. Specificare mediante un commento il significato di ciascun metodo. In base ai metodi descritti, definire gli attributi d'istanza adatti a gestire un ascensore.

9.39 Un *lucchetto* è un oggetto che può trovarsi in uno stato di *aperto* o *chiuso*. Può essere chiuso mediante una chiave costituita da un numero intero e può essere aperto solo precisando il valore corretto della chiave con la quale è stato chiuso. Descrivere un'interfaccia che descriva gli oggetti lucchetto. Implementare l'interfaccia. Scrivere una significativa porzione di applicazione per testare la classe.

9.40 Descrivere un'interfaccia di oggetti *urna* costituiti da contenitori di numeri naturali, dal numero 1 fino ad un certo numero specificato al momento della creazione dell'urna. Deve essere possibile estrarre i numeri ad uno ad uno e reimbussolare tutti i numeri, ripristinando la situazione originale.

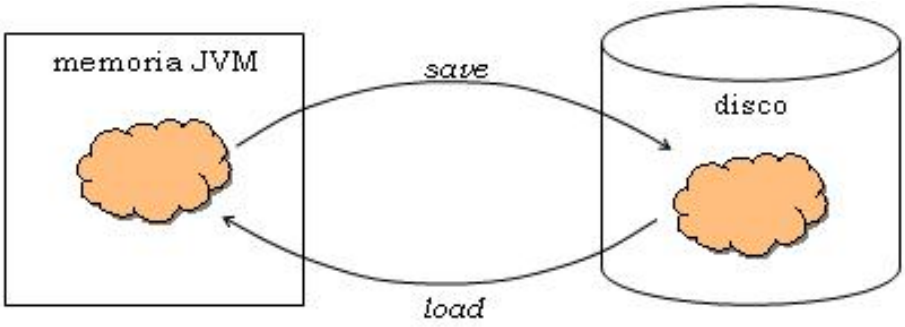
9.41 Definire un'interfaccia che specifichi le usuali frazioni composte da numeratore e denominatore interi. Deve essere possibile eseguire le operazioni sulle frazioni. Implementare una parte significativa dell'interfaccia.

9.42 Si consideri il contesto della *geometria della tartaruga* dove si disegnano delle figure del piano muovendo una tartaruga. Le tartarughe, in questo contesto semplificato, possono essere orientate solo lungo le quattro direzioni parallele agli assi coordinati.

1. Definire un'interfaccia *Tarta4D* per descrivere le tartarughe (dell'omonima geometria) che si muovono lungo le 4 direzioni $N/E/S/W$.
2. Implementare l'interfaccia, supponendo di poter gestire la grafica mediante il metodo statico $G.linea(x_1, y_1, x_2, y_2)$ che disegna un segmento congiungente i due punti di coordinate (x_1, y_1) e (x_2, y_2).

9.43 Rappresentare mediante una sequenza il polinomio $x^4 - 7x^2 + x - 3$. Scrivere un algoritmo per determinare la derivata di un polinomio rappresentato come sopra. Descrivere un diagramma UML della classe i cui oggetti sono i *polinomi in una indeterminata*.

9.44 Studiare e realizzare il pattern *Persistente*. La caratteristica di persistenza di un oggetto consiste nella capacità di un oggetto a sopravvivere al processo generato dall'applicazione che lo ha creato, in modo da essere utilizzato anche da successivi processi. La persistenza può essere realizzata salvando su disco (*save*) e caricando da disco (*load*) l'immagine dell'oggetto, secondo uno schema descritto dalla seguente figura:



9.45 Scrivere l'interfaccia di una *regola* della forma

$$premessa \rightarrow conclusione$$

e l'interfaccia di una macchina *L-trasformatore* di stringhe schematizzato nella seguente figura:

10

ASSOCIARE

10.1 Associare

Una *associazione* α fra due classi o più è un legame logico che esprime una qualche forma di connessione fra gli oggetti delle classi. Un'associazione α fra due classi A e B viene rappresentata graficamente da un segmento che collega i riquadri delle due classi.

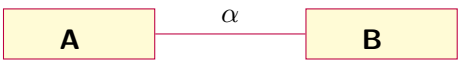


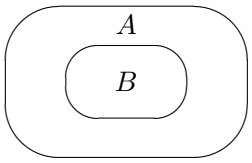
Figura 10.1: Diagramma UML di una generica associazione α fra le due classi A e B.

Il segmento che unisce i due riquadri delle due classi può assumere diverse forme a seconda della particolare associazione considerata. Nel contesto della programmazione orientata agli oggetti sono particolarmente interessanti ed utilizzate le forme di associazione descritte nei seguenti paragrafi.

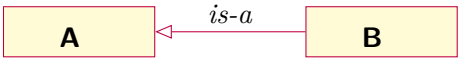
10.2 Associazione di specializzazione

La *specializzazione* è una relazione fra una classe generale (detta *superclasse*) e una versioni specializzata di quella stessa classe (detta *sottoclasse*). Useremo la notazione $A \leftarrow B$ per indicare che *Gli elementi della classe B sono elementi della classe A* e diremo che *la classe B è in relazione is-a con la classe A*. Nel contesto della metodologia orientata agli oggetti la relazione di specializzazione è designata con il termine *ereditarietà* in quanto la sottoclasse eredita gli attributi e le operazioni della superclasse.

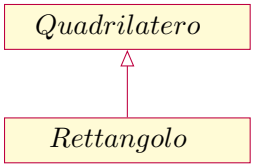
L'associazione *is-a* esprime il fatto che gli oggetti della classe B sono un sottoinsieme degli oggetti delle classe A. Questa relazione può essere efficacemente rappresentata anche mediante i diagrammi di Venn, con due insiemi uno incluso nell'altro:



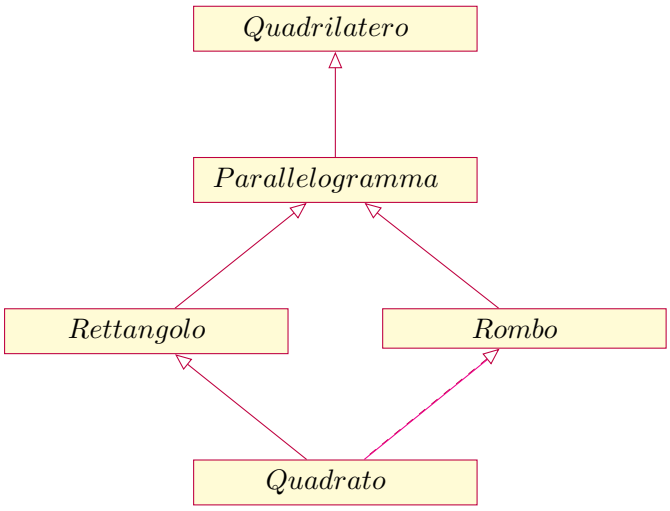
oppure mediante il seguente diagramma UML:



Esempio 10.2.1 - Nel contesto delle figure geometriche del piano vale la seguente associazione *is-a*:



Esempio 10.2.2 - Il grafo che segue descrive la relazione di ereditarietà fra alcune classi di figure geometriche, inducendo una classificazione dei quadrilateri.



Osservazione. La relazione di derivazione fra classi gode della proprietà transitiva; per questo motivo in un grafo di derivazione non vengono indicate le frecce di derivazione che sono deducibili in base alla proprietà transitiva. Ad esempio, nel precedente diagramma non serve che vengano indicate le derivazioni $Quadrilatero \leftarrow Rettangolo$, $Parallelogramma \leftarrow Quadrato$ ed altre.

Nella programmazione orientata agli oggetti l'associazione di ereditarietà si presenta nei seguenti casi:

- quando una classe implementa un'interfaccia
- quando un'interfaccia estende un'interfaccia
- quando una classe estende una classe

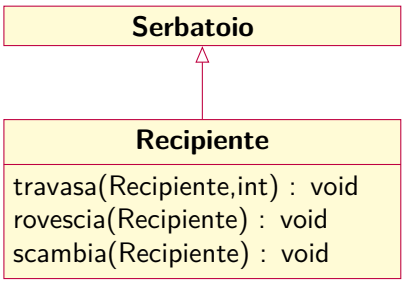
10.3 Estensione

Nei linguaggi di programmazione orientati agli oggetti la generalizzazione e la specializzazione fra classi di oggetti si concretizza mediante il meccanismo dell'*estensione*. L'estensione diventa un meccanismo operativo che si basa su una gerarchia di classi legate da una relazione di specializzazione. Questo meccanismo può essere applicato indifferentemente fra interfacce o fra classi.

Estensione di un'interfaccia

Estendere un'interfaccia significa aggiungere nuovi metodi ad un'interfaccia già definita.

Esempio 10.3.1 - L'interfaccia *Serbatoio* definita nell'esempio 9.5.1 può essere estesa con un'interfaccia *Serbatoio* aggiungendo dei metodi per eseguire operazioni di travaso fra serbatoi:



travasa : *travasa in un altro recipiente una data quantità*
rovescia : *rovescia tutto un recipiente in un altro*
scambia : *scambia le quantità fra due recipienti*

Anche per l'interfaccia *Recipiente*, similmente a quanto osservato per l'interfaccia *Serbatoio* precedentemente esaminata, i metodi *travasa*, *rovescia*, *scambia* possono ritornare un opportuno valore da interpretarsi come risultato si come sono state eseguite queste operazioni e se sono andate a buon fine.

Estensione di una classe ed ereditarietà fra le classi

L'estensione può essere applicata anche fra classi: *estendere* una classe significa realizzare una nuova classe con aggiunta di funzionalità rispetto ad una classe base già definita. Le possibilità sono:

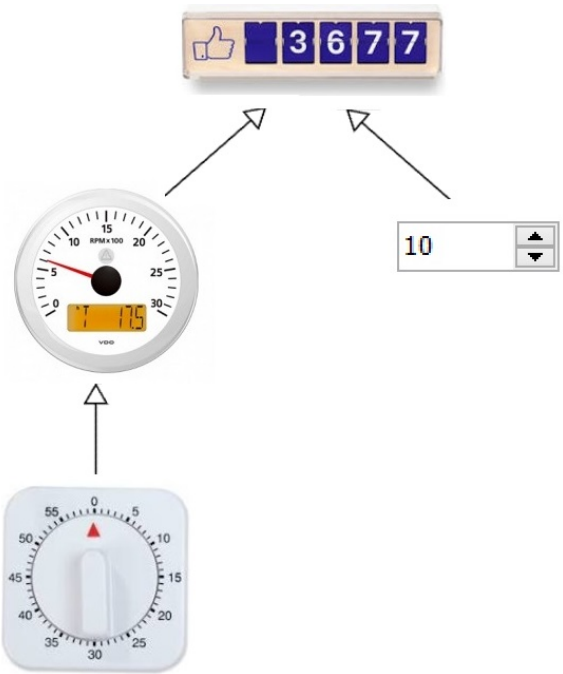
- aggiungere nuovi attributi rispetto alla classe base
- aggiungere nuovi metodi rispetto alla classe base
- ridefinire metodi della classe base

La classe che estende eredita tutte le caratteristiche della classe padre. L'*ereditarietà* è un concetto fondamentale della OOD/OOP in quanto sta a

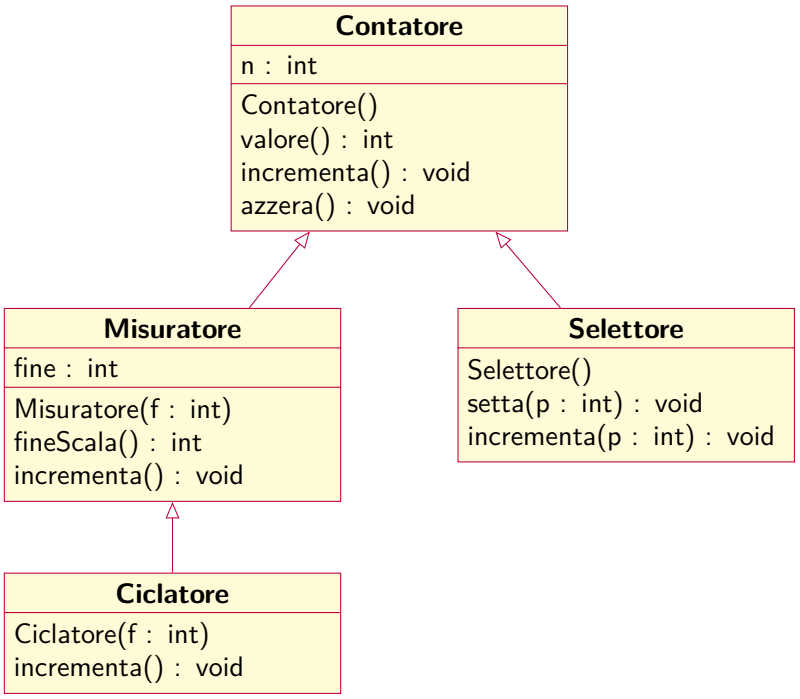
fondamento della possibilità di riutilizzare componenti software. In un linguaggio OO questo concetto si concretizza mediante un meccanismo che consente di creare delle gerarchie di classi; permette di creare nuove classi a partire da classi già esistenti ereditandone le caratteristiche (attributi e metodi) aggiungendone di nuove e di ridefinirle alcune. Nei linguaggi di programmazione questo meccanismo viene detto *estensione*.

Esempio 10.3.2 - Il concetto di ereditarietà corrisponde alla relazione *IsA* fra oggetti. Data una classe di oggetti A (classe base), estendendo la classe A si può ottenere una classe B (classe derivata) che mantiene le caratteristiche (attributi ed operazioni) della classe base ed a questa ne aggiunge di nuove.

Esempio 10.3.3 - Un *contatore* è un oggetto che memorizza un numero naturale e può essere manipolato mediante specifici metodi che permettono di ottenere, modificare, incrementare di un'unità ed azzerare il valore. Un *misuratore* è un contatore con un valore di finescala (tipo una bilancia). Un *ciclatore* è un misuratore che arrivato a fine scala si riposiziona sullo zero. Un *selettore* è un contatore azionato manualmente con possibilità di essere settato ad un generico valore naturale ed incrementato di un generico numero intero. Tutte queste specificazioni sono sintetizzate nella seguente figura.



Il diagramme delle classi che segue traduce il precedente grafo di ereditarità degli oggetti.



Osservazione. Nella classe *Misuratore* il metodo *incrementa*, già definito nella classe *Contatore*, va ridefinito in modo da controllare che il valore non ecceda il valore di fondoscala, nel qual caso deve essere azzerato. Nella classe Segnapunti i metodi *setta* e *decrementa* risultano particolarmente poco efficienti in quanto devono essere realizzati mediante un ciclo in cui richiamare più volte il metodo *incrementa*; per questo motivo l'attributo *n* della classe *Contatore* viene qualificato con *protected* affinché possa essere acceduto e modificato direttamente dall'interno delle classi che estendono la classe *Contatore*.

10.4 L'estensione nei linguaggi programmazione

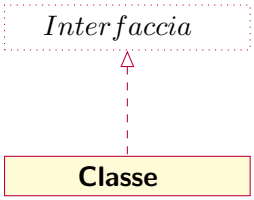
I linguaggi di programmazione OO offrono dei meccanismi sintattici per codificare l'estensione fra classi. A seguire è riportata la codifica in Java delle classi *Misuratore* (che estende Contatore) e *Ciclatore* che estende la classe *Misuratore*, come descritta nell'esempio 10.3.3.

Java

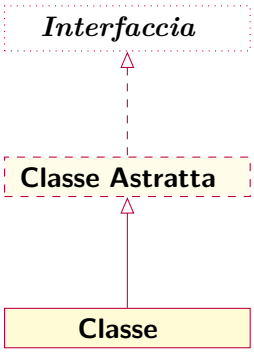
```
public class Misuratore extends Contatore
{
    private int fine;    // valore di finescala

    Misuratore(int f)
    {
        super();  // richiama Contatore()
        if (f < 0) f = 0;
        fine = f;
    } // [c] Misuratore

    public int fineScala()
    {
        return fine;
    } // [m] fineScala

    @Override
    public void incrementa()
    {
        if (valore() < fine)
            super.incrementa();
    } // [m] incrementa

} // [class] Misuratore

public class Ciclatore extends Misuratore
{
    Ciclatore(int f)
    {
        super(f);  // richiama Misuratore(int)
    } // [c] Ciclatore

    @Override
    public void incrementa()
    {
        if (valore() < fineScala())
            super.incrementa();
        else
            azzera();
    } // [m] incrementa

} // [class] Ciclatore
```

10.5 Interfacce, classi astratte e classi concrete

Mediante la notazione UML l'implementazione di un'interfaccia da parte di una classe viene descritta mediante il seguente diagramma.



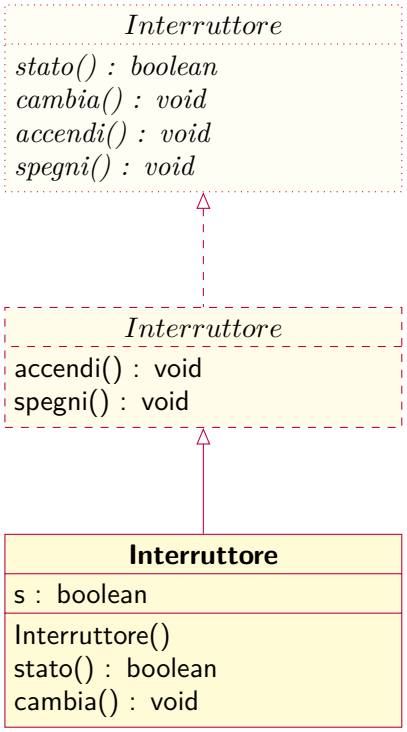
E' possibile che una classe implementi solo alcuni dei metodi previsti nell'interfaccia; in questi casi si ottiene una *classe astratta* che non permette di creare oggetti.



10.6 Classi astratte

Il meccanismo dell'estensione è utilizzato anche per la realizzazione di *classi astratte*, ossia di classi che implementano parzialmente l'interfaccia, ossia non tutti i metodi specificati nell'interfaccia. In una classe astratta vengono solitamente definiti i metodi che si appoggiano su altri metodi dell'interfaccia, senza fare riferimento agli attributi degli oggetti. Una classe che estende una classe astratta senza definire tutti i metodi che mancano rimane una classe astratta. Coerentemente con il fatto che in una classe astratta non sono definiti tutti i metodi dell'interfaccia, non è possibile creare oggetti di una classe astratta. Una classe astratta deve essere estesa mediante un'altra classe (concreta) che realizza tutti i metodi mancanti (ed eventualmente può ridefinirne alcuni ed aggiungerne altri).

Richiamando l'interfaccia *Interuttore* definita nel paragrafo 9.5, il seguente grafo UML descrive l'interfaccia *Interuttore*, una sua estensione mediante una classe astratta che implementa parzialmente l'interfaccia e la classe concreta che realizza i metodi mancanti nella classe astratta.



10.7 Associazione di aggregazione

L'*aggregazione* è un particolare tipo di associazione in cui un oggetto di una classe B usa oggetti di una classe B. Gli oggetti della classe B vengono detti *aggregati* mentre gli oggetti della classe A vengono detti *componenti*. Questo tipo di associazione è nota anche come *use-a*.

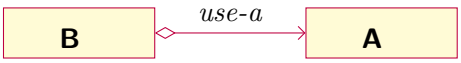
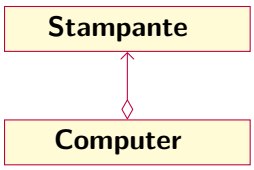


Figura 10.2: Schema grafico dell'associazione B *use-a* A.

Esempio 10.7.1 - Nel contesto degli strumenti di un sistema informatico vale l'associazione *Computer use-a Stampante*, descritta nel seguente schema grafico:



Il simbolo di *diamante vuoto* nel diagramma della figura 10.2 indica che un'istanza della classe B ha 0 o più istanze della classe A. L'aggregazione è denominata anche *aggregazione condivisa*, per sottolineare il fatto che un'istanza della classe B può essere condivisa fra più oggetti della classe A. Ciò significa che quando un oggetto della classe B viene eliminato, non necessariamente vengono eliminate le istanze della classe A associate all'oggetto eliminato. In dettaglio, la semantica dell'aggregazione è descritta mediante i seguenti punti:

- i componenti possono esistere indipendentemente dall'aggregato
- gli aggregati possono condividere uno stesso componente
- l'aggregazione è transitiva
- l'aggregazione è asimmetrica

10.8 Associazione di composizione

La *composizione* è uno speciale tipo di aggregazione in cui l'oggetto contenuto appartiene ad un solo aggregato e deve essere eliminato quando l'oggetto aggregante viene eliminato. L'oggetto contenuto viene creato direttamente dall'aggregato e non viene condiviso con altri oggetti. Questo tipo di associazione è nota anche come *has-a*.

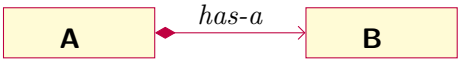
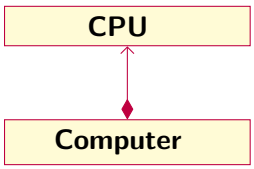


Figura 10.3: Schema grafico dell'associazione A *has-a* B.

Esempio 10.8.1 - Nel contesto degli strumenti di un sistema informatico vale l'associazione *Computer has-a CPU*, descritta nel seguente schema grafico:



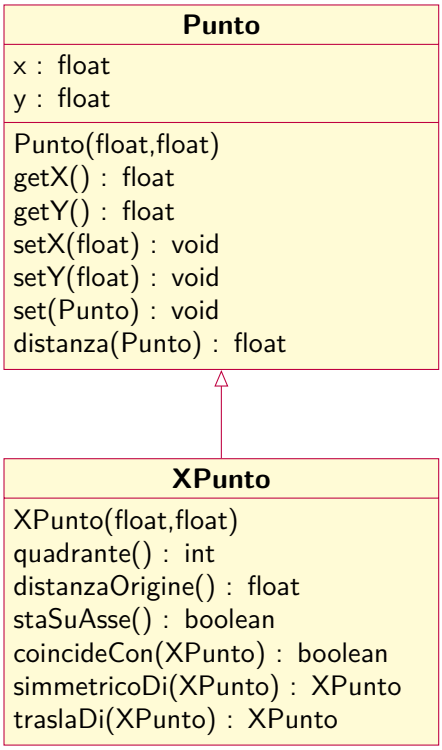
La notazione del diagramma della figura 10.3 indica che un'istanza della classe B (composito) ha 0 o più istanze della classe A (componente); inoltre valgono i seguenti vincoli:

- ogni componente può appartenere ad un solo composito per volta
- il composito è l'unico responsabile di tutte le sue parti; è responsabile della loro creazione e distruzione
- il composito può rilasciare una sua parte ad un altro oggetto che si prenda la relativa responsabilità
- se il composito viene distrutto, deve distruggere tutte le sue parti o cederne la responsabilità a qualche altro oggetto

Problema 10.8.1 Ampliare il diagramma della classe *Punto* presentato nell'esempio 9.4.1 aggiungendo i seguenti metodi:

- quadrante del piano cartesiano $(1, 2, 3, 4)$ in cui si trova il punto
- distanza del punto dall'origine degli assi
- controllo se il punto sta su un asse
- controllo se due punti coincidono
- punto simmetrico di un altro rispetto all'origine
- traslazione del punto di un dato tratto

Soluzione. La soluzione potrebbe essere svolta inserendo semplicemente i metodi richiesti direttamente all'interno del terzo riquadro del diagramma della classe *Punto*. Dal punto di vista concettuale ed operativo risulta preferibile specificare questi metodi esternamente, in un altro diagramma della classe, senza modificare il precedente diagramma della classe. Questo nuovo diagramma della classe avrà un suo nome identificativo, che spesso richiama il nome della classe genitore (la X usata nella soluzione che segue sta per *eXtended*, per richiamare il fatto che la classe *XPunto* estende la classe *Punto*; ma il nome potrebbe essere diverso). In notazione UML, per indicare che questi metodi si aggiungono a quelli della classe padre, si mette una freccia, come si vede a seguire. Per documentare i nuovi metodi si può sempre aggiungere un'ulteriore annotazione (che in questo caso conterrebbe le informazioni specificate nel testo dell'esercizio).

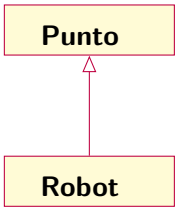


Osservazione. Nella situazione indicata sopra si dice che "la classe *XPunto* estende la classe *Punto*"; gli oggetti della classe *XPunto* sono anche oggetti della classe *Punto* (ma non viceversa); sugli oggetti della classe *Punto* si possono applicare solo i metodi specificati nella classe *Punto*; sugli oggetti della classe *XPunto* si possono applicare sia i metodi della classe *Punto* che i metodi della classe *XPunto*.

Non sempre la soluzione di estendere una classe risulta la soluzione più ideonea, come si deduce dai ragionamenti riportati nel seguente esempio.

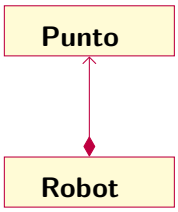
Esempio 10.8.2 - Supponiamo di aver già realizzato la classe *Punto* e di voler realizzare la classe *Robot*, come sono state definite nei capitoli precedenti. Passando alla fase di implementazione della classe *Robot* si aprono due possibilità distinte in base alle diverse associazioni che si possono instaurare fra le due classi *Robot* e *Punto*.

Una prima soluzione consiste nel vedere una relazione di ereditarietà fra un *Punto* ed un *Robot*, ossia vedere un robot come un *punto che si muove*. Il corrispondente diagramma delle classi è il seguente:



Questa soluzione offre il vantaggio di ereditare tutti i metodi già realizzati nella classe *Punto*; d'altra soffre dell'incoerenza di poter modificare la posizione di un robot modificandone direttamente le coordinate della posizione senza usare i comandi di movimento.

Un'altra soluzione che previene la modifica diretta della posizione del robot si basa sul seguente diagramma delle classi:



ESERCIZI

10.1 Descrivere mediante dei diagrammi di Venn la gerarchia delle classi di figure geometriche riportate nell'esempio 10.2.2.

10.2 Inserire nel diagramma delle classi di figure geometriche riportate nell'esempio 10.2.2 la classe *Trapezio* delle figure geometriche dei trapezi.

10.3 Estendere la classe *Robot* aggiungendo dei metodi per gestire la distanza percorsa.

10.4 Implementare il sistema di classe *Caso-Dado-Urna*.

10.5 Per un'applicazione si rendono necessari degli oggetti costituiti da una *coppia di dadi*. I dadi costituenti una coppia devono, all'occorrenza, poter essere percepiti sia come un unico oggetto (una coppia), nel caso in cui interessi solamente sapere il punteggio complessivo totalizzato dalla coppia di dadi, sia come oggetti individuali (anche se vengono lanciati contemporaneamente), nel caso in cui si desideri sapere il valore di ciascun dado.

1. Individuare eventuali relazioni *is-a*/*has-a* fra le classi di oggetti *Dado* e *CoppiaDiDadi*.
2. Definire un'interfaccia `Dado_` ed un'interfaccia `CoppiaDiDadi_`.
3. Nell'ipotesi di disporre di una classe `Dado` che implementa l'interfaccia `Dado_`, implementare l'interfaccia `CoppiaDiDadi_`.
4. Discutere se, limitandosi a considerare il punteggio complessivo totalizzato da una coppia di dadi, una *coppia di dadi a* 6 *facce* è equivalente ad un *unico dado a* 12 *facce*.

10.6 Definire un'interfaccia `CoppiaDiDadi_` i cui oggetti sono costituiti da 2 dadi. Una coppia di dadi può essere lanciata; una volta lanciata è possibile accedere sia alla somma dei valori ottenuti, sia ai valori ottenuti in ciascun dado. Implementare l'interfaccia `CoppiaDiDadi_` basandosi sulla classe `Dado`, i cui oggetti sono degli usuali dadi a 6 facce. La classe `Dado` implementa la seguente interfaccia:

```
interface Dado_
{
    /** lancio del dado e ritorno del valore */
    int lancia();
} // [interface] Dado_
```

10.7 Spiegare quali inconvenienti potrebbero verificarsi nel caso il generatore di numeri casuali, utilizzato nell'implementazione della classe `Urna`, non generasse un particolare numero. Rendere più robusta l'implementazione della classe `Urna` in modo da gestire questa possibile deficienza del generatore di numeri casuali.

10.8 Individuare le principali classi di entità di un sistema di classi per l'implementazione della *grafica della tartaruga*.

10.9 Analizzare le classi di oggetti di seguito descritte. Definire una gerarchia di relazioni fra le varie classi.

- *penna*: può essere mossa sul piano cartesiano; può essere alzata ed abbassata; può essere collegata ad un foglio; se è collegata ad un foglio e se è abbassata, quando viene spostata lascia una linea sul foglio
- *tartaruga*: è il tradizionale oggetto che permette di gestire l'omonima grafica; gestisce un angolo di orientamento modificabile e può essere fatta avanzare
- *disegno*: costituisce la struttura dati che memorizza le linee tracciate sul foglio
- *foglio*: è una finestra grafica sulla quale è possibile tracciare linee

Coerentemente con il grafo descritto al punto precedente, definire delle interfacce degli oggetti *penna*, *tartaruga*, *disegno*, *foglio*.

10.10 Definire l'interfaccia dei seguenti oggetti per realizzare un sistema per giocare a scacchi. Suggerimento: alcune delle classi coinvolte potrebbero essere le seguenti: *Scacchiera*, *Pezzo*, *Re*, *Regina*, ..., *Mossa*, *Posizione*, *Configurazione*, *Motore*, ...

10.11 Realizzare il sistema di classi descritto composto dalle seguenti classi di entità geometriche geometriche, considerate non ancorate ad un piano cartesiano, ma individuate solamente dalle lunghezze dei lati, secondo il grafo descritto nell'esempio 10.2.2.

1. *Parallelogramma*
2. *Rettangolo*
3. *Quadrato*

10.12 Progettare e realizzare una classe *Ciclatore* i cui oggetti sono dei contatori vincolati ad assumere valori naturali fra due dati estremi; un ciclatore può essere incrementato o decrementato di un'unità.

10.13 Progettare e realizzare una classe *Orologio* i cui oggetti sono simili a dei contatori, in grado di segnare il tempo in o minuti ed ore.

10.14 Discutere la situazione in cui *Urna extends Dado* ma il metodo di estrazione si chiama *estrai*.

10.15 Descrivere il grafo delle relazioni (*IsA*, *HasA*, *UseA*) fra le seguenti classi di oggetti:

- *Colore* : colore individuato dalle usuali tre componenti (r,g,b)
- *Punto* : punto del piano cartesiano, individuato dalle sue due coordinate
- *Foglio*: pannello grafico, dotato di un sistema di riferimento cartesiano, sul quale è possibile disegnare mediante una penna
- *Penna* : penna colorata, usata per disegnare su un foglio; può essere alzata ed abbassata sul foglio e spostata su un punto del foglio
- *Tartaruga*: è il tradizionale oggetto della grafica della tartaruga, dotato di una penna, mediante il quale è possibile disegnare su un foglio

- *Disegno*: struttura dati che memorizza quanto disegnato su un foglio

Coerentemente con questo grafo delle relazioni, definire l'interfaccia *Penna*.

10.16 Descrivere il grafo delle relazioni (*IsA*, *HasA*, *UseA*) fra le seguenti classi di oggetti:

- *Colore* : colore individuato dalle usuali tre componenti (r,g,b)
- *Punto* : punto del piano cartesiano, individuato dalle sue due coordinate
- *Foglio*: pannello grafico, dotato di un sistema di riferimento cartesiano, sul quale è possibile disegnare mediante una penna
- *Penna* : penna colorata, usata per disegnare su un foglio; può essere alzata ed abbassata sul foglio e spostata su un punto del foglio
- *Tartaruga*: è il tradizionale oggetto della grafica della tartaruga, dotato di una penna, mediante il quale è possibile disegnare su un foglio
- *Disegno*: struttura dati che memorizza quanto disegnato su un foglio

Basandosi su questo grafo delle relazioni, definire l'inizio della classe *Tartaruga*, limitandosi a quanto direttamente deducibile dal grafo delle relazioni che si è descritto.

10.17 Studiare il grafo delle relazioni (*IsA*, *HasA*, *UseA*) fra le seguenti classi: *Coppia* (coppia di numeri reali), *Punto* (punto del piano cartesiano), *Complesso* (numero complesso), *Segmento* (segmento del piano cartesiano).

10.18 Descrivere un grafo di ereditarietà composto dalle seguenti classi di entità: *Stella*, *Poligono*, *Spirale*.

10.19 Descrivere un grafo di ereditarietà per il seguente sistema di classi di entità geometriche geometriche, considerate non ancorate ad un piano cartesiano, ma individuate solamente dalle lunghezze dei lati ed ampiezza degli angoli: *Parallelogramma*, *Rombo*, *Rettangolo*, *Poligono*, *FiguraPiana*, *PoligonoRegolare*, *Quadrato*, *Triangolo*, *TriangoloEquilatero*, *TriangoloIsoscele*, *Quadrilatero*.

11

CONTENERE

Vedremo che non c'è niente di magico, misterioso o difficile sui metodi per trattare strutture complesse.

D. Knuth,
The Art of Computer Programming

Un *design pattern* è un modello di programmazione generico, uno schema applicabile a delle situazioni similari. Spesso i design pattern si fondano su dei concetti che risultano potenti proprio per il loro alto grado di astrazione e generalità. I design pattern costituiscono uno strumento di progettazione: essi descrivono il progetto in termini di oggetti, del ruolo e del problema che essi risolvono. Il meccanismo dei design pattern è bene supportato dai linguaggi di programmazione orientati agli oggetti (Java, Python, C++, ...) nei quali la specifica astratta di un pattern viene descritta mediante un'interfaccia.

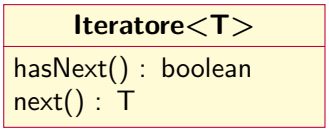
Nella letteratura informatica sono stati evidenziati ed analizzati diversi design pattern. In questo capitolo ne vengono presentati alcuni fra quelli più importanti connessi alla programmazione orientata agli oggetti ed alle strutture dei dati. Fra questi, *contenitore* è un tipico design pattern che caratterizza la maggior parte delle strutture dati e caratterizza a livello astratto qualsiasi struttura composta da oggetti più elementari.

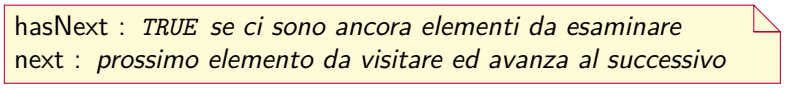
11.1 Iteratori

Un *iteratore* [1] è un oggetto che consente di visitare sequenzialmente tutti gli elementi contenuti in un altro oggetto detto *iterabile*. L'utilità di un iteratore risiede nella possibilità di indagare la composizione dell'iterabile indipendentemente dalla sua struttura interna e dai dettagli della sua implementazione. Gli iteratori esplicano il loro effetto in sinergia con gli iterabili: gli iterabili generano (o memorizzano) gli elementi da fornire e gli iteratori attraversano l'iterabile fornendo all'esterno i suoi elementi.

Gli iteratori possono fornire delle operazioni aggiuntive; ad esempio gli *iteratori bidirezionali* consentono di muoversi in entrambi i versi; gli *iteratori filtranti* consentono di visitare selettivamente solo gli elementi soddisfacenti ad una specificata condizione; gli *iteratori resettabili* possono essere riposizionati sul primo elemento dell'iterabile al quale sono riferiti.

Un iteratore, nella sua forma minimale, può essere descritto mediante la seguente interfaccia:





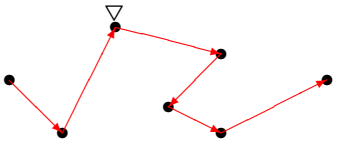


Figura 11.1: Schema di un iteratore $\triangledown$ che visita gli oggetti $\bullet$ di un iterabile.

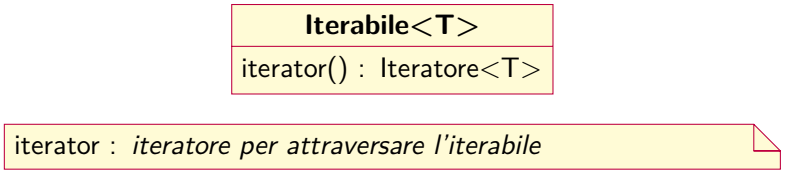
La creazione di un iteratore avviene a carico dell'oggetto che si rende visitabile. Viene assunta l'ipotesi che ogni iteratore, appena creato, venga posizionato sul primo elemento dell'iterabile; successivamente, dopo un suo uso, l'iteratore diventa inutilizzabile, in quanto non è possibile riposizionarlo sul primo elemento.

[1] Nel contesto delle basi di dati un iteratore viene spesso detto *cursore*.

11.2 Iterabili

Un *iterabile* è un oggetto composto da altri oggetti, che può essere attraversato ed analizzato, fornendo l'accesso, uno dopo l'altro, ai suoi elementi. L'attraversamento di un iterabile viene realizzato mediante degli iteratori che consentono, dall'esterno, il movimento attraverso l'iterabile, fornendo, una dopo l'altra, le componenti che incontra. Questa tecnica, introdotta agli inizi degli anni '90, costituisce un potente design pattern relativo alle strutture dati.

Un iterabile è in grado di fornire un iteratore mediante il quale permette di essere attraversato. Coerentemente con questa tecnica, un iterabile risulta caratterizzato dalla seguente interfaccia:



Nel caso in cui un iterabile a sia composto da un numero finito di elementi, è possibile elaborarli con uno schema di algoritmo come il seguente:

Algoritmo 1 - Visita di un iterabile a mediante iteratore esplicito

```
1: k ← a.iterator()
2: while k.hasNext() do
3:     x ← k.next()
4:     elabora x
5: end while
```

È possibile visitare un iterabile con un meccanismo che rende invisibile l'iteratore, mediante un algoritmo avente la seguente struttura [2]:

Algoritmo 2 - Visita di un iterabile a mediante iteratore implicito

```
1: for x in a do
2:     elabora x
3: end for
```

[2] In Java la sintassi è la seguente: `for (E x:a) { elabora x }`

11.3 Generatori

Un *generatore* è un particolare iteratore che genera altri oggetti. Corrisponde all'idea di un oggetto che contiene "potenzialmente" altri oggetti, che non vengono necessariamente memorizzati, ma generati nel momento in cui vengono richiesti, in base ad un criterio intrinseco e specifico del generatore stesso. Per questa caratteristica un generatore è in grado di fornire anche una sequenza illimitata di oggetti.

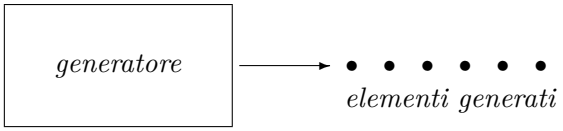


Figura 11.2: Schema della generazione di elementi.

Un generatore non ha bisogno di rapportarsi ad un iterabile di cui fornire gli elementi.

Esempio 11.3.1 - In molte situazioni serve generare delle sequenze o successioni di oggetti; risulta indicato il meccanismo dei generatori:

- *sequenze*: sequenza delle stringhe che costituiscono gli anagrammi di una data parola, righe del risultato di una interrogazione su una base di dati relazionale, ...
- *successioni*: numeri naturali, numeri di Fibonacci, numeri casuali, password, ...

11.4 Contenitori

Un *contenitore* [3] è un oggetto contenente altri oggetti. Per indicare che C è un contenitore di generici oggetti di tipo E si utilizza la notazione $C<E>$ oppure C_E. Graficamente un generico contenitore di oggetti può essere descritto come nella figura 11.3.

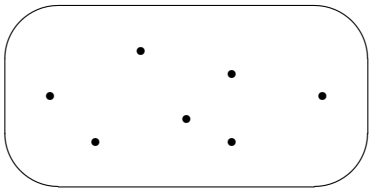


Figura 11.3: Contenitore di oggetti.

[3] In alcuni linguaggi di programmazione (Java, Python, ...) i contenitori vengono denominati *collezioni*.

Un contenitore si caratterizza per il fatto che gli elementi contenuti non vengono generati dal contenitore ma vengono dapprima inseriti nel contenitore e successivamente acceduti ed estratti. In base a questa caratterizzazione, in generale, su un contenitore risultano applicabili le seguenti categorie di operazioni: *inserimento*, *eliminazione* ed *accesso* agli elementi, *interrogazione* dello stato del contenitore.

Un contenitore, indipendentemente dalla sua struttura organizzativa interna, deve poter essere ispezionato e consentire l'accesso ai suoi elementi. L'esigenza di attraversare un contenitore è così sentita che, nei vari linguaggi di programmazione, i contenitori sono intrinsecamente iterabili.

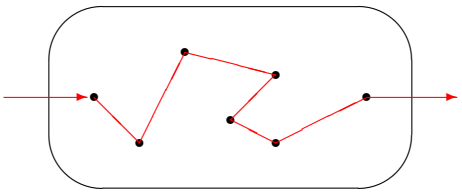
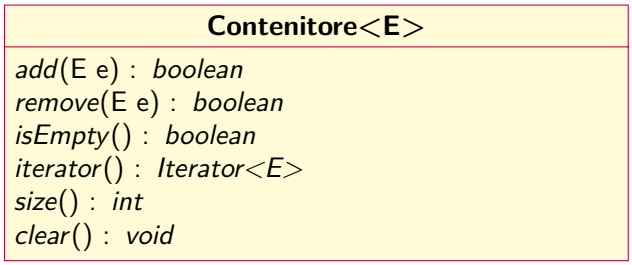
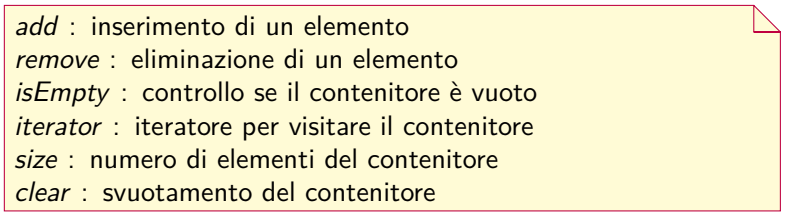


Figura 11.4: Contenitore iterabile.

In base alle precedenti considerazioni un contenitore iterabile composto da elementi di tipo E risulta caratterizzato dalla seguente interfaccia:





Un contenitore iterabile consente l'implementazione di interessanti algoritmi di elaborazione, indipendentemente dalla struttura dati fisica usata per memorizzare gli elementi; ad esempio: contare gli elementi, ricercare se un dato elemento è presente, determinare l'elemento minimo.

L'attraversamento di un contenitore iterabile viene solitamente svolto mediante un ciclo come descritto nell'algoritmo 1 e nell'algoritmo 2.

Nota. Un fastidioso problema di interferenza fra i metodi di un contenitore ed i metodi degli iteratori che lo attraversano insorge quando i metodi modificatori di un contenitore ed i metodi di accesso al contenitore mediante degli iteratori vengono usati contemporaneamente e non operano in modo mutuamente esclusivo, nel senso che, ad esempio, mentre si sta attraversando un contenitore con un iteratore, il contenitore viene modificato. A seguire sono descritte delle possibili soluzioni:

- creare una copia del contenitore nel momento in cui viene creato l'iteratore e, per tutta la sua vita, l'iteratore fa riferimento a questa copia
- il contenitore viene bloccato e reso non modificabile (*read-only*) quando ci sono degli iteratori attivi su di lui
- gli iteratori attivi sul contenitore vengono invalidati nel momento in cui il contenitore viene modificato
- gli iteratori attivi sul contenitore vengono opportunamente adattati per tener conto della sopravvenuta modifica al contenitore

11.5 Contenitori di contenitori

I contenitori possono essere classificati come segue:

- *contenitori semplici o piatti*: gli elementi contenuti nel contenitore sono oggetti elementari atomici.

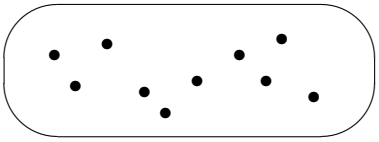


Figura 11.5: Contenitore semplice o piatto.

- *contenitori composti o strutturati*: gli elementi contenuti nel contenitore possono essere a loro volta dei contenitori; questa possibilità consente l'aggregazione di elementi secondo schemi gerarchici.

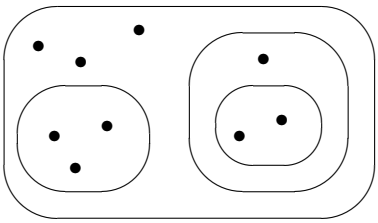


Figura 11.6: Contenitore composto o strutturato.

Un contenitore può contenere oggetti generici; in particolare un contenitore può contenere altri contenitori come suoi elementi. Quando si elabora un contenitore considerando i suoi elementi "di primo livello" si parla di *elaborazione superficiale*; quando l'elaborazione viene trasmessa ricorsivamente sugli elementi che sono a loro volta dei contenitori si parla di *elaborazione in profondità*. Ad esempio, con riferimento all'esempio riportato nella figura 11.7, il "conteggio superficiale degli elementi" fornisce il valore 5; il "conteggio in profondità" fornisce il valore 9; la "ricerca superficiale del valore 6" fornisce il valore *false*, mentre la "ricerca in profondità del valore 6" fornisce il valore *true*.

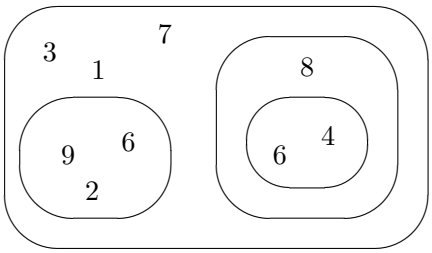


Figura 11.7: Contenitore di contenitori.

Molte operazioni di elaborazione di un contenitore possono essere svolte utilizzando il meccanismo di attraversamento predisposto dal contenitore stesso; per elaborare un contenitore di contenitori è necessario differenziare l'elaborazione superficiale dall'elaborazione in profondità. Per questo si ricorre ad un predicato per decidere se un elemento x è un atomo:

$$x.isAtom()$$

ed a un predicato per decidere se un contenitore è vuoto:

$$x.isEmpty()$$

Questo schema si fonda sulla possibilità di indagare se un oggetto è istanza di una data classe (o interfaccia) mediante l'operatore binario booleano infisso *instanceof*. Questa possibilità evidenzia, fra l'altro, l'utilità delle interfacce vuote, le quali costituiscono un meccanismo per "timbrare" gli oggetti appartenenti a delle specifiche classi.

L'elaborazione può basarsi sullo schema descritto nell'algoritmo 3.

Algoritmo 3 - Elaborazione di un elemento x di un contenitore iterabile

```
1: if x.isAtom() then
2:     elaborazione dell'elemento x
3: else
4:     elaborazione ricorsiva del contenitore x
5: end if
```

Innestando l'algoritmo 3 al posto della linea 2 nell'algoritmo 1 si ottiene un algoritmo per elaborare in profondità un generico contenitore iterabile.

11.6 Tipologie di contenitori

Come tutte le strutture astratte, i contenitori non rendono visibile all'esterno la loro struttura organizzativa interna; essi risultano manipolabili solo attraverso delle operazioni che agiscono secondo specifici criteri e vincoli che caratterizzano le diverse tipologie di questi contenitori. In base a questi criteri, i contenitori possono essere classificati in sottocategorie che si diversificano fra loro per le specifiche operazioni di inserimento, accesso, modifica, eliminazione, estrazione ed interrogazione sugli elementi. Il comportamento ed il risultato di questi metodi dipende dai seguenti parametri:

- caratteristiche del contenitore al quale viene applicata l'operazione
- ordine temporale con cui gli elementi vengono inseriti
- valore dell'elemento che si tenta di inserire/modificare/eliminare
- valori degli elementi presenti nel contenitore
- politica di gestione dei duplicati
- politica di gestione delle modalità di accesso agli elementi del contenitore (secondo il pattern *iterabile-iteratore* oppure mediante metodi specifici)

Imponendo alcuni vincoli alle operazioni su un contenitore si ottengono diverse tipologie di contenitori. Queste tipologie sono caratterizzate da un insieme di operazioni generali applicabili ad ogni contenitore e da un insieme di operazione specifiche che caratterizzano ciascuna tipologia, come descritto nei paragrafi che seguono.

Nei linguaggi di programmazione moderni queste diverse tipologie di contenitori sono generalmente disponibili mediante appositi pacchetti di classi.

11.7 Sequenze

Le *sequenze* costituiscono una delle forme più utilizzate di contenitori. Sono caratterizzate da una struttura lineare di elementi disposti in fila; le operazioni di inserimento, accesso e modifica vengono precisate in funzione di un indice di posizione.

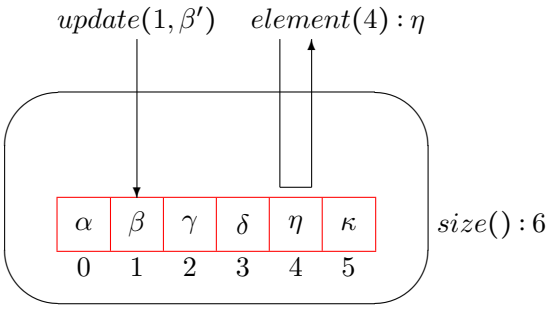
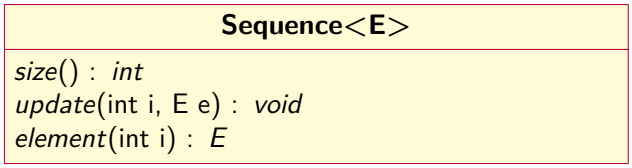
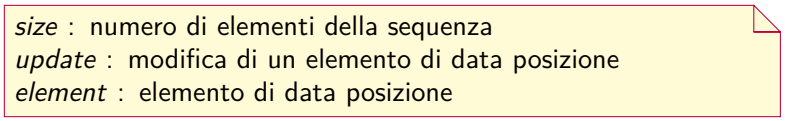


Figura 11.8: Una *sequenza* composta da 6 elementi. Sono evidenziate le operazioni di modifica dell'elemento presente ad una data posizione, l'operazione di accesso all'elemento posto ad una data posizione e l'operazione che fornisce il numero di elementi presenti.

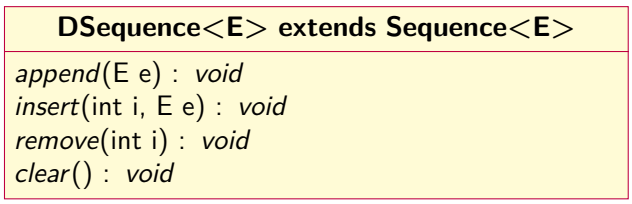
Le seguenti operazioni caratterizzano una *sequenza statica*, ossia una sequenza la cui dimensione viene fissata, in modo definitivo, al momento della creazione. Il seguente diagramma descrive l'interfaccia di una classe generica *Sequence<E>* costituita dalle sequenze di elementi di una classe E.

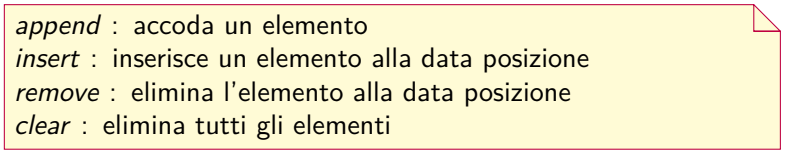




Basandosi sull'innterfaccia definita sopra si possono realizzare i tradizionali algoritmi di ordinamento, ricerca e molti altri.

La precedente interfaccia può essere estesa per descrivere delle sequenze dinamiche in cui, oltre alle modifiche agli elementi, sono previste delle operazioni di modifica della struttura della sequenza.





Una sequenza viene generalmente realizzata mediante un array in cui memorizzare gli elementi. Nel caso di una sequenza dinamica risulta preferibile, per questioni di efficienza nell'esecuzione delle operazioni, utilizzare una struttura fisica dinamica costituita da una lista concatenata di nodi.

11.8 Pile

Le *pile* o *stack* sono dei contenitori nei quali l'inserimento e l'estrazione di elementi avviene con la politica *L*ast-*I*n-*F*irst-*O*ut (*LIFO*); è accessibile ed è possibile estrarre solo l'ultimo elemento inserito. Considerando uno stack come una struttura lineare di elementi, l'inserimento, l'accesso e la cancellazione di elementi sono possibili solo su un'estremità.

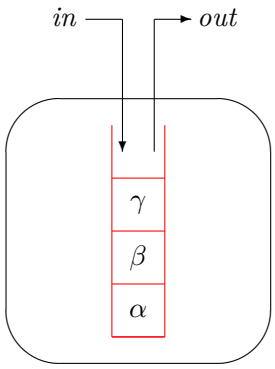
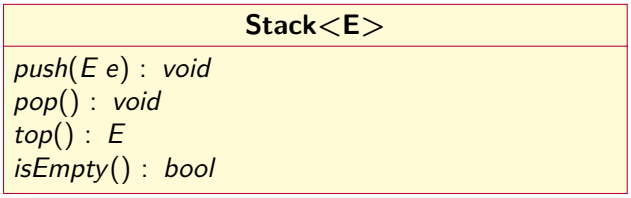
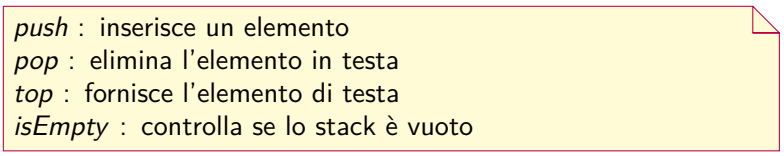


Figura 11.9: Uno *stack*, ottenuto con l'inserimento sequenziale degli elementi α, β e γ. L'elemento γ è l'unico elemento accessibile.

Il seguente diagramma descrive l'interfaccia di una classe generica *Stack<E>* costituita dalle pile elementi di una classe *E*.





Uno stack viene generalmente realizzato mediante un array per memorizzare gli elementi ed una variabile intera (*stack pointer*) che indica la posizione nell'array dell'ultimo elemento inserito.

11.9 Code

Le *code* o *queue* sono dei contenitori nei quali l'inserimento e l'estrazione di elementi avviene con la politica *F*irst-*I*n-*F*irst-*O*ut (*FIFO*); è accessibile ed è possibile estrarre solo il primo elemento inserito. Considerando una coda come una struttura lineare di elementi, l'inserimento avviene su un estremo e l'accesso e la cancellazione sull'altro estremo.

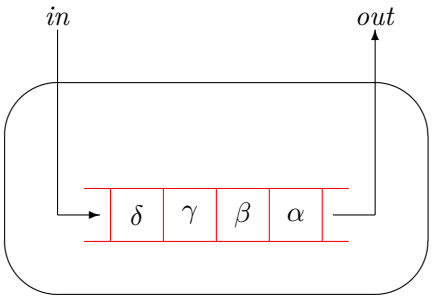
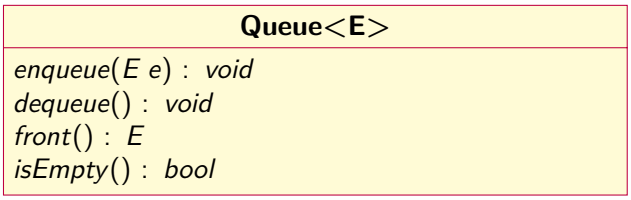
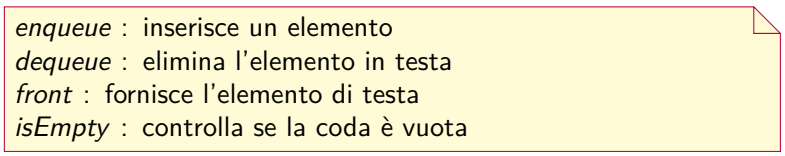


Figura 11.10: Una *coda*, ottenuta con l'inserimento sequenziale degli elementi α, β, γ e δ. L'elemento α è l'unico elemento accessibile.

Il seguente diagramma descrive l'interfaccia di una classe generica *Queue<E>* costituita dalle code di elementi di una classe *E*.





Una coda viene solitamente realizzata mediante un *array circolare* dove gli elementi della coda vengono inseriti in una porzione dell'array le cui estremità sono individuate da due indici di posizione, considerando l'array circolare ottenuto saldando virtualmente fra loro le due estremità dell'array e gestendo lo sforamento degli indici di posizione rientrando dall'altro estremo.

11.10 Code a priorità

Una *coda a priorità* o *priority queue* è un contenitore di elementi sui quali è definito un criterio d'ordine. È possibile inserire gli elementi senza alcun vincolo ed è possibile accedere ed estrarre l'elemento minimo fra quelli presenti. Generalmente una cosa a priorità viene configurata come contenitore iterabile.

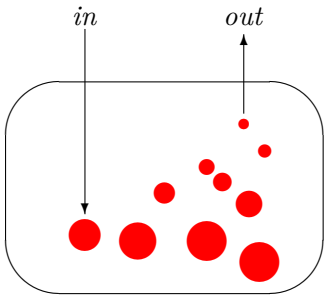
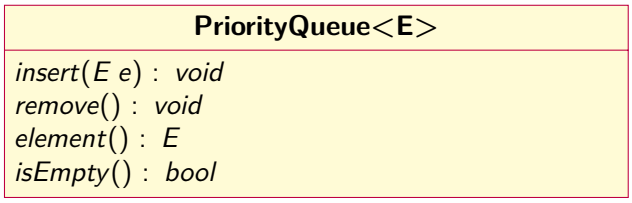
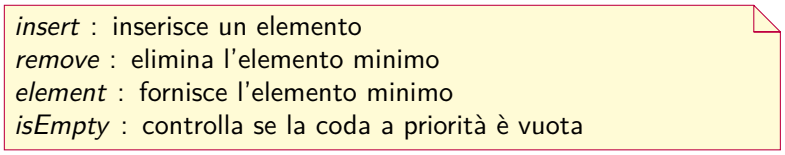


Figura 11.11: Una *coda a priorità*.

Il seguente diagramma descrive l'interfaccia di una classe generica *PriorityQueue<E>* costituita dalle code a priorità di elementi di una classe E.

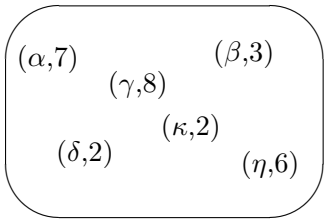




Per rendere efficienti le operazioni di inserimento e cancellazione, una coda a priorità viene rappresentata mediante una struttura ad albero (*heap*) avente la caratteristica di mantenere gli elementi ordinati, garantendo una complessità costante $O(1)$ per l'operazione di accesso (all'elemento minimo) e, per le operazioni di inserimento e cancellazione, con complessità asintotica pari a $O(\log n)$, indicando con n il numero di elementi inseriti.

11.11 Mappe e Dizionari

Mappe e *dizionari* sono contenitori di coppie della forma (k, v) dove k è una *chiave* e v è il *valore* associato alla chiave k. Le chiavi possono essere di un generico tipo (naturali, interi, stringhe, oggetti, ...). Nelle mappe non sono ammessi valori duplicati delle chiavi, mentre ciò è ammesso nei dizionari.



Dal punto di vista logico una mappa si comporta come un array con la differenza che l'accesso agli elementi avviene mediante una chiave, invece di un indice naturale come avviene per gli array. Per questa somiglianza una mappa viene detta anche *array associativo*.

α	β	γ	δ	η	κ
7	3	8	2	6	2

Il seguente diagramma descrive le interfacce di due classi generiche *Map<K,E>* e *Dictionary<K,E>*, costituite dalle mappe e dai dizionari con chiavi di classe K e valori di classe E.

Map<K,E> - Dictionary<K,E>

insert(*K*, *E*) : *void*
remove(*K*) : *void*
element(*K*) : *E*
isEmpty() : *bool*
size() : *int*
iterator() : *Iterator<K>*

insert : inserisce un elemento (coppia (k, e))
remove : elimina l'elemento associato alla data chiave
element : fornisce il valore associato alla data chiave
isEmpty : controlla se il contenitore è vuoto
size : numero di elementi del contenitore
iterator : iteratore per visitare le chiavi degli elementi

11.12 Implementazione di contenitori

Un contenitore può essere realizzato memorizzando gli elementi in una struttura dati fisica; tipicamente vengono usati array, catene e strutture con puntatori. Alcuni linguaggi di programmazione predisposngono questi contenitori in modo nativo oppure mediante librerie di funzioni o classi di oggetti [4].

Le varie classi di contenitori vengono realizzate mediante una classica tecnica implementativa che si svolge attraverso i seguenti passi:

1. definizione di un'interfaccia I che caratterizza il contenitore, definendo i prototipi delle operazioni [5]
2. realizzazione di una classe astratta A che implementa l'interfaccia I e che realizza i metodi indipendenti dalla struttura dati concreta che verrà utilizzata
3. scelta di una struttura dati concreta S (già realizzata o da realizzarsi) adeguata per supportare in modo efficiente le operazioni nell'interfaccia e definizione di una classe concreta C che estende la classe astratta A e che, usando la struttura dati concreta S, definisce i metodi mancanti non ancora definiti (in quanto richiedenti la conoscenza dei dettagli implementativi della specifica struttura
4. codifica in un linguaggio di programmazione delle operazioni specificate al punto 1. adottando la struttura dati descritta al punto 2.

In taluni casi le prime due fasi vengono fuse in un'unica fase consistente nella realizzazione diretta di una classe astratta.

Il criterio in base al quale gli elementi vengono inseriti nel e forniti dal contenitore dipende e caratterizza ciascuno specifico contenitore. Tale criterio indirizza inoltre la scelta della struttura dati fisica utilizzata per memorizzare gli elementi nel contenitore.

[4] Ad esempio la classe *Vector* o la classe *ArrayList* delle librerie Java.
[5] Oppure usando un costrutto sintattico specifico del linguaggio; ad esempio `interface` nel linguaggio Java.

ESERCIZI

11.1 Realizzare un *generatore di numeri di Fibonacci*, adottando il pattern *generatore*.

11.2 Implementare l'interfaccia `Generatore` per realizzare dei *generatori ciclici* di numeri da 1 a 12, secondo l'usuale avanzare delle ore sull'orologio. Scrivere una porzione di applicazione che visualizzi le ore dalla posizione 100 alla 120.

11.3 Realizzare un *generatore di password*, adottando il pattern *generatore*.

11.4 Adottando la tecnica del pattern *generatore*, realizzare un *mazzo di carte da gioco* dal quale è possibile estrarre delle carte.

11.5 Implementare l'interfaccia *Generatore* per realizzare una classe *Tabellina* i cui oggetti sono dei generatori dei multipli positivi di un dato numero naturale. Scrivere una porzione di applicazione che utilizzi la tabellina del 13 al fine di visualizzare i primi 10 multipli del 13 maggiori di 100.

11.6 Realizzare un generatore di anagrammi (*anagrammatore*) di una data stringa.

11.7 Realizzare un generatore illimitato che fornisce la successione di frazioni secondo l'ordinamento di Dedekind.

11.8 Discutere quali inconvenienti potrebbero insorgere se la definizione del metodo `minoreUguale` di un comparatore non soddisfacesse alle proprietà di una relazione d'ordine.

11.9 Dato un iterabile (finito) generare tutte le coppie dei suoi elementi.

11.10 Stabilire se in un contenitore iterabile è presente superficialmente un dato elemento.

11.11 Convertire un contenitore iterabile in stringa (adottando delle idonee convenzioni).

11.12 Decidere se un contenitore iterabile è piatto.

11.13 Decidere se un contenitore iterabile è composto da elementi tutti distinti fra loro.

11.14 Determinare il numero di elementi elementari (non contenitori) presenti in un contenitore iterabile, mediante un'elaborazione superficiale e mediante un'elaborazione in profondità.

11.15 Determinare l'elemento minimo presente in un contenitore iterabile, mediante un'elaborazione superficiale e mediante un'elaborazione in profondità.

11.16 Decidere se in un contenitore iterabile è presente un dato elemento, mediante un'elaborazione superficiale e mediante un'elaborazione in profondità.

11.17 Determinare la frequenza con la quale è presente un dato elemento in un contenitore iterabile, mediante un'elaborazione superficiale e mediante un'elaborazione in profondità.

11.18 Determinare il prodotto cartesiano di due *iterabili* ($A \times B$ oppure A^2).

11.19 Discutere cosa possa significare il concetto di *uguaglianza* fra due contenitori. Stabilire se due contenitori sono uguali.

11.20 Determinare il massimo livello di annidamento di un contenitore di contenitori; si assuma la convenzione che un contenitore piatto abbia livello di annidamento pari a 0.

11.21 Descrivere la situazione finale che si ottiene, a partire da una situazione iniziale di *stack vuoto*, eseguendo la seguente sequenza di operazioni: *push*(A), *push*(B), *pop*, *push*(C), *push*(D), *pop*, *push*(E). Con riferimento alla situazione finale, indicare quali sono gli elementi direttamente accessibili.

11.22 Descrivere la situazione finale che si ottiene, a partire da una situazione iniziale di *coda vuota*, eseguendo la seguente sequenza di operazioni: *enqueue*(A), *enqueue*(B), *dequeue*, *enqueue*(C), *enqueue*(D), *dequeue*, *enqueue*(E). Con riferimento alla situazione finale, indicare quali sono gli elementi direttamente accessibili.

11.23 Descrivere la situazione finale che si ottiene, a partire da una situazione iniziale di *coda a priorità vuota*, eseguendo la seguente sequenza di operazioni: *insert*(5), *insert*(7), *remove*, *insert*(4), *insert*(2), *remove*, *insert*(6). Con riferimento alla situazione finale, indicare quali sono gli elementi direttamente accessibili.

11.24 Stabilire se in una coda è presente un dato elemento. Risolvere il problema nel caso in cui la coda sia iterabile e nel caso in cui non lo sia.

11.25 Valutare un'espressione aritmetica espressa in notazione RPN, usando una pila.

11.26 Ordinare una sequenza, usando una coda a priorità.

11.27 Una *doppia coda* o *deque* (*d*ouble *e*nded *que*ue) è una coda in cui si possono inserire, eliminare ed accedere agli elementi da entrambe le estremità. Scrivere un diagramma UML che specifichi le operazioni che caratterizzano una doppia coda.

11.28 Discutere come un array può essere utilizzato per realizzare le seguenti forme di contenitori: sequenze, sequenze dinamiche, pile, code, code a priorità, mappe e dizionari. In particolare si precisi la complessità delle operazioni di inserimento, accesso e modifica degli elementi.